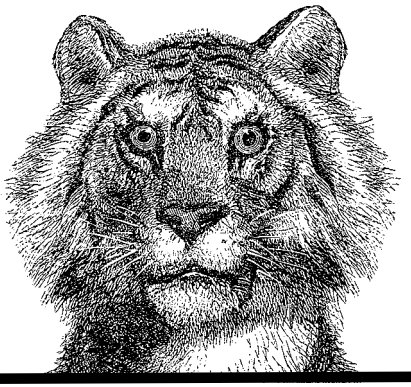


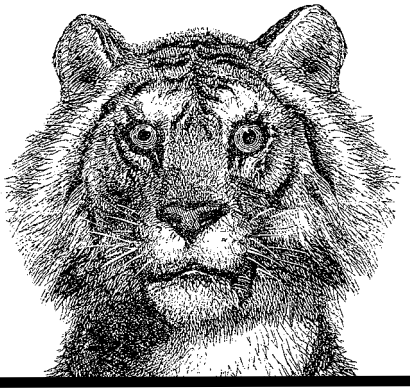
JAVA™ IN A NUTSHELL

A Desktop Quick Reference

Fourth Edition

David Flanagan

O'REILLY®



JAVA™

СПРАВОЧНИК

Четвертое издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2004

Дэвид Флэнаган

Java. Справочник
4-е издание

Перевод К. Волкова и В. Шальнева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*И. Васильев*
Редактор	*Ю. Кунивер*
Корректор	*С. Беляева*
Верстка	*Ю. Кунивер*

Флэнаган Д.
Java. Справочник, 4-е издание – Пер. с англ. – СПб: Символ-Плюс, 2004. – 1040 с., ил.
ISBN 5-93286-067-7

Этот бестселлер представляет собой краткий справочник, необходимый каждому Java-программисту. Книга содержит ускоренный вводный курс в язык Java и обзор ключевых API, благодаря чему опытные программисты смогут сразу перейти к написанию Java-кода. Четвертое издание «Java. Справочник» посвящено Java 1.4 и включает краткое описание синтаксиса Java, изложение объектно-ориентированных возможностей Java и обзор основных API Java, в котором объясняется, как выполнять такие стандартные задачи, как работа со строками, ввод/вывод, обработка XML, SSL и поддержка потоков при помощи классов и интерфейсов, составляющих платформу Java 2.

Книга также содержит заслуживающий доверия справочник O'Reilly по всем классам, входящим в базовые Java-пакеты, такие как *java.lang*, *java.io*, *java.beans*, *java.math*, *java.net*, *java.text* и *java.util*. Справочник охватывает множество новых классов Java 1.4, включая NIO (новый интерфейс ввода/вывода), протоколирование и средства работы с XML.

При изучении Java неоценимую помощь окажет вам и книга «Java в примерах. Справочник» (Символ-Плюс, 2003), которая отлично дополняет данное издание.

ISBN 5-93286-067-7
ISBN 0-596-00283-1 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 29.12.2003. Формат 70x100 1/16. Печать офсетная.
Объем 65 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Эта книга посвящается всем,
кто учит жить мирно и противостоит насилию.

Оглавление

Предисловие

Данная книга представляет собой настольный справочник для программистов, пишущих на Java™. Она предназначена для того, чтобы лежать рядом с вашей клавиатурой и быть под рукой, когда вы программируете. Первая часть книги – краткое, «сухое» введение в язык программирования Java и в базовый набор функций *прикладного программного интерфейса* (API) платформы Java. Вторая часть – это справочный раздел, который дает лаконичное описание деталей большинства классов и интерфейсов базовых API Java. Книга охватывает версии Java 1.0, 1.1, 1.2, 1.3 и 1.4.

Изменения в четвертом издании

Так как платформа Java снова значительно расширилась с выпуском Java 1.4, соответственно увеличился и объем данной книги. Далее приводятся некоторые из наиболее важных особенностей, появившихся в Java 1.4 (а также в этой книге):

Оператор assert (Assert statement)

Язык Java был расширен для поддержки утверждений при помощи оператора `assert`. Этот новый оператор описан в главе 2.

Постоянство JavaBeans (JavaBeans persistence)

Компоненты JavaBeans и связанные с ними объекты могут быть сериализованы в XML-документы. Более подробную информацию можно получить в разделе `java.beans.XMLEncoder` главы 9.

Новый прикладной программный интерфейс ввода/вывода (I/O API)

Для поддержки высокопроизводительного и неблокирующего файлового и сетевого ввода/вывода в версию Java 1.4 включен новый прикладной программный интерфейс. См. описание пакета `java.nio` и его подпакетов в главе 14. В главе 4 приведено несколько примеров использования этого важного интерфейса.

Прикладной программный интерфейс маршрутов сертификации (Certification path API)

Пакет `java.security.cert` был расширен новыми классами и интерфейсами для создания цепочек сертификатов или «маршрутов сертификации», обычно используемых в сетевой аутентификации.

Прикладной программный интерфейс протоколирования (Logging API)

Новый пакет `java.util.logging` определяет мощный и гибкий каркас протоколирования для приложений Java.

Прикладной программный интерфейс для работы с настройками пользователя (Preferences API)

Пакет `java.util.prefs` определяет прикладной программный интерфейс, позволяющий приложениям хранить и запрашивать значения настроек пользователей и общесистемных опций конфигурации.

Регулярные выражения (Pattern matching with regular expressions)

Новый пакет `java.util.regex` позволяет обрабатывать текстовые шаблоны с помощью регулярных выражений в стиле Perl.

Защищенные сетевые сокеты (Secure network sockets)

Расширение прикладного программного интерфейса протокола защищенных сокетов (Java Secure Sockets Extension (JSSE) API), определенное в новых пакетах `javax.net` и `javax.net.ssl`, обеспечивает защиту при работе в сети посредством протоколов SSL и TLS.

Сетевая аутентификация и авторизация (Network authentication and authorization)

Служба аутентификации и авторизации Java (Java Authentication and Authorization Service (JAAS)) определяется в пакете `javax.security.auth` и его подпакетах. JAAS позволяет приложениям Java безопасно устанавливать личность пользователя и выполнять код в соответствии с политикой безопасности на базе прав данного пользователя.

Синтаксический анализ и преобразование XML (XML parsing and transformations)

Прикладной программный интерфейс Java для обработки данных XML (Java API for XML Processing (JAXP)) определяется в пакетах `javax.xml.parsers`, `javax.xml.transform` и их подпакетах. JAXP предоставляет средства для синтаксического анализа XML-документов с использованием стандартных прикладных программных интерфейсов SAX и DOM, а также преобразование содержимого этих документов с использованием XSLT. Подобно JAXP, стандартные прикладные программные интерфейсы DOM и SAX являются частью платформы Java 1.4. Вы сможете найти их в пакете `org.w3c.dom`, описанном в главе 23, а также в пакете `org.xml.sax` и подпакетах, представленных в главе 24.

В главе 4 вы найдете примеры использования большинства прикладных программных интерфейсов.

Помимо добавления нового материала, было внесено несколько изменений и в структуру книги. В предыдущих изданиях на каждый пакет выделялась отдельная глава справочного раздела. Сорок шесть отдельных пакетов, освещенных в данном издании, привели бы к чрезмерному количеству глав. Поэтому связанные пакеты (с общим префиксом) сгруппированы в одну главу, благодаря чему количество глав справочного раздела сокращено до 15. Однако из-за того что краткий справочник организован в строгом алфавитном порядке, разбиение на главы выполнено не по тематическому признаку. Интересующий вас раздел можно найти путем пролистывания раздела, как в случае словаря или телефонного справочника.

Кроме того, по причине большого количества новых пакетов пришлось убрать рисунки, представляющие иерархию пакетов, которые располагались в начале каждой главы в прошлых изданиях. Данные рисунки легли непомерным грузом на плечи группы технических иллюстраторов компании O'Reilly & Associates, Inc., поскольку их нужно было тщательно вырисовывать вручную. Более того, рисунки оказались не

так полезны, как это представлялось вначале. Для данного издания было решено, что рисунки не стоят затрачиваемых на них средств. Если вы все же принадлежите к числу тех немногочисленных читателей, которым пригодились эти диаграммы, приношу свои извинения за их удаление.

Есть две новые особенности в справочном разделе, способные компенсировать потерю иерархических диаграмм. Во-первых, раздел ссылок для каждого пакета включает в себя перечень всех интерфейсов и классов пакета. Данные в этом списке будут группироваться по категориям (например, интерфейсы, классы и исключения) и по иерархическому представлению. Такой перечень, не будучи графическим, предоставляет такое же количество информации, что и прежние иерархические диаграммы. Во-вторых, для каждого класса и интерфейса иерархический подраздел был переведен из неудобного текстового формата в более совершенный графический.

Структура книги

В первых восьми главах данной книги описаны язык Java, платформа Java и средства разработки Java, которые входят в набор инструментальных средств разработки Java, поставляемый компанией Sun (Java SDK). Первые четыре главы являются базовыми, а последующие четыре будут интересны не всем Java-программистам.

Глава 1 «Введение»

Эта глава содержит обзор языка Java и Java-платформы, с объяснением важных возможностей и преимуществ Java. В конце главы новоиспеченный Java-программист знакомится с примером Java-программы и проходит ее шаг за шагом.

Глава 2 «Синтаксис Java: от подножия к вершине»

В этой главе подробно описывается язык программирования Java. Глава большая и обстоятельная. Опытные Java-программисты могут использовать ее в качестве справочника по языку. Прочитав данную главу, программисты со значительным опытом работы с языком C или C++ ознакомятся с синтаксисом Java. Однако глава не подразумевает обязательное наличие большого опыта программирования и не требует знания C или C++. Данное издание разработано таким образом, чтобы даже начинающие программисты со скромным опытом смогли разобраться с программированием на Java, тщательно изучив данную главу.

Глава 3 «Объектно-ориентированное программирование в Java»

В этой главе описаны пути использования базового синтаксиса Java, рассматриваемого в главе 2, при написании объектно-ориентированных программ на языке Java. Данная глава не предполагает наличия какого-либо предварительного опыта в ООП. Она может служить учебным пособием для начинающих программистов и справочником для опытных Java-разработчиков.

Глава 4 «Java-платформа»

В этой главе дается краткий обзор основных прикладных программных интерфейсов Java, рассматриваемых в данной книге. Глава содержит множество небольших примеров решения типовых задач с использованием классов и интерфейсов, входящих в платформу Java. Программисты, не знакомые с Java, особенно те, кому легче учиться на примерах, по достоинству оценят эту главу.

Глава 5 «Безопасность в Java»

В данной главе описана архитектура безопасности Java, которая позволяет запускать ненадежный код в безопасной среде, из которой он не сможет намеренно

причинить вред системе. Для каждого Java-программиста важно иметь хотя бы общее представление о механизмах безопасности Java.

Глава 6 «Компоненты JavaBeans»

В данной главе представлена структура компонентов Java (JavaBeans™) и объяснено, что должен знать программист для создания и применения многократно используемых встраиваемых классов Java, известных как компоненты (beans).

Глава 7 «Соглашения по программированию и документированию в Java»

В данной главе изложены важные общепринятые соглашения по написанию Java-программ. Здесь же описано, как сделать Java-код самодокументируемым путем включения в него специальных комментариев.

Глава 8 «Средства разработки Java»

В набор средств разработки Java (Java SDK) входит большое количество полезных инструментальных средств разработки. В частности, можно выделить интерпретатор и компилятор Java. В данной главе описаны эти средства.

Первые восемь глав помогут вам изучить язык Java и дадут возможность освоить и начать работать с прикладными программными интерфейсами Java. Однако большая часть книги (главы 9–24) является справочником по прикладным программным интерфейсам – детальным и одновременно лаконичным описанием, удобным в применении. Важно внимательно изучить первую главу в начале справочного раздела, где объясняется, как наиболее продуктивно использовать справочный материал. Также обратите внимание, что за главами справочного раздела следует заключительная глава под названием «Алфавитный указатель классов, методов и полей». По этому указателю можно отыскать имя класса и найти пакет, в котором он определен, или отыскать имя метода или поля и найти соответствующий пакет.

Книги по теме

O'Reilly издает целую серию книг по Java-программированию, включая несколько книг, дополняющих данное издание. Вот эти книги:

«Java Enterprise in a Nutshell»

Данная книга является кратким учебным пособием и справочником по таким прикладным программным интерфейсам Java для приложений масштаба предприятия, как JDBC, RMI, JNDI и CORBA.

«Java Foundation Classes in a Nutshell»

Эта книга является кратким учебным пособием и справочником по графике, графическому интерфейсу пользователя и по соответствующим прикладным программным интерфейсам платформы Java. Сюда входят апплеты, AWT, Java2D и Swing.

«Java Examples in a Nutshell»[1]

Книга содержит сотни законченных рабочих примеров, иллюстрирующих решения большинства распространенных задач, использующих базовый прикладной программный интерфейс Java, дополнительные программные интерфесы и программные интерфейсы для приложений уровня предприятия. Подобно главе 4 данного издания, книга «Java Examples in a Nutshell» значительно расширена, а все фрагменты кода доведены до уровня работающих примеров. Эта книга особенно по-

[1] Д. Флэнаган «Java в примерах. Справочник». – Пер. с англ. – СПб: Символ-Плюс, 2003.

лезна тем читателям, которые легче обучаются, экспериментируя с уже существующим кодом.

«J2ME in a Nutshell»

Данная книга является кратким учебным пособием и справочником по графике, работе в сети и прикладным программным интерфейсам баз данных платформы Java 2 Micro Edition (J2ME).

Полный перечень книг о Java от O'Reilly можно найти на сайте *http://java.oreilly.com/*. Ниже представлен список книг об основных прикладных программных интерфейсах Java (как, например, данная книга):

«Learning Java», Пэт Нимейер (Pat Niemeyer) и Джонатан Кнадсен (Jonathan Knudsen)

Исчерпывающее введение в Java с акцентом на создание клиентских приложений.

«Java Threads», Скотт Оукс (Scott Oaks) и Генри Уонг (Henry Wong)

Хотя Java и облегчает многопоточное программирование, оно все еще требует определенной сноровки. В данной книге изложена вся необходимая информация.

«Java I/O», Эллиот Расти Гарольд (Elliotte Rusty Harold)

Архитектура ввода/вывода в Java, основанная на концепции потоков, прекрасна сама по себе. В данной книге она рассматривается с той тщательностью, которую заслуживает.

«Java Network Programming», Эллиот Расти Гарольд (Elliotte Rusty Harold)

В книге детально рассматриваются сетевые прикладные программные интерфейсы Java.

«Java Security», Скотт Оукс (Scott Oaks)

В данной книге детально рассматриваются механизмы управления доступом, а также описываются механизмы проверки достоверности цифровых подписей и дайджестов сообщений.

«Java Cryptography», Джонатан Кнадсен (Jonathan Knudsen)

Всестороннее рассмотрение криптографических расширений Java, пакетов `javax.crypto.*` и всего того, что необходимо знать о криптографии в Java.

«Developing Java Beans», Роберт Ингландер (Robert Englander)

Полное руководство по написанию компонентов, работающих с прикладным программным интерфейсом компонентов Java (JavaBeans API).

Программные ресурсы Java в Интернете

Эта книга является пособием для быстрого доступа к часто используемой информации. Она не содержит, да и не может содержать всей информации о Java. В дополнение к уже перечисленным выше книгам далее представлены еще несколько ценных (и бесплатных) электронных источников информации по Java-программированию.

Главный сайт компании Sun, посвященный Java: *http://java.sun.com/*. Веб-сайт для Java-разработчиков: *http://developer.java.sun.com/*. Большая часть информации, представленной на этом сайте, защищена паролем. Чтобы получить к ней доступ, необходимо зарегистрироваться (бесплатно).

Компания Sun распространяет в электронном виде документацию для всех классов и методов Java (HTML-формат *javadoc*). Хотя искать необходимые данные в этой документации довольно неудобно, а информация местами неточная либо устаревшая, все же она станет отличным отправным пунктом в поиске более подробных сведений о

каком-либо отдельном пакете, классе, методе или поле Java. Если вы еще не обзавелись файлами *javadoc*, то последнюю доступную версию можно найти по ссылке на сайте *http://java.sun.com/docs/*. Кроме того, компания Sun распространяет в Интернете свои высококачественные учебные пособия по Java (Java Tutorial). Вы сможете просмотреть и загрузить их с сайта *http://java.sun.com/docs/books/tutorial/*.

Если вы хотите обсудить Java (на английском) в сети Usenet, обратите внимание на сетевую конференцию *comp.lang.java.programmer* и связанные с ней конференции *comp.lang.java.**. По адресу *http://www.afu.com/javafaq.htm* вы сможете найти исчерпывающие ответы на часто задаваемые вопросы (FAQ). Этот документ составляется Питером ван дер Линденом.

И наконец, не следует забывать о веб-сайте O'Reilly, посвященном Java, – *http://java.oreilly.com*. Здесь публикуются новости и комментарии по Java. В состав сети O'Reilly (*www.oreillynet.com*) входит сайт *onjava.com*, посвященный Enterprise Java.

Примеры в Интернете

Примеры из рассматриваемой нами книги доступны в сети Интернет. Их можно загрузить с домашней страницы книги, расположенной по адресу *http://www.oreilly.com/catalog/javanut4/*. Также на этом сайте публикуются важные замечания по книге и список найденных опечаток.

Типографские соглашения

В данной книге приняты следующие правила оформления:

Курсив

Используется для обозначения важных моментов в тексте, выделения терминов при их первом применении, а также для выделения команд, адресов электронной почты, веб- и FTP-сайтов, имен каталогов, файлов и названий сетевых конференций.

`Моноширинный шрифт`

Применяется для имен классов, констант, методов, пакетов и ключевых слов Java, а также для элементов XML, тегов и команд SQL.

`Моноширинный курсив`

Используется для имен аргументов функций, переменных, а также для выделения комментариев и обозначения элемента, подлежащего замене на фактическое значение в вашей программе.

`Моноширинный полужирный шрифт`

Используется в примерах программ и их фрагментах для выделения ключевых слов Java, имен классов, методов, полей, свойств и конструкторов.

Вопросы и комментарии

Свои комментарии и вопросы по данной книге направляйте издателю по адресу:

O'Reilly & Associates, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (для Соединенных Штатов и Канады)

(707) 829-0515 (международный либо местный)
(707) 829-1014 (факс)

Существует веб-страница, посвященная данной книге, которая содержит список опечаток, примеры и другую дополнительную информацию. Страница расположена по адресу:

http://www.oreilly.com/catalog/javanut4/

Задать технические вопросы либо прислать комментарии по этой книге можно по адресу:

bookquestions@oreilly.com

Более подробно о книгах, конференциях, центрах ресурсов и Сети O'Reilly можно узнать, посетив веб-сайт O'Reilly:

http://www.oreilly.com/

Как формируется справочник

Для самых любопытных читателей в этом разделе кратко объясняется процесс подготовки справочного материала для книги «Java. Справочник».

Моя система подготовки материала для справочного раздела эволюционировала по мере развития Java. Настоящая система является частью большой системы просмотра коммерческой документации, которую я сейчас разрабатываю (для получения более подробной информации посетите сайт *http://www.davidflanagan.com/Jude/*). Программа работает в два этапа: на первом этапе собирается и систематизируется информация по прикладным программным интерфейсам, а на втором эта информация выводится в виде глав справочника.

Первый этап начинается с прочтения файлов для всех классов и интерфейсов, подлежащих документированию. Практически вся информация по прикладным программным интерфейсам в справочнике берется из файлов классов. Главное исключение составляют имена аргументов методов, которые не сохраняются в файлах классов. Имена аргументов добываются путем синтаксического анализа исходного файла Java для каждого класса и интерфейса. Если исходные файлы недоступны, я получаю имена аргументов методов путем синтаксического анализа документации по прикладному программному интерфейсу, сгенерированной утилитой *javadoc*. Для получения информации по прикладному программному интерфейсу из исходных файлов и файлов *javadoc* я использую парсеры (parsers). Они созданы при помощи генератора парсеров Antlr, разработанного Терренсом Парром из Института Магеланг (Magelang Institute). (Более подробную информацию об этом мощном инструментальном программном средстве можно найти по адресу *http://www.antlr.org/*.)

Когда после прочтения файлов классов, исходных файлов и файлов *javadoc* получена информация по API, программа тратит некоторое время на создание и сортировку перекрестных ссылок. Затем вся информация по API сохраняется в одном большом файле данных.

На втором этапе информация считывается из файла данных и выводится в виде глав справочника посредством специализированного XML-документа. После того как информация выведена в формате XML, я передаю ее производственной команде компании O'Reilly. Там ее обрабатывают и преобразовывают в исходный код для troff. Исходный troff-код обрабатывается с помощью программы *groff* фонда GNU (*ftp://*

ftp.gnu.org/gnu/groff/) и специализированного набора troff-макросов. В результате получается текст на языке PostScript, который выводится на принтер.

Благодарности

Много людей помогали мне в создании этой книги, и я всем им признателен. Я в долгу перед множеством читателей первых трех изданий, приславших свои комментарии, предложения, сообщения об ошибках и похвальные отзывы. Множество песчинок их общего вклада рассеяны по всей книге. Я хочу извиниться перед теми читателями, чьи ценные предложения не вошли в данное издание.

Редактором первых трех изданий книги была Паула Фергюсон (Paula Ferguson), мой друг и коллега. Книга стала более солидной, понятной и полезной благодаря тому, Паула внимательно прочла текст и дала ценные советы. Редакторские обязанности увели Паулу от книг по Java к книгам по веб-программированию, и четвертое издание книги вышло под редакцией Боба Экстэйна (Bob Eckstein), внимательного редактора с потрясающим чувством юмора.

Рецензию на новый материал для этого издания давала группа специалистов компании Sun. Очень часто случалось так, что они рецензировали именно те части прикладного программного интерфейса, создателями которых они являлись. Мне очень повезло, что я смог обратиться за рецензией «прямо к первоисточнику». Я очень признателен этим разработчикам. Несмотря на свой загруженный график они нашли время прочесть и прокомментировать мои наброски. Я перечислю рецензентов в алфавитном порядке:

- Джош Блок (Josh Bloch), автор потрясающей книги «*Effective Java Programming Language Guide*», рецензировал новый материал по утверждениям и прикладному программному интерфейсу для работы с настройками пользователя.
- Грэхэм Гамильтон (Graham Hamilton) рецензировал материал по прикладному программному интерфейсу протоколирования.
- Джонатан Кнадсен (Jonathan Knudsen) рецензировал материалы по JSSE и маршрутам сертификации (он также является автором книг O'Reilly).
- Чарли Лай (Charlie Lai) рецензировал материалы по JAAS.
- Рэм Марти (Ram Marti) рецензировал материалы по JGSS.
- Филипп Милн (Philip Milne), прежде работавший в Sun, а теперь являющийся сотрудником компании Dresdner Kleinwort Wasserstein, рецензировал материалы по механизму постоянства Java-компонентов.
- Марк Рэйнхольд (Mark Reinhold) рецензировал материалы по `java.nio`. Марк заслуживает особой благодарности за рецензирование второго, третьего и четвертого издания книги.
- Андреас Стербенц (Andreas Sterbenz) и Брэд Вэтмор (Brad Wetmore) рецензировали материалы по JSSE.

В дополнение к уже перечисленным рецензентам из Sun следует упомянуть Рона Хитченса (Ron Hitchens), рецензировавшего мой новый материал по вводу/выводу, и моего редактора, Боба Экстайна, который проделал двойную работу в качестве технического рецензента по XML. Я искренне признателен каждому из перечисленных здесь людей за их кропотливо проделанную работу. Конечно, любые ошибки в книге допущены только мной.

Свою неоспоримую лепту в создание третьего издания внесли рецензенты, хорошо разбирающиеся в Java-платформе. Джошуа Блок (Joshua Bloch), один из ведущих авторов структуры Java-коллекций, рецензировал описания классов и интерфейсов коллекций. Джошуа также помог исследовать классы Java 1.3 `Timer` и `TimerTask`. Марк Рэйнхольд, создатель пакета `java.lang.ref`, объяснил его структуру и отрецензировал созданную мной документацию, посвященную этому пакету. Скотт Оукс рецензировал мои описания классов и интерфейсов защиты и криптографии Java. Джошуа, Марк и Скотт – специалисты компании Sun Microsystems, и я очень признателен им за уделенное мне внимание. Кроме них документацию по пакету `javax.crypto` и его подпакетам рецензировал Йон Ивз. Йон работал над технологическим применением криптографических расширений Java (об этом можно прочесть по адресу *http://www.aha.net.au/*). Его предложения оказались очень полезны. В настоящее время Йон работает в компании Fluent Technologies (*http://www.fluent.com.au/*) консультантом по Java и электронной коммерции. И наконец, в первой главе использованы комментарии от рецензентов, не знакомых с платформой Java. Кристина Берн (Christina Byrne) дала рецензию с точки зрения неопытного программиста, а Джудита Берн (Judita Byrne) из Virginia Power высказала свое мнение как квалифицированный COBOL-программист.

Для второго издания Джон Зуковски (John Zukowski) отрецензировал мой справочник по Java 1.1 AWT, а Джорж Риз (George Reese) – большую часть остального нового материала. Второе издание получило свое благословение от «команды-мечты» технических рецензентов компании Sun. Джон Роуз (John Rose), автор спецификации внутренних классов Java, рецензировал главу, посвященную внутренним классам. Марк Рэйнхольд, автор новых классов символьных потоков в `java.io`, рецензировал мою документацию по этим классам. Накул Сарайа (Nakul Saraiya), дизайнер нового прикладного программного интерфейса Java Reflection, рецензировал документацию по пакету `java.lang.reflect`. Я глубоко признателен этим разработчикам и архитекторам; благодаря их усилиям книга обрела достоверный и исчерпывающий материал.

Майк Лукидес (Mike Loukides) осуществлял общее руководство выпуском первого издания книги. Эрик Рэймонд (Eric Raymond) и Трой Даунинг (Troy Downing) рецензировали первое издание – они помогли мне увидеть ошибки и упущения и дали дельные советы по улучшению книги.

Как обычно, производственная команда O'Reilly с успехом справилась с задачей создания книги из предоставленных мною электронных файлов. Выражаю всем сотрудникам рабочей группы свою признательность.

Как всегда, признаюсь в любви к Кристи.

Дэвид Флэнаган (David Flanagan)
http://www.davidflanagan.com/
январь 2002

Часть I

Введение в Java

Часть I представляет введение в язык программирования Java и Java-платформу. В главах этой части содержится достаточно информации для того, чтобы немедленно приступить к работе с Java.

Глава 1

Введение

Добро пожаловать в мир Java! Начало этого раздела объясняет, что такое Java. Здесь описаны некоторые особенности языка Java, которые отличают его от других языков программирования. Далее следует введение в язык, представленное в форме учебного пособия. В нем содержится Java-программа, которую можно будет набрать, скомпилировать и запустить.

Что такое Java?

При обсуждении Java важно понимать различия между языком программирования Java, виртуальной машиной Java и Java-платформой. Язык программирования Java – это язык, на котором написаны Java-приложения (включая апплеты, сервлеты и JavaBeans-компоненты). Когда компилируется Java-программа, она переводится в байт-коды, представляющие собой переносимый машинный язык архитектуры центрального процессора, который известен как *виртуальная машина Java* (другие названия: Java VM и JVM). JVM может быть реализована напрямую в аппаратном обеспечении, но обычно она реализуется в форме программного обеспечения, которое интерпретирует и выполняет байт-коды.

Java-платформа отличается как от языка Java, так и от Java VM. Java-платформа является предопределенным набором Java-классов, которые существуют в каждой Java-инсталляции; эти классы могут использоваться всеми Java-программами. Еще Java-платформу иногда называют средой выполнения Java или основными Java APIs (прикладными программными интерфейсами). Java-платформа может быть расширена при помощи дополнительных стандартных расширений. Такие API-расширения существуют в некоторых Java-инсталляциях, но нет никакой гарантии, что они присутствуют в каждой из них.

Java: язык программирования

Язык программирования Java является современным объектно-ориентированным языком, синтаксис которого похож на синтаксис языка C. Люди, разрабатывавшие язык, старались сделать Java мощным языком и в то же время избежать чрезмерно сложной функциональности, в которой увязли другие объектно-ориентированные языки, такие как C++. Стремясь к простоте языка, создатели дали программистам возможность писать надежный код, лишенный ошибок. Благодаря хорошо продуманной структуре и передовым нововведениям, язык Java заслужил доверие про-

граммистов, которые обычно получают удовольствие от работы с Java после мучений с более сложными и менее мощными языками.

Виртуальная машина Java

JVM, или Java-интерпретатор, является важной частью каждой инсталляции Java. Java-программы задумывались как машинно-независимые, но их можно переносить только на те платформы, на которые был перенесен Java-интерпретатор. Sun поставляет VM-реализации для своей операционной системы Solaris, для платформ Microsoft Windows и Linux. Много других производителей, включая Apple и различных коммерческих поставщиков ПО под Unix, предлагают Java-интерпретаторы для своих платформ. Тем не менее Java VM предназначена не только для настольных компьютеров. Ее перенесли на игровые приставки; имеются усеченные версии даже для портативных устройств, работающих под Windows CE и PalmOS.

Хотя интерпретаторы обычно не считаются системами, обладающими большой производительностью, Java VM демонстрирует удивительно высокую производительность, которая постоянно повышается. Особенно следует отметить VM-технологию, известную как оперативная (just-in-time, JIT) компиляция, посредством которой байт-коды Java сразу же переводятся в машинный язык «родной» платформы, что увеличивает скорость выполнения циклически повторяющегося кода. Hotspot-технология от Sun является особенно удачной реализацией JIT-компиляции.

Java-платформа

Java-платформа так же важна, как и язык программирования Java и виртуальная машина Java. Все программы, написанные на языке Java, зависят от набора предопределенных классов[1], которые составляют платформу Java. Java-классы организованы в форме связанных групп, известных как *пакеты* (packages). В Java-платформе пакеты сгруппированы по функциональности: ввод/вывод, работа в сети, графика, создание пользовательского интерфейса, безопасность и многое другое.

Выход Java 1.2 был важной вехой для Java-платформы. Этот выпуск почти утроил количество классов и привнес новую функциональность. Для того чтобы этот момент не остался незамеченным, компания Sun назвала новую версию «платформа Java 2». Это название является торговой маркой и служит целям продвижения продукта на рынке; оно подчеркивает, как сильно изменилась платформа Java со времени выхода своей первой версии. И все же большинство программистов, говоря о Java-платформе, упоминают и ее официальный номер версии (1.4 на момент написания книги).

Важно понимать, что подразумевается под термином «платформа». Для программиста «платформа» – это те прикладные программные интерфейсы, которые он может использовать при написании программ. Эти интерфейсы обычно зависят от ОС какого-либо конкретного компьютера. То есть программист, пишущий программу, которая будет работать под MS Windows, должен использовать набор интерфейсов, отличный от того, который будет использовать программист, пишущий ту же программу, но уже под систему, основанную на Macintosh или Unix. С этой точки зрения Windows, Macintosh и Unix являются тремя различными платформами.

Java не является ОС. Тем не менее Java-платформа предоставляет прикладные интерфесы в таком объеме, который соответствует операционной системе. С помощью Java-

[1] Класс – это модуль Java-кода, который определяет структуру данных и набор методов (также называемых процедурами, функциями или подпрограммами), которые работают с этими данными.

платформы можно создавать приложения без ущерба для расширенных возможностей и функциональности, доступной тем программистам, которые пишут так называемые «родные приложения» (native applications), предназначенные для конкретной ОС. Приложение, написанное для Java-платформы, работает на любой ОС, которая поддерживает Java-платформу. На практике это означает, что вам не придется создавать отдельные версии ваших программ отдельно под Unix, Windows или Macintosh. Одна Java-программа может быть запущена на всех этих ОС. Вот почему девиз Sun для Java звучит так: «Напиши один раз, запускай где угодно!».

Становится понятно, почему такие компании, как Microsoft, несколько опасаются Java. Java-платформа – это не ОС, но для программистов она является альтернативной платформой для разработки приложений, и в этом контексте Java-платформа довольно популярна. Она уменьшает зависимость программистов от какой-либо конкретной ОС и, позволяя запускать программы на любой ОС, дает конечным пользователям больше свободы действий в выборе операционной системы.

Версии Java

На момент написания этой книги было известно пять основных версий Java:

Java 1.0

Это первая официальная версия Java. Она содержала 212 классов, организованных в форме восьми пакетов. Она была проста и удобна, но сейчас полностью устарела.

Java 1.1

Этот выпуск Java удвоил размер Java-платформы до 504 классов в 23 пакетах. Были введены внутренние классы (inner classes), что было важным изменением в самом языке Java; кроме того, выпуск привнес значительные нововведения, увеличившие производительность виртуальной машины Java. Существует большой набор веб-броузеров, совместимых с Java 1.1, поэтому несмотря на то, что эта версия устарела, она все еще используется для написания простых апплетов (апплет – это Java-программа, которая включается в веб-страницы). См. книгу «Java Foundation Classes in a Nutshell» (O'Reilly), в которой описываются апплеты.

Java 1.2

Это очень важный Java-выпуск; он утроил размер Java-платформы до 1520 классов в 59 пакетах. Важные дополнения: новый Swing GUI API плюс мощный и гибкий программный интерфейс коллекций (collections), предназначенный для работы с множествами (set), списками (list) и ассоциативными массивами (map) объектов. Из-за значительно расширенной функциональности, которую содержал выпуск 1.2, платформа получила новое название «платформа Java 2».

Java 1.3

Это по большей части сопровождающий выпуск, сфокусированный на исправлении ошибок и улучшении стабильности и скорости работы (улучшения включают в себя мощную виртуальную машину «HotSpot»). К платформе добавлены программные интерфейсы JNDI и Java Sound, которые ранее поставлялись в качестве расширений. Вот самые интересные новые классы: java.util.Timer и java.lang.reflect.Proxy. В общей сложности версия 1.3 Java-платформы содержит 1842 класса в 76 пакетах.

Java 1.4

Это еще один большой выпуск, который добавил важную функциональность и увеличил размер платформы на 62% – 2991 класс и интерфейс в 135 пакетах. Новая функциональность: мощный, низкоуровневый программный интерфейс вво-

да/вывода данных; поддержка паттернов регулярных выражений; программный интерфейс логирования; интерфейс для работы с настройками пользователя; новые классы-коллекции; механизм постоянства для JavaBeans, основанный на XML; поддержка XML-парсинга (parsing) с использованием DOM- и SAX-интерфейсов; аутентификация пользователя при помощи интерфейса JAAS; поддержка безопасного сетевого соединения с использованием протокола SSL; поддержка криптографии; новый программный интерфейс для чтения и записи графических файлов; интерфейс для сетевой печати; ряд новых компонентов графического пользовательского интерфейса в интерфейсе Swing и упрощенная архитектура «перетаскивания» (drag-and-drop) для Swing. В дополнение ко всем этим изменениям в платформе выпуск Java 1.4 содержит оператор `assert`, введенный в язык Java.

Для работы с Java 1.0 или Java 1.1 необходим соответствующий JDK (инструментальный комплект разработки в среде Java). С выходом Java 1.2 комплект JDK был переименован и теперь называется SDK (набор инструментальных средств разработки ПО), то есть Java 2 SDK, или, если точнее, Java 2 SDK, стандартный выпуск, версия 1.4. Несмотря на новое название многие программисты по-прежнему называют этот инструментальный набор «JDK».

Старайтесь не путать JDK (или SDK) со средой выполнения Java (Java Runtime Environment, JRE). JRE содержит все, что необходимо для запуска Java-программ, но не содержит инструментарий, который необходим для их разработки (в первую очередь, компилятор). Вам также следует помнить о Java-плагине (Java Plug-in), версии JRE, которая была разработана для интеграции в веб-броузеры Netscape Navigator и Microsoft Internet Explorer.

Помимо стандартного выпуска Java, который используется большинством программистов, пишущих на Java, и документация к которому приводится в этом издании, компания Sun также выпустила платформу Java 2, предназначенную для создания корпоративных приложений (J2EE), и платформу Java 2, мини-выпуск (J2ME) для потребительских электронных систем, таких как портативные PDA и сотовые телефоны. См. книги «Java Enterprise in a Nutshell» и «Java Micro Edition in a Nutshell» (обе книги выпущены O'Reilly) для лучшего ознакомления с этими выпусками.

Основные преимущества Java

Почему лучше выбрать именно Java? Стоит ли изучать новый язык и новую платформу? В этом разделе рассматриваются основные преимущества Java.

Напиши один раз и запускай где угодно!

Sun рассматривает возможность «Напиши один раз и запускай где угодно!» как ключевое преимущество Java-платформы. В переводе с жаргона этот девиз означает следующее: самое важное в Java-технологии состоит в том, что написать свое приложение нужно только один раз – для Java-платформы, – а затем его можно запускать где угодно.

«Где угодно» означает любую ОС, поддерживающую Java-платформу. К счастью, поддержка Java становится повсеместной. Она уже интегрирована или интегрируется практически во все основные ОС. Поддержка встроена в популярные веб-броузеры, что фактически означает ее присутствие на каждом компьютере, имеющем выход в Интернет. Поддержка Java встроена в бытовые электронные устройства, такие как телевизионные приставки, PDA и сотовые телефоны.

Безопасность

Еще одна отличительная особенность Java – средства безопасности. Как язык, так и сама платформа с самого начала разрабатывались с учетом безопасности. Java-платформа позволяет пользователям загружать по сети непроверенный код и запускать его в безопасной среде, в которой он не сможет нанести какой-либо ущерб. Непроверенный код не сможет заразить базовую систему каким-либо вирусом, не сможет считывать или записывать файлы на жесткий диск и т. п. Одни эти возможности делают Java уникальной платформой.

В Java 1.2 модель механизма защиты от несанкционированного доступа была доработана и улучшена. В этой модели уровни защиты и ограничения доступа стали более гибкими, а применять их можно не только в рамках апплетов. В Java 1.2 любой Java-код – апплет, сервлет, JavaBeans-компонент или все Java-приложение – можно запускать с ограниченными правами доступа, что предотвратит возможность нанесения какого-либо ущерба системе.

Вопросы безопасности, связанные с языком и платформой Java, тщательно рассматривались специалистами во всем мире. На ранних этапах существования Java ошибки в системе безопасности – некоторые из них были достаточно серьезные – не раз обнаруживались, но в дальнейшем были исправлены. Java предоставляет надежные гарантии, связанные с безопасностью, и поэтому случаи, когда находится новая ошибка в системе безопасности, становятся сенсацией. Ни одна другая распространенная платформа не предоставляет таких серьезных гарантий безопасности, как Java. Никто не утверждает, что в будущем не будут обнаружены новые прорехи в системе безопасности Java, но даже если безопасность Java не является совершенной, эта платформа на практике показала себя достаточно надежной в повседневном использовании. Безусловно, она лучше всех других альтернатив.

Сетевое программирование

Корпоративным девизом Sun всегда было высказывание «Сеть – это компьютер». Разработчики Java-платформы признавали важность сетевой работы и сделали Java-платформу «сетецентричной». С точки зрения программиста Java облегчает работу с ресурсами в сети и помогает в создании сетевых приложений с использованием клиент-серверных или многоуровневых архитектур.

Динамические, расширяемые программы

Java является как динамической, так и расширяемой средой. Java-код представлен в форме модульных объектно-ориентированных единиц, которые называются *классами*. Классы хранятся в отдельных файлах и загружаются в Java-интерпретатор только при необходимости. Это означает, что приложение может само определить во время своей работы, какие классы ему нужны, и загрузить их тогда, когда это необходимо. Это также означает, что программа может динамично расширять саму себя путем загрузки классов, необходимых для расширения функциональности.

Сетецентричный характер Java-платформы означает, что Java-приложение может динамично расширять себя путем загрузки новых классов из сети. Приложение, которое использует преимущества этой функциональности, перестает быть монолитным блоком кода. Напротив, оно становится интерактивным собранием независимых программных компонентов. Таким образом, Java представляет новый подход в области проектирования и разработки приложений.

Локализация

Как язык, так и сама Java-платформа с самого начала разрабатывались с учетом того, что помимо английского существуют и другие языки. Когда создавалась платформа Java, язык Java фактически являлся единственным распространенным языком программирования, в котором поддержка локализации была заложена с самого начала, а не добавлена задним числом. В то время как большинство языков программирования используют 8-битовые символы, которые представляют алфавиты только английского и западноевропейских языков, Java использует 16-битовые символы Unicode, которые представляют фонетические алфавиты и идеограммы языков всего мира. Функциональность Java в области локализации не ограничена символами. Эта функциональность пронизывает всю Java-платформу, что облегчает написание локализованных программ на Java (в отличие от других сред).

Производительность

Как уже упоминалось выше, Java-программы компилируются не в «родные» машинные команды, а в переносимый промежуточный код, известный как байт-код. Виртуальная машина Java (Java VM) выполняет Java-программу путем интерпретации машинно-независимых команд, представленных в форме байт-кода. Эта архитектура означает, что Java-программы работают быстрее, чем программы или сценарии, написанные на интерпретируемых языках, но они, как правило, работают медленнее, чем программы C и C++, скомпилированные в виде инструкций на машинно-зависимом языке. Следует также иметь в виду, что хотя Java-программы и компилируются в байт-код, не вся Java-платформа реализована посредством интерпретируемых байт-кодов. С целью обеспечения эффективности трудоемкие (с точки зрения вычислений) части Java-платформы – такие как методы обработки строк – реализуются с использованием машинно-зависимого кода.

Ранние выпуски Java содержали в себе ряд проблем, связанных с производительностью, однако с каждым новым выпуском скорость Java VM значительно возрастала. Виртуальная машина прошла серьезную настройку и оптимизацию в самых критичных местах. Более того, большинство современных реализаций включают в себя JIT-компилятор, который переводит байт-коды Java в машинно-зависимые инструкции непосредственно перед выполнением программы.

Используя изощренные JIT-компиляторы, Java-программы могут работать со скоростью, сравнимой со скоростью машинно-зависимых приложений, написанных на C и C++.

Java – это машинно-независимый, интерпретируемый язык; Java-программы работают почти так же быстро, как и машинно-зависимые программы, написанные на C и C++. Ранее производительность являлась причиной, из-за которой некоторые программисты избегали Java. Теперь, принимая во внимание улучшения, внесенные в Java 1.2, 1.3 и 1.4, вопросы, связанные с производительностью, уже не должны никого волновать.

Эффективность программирования и период до выхода продукта на рынок

Наконец, одним из самых важных стимулов для использования Java является тот факт, что программистам по душе этот язык. Java – гибкий язык, в который интегрирован мощный, хорошо разработанный набор программных интерфейсов. Программистам нравится программировать на Java. Их часто поражает то, насколько

быстро можно добиться результатов, применяя этот язык. Java является простым и красивым языком с тщательно разработанными, интуитивно понятными программными интерфейсами. Как следствие, программисты пишут код, в котором содержится меньше ошибок, чем в случае с другими платформами, что, в свою очередь, сокращает время разработки программ.

Пример программы

Пример 1.1 показывает Java-программу, которая подсчитывает факториалы.[1]

Цифры, с которых начинается каждая строка, не являются частью программы; они служат для облегчения построчного рассмотрения программы.

Пример 1.1. Factorial.java: программа для вычисления факториалов

```
1 /**
2   Эта программа подсчитывает факториал числа.
3 */
4 public class Factorial { // Определить класс
5   // Программа начинается здесь
6   public static void main(String[] args) {
7     // Получить входные данные от пользователя
8     int input = Integer.parseInt(args[0]);
9     double result = factorial(input);      // Посчитать факториал
10    System.out.println(result);            // Распечатать результат
11  } // метод main() заканчивается здесь
12
13  // Этот метод подсчитывает x!
14  public static double factorial(int x) {
15    if (x < 0)         // Проверить правильность входных данных
16      return 0.0;      // Если данные неправильные, отобразить 0
17    double fact = 1.0; // Начинать с исходной величины
18    while(x > 1) {     // цикл выполняется, пока x больше 1
19      fact = fact * x; // умножать на x каждый раз
20      x = x - 1;       // а затем уменьшить x
21    }                  // Вернуться к началу цикла
22    return fact;       // возвратить результат
23  }                    // метод factorial() заканчивается здесь
24 }                     // класс заканчивается здесь
```

Компиляция и запуск программы

Прежде чем рассматривать работу программы, следует обсудить, как ее запускать. Для того чтобы скомпилировать и запустить программу, вам потребуется какой-либо набор инструментальных средств разработки ПО (SDK) для Java. Компания Sun Microsystems разработала язык Java. Она поставляет Java SDK для своей ОС Solaris, а также для платформ Linux и Microsoft Windows. На момент написания этой книги последней версией Sun SDK была Java 2 SDK, Standard Edition, версия 1.4 (можно загрузить с сайта *http://java.sun.com/*). Удостоверьтесь в том, что вы загружаете SDK, а не среду выполнения Java (JRE). JRE позволяет запускать существующие Java-программы, но не позволяет писать и компилировать свои собственные.

Sun поддерживает SDK только для платформ Solaris, Linux и Windows. Тем не менее много других компаний лицензировали и перенесли SDK на свои платформы. Свяжитесь с вашими дилерами, чтобы узнать, имеется ли Java SDK для вашей системы.

---

[1] Факториал целого числа – это произведение данного числа и всех положительных целых чисел, которые меньше этого числа. Например: факториалом 4, который обозначается как «4!», будет 24 (4 умножить на 3, затем на 2 и на 1). По определению 0! равен 1.

Sun SDK – это не единственная среда программирования на Java, которую вы можете использовать. Такие компании, как Borland, Inprise, Metrowerks, Oracle, Sybase и Symantec, также предлагают свои коммерческие продукты для написания Java-программ. В этой книге мы предполагаем, что вы используете Sun SDK. Если вы используете продукт какого-либо другого поставщика, непременно прочтите его документацию, чтобы узнать, как компилировать и запускать простую программу, такую как показана в примере 1.1.

Как только вы установили среду программирования на Java, первый шаг, который вам необходимо сделать для запуска программы, – это набрать ее. В своем любимом текстовом редакторе введите программу так, как она отображена в примере 1.1. Вы должны пропускать номера строк, поскольку они приводятся для удобства. Помните, что Java – это язык, чувствительный к регистру символов, и поэтому верхний и нижний регистры должны соблюдаться. Обратите внимание, что многие строки этой программы заканчиваются точкой с запятой. Очень распространена следующая ошибка: об этих символах просто забывают, но программа не может работать без них, поэтому будьте внимательны! Если вы печатаете не очень быстро, то можно опустить все, начиная от // и до конца строки. Это *комментарии*; они приводятся для удобства и игнорируются компилятором.[1] При написании Java-программ следует использовать текстовый редактор, который сохраняет файлы в виде простого текста, а не текстовый процессор, который поддерживает шрифты и форматирование, сохраняя файлы в своем собственном формате. Мой излюбленный текстовый редактор в системах Unix – это *emacs*. Если вы используете Windows, то вы, возможно, выберете Notepad или WordPad, если у вас нет специализированного редактора. Если вы используете коммерческую среду программирования в Java, то в ней, вероятно, уже есть подходящий текстовый редактор; прочтите документацию, которая прилагается к продукту. Когда вы завершили ввод программы, сохраните ее в файле с именем `Factorial.java`. Это важно: программа не будет работать, если ее сохранить под каким-то другим именем.

После написания подобной программы следующим шагом является ее компиляция. В состав Sun SDK входит Java-компилятор, который известен как *javac*. Это инструмент командной строки, поэтому вы можете вызывать его только из терминального окна, например из окна MS-DOS в Windows или окна *xterm* в Unix. Скомпилируйте программу, набрав следующую командную строку:[2]

```
C:\>javac Factorial.java
```

Если такая команда выдает какие-либо ошибки, то вы, вероятно, что-то не так набрали в самой программе. Если не выдается никаких сообщений об ошибке, то компиляция удалась – *javac* создает файл, который называется `Factorial.class`. Это скомпилированная версия программы.

После компиляции Java-программы ее нужно запустить. В отличие от программ, написанных на других языках, Java-программы не компилируются в код на машинно-зависимом языке, поэтому система не может выполнять их непосредственно. Вместо этого их запускает другая программа, известная как Java-интерпретатор. В Sun SDK ин-

---

[1] Я рекомендую вам набрать этот пример вручную, с тем чтобы «прочувствовать» язык. Однако если вы *действительно* не хотите набирать вручную, вы можете загрузить этот пример, а также все примеры из книги с сайта *http:// www.oreilly.com/catalog/javanut4/*.

[2] Символы «`C:\>`» представляют собой подсказку командной строки; не печатайте эти символы сами.

терпретатор – это программа командной строки, которая по вполне понятным причинам называется *java*. Чтобы запустить программу вычисления факториала, наберите:

```
C:\>java Factorial 4
```

java – это команда, которая запускает Java-интерпретатор, а *Factorial* – это название Java-программы, которую мы хотим запустить через интерпретатор; `4` – это входные данные, а именно число, факториал которого мы хотим узнать при помощи интерпретатора. Программа выводит одну строку результата, которая показывает, что факториал 4 равен 24.

```
C:\> java Factorial 4
24.0
```

Поздравляем! Вы только что написали, скомпилировали и запустили вашу первую Java-программу. Попробуйте запустить ее снова, чтобы подсчитать факториалы других чисел.

Анализ программы

Теперь, когда вы запустили программу вычисления факториала, давайте проанализируем ее построчно и выясним, что именно приводит в действие Java-программы.

Комментарии

Первые три строки программы – это комментарии. Java их проигнорирует, но программисту они дадут информацию о том, что делает программа. Комментарий начинается символами `/*` и заканчивается символами `*/`. Любое количество текста, включая многострочный текст, может находиться между этими символами. Java также поддерживает другой тип комментариев, которые можно видеть в строках с 4 по 24. Если в Java-программе появятся символы `//`, то Java проигнорирует эти символы, равно как и другой текст, который появится между этими символами и концом строки.

Как определить класс

Строка 4 – это начало программы. В ней указывается на то, что мы определяем класс, который называется `Factorial`. Это объясняет, почему программа должна храниться в файле с названием `Factorial.java`. Это имя файла означает, что файл содержит исходный код Java для класса, названного `Factorial`. Слово `public` является *модификатором* (*modifier*); оно говорит о том, что класс доступен для всех и его может использовать кто угодно. Открывающая фигурная скобка (`{`) отмечает начало самого класса, который продолжается до строчки 24, где мы находим закрывающую фигурную скобку (`}`). Программа содержит ряд пар фигурных скобок; отступы строк показывают вложенность внутри этих фигурных скобок.

Класс – это основная единица структуры программы, написанной на Java, поэтому не удивительно, что первая строка нашей программы объявляет класс. Все Java-программы представляют собой классы, хотя некоторые программы используют много классов вместо одного. Java – это объектно-ориентированный язык программирования, а классы являются основной частью объектно-ориентированной парадигмы. Каждый класс определяет уникальный вид объекта. Пример 1.1 на самом деле не является объектно-ориентированной программой, поэтому я не буду вдаваться в подробности по поводу классов и объектов. Это, в сущности, является темой главы 3. Пока нужно лишь понять, что класс определяет набор взаимодействующих *членов* (*members*). Эти члены могут быть полями, методами или другими классами. `Factorial` как класс содержит два члена, оба являются методами. Они будут описаны в последующих разделах.

Как определить метод

Строка 6 начинает определение *метода* нашего класса `Factorial`. Метод – это именованный фрагмент Java-кода. Java-программа может вызвать метод для выполнения содержащегося в нем кода. Если вы программировали на других языках, то вам уже, вероятно, попадались методы – они могли называться функциями, процедурами или подпрограммами. Интересно, что методы имеют параметры и возвращаемые значения. При вызове метода вы передаете ему какие-либо данные, с которыми он должен работать, а он возвращает вам результат. Метод напоминает алгебраическую функцию:

```
y = f(x)
```

В данном случае математическая функция `f` выполняет вычисление с величиной, представленной `x`, и возвращает величину, которая представлена `y`.

Вернемся к строке 6. Ключевые слова `public` и `static` являются модификаторами. `public` означает, что метод доступен всем; его может использовать кто угодно. Значение модификатора `static` в данном случае не важно; оно объясняется в главе 3. Ключевое слово `void` описывает возвращаемое значение метода. В данном случае оно обозначает то, что этот метод не имеет возвращаемого значения.

Слово `main` является названием метода. `main` – это специальное имя. Когда вы запускаете Java-интерпретатор, он читает указанный класс, а затем ищет метод с именем `main()`.[1] Когда интерпретатор находит этот метод, он начинает запуск программы с этого метода. Когда метод `main()` заканчивается, программа завершена, а Java-интерпретатор прекращает работу. Другими словами, метод `main()` – это главная точка входа в Java-программу. Однако для метода недостаточно иметь название `main()`. Метод следует объявить как `public static void` (точно так, как показано в строке 6). Фактически в строке 6 можно изменить лишь слово `args`, которое можно заменить любым другим словом на ваше усмотрение. Вы будете использовать эту строку во всех своих Java-программах, так что ее следует заучить наизусть прямо сейчас![2]

Вслед за названием метода `main()` следует список параметров метода, заключенных в круглые скобки. Данный метод `main()` имеет только один параметр. `String[]` указывает на тип параметра, который является массивом строк (то есть пронумерованным списком строк текста). `args` указывает на название параметра. В алгебраическом выражении f(x) символ x – это просто способ указать на неизвестную величину. `args` служит той же цели, только для метода `main()`. Как мы увидим в дальнейшем, название `args` применяется в самом методе для ссылки на неизвестную величину, которая передается методу.

Как я уже упоминал, метод `main()` – это особый метод, вызываемый Java-интерпретатором, когда он начинает выполнять Java-класс (программу). Когда вы вызываете Java-интерпретатор следующим образом:

---

[1] В соответствии с принятыми условными обозначениями, когда в этой книге упоминается метод, он сопровождается парой круглых скобок. Как вы увидите впоследствии, круглые скобки – это важная часть синтаксиса метода, они служат для отделения имен методов от имен классов, полей, переменных и т. п.

[2] Все Java-программы, которые запускаются непосредственно Java-интерпретатором, должны содержать метод `main()`. Программы такого типа обычно называются приложениями. Можно писать программы, которые не запускаются напрямую интерпретатором, а динамически загружаются в какую-либо уже работающую Java-программу. Примерами могут служить апплеты – программы, которые запускаются веб-броузером, и сервлеты – программы, которые запускаются веб-сервером. Апплеты рассматриваются в книге «Java Foundation Classes in a Nutshell» (O'Reilly), а сервлеты – в «Java Enterprise in a Nutshell» (O'Reilly). В этой книге рассматриваются только приложения.

```
C:\>java Factorial 4
```

строка «4» передается методу `main()` в качестве значения параметра с названием `args`. Более точно выражаясь, в `main()` передается массив строк, содержащий только одну позицию «4». Когда вы вызываете программу следующим образом:

```
C:\>java Factorial 4 3 2 1
```

то массив из четырех строк – «4», «3», «2» и «1» – передается методу `main()` в качестве значения параметра с названием `args`. Наша программа смотрит только на первую строку в массиве, остальные строки игнорируются.

И последнее: в строке 6 есть открытая фигурная скобка. Этим отмечается начало самого метода `main()`, который продолжается вплоть до соответствующей закрывающей фигурной скобки в строке 11. Методы состоят из операторов (statements), которые Java-интерпретатор выполняет последовательно. В данном случае строки 8, 9, 10 – это три оператора, которые составляют тело метода `main()`. Каждый оператор заканчивается точкой с запятой, отделяющей его от следующего. Это важная часть синтаксиса Java, но начинающие программисты часто забывают о точке с запятой.

Объявление переменной и анализ входных данных

Первая инструкция метода `main()`, строка 8, объявляет переменную и присваивает ей значение. В любом языке программирования *переменная* (*variable*) просто является символическим именем некоторой величины. Мы уже видели в этой программе, что имя `args` ссылается на значение параметра, передаваемого методу `main()`. Параметры метода являются одной разновидностью переменных. Кроме того, методы могут объявлять дополнительные «локальные» переменные. В методах локальные переменные можно применять для сохранения промежуточных значений, которые используются при вычислениях.

Именно этим мы занимаемся в строке 8. Она начинается со слов `int input`, объявляющих переменную с именем `input` и указывающих на то, что тип этой переменной – `int`, то есть это целое число. Java может работать с переменными различных типов, включая целые числа, действительные числа или числа с плавающей точкой, символы (например, цифры и буквы) и строки текста. Java – это строго типизированный язык, что означает следующее: все переменные должны иметь конкретный тип и могут ссылаться только на значения этого типа. Наша переменная `input` всегда ссылается на целое число; она не может ссылаться на число с плавающей точкой или на строку. Есть типы и у параметров метода. Вспомним, что параметр `args` имел тип `String[]`.

Вернемся к строке 8: объявление переменной `int input` сопровождается символом «=». Это выражение присваивания в Java; оно устанавливает значение переменной. Когда вы начнете работу с Java-кодом, не следует читать «=» как «равно»; воспринимайте этот знак как «присвоена такая-то величина». Как будет показано в главе 2, для «равно» существует другой оператор.

Значение, присваиваемое нашей переменной `input`, – `Integer.parseInt(args[0])`. Это вызов метода. Первый оператор метода `main()` вызывает другой метод с названием `Integer.parseInt()`. Как вы уже, наверное, догадались, этот метод анализирует целое число, то есть переводит строчное представление целого числа, например «4», в целое число. Метод `Integer.parseInt()` не является частью языка Java, но он входит в ядро программного интерфейса Java. Каждая Java-программа может задействовать мощный набор классов и методов, содержащихся в этом ключевом программном интерфейсе. Вторая часть данной книги содержит справочник по базовым функциям прикладного программного интерфейса Java.

Когда вы вызываете метод, вы передаете значения (*аргументы*), присваиваемые соответствующим параметрам, которые определяются методом, а метод вовращает некоторое значение. Аргумент, переданный `Integer.parseInt()`, – это `args[0]`. Вспомним, что `args` – это название параметра для `main()`; оно указывает на массив (или список) строк. Элементы массива последовательно пронумерованы, первый всегда имеет номер 0. Нам интересна только первая строка в массиве `args`, поэтому мы применяем выражение `args[0]` для ссылки на эту строку. Таким образом, когда мы вызываем программу так, как описано ранее, строка 8 берет первую строку, указанную после названия класса («4»), и передает ее методу с именем `Integer.parseInt()`. Этот метод преобразует строку в соответствующее целое число и отдает его в качестве возвращаемого значения. Наконец, возвращенное целое число присваивается переменной с именем `input`.

Вычисление результата

Содержимое строки 9 очень похоже на то, что находится в строке 8. Объявляется переменная, и ей присваивается значение. Значение, присвоенное переменной, подсчитывается путем вызова метода. Переменная называется `result`, а ее тип – `double`. `double` означает число с плавающей запятой двойной точности. Значение, присвоенное переменной, вычисляется методом `factorial()`. Тем не менее метод `factorial()` не является частью стандартного программного интерфейса Java. Напротив, он определен как часть нашей программы (строками 14–23). Аргумент, переданный `factorial()`, является значением, на которое ссылается переменная `input`, подсчитанная в строке 8. Вскоре мы рассмотрим сам метод `factorial()`, но вы, вероятно, уже можете предположить по его названию, что он берет входное значение, подсчитывает факториал этой величины и возвращает результат.

Отображение результата

Строка 10 просто вызывает метод с названием `System.out.println()`. Этот часто применяемый метод является частью ядра Java API; он дает команду Java-интерпретатору на отображение значения. В данном случае значение, которое он выводит, представляет собой значение, на которое ссылается переменная с именем `result`. Это результат нашего вычисления факториала. Если переменная `input` имеет значение 4, то переменная `result` хранит значение 24, а данная строка выводит это значение.

Метод `System.out.println()` не возвращает значения, поэтому в этом операторе отсутствует объявление переменной и выражение присваивания (=), так как отсутствует какое-либо значение, которое можно чему-либо присвоить. Эту ситуацию можно описать и по-другому: как и метод `main()` в строке 6, `System.out.println()` объявлен как `void`.

Конец метода

Строка 11 содержит только один символ }. Этот символ отмечает конец метода. Когда Java-интерпретатор доберется до этого места, он уже закончит выполнение метода `main()`, прекратив работу. Конец метода `main()` также является концом области определения переменных `input` и `result`, которые были объявлены в `main()`, и параметра `args` метода `main()`. Эти имена переменных и параметра имеют смысл только в методе `main()` и не могут быть использованы где-либо еще внутри программы, если только другие части программы не объявят другие переменные или параметры, которые по совпадению будут иметь те же имена.

Пустые строки

Строка 12 – это пустая строка. Вы можете вставлять пустые строки, пробелы и символы табуляции в любой части программы. С их помощью программу можно сделать удобной для чтения. Пустая строка появляется здесь для того, чтобы отделить метод `main()` от метода `factorial()`, который начинается в строке 13. Вы также заметите, что программа использует пробелы и табуляции для отступов в различных строках кода. Эти отступы необязательны, но они подчеркивают структуру программы и в значительной степени улучшают удобство чтения кода.

Еще один метод

Строка 13 начинается с определения метода `factorial()`, который применялся в методе `main()`. Сравните эту строку со строкой 6, и вы заметите сходства и различия. Метод `factorial()` имеет те же самые модификаторы `public` и `static`. Он принимает один параметр целого типа, который мы назвали `x`. В отличие от метода `main()`, который не возвращает значения (void), `factorial()` возвращает значение типа `double`. Открытая фигурная скобка отмечает начало тела метода, которое продолжается через вложенные фигурные скобки в строках 18 и 21 до строки 23, где находится соответствующая закрывающая фигурная скобка. Тело метода `factorial()` как и тело метода `main()` состоит из операторов, которые находятся в строчках 15–22.

Проверка достоверности входных данных

В методе `main()` мы видели объявление переменных, присваивание и вызов методов. Оператор в строке 12 – другой. Это оператор if, который при определенных условиях выполняет другой оператор. Ранее мы заметили, что Java-интерпретатор один за другим выполняет три оператора метода `main()`. Он всегда выполняет их именно таким образом, именно в таком порядке. Оператор `if` – это оператор управления потоком; он может влиять на то, как интерпретатор выполняет программу.

За ключевым словом `if` следует выражение, содержащее скобки и оператор. Сначала Java-интерпретатор вычисляет выражение. Если оно верно, интерпретатор выполняет оператор. Если выражение не верно, интерпретатор пропускает оператор и переходит к следующему. Условие для оператора `if` в строке 15 – `x<0`. Оно проверяет, является ли величина, переданная методу `factorial()`, меньшей, чем ноль. Если да, выражение является верным (`true`); выполняется оператор в строке 16. Строка 15 не заканчивается точкой с запятой, потому что оператор в строке 16 является частью оператора `if`. Точка с запятой требуется только в конце оператора.

Строка 16 – это оператор `return`. Она говорит о том, что возвращаемое значение метода `factorial()` равно 0.0. `return` также является оператором управления потоком. Когда Java-интерпретатор видит `return`, он прекращает выполнять текущий метод и немедленно возвращает указанное значение. Оператор `return` можно употреблять как отдельный оператор, но в данном случае он является частью оператора `if` в строке 15. Отступ в строке 16 помогает обратить внимание на этот факт. (Java не будет принимать во внимание этот отступ, но он очень пригодится людям, которые будут читать Java-код!) Строка 16 выполняется только в том случае, если выражение в строке 15 верно.

Прежде чем продолжить, нам следует немного вернуться назад и разобраться, для чего необходимы строки 15 и 16. Подсчет факториала для отрицательного числа является ошибкой, поэтому данные строки гарантируют, что вводимое значение `x` является допустимым. Если оно недопустимо, эти строки заставляют `factorial()` возвращать заведомо невозможный результат, то есть 0.0.

Важная переменная

Строка 17 является еще одним объявлением переменной; в этой строке объявляется переменная с названием `fact` типа `double` и ей присваивается начальное значение 1.0. Эта переменная хранит значение факториала в то время, как мы подсчитываем его в последующих инструкциях. В Java переменные могут быть объявлены где угодно; они не ограничиваются началом метода или блоком кода.

Организация цикла расчета факториала

Строка 18 вводит еще один тип оператора – цикл `while`. Как и оператор `if`, оператор `while` состоит из выражения в скобках и оператора. Когда Java-интерпретатор видит оператор `while`, он вычисляет связанное с ним выражение. Если оно верно, интерпретатор выполняет оператор. Интерпретатор повторяет этот процесс, вычисляя значение выражения и выполняя оператор в том случае, если выражение верно, до тех пор, пока значение выражения не станет неверным. В строке 18 стоит выражение `x>1`, поэтому оператор `while` повторяет цикл, пока параметр `x` содержит значение, которое больше 1. Это можно выразить и другими словами: цикл повторяется до тех пор, пока переменная `x` не станет меньше или равна 1. Судя по этому выражению, можно предположить, что, если цикл когда-нибудь должен прекратиться, то величина `x` будет каким-то образом модифицироваться при помощи оператора, исполняемого циклом.

Главное отличие между оператором `if` в строках 15–16 и циклом `while` в строках 18–21 состоит в том, что оператор, связанный с циклом `while`, является *составным оператором*. Составной оператор – это ноль или более операторов, заключенных между двумя фигурными скобками. За ключевым словом `while` в строке 18 следует выражение в круглых скобках, а затем открывающая фигурная скобка. Это означает, что тело цикла состоит из всех операторов, находящихся между этой открывающей скобкой и закрывающей в строке 21. Ранее в этом разделе я уже упоминал, что все операторы Java заканчиваются точкой с запятой. Тем не менее это правило не относится к составным операторам, как вы могли заметить по отсутствию точки с запятой в конце строки 21. Конечно же, операторы внутри составного оператора (строки 19 и 20) заканчиваются точкой с запятой.

Тело цикла `while` состоит из операторов в строках 19 и 20. Строка 19 умножает величину `fact` на величину `x` и заносит результат обратно в `fact`. Строка 20 выполняет аналогичные действия. Она вычитает 1 из величины `x` и сохраняет результат в `x`. Важным является символ * в строке 19. Это *оператор* умножения. Как вы догадались, «-» в строке 17 является оператором вычитания. Оператор – это ключевая часть синтаксиса в Java: он выполняет вычисления с одним или двумя *операндами* для того, чтобы получить новое значение. Операторы и операнды комбинируются для формирования *выражений*, таких как `fact*x` или `x-1`. В программе мы видели и другие операторы. Строка 18, например, использует оператор «больше чем» (>) в выражении `x>1`, которое сравнивает величину переменной `x` с 1. Значением этого выражения является булева величина, которая может принимать значения `true` или `false` в зависимости от результата сравнения.

Чтобы понять работу цикла `while`, полезно подумать так, как думает Java-интерпретатор. Предположим, что мы подсчитываем факториал 4. До начала цикла переменная `fact` равна 1.0, а `x` равна 4. После того как тело цикла было выполнено один раз – после первой итерации, – `fact` будет равна 4.0, а `x` равна 3. После второй итерации `fact` равна 12.0, а `x` равна 2. После третьей итерации переменная `fact` равна 24.0, а `x` равна 1. Когда интерпретатор проверит условие цикла после третьей итерации, он обнаружит, что выражение `x>1` больше не является верным, а потому прекратит повторение цикла; выполнение программы будет продолжено со строки 22.

Возвращение результата

Строка 22 – это еще один оператор `return`, такой же, как в строке 16. Этот оператор не будет возвращать постоянное значение типа 0.0, а возвратит значение переменной `fact`. Если значение `x`, переданное в метод `factorial()`, равно 4, то, как мы заметили ранее, значение `fact` равно 24.0, так что это и есть возвращаемое значение. Вспомните, что метод `factorial()` был вызван в строке 9 программы. Когда выполняется оператор `return`, управление возвращается в строку 9, где возвращаемое значение присваивается переменной с названием `result`.

Исключения

Если вы проследили за построчным анализом примера 1.1, то вы уже прошли значительную дистанцию на пути к пониманию основ языка Java.[1] Это простая и в то же время нетривиальная программа, которая иллюстрирует многие возможности Java. Есть еще одна важная особенность программирования на Java, на которой я хотел бы остановиться. Эта особенность не видна в самом листинге программы. Вспомним следующее: программа подсчитывает факториал числа, вводимого в командной строке. Что произойдет, если запустить программу без указания числа?

```
C:\>java Factorial
java.lang.ArrayIndexOutOfBoundsException: 0
        at Factorial.main(Factorial.java:8)
C:\>
```

А если указать величину, которая не является числом?

```
C:\>java Factorial ten
java.lang.NumberFormatException: ten
        at java.lang.Integer.parseInt(Integer.java)
        at java.lang.Integer.parseInt(Integer.java)
        at Factorial.main(Factorial.java:8)
C:\>
```

В обоих случаях происходит ошибка, то есть, выражаясь на языке Java, генерируется *исключение* (*exception*). Java-интерпретатор выводит сообщение, в котором объясняется тип исключения и указывается, где оно имело место (оба вышеупомянутых исключения произошли в строке 8). В первом случае исключение генерируется из-за отсутствия строк в списке `args`, то есть у `args[0]` запрошена несуществующая строка. Во втором случае `Integer.parseInt()` не может преобразовать строку «ten» в число. Более подробно исключения рассмотрены в главе 2, из которой мы узнаем, как их правильно обрабатывать.

---

[1] Не стоит волноваться, если вы не поняли всех деталей программы подсчета факториала. Мы подробно рассмотрим язык Java в главе 2 и главе 3. Тем не менее, если вам кажется, что вы не поняли ни одной строчки из этого анализа, то, вероятно, последующие главы могут оказаться для вас затруднительными. В таком случае вам, возможно, следует обратиться к другим источникам в целях изучения языка Java, а затем вернуться к этой книге для закрепления ваших знаний, и, конечно, использования данного материала в качестве справочного пособия. Вот ссылка на сайт, который может оказаться полезным при изучении языка Java (это учебное пособие от Sun, доступное в режиме online): *http://java.sun.com/docs/books/tutorial/*.

Глава 2

Синтаксис Java: от подножия к вершине

Данная глава представляет собой краткое, но содержательное введение в синтаксис Java. В первую очередь она предназначена для читателей, не знакомых с языком, но имеющих небольшой опыт программирования. Полные решимости новички также могут почерпнуть здесь много полезного. Если же вы уже знакомы с Java, эта глава станет удобным справочным пособием по языку. В предыдущих изданиях книги данная глава предназначалась для программистов, переходящих от C и C++ к Java. Для данного издания она была переписана и стала полезной более широкому кругу читателей, хотя все еще содержит сравнения с языками C и C++ для удобства разработчиков с опытом программирования на этих языках.[1]

В этой главе описывается синтаксис Java. Изложение начинается с самого простого уровня, а рассматриваемые структуры постепенно усложняются. Глава охватывает:

- Символы, используемые при написании программ на Java, а также кодировку этих символов.
- Типы данных, литеральные значения (literal values), идентификаторы и другие лексемы, составляющие Java-программу.
- Операторы для группирования отдельных лексем в большие выражения.
- Операторы, образующие выражения и другие операторы для формирования логических фрагментов Java-кода.
- Методы (они же функции, процедуры или подпрограммы), являющиеся именованными наборами Java-операторов, которые можно вызвать из другого Java-кода.
- Классы, являющиеся коллекциями методов и полей. Будучи центральным элементом Java-программы, классы составляют основу объектно-ориентированного программирования. Глава 3 полностью посвящена обсуждению классов и объектов.
- Пакеты, которые являются наборами соответствующих классов.

---

[1] Читатели, которым недостаточно предоставленного материала по языку Java, могут прочитать книгу «The Java Programming, Second Edition». Авторы – Кен Арнольд (Ken Arnold) и Джеймс Гослинг (James Gosling), создатель Java. Книга выпущена издательством Addison Wesley Longman. А читатели, желающие основательно разобраться в этой теме, могут обратиться к первоисточнику: «The Java Language Specification», Джеймс Гослинг, Билл Джой (Bill Joy) и Гай Стил (Guy Steele) (Addison Wesley Longman). Данная спецификация доступна в бумажном и электронном вариантах; вы можете загрузить ее с веб-сайта компании Sun: *http://java.sun.com/docs/books/jls/*. При написании главы мне пригодились оба документа.

- Java-программы, содержащие один или несколько взаимодействующих классов, которые можно извлечь из одного или нескольких пакетов.

Синтаксис большинства языков программирования сложен, и Java – не исключение. Вообще говоря, невозможно описать все элементы языка, не задействуя еще не рассмотренные элементы. Например, довольно сложно объяснить значение операторов в Java, не опираясь на объекты. Но в то же время невозможно полностью описать объекты, не опираясь на операторы. Следовательно, процесс изучения Java, так же как и любого другого языка, итеративен. Если вы еще не знакомы с Java (либо с языком программирования в стиле Java), то вам стоит проработать эту и следующую главы *дважды*, чтобы усвоить взаимосвязанные понятия.

Символьный набор Unicode

При написании Java-программ применяется символьный набор Unicode. В отличие от 7-битного кода ASCII, пригодного только для английского языка, и 8-битного кода ISO Latin-1, подходящего лишь для основных западно-европейских языков, 16-битный код Unicode может представлять фактически любой письменный язык, используемый на планете. Однако Unicode поддерживается немногими текстовыми редакторами; в действительности большинство Java-программ написаны на простом ASCII. Символы 16-битного Unicode обычно записывают в файл, используя кодирование, известное как UTF-8, которое переводит 16-битные символы в поток байтов. Формат разработан таким образом, что простой текст ASCII и Latin-1 представляет собой допустимый поток байтов UTF-8. Таким образом, вы просто пишете программы на простом ASCII, а затем они работают как Unicode.

Если вы хотите вставить символ Unicode в программу Java, написанную на простом ASCII, используйте специальную управляющую последовательность `\uxxxx`, то есть последовательность, состоящую из обратной косой черты, буквы «u» нижнего регистра и четырех шестнадцатеричных символов. Например, `\u0020` означает символ пробела, а `\u03c0` означает символ π. Вы можете использовать символы Unicode в любой части Java-программы, включая комментарии и имена переменных.

Комментарии

Java поддерживает комментарии трех видов. Первый вид – однострочный комментарий, который начинается символами // и продолжается до конца текущей строки. Например:

```
int i = 0;          // Инициализация переменной цикла
```

Второй вид – многострочный комментарий. Он начинается символами /* и занимает несколько строк до символов */. Java-компилятор игнорирует любой текст между /* и */. Хотя этот стиль обычно используют для многострочных комментариев, его также можно применять и для однострочных. Комментарии этого вида не подлежат вложению (то есть комментарий /* */ нельзя размещать внутри другого комментария). При написании многострочных комментариев программисты часто применяют дополнительные символы *, чтобы таким образом выделить комментарий. Ниже представлен типичный многострочный комментарий:

```
/*
 *   Шаг 4: Вывести статические методы - как открытые, так
 *   и защищенные, но не перечислять устаревшие.
 */
```

Третий вид комментария является особой разновидностью второго вида. Если комментарий начинается с `/**`, он рассматривается как особый тип комментария – *документирующий комментарий* (*doc comment*). Как и обычные многострочные комментарии, документирующие комментарии завершаются символами `*/` и не подлежат вложению. При создании Java-класса, который будет применяться другими программистами, используйте документирующие комментарии для включения документации по классу и его методам в исходный код. Программа под названием *javadoc* извлекает комментарии и обрабатывает их с целью создания документации по вашему классу в формате HTML. Документирующий комментарий может содержать HTML-теги и использовать дополнительный синтаксис, понимаемый *javadoc*. Например:

```
/**
 * Вывести на экран список классов, по нескольку на строку.
 *
 * @param classes Классы, подлежащие выводу на экран
 * @return <tt>true</tt> в случае успеха
 * <tt>false</tt> в случае неудачного исхода.
 * @author Дэвид Флэнаган
 */
```

Более подробно о синтаксисе документирующего комментария и о программе *javadoc* рассказано в главе 7.

Идентификаторы и зарезервированные слова

Идентификатором (*identifier*) является любое символическое имя какого-либо элемента в Java-программе. Идентификатором может выступать имя класса, метода, параметра и переменной. Идентификатор должен начинаться с буквы, символа подчеркивания (_) или с валютного символа Unicode (например, $, £, ¥). За начальным символом может следовать любое количество букв, цифр, символов подчеркивания или валютных символов. Следует помнить, что Java использует множество символов Unicode, куда входит довольно много букв и цифр, не входящих во множество символов ASCII. Ниже представлены разрешенные идентификаторы:

```
i
engine3
theCurrentTime
the_current_time
θ
```

Идентификаторы могут содержать цифры, но не могут начинаться с цифры. Более того, идентификаторы не могут содержать знаков пунктуации, за исключением подчеркивания и валютных знаков. Обычно знак доллара и другие валютные знаки остаются в резерве для автоматического генерирования кода компилятором или другим препроцессором. Лучше избегать этих символов в создаваемых идентификаторах.

Еще одно существенное ограничение: в качестве идентификаторов нельзя применять ключевые слова и литералы, являющиеся частью самого языка Java. Зарезервированные слова перечислены в табл. 2.1.

Обратите внимание, что `const` и `goto` являются зарезервированными словами, но не представляют собой часть языка Java. `assert` – это зарезервированное слово начиная с версии Java 1.4.

Таблица 2.1. Зарезервированные слова в Java

Abstract	default	If	package	synchronized
Assert	do	Implements	private	this
Boolean	double	Import	protected	throw
Break	else	Instanceof	public	throws
Byte	extends	int	return	transient
Case	false	interface	Short	true
Catch	final	long	static	try
Char	finally	native	strictfp	void
Class	float	new	Super	volatile
Const	for	null	switch	while
Continue	goto			

Примитивные типы данных

Java поддерживает восемь базовых типов данных, известных как *примитивные типы*. Кроме того, Java поддерживает классы и массивы как составные, или ссылочные, типы данных. Классы и массивы описаны ниже в этой главе. Существуют такие примитивные типы: булев тип, символьный тип, четыре целых типа и два типа с плавающей точкой. Четыре целых типа и два типа с плавающей точкой различаются по количеству бит, которые их образуют, и, следовательно, по диапазону чисел, которые они представляют. Примитивные типы данных собраны в табл. 2.2.

Таблица 2.2. Примитивные типы данных в Java

Тип	Содержит	По умолчанию	Размер	Диапазон
boolean	true или false	false	1 бит	не доступен
char	символ Unicode	\u0000	16 бит	от \u0000 до \uFFFF
byte	целое число со знаком	0	8 бит	от –128 до 127
short	целое число со знаком	0	16 бит	от –32768 до 32767
int	целое число со знаком	0	32 бита	от –2147483648 до 2147483647
long	целое число со знаком	0	64 бита	от –9223372036854775808 до 9223372036 854775807
float	IEEE 754, с плавающей точкой	0.0	32 бита	от 1.4E–45 до 3.4028235 E+38
double	IEEE 754, с плавающей точкой	0.0	64 бита	от 4.9E–324 до 1.7976931348623157 E+308

Тип boolean

Тип `boolean` представляет значения истинности. Существует только два возможных значения для данного типа, представляющих два булевых состояния: включено или выключено, да или нет, истина или ложь. Java резервирует слова `true` и `false` для представления этих булевых значений.

Программисты, пришедшие из мира языков C и C++, должны помнить, что в Java существуют довольно строгие ограничения по отношению к типу `boolean`: значения типа `boolean` нельзя преобразовать ни в какой другой тип данных, и наоборот. В частности, `boolean` не является интегральным типом, а целые значения нельзя применять вместо булевых. Другими словами, в Java нельзя использовать такие сокращения:

```
if (o) {
  while(i) {
  }
}
```

Наоборот, в Java нужно писать более чистый код, явно обозначая необходимые сравнения:

```
if (o != null) {
  while(i != 0) {
  }
}
```

Тип char

Тип `char` представляет символы Unicode. Многие опытные программисты удивляются, узнав, что в Java значения типа `char` 16-битные, но на практике этот факт очевиден. Чтобы включить символьную константу в Java-программу, необходимо просто поместить ее между двумя одинарными кавычками (апострофами):

```
char c = 'A';
```

Безусловно, в качестве символьной константы вы можете использовать любой символ Unicode. Можно также задействовать управляющую последовательность Unicode `\u`. Кроме того, Java поддерживает несколько других управляющих последовательностей, которые облегчают представление распространенных неотображаемых символов ASCII, таких как символ новой строки, и позволяют вводить некоторые знаки пунктуации, имеющие особое значение в Java. Например:

```
char tab='\t', apostrophe='\'', nul='\000', aleph='\u05D0';
```

В табл. 2.3 перечислены управляющие символы, которые можно применять в константах типа `char`. Эти знаки можно также использовать в строковых литералах, которые более подробно описаны далее в этой главе.

Значения типа `char` можно преобразовывать в различные целые типы, и наоборот. Однако, в отличие от значений типа `byte`, `short`, `int` и `long`, величина типа `char` не имеет знака. Класс `Character` определяет несколько удобных статических методов для работы с символами, в том числе `isDigit()`, `isJavaLetter()`, `isLowerCase()` и `toUpperCase()`.

Таблица 2.3. Управляющие символы в Java

Управляющая последовательность	Значение последовательности
\b	Возврат на одну позицию
\t	Горизонтальная табуляция
\n	Новая строка
\f	Подача страницы
\r	Возврат каретки
\"	Двойная кавычка
\'	Одинарная кавычка
\\	Обратная косая черта
xxx	Символ Latin-1 с кодом *xxx*, где *xxx* является восьмеричным числом (основание 8) между 000 и 377. Формы *x* и *xx* разрешаются (например, '\0'), но не рекомендуется, так как могут возникнуть сложности в строковых константах, где за управляющей последовательностью следует обычная цифра.
\u*xxxx*	Символ Unicode с кодом *xxxx*, где *xxxx* выражено четырьмя шестнадцатеричными цифрами. Переключения Unicode (Unicode escapes) можно использовать во всей Java-программе, а не только в символьных константах и строковых литералах.

Целые типы

Целыми типами в Java являются `byte`, `short` и `long`. Как показано в табл. 2.2, эти типы различаются только по количеству битов и, следовательно, по диапазону представляемых чисел. Все целые типы представляют числа со знаком; в Java отсутствует ключевое слово `unsigned`, имеющееся в C и C++.

Как и следовало ожидать, литералы для каждого из перечисленных типов выражены в виде строки десятичных цифр, которой может предшествовать необязательный знак минус.[1] Ниже перечислены некоторые разрешенные литералы:

```
0
1
123
-42000
```

Целые литералы также можно выразить посредством шестнадцатеричных или восьмеричных чисел. Литерал, который начинается с `0x` или `0X`, считается шестнадцатеричным числом, использующим буквы от `A` до `F` (или от `a` до `f`) в качестве дополни-

[1] С технической точки зрения знак минус является оператором, работающим с литералом, а не частью самого литерала. Кроме того, все целые литералы являются либо 32-битными целыми значениями, либо, если за литералом следует буква L, 64-битными длинными значениями. Для литералов типа `byte` и `short` не существует отдельного синтаксиса, но при необходимости литералы типа `int` можно преобразовать в величины более «коротких» типов. Например, в коде:

```
byte b = 123;
```

где `123` — 32-битный литерал типа `int`, который в операторе присваивания автоматически преобразуется в байт без необходимости использования оператора приведения.

тельных символов, необходимых для записи чисел с основанием 16. Целые литералы, которые начинаются с ведущего символа 0, считаются восьмеричными числами (с основанием 8) и не могут содержать цифры 8 или 9. Java не позволяет выражать целые литералы в двоичной системе счисления (с основанием 2). Разрешенными шестнадцатеричными и восьмеричными литералами являются:

```
0xff        // Десятичное 255, выраженное шестнадцатеричным
0377        // То же число, выраженное восьмеричным (с основанием 8)
0xCAFEBABE // Магическое число для идентификации файлов классов Java
```

Целые литералы являются либо 32-битными значениями типа `int`, либо, когда они заканчиваются буквой `L` или `l`, 64-битными значениями типа `long`.

```
1234      // Целое значение
1234L     // Длинное значение
0xffL     // Другое длинное значение
```

Целочисленная арифметика в Java модульная, то есть она никогда не генерирует переполнение, если вы вышли за диапазон данного целого типа. Вместо этого числа просто циклически переносятся. Например:

```
byte b1 = 127, b2 = 1;          // Максимальное значение байта 127
byte sum = (byte)(b1 + b2);     // Сумма переносится до -128. Это минимальное значение байта
```

Ни компилятор, ни интерпретатор Java никак не предупреждают о происходящих событиях. В целочисленной арифметике важно убедиться, что диапазон используемого типа удовлетворяет поставленным целям. Целочисленное деление на ноль и взятие по модулю ноль запрещено и влечет за собой генерирование `ArithmeticException`.

Для каждого целого типа существует соответствующий класс-обертка: `Byte`, `Short`, `Integer` и `Long`. Каждый из перечисленных классов определяет константы `MIN_VALUE` и `MAX_VALUE`, которые описывают диапазон типа. Классы также определяют такие полезные статические методы, как `Byte.parseByte()` и `Integer.parseInt()`, служащие для преобразования строк в целые значения.

Типы с плавающей точкой

Вещественные числа в Java представлены типами данных `float` и `double`. Как показано в табл. 2.2, `float` является 32-битным значением с плавающей точкой, с обычной точностью, а `double` представляет 64-битное значение с плавающей точкой, с двойной точностью. Оба типа соответствуют стандарту IEEE 754-1985, который определяет формат чисел и арифметические операции, применимые к этим числам.

Значения с плавающей точкой можно непосредственно включать в Java-программу. В такой величине за необязательной последовательностью цифр следует десятичная точка и другая последовательность цифр. Ниже представлены несколько примеров:

```
123.45
0.0
.01
```

Литералы с плавающей точкой можно также представить в экспоненциальной, или научной, нотации, в которой за числом следует буква `e` или `E` (показатель степени) и другое число. Второе число представляет степень десятки, на которую умножается первое число. Например:

```
1.2345E02       // 1.2345 × 10^2, или 123.45
1e-6            // 1 × 10^-6, или 0.000001
```

```
6.02e23          // Число Авогадро: 6.02 × 10^23
```

Литералы с плавающей точкой по умолчанию являются значениями типа `double`. При включении значения типа `float` в программу за числом следует поставить символ `f` или `F`:

```
double d = 6.02E23;
float f = 6.02e23f;
```

Литералы с плавающей точкой нельзя представить в шестнадцатеричной или восьмеричной системе счисления.

Большинство вещественных чисел, по самой их природе, нельзя точно представить каким-либо конечным количеством битов. Таким образом, необходимо помнить, что значения `float` и `double` являются только приближенными значениями представляемых ими чисел. `float` – это 32-битное приближение, которое дает как минимум 6 значимых десятичных разрядов, а `double` – это 64-битное приближение, которое представляет по крайней мере 15 значимых десятичных разрядов. На практике эти числа подходят для большинства вычислений с вещественными числами.

Кроме представления обычных чисел, типы данных `float` и `double` могут представлять четыре специальных значения: положительную и отрицательную бесконечность, нуль и NaN. Значения бесконечности получаются, если результат вычисления с плавающей точкой превышает диапазон `float` или `double`. Нулевое значение представляет собой результат вычисления с плавающей точкой, который меньше нижней границы диапазона `float` или `double`. В типах с плавающей точкой в Java различают положительный и отрицательный нуль в зависимости от места, в котором возникло значение ниже минимального. На практике поведение положительного и отрицательного нуля не сильно различается. И наконец, последним специальным значением является NaN, то есть нечисловое значение. Значение NaN получается при выполнении недопустимой операции с плавающей точкой, например 0.0/0.0. Далее приводятся примеры операторов, приводящих к этим частным значениям:

```
double inf = 1.0/0.0;          // Бесконечность
double neginf = -1.0/0.0;      // Отрицательная бесконечность
double negzero = -1.0/inf;     // Отрицательный нуль
double NaN = 0.0/0.0;          // Не числовое значение
```

Поскольку типы с плавающей точкой в Java могут сводить переполнение к бесконечности, антипереполнение – к нулю, а также имеют особое NaN-значение, то арифметические операции с плавающей точкой никогда не генерируют исключений, даже при выполнении недопустимых операций, например при делении нуля на нуль либо при вычислении корня отрицательного числа.

Примитивные типы `float` и `double` имеют соответствующие классы-обертки, которые называются `Float` и `Double`. Каждый из этих классов определяет следующие полезные константы: `MIN_VALUE`, `MAX_VALUE`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY` и `NaN`.

Бесконечные значения с плавающей точкой ведут себя вполне логично. Например, прибавление к бесконечности или вычитание из нее любого конечного значения дает бесконечность. Поведение отрицательного нуля почти не отличается от положительного нуля; фактически оператор равенства == сообщает о равенстве отрицательного и положительного нуля. Единственный способ отличить отрицательный нуль от положительного или обычного нуля – разделить на него какое-либо число. 1.0/0.0 дает положительную бесконечность, а деление 1.0 на отрицательный нуль дает отрицательную бесконечность. И наконец, поскольку NaN не является числом, оператор == сообщает, что это значение не равно ни одному другому числу, включая само значе-

ние! Чтобы проверить, являются ли значения `float` и `double` нечисловыми (NaN), следует вызвать методы `Float.isNaN()` и `Double.isNaN()`.

Строки

Кроме булевых, символьных, целых типов данных и типов данных с плавающей точкой, в Java существует тип для работы со строками текста (или просто *строками*). Однако тип `String` представляет собой класс и не относится к примитивным типам языка. Тем не менее в Java есть синтаксис для непосредственного включения строковых значений в программу, необходимый по причине широкого употребления строк. Литерал типа `String` состоит из произвольного текста в двойных кавычках.

```
"Hello, world"
"'This' is a string!"
```

Строковые литералы могут содержать любые управляющие последовательности, которые могут представлять литералы типа `char` (см. табл. 2.3). Для включения в литерал типа `String` двойных кавычек используйте последовательность `\"`. Далее в этой главе строки и строковые литералы рассмотрены более подробно. В главе 4 показаны несколько возможных способов работы с объектами `String` в Java.

Преобразования типов

В Java возможны преобразования между целыми значениями и значениями с плавающей точкой. Кроме того, можно преобразовывать значения целых типов и типов с плавающей точкой в значения типа `char` и наоборот, поскольку каждый символ соответствует цифре в кодировке Unicode. Фактически тип `boolean` является единственным примитивным типом в Java, который нельзя преобразовать в другой примитивный тип. Кроме того, любой другой примитивный тип нельзя преобразовать в `boolean`.

Существует два основных типа преобразований. *Расширяющее преобразование* (*widening conversion*) происходит, если значение одного типа преобразовывается в более широкий тип, с большим диапазоном допустимых значений. Java выполняет расширяющие преобразования автоматически, например, если вы присвоили литерал типа `int` переменной типа `double` или литерал типа `char` переменной типа `int`.

Однако *сужающее преобразование* (*narrowing conversion*) – совсем другое дело. Сужающее преобразование происходит, если значение преобразуется в значение типа, диапазон которого не шире изначального. Сужающие преобразования не всегда безопасны: например, преобразование целого значения 13 в `byte` имеет смысл, а преобразование 13000 в `byte` неразумно, поскольку `byte` может хранить только числа от –128 до 127. Поскольку во время сужающего преобразования могут быть потеряны данные, Java-компилятор возражает против любого такого преобразования, даже если преобразуемое значение укладывается в более узкий диапазон указанного типа:

```
int i = 13;
byte b = i;     // Компилятор не разрешит это выражение
```

Единственное исключение из правила – присвоение целого литерала (значения типа `int`) переменной `byte` или `short`, если литерал соответствует диапазону переменной.

Если вам все же необходимо выполнить сужающее преобразование, и вы уверены, что не потеряете и не исказите данные, то можно заставить Java выполнить преобразование посредством языковой конструкции *приведение* (*cast*). Для этого перед преобразуемым значением введите в круглых скобках имя нужного типа. Например:

```
int i = 13;
byte b = (byte) i;       // Принудительное преобразование int в byte
i = (int) 13.456;        // Принудительное преобразование литерала типа double в int 13
```

Приведение примитивных типов чаще всего используют для преобразования значений с плавающей точкой в целые числа. При этом дробная часть значения с плавающей точкой просто отбрасывается (то есть значение с плавающей точкой округляется по направлению к нулю, а не к ближайшему целому числу). Методы `Math.round()`, `Math.floor()` и `Math.ceil()` выполняют другие типы округления.

Поведение величины типа `char` в большинстве случаев совпадает с поведением величины целого типа, следовательно, значение типа `char` можно использовать везде, где требуются значения `int` или `long`. Однако напомним, что тип `char` не имеет знака, поэтому он ведет себя отлично от типа `short`, несмотря на то что диапазон обоих типов равен 16 бит.

```
short s = (short) 0xffff;      // Данные биты представляют число -1
char c = '\uffff';             // Те же биты представляют символ юникода
int i1 = s;                    // Преобразование типа short в int дает -1
int i2 = c;                    // Преобразование char в int дает 65535
```

Таблица 2.4 представляет собой сетку, где для каждого примитивного типа указаны типы, в которые их можно преобразовать, и способ преобразования. Буква N в таблице означает невозможность преобразования. Буква Y означает расширяющее преобразование, которое выполняется автоматически. Буква C означает сужающее преобразование, требующее явного приведения. Наконец, Y* означает автоматическое расширяющее преобразование, в процессе которого значение может потерять некоторые из наименее значимых разрядов. Это может произойти при преобразовании `int` или `long` во `float` или `double`. Типы с плавающей точкой имеют больший диапазон, чем целые типы, поэтому `int` или `long` можно представить посредством `float` или `double`. Однако типы с плавающей точкой являются приближенными числами и не всегда могут содержать так много значащих разрядов, как целые типы.

Таблица 2.4. Преобразование примитивных типов в Java

Преобразовать из:	**Преобразовать в:**							
	boolean	byte	Short	Char	Int	long	float	double
boolean	–	N	N	N	N	N	N	N
byte	N	–	Y	C	Y	Y	Y	Y
short	N	C	–	C	Y	Y	Y	Y
char	N	C	C	–	Y	Y	Y	Y
int	N	C	C	C	–	Y	Y*	Y
long	N	C	C	C	C	–	Y*	Y*
float	N	C	C	C	C	C	–	Y
double	N	C	C	C	C	C	C	–

Ссылочные типы

Кроме восьми примитивных типов, в Java определены две дополнительные категории типов данных: классы и массивы. Java-программы содержат определения классов; каждый класс определяет новый тип данных, который можно использовать в Java-программах. Например, программа может определить класс с именем `Point` и при-

менять его для управления точками X, Y в декартовой системе координат. Таким образом, `Point` становится новым типом данных для этой программы. Массив (array) представляет собой список значений другого типа. `char` является одним типом данных, а массив значений типа `char` – другим типом, который обозначается `char[]`. Массив объектов `Point` – это тип данных `Point[]`. А массив массивов `Point` представляет другой тип данных, который обозначается как `Point[][]`.

Как вы заметили, существует бесконечное множество возможных классов и массивов. Вместе эти типы данных называются *ссылочными типами* (*reference types*). Причину такого названия вы поймете, прочитав данную главу. А пока важно понять, что класс и массив значительно отличаются от примитивных типов, поскольку представляют собой составные, или сложные, типы. Величина примитивного типа содержит только одно значение. Классы и массивы принадлежат к составным типам, содержащим множество значений. Например, величина типа `Point` содержит два значения типа `double`, представляющих координаты X и Y. Безусловно, тип `char[]` является составным типом, поскольку представляет список символов. По своей сути класс и массив намного сложнее, чем примитивные типы данных. Далее в этой главе класс и массив будут рассмотрены более подробно. В главе 3 мы исследуем классы еще скрупулезнее.

Выражения и операторы

Пока что из данной главы мы узнали о примитивных типах, которыми может манипулировать Java-программа. Мы увидели, как в Java-программы включать примитивные значения в виде *литералов*. Мы также использовали *переменные* в качестве символических имен, представляющих, или содержащих, значения. Данные литералы являются лексемами, из которых строятся Java-программы.

Выражение (*expression*) представляет собой следующий, более высокий уровень структуры Java-программы. Java-интерпретатор *вычисляет* (*evaluates*) выражение для определения его значения. Наиболее простые выражения называются *первичными выражениями* (*primary expressions*). Они состоят из литералов и переменных. Все примеры, приведенные ниже, представляют выражения:

```
1.7    // Литерал с плавающей точкой
true   // Булевый литерал
sum    // Переменная
```

Значение, полученное в результате вычисления литерального выражения Java-интерпретатором, само является литералом. В результате вычисления выражения, содержащего переменные, получается значение, хранимое в переменной.

Первичные выражения не представляют большого интереса. Более сложные выражения создаются посредством операторов, соединяющих первичные выражения. Например, следующее выражение использует оператор присваивания для комбинирования двух первичных выражений – переменной и литерала с плавающей точкой – в выражение присваивания:

```
sum = 1.7
```

Но операторы применяют не только в первичных выражениях; их можно использовать для выражений любого уровня сложности. Поэтому все примеры, приведенные ниже, являются разрешенными выражениями:

```
sum = 1 + 2 + 3*1.2 + (4 + 8)/3.0
sum/Math.sqrt(3.0 * 1.234)
(int)(sum + 33)
```

Список операторов

Виды выражений, которые вы можете писать на языке программирования, полностью зависят от множества доступных операторов. В табл. 2.5 приведены доступные в Java операторы. Колонки P и A определяют приоритет и ассоциативность для каждой группы родственных операторов соответственно.

Таблица 2.5. Операторы Java

P	A	Оператор	Тип(ы) операнда(ов)	Выполняемая операция
15	L	`.`	объект, член (member)	доступ к члену объекта
		`[]`	массив, int	доступ к элементу массива
		`( args )`	метод, список аргументов	вызов метода
		`++, --`	переменная	постфиксная форма инкремента и декремента
14	R	`++, --`	переменная	префиксная форма инкремента и декремента
		`+, -`	число	унарный плюс, унарный минус
		`~`	целое число	битовое дополнение
		`!`	булевый	логическое отрицание (NOT)
13	R	`new`	класс, список аргументов	создание объекта
		`( type )`	тип, любой	приведение (преобразование типа)
12	L	`*, /, %`	число, число	умножение, деление, остаток
11	L	`+, -`	число, число	прибавление, вычитание
		`+`	строка, любой	сцепление строк
10	L	`<<`	целое число, целое число	сдвиг влево
		`>>`	целое число, целое число	сдвиг вправо с дополнением знака
		`>>>`	целое число, целое число	сдвиг вправо с дополнением нулями
9	L	`<, <=`	число, число	меньше чем, меньше или равно
		`>, >=`	число, число	больше чем, больше или равно
		`instanceof`	ссылка, тип	сравнение типов
8	L	`==`	примитив, примитив	равны (имеют идентичные значения)
		`!=`	примитив, примитив	не равны (имеют разные значения)
		`==`	ссылка, ссылка	равны (относятся к одному объекту)
		`!=`	ссылка, ссылка	не равны (относятся к разным объектам)
7	L	`&`	целое число, целое число	поразрядное умножение (AND)
		`&`	булевый, булевый	логическое умножение (AND)
6	L	`^`	целое число, целое число	поразрядное исключающее ИЛИ (XOR)
		`^`	булевый, булевый	логическое исключающее ИЛИ (XOR)
5	L	`|`	целое число, целое число	поразрядное сложение (OR)
		`|`	булевый, булевый	логическое сложение (OR)
4	L	`&&`	булевый, булевый	условное умножение (AND)

Таблица 2.5 (продолжение)

P	A	Оператор	Тип(ы) операнда(ов)	Выполняемая операция
3	L	`||`	булевый, булевый	условное сложение (OR)
2	R	`?:`	булевый, любой, любой	условный (тернарный) оператор
1	R	`=`	переменная, любой	присваивание
		`*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=`	переменная, любой	присваивание с операцией

Приоритет

В колонке P табл. 2.5 определен *приоритет* (*precedence*) для каждого оператора. Приоритет определяет порядок выполнения операторов. Рассмотрим следующее выражение:

```
a + b * c
```

Оператор умножения имеет более высокий приоритет, чем оператор сложения, поэтому `a` прибавляется к произведению `b` и `c`. Приоритет операторов можно считать показателем степени связанности операторов с их операндами. Чем больше число, тем сильнее они связаны.

Приоритет операторов по умолчанию можно изменить с помощью круглых скобок, которые явно укажут порядок операций. Чтобы сложение выполнялось перед умножением, предыдущее выражение можно переписать таким образом:

```
(a + b) * c
```

В Java приоритет операторов был выбран с намерением обеспечить совместимость с C; разработчики языка C выбрали такой приоритет, чтобы писать большую часть выражений естественно, без круглых скобок. В Java существует несколько общеупотребительных идиом, требующих наличия круглых скобок. Среди них:

```
// Приведение класса, объединенное с доступом к члену
((Integer) o).intValue();

// Присваивание в совокупности со сравнением
while((line = in.readLine()) != null) { ... }

// Побитовые операторы в совокупности со сравнением
if ((flags & (PUBLIC | PROTECTED)) != 0) { ... }
```

Ассоциативность

Если выражение включает несколько операторов с одинаковым приоритетом, то порядком выполнения операций управляет ассоциативность операторов. Большинство операторов ассоциативны слева направо, то есть операции выполняются слева направо. Однако операторы присваивания и унарные операторы обратно ассоциативны (справа налево). В колонке A табл. 2.5 определена ассоциативность для каждого оператора или группы операторов. Значение L означает ассоциативность слева направо, а R означает обратную ассоциативность.

Аддитивные операторы ассоциативны слева направо, то есть выражение `a+b-c` вычисляется слева направо: `(a+b)-c`. Унарные операторы и операторы присваивания вычисляются справа налево. Рассмотрим более сложное выражение:

```
a = b += c = -~d
```

Оно вычисляется следующим образом:

```
a = (b += (c = -(~d)))
```

Так же как и приоритет операторов, ассоциативность устанавливает порядок вычисления выражений по умолчанию. Порядок по умолчанию можно изменить при помощи круглых скобок. Впрочем, в Java ассоциативность операторов по умолчанию предоставляет естественный синтаксис выражений. Очень редко возникает необходимость изменять его.

Количество и типы операндов

В четвертой колонке табл. 2.5 определены возможные типы и количество операндов для каждого оператора. Некоторые операторы работают только с одним операндом; они называются унарными операторами. Например, оператор «унарный минус» меняет знак отдельного числа.

```
-n        // Оператор «унарный минус»
```

Однако большинство операторов являются бинарными; они работают с двумя операндами. Оператор - (минус) может выступать в обеих формах:

```
a - b     // Оператор вычитания является бинарным
```

В Java также определен один тернарный (ternary) оператор, часто называемый условным оператором. Он похож на условный оператор `if`, но стоит внутри выражения. Эти три операнда разделяются знаком вопроса и двоеточием; второй и третий операнды должны относиться к одному типу.

```
x > y ? x : y // Тернарное выражение, возвращает большее число из x и y.
```

Кроме определенного количества операндов, каждый оператор ожидает конкретные типы операндов. В четвертой колонке таблицы перечислены типы операндов. Некоторые коды из этой колонки необходимо объяснить подробнее:

число

Целое число, значение с плавающей точкой или символ (то есть любой примитивный тип, кроме boolean).

целое число

Значения: `byte`, `short`, `int`, `long` или `char` (значения `long` не допускаются в операторе доступа к массиву `[]`).

ссылка

Объект или массив.

переменная

Переменная или любая другая величина (например, элемент массива), которой можно присвоить значение.

Возвращаемый тип

Подобно тому как каждый оператор ожидает операнды определенных типов, любой оператор выдает значение определенного типа. Арифметические, побитовые операто-

ры, а также операторы инкремента, декремента и сдвига возвращают double, если хотя бы один из операндов является double. В противном случае они возвращают float, если хотя бы один из операндов представляет float. Иначе они возвращают long, если хотя бы один из операндов является long. В других случаях возвращается int, даже если оба операнда представляют byte, short или char, диапазон которых более узкий, чем у int.

Операторы сравнения, равенства и булевые операторы всегда возвращают булевые значения. Все операторы присваивания возвращают присваиваемое значение, тип которого должен быть совместим с переменной, находящейся в левой части выражения. Условный оператор возвращает значение второго или третьего аргумента, которые должны относиться к одному типу.

Побочные эффекты

Каждый оператор вычисляет значение на основе одного или нескольких операндов. Однако некоторые операторы, в дополнение к основному вычислению, приводят к *побочным эффектам* (*side effects*). Если выражение подразумевает побочные эффекты, то при его вычислении состояние Java-программы изменяется настолько, что повторное вычисление выражения может привести к результату, отличному от первого. Например, оператор инкремента ++ имеет побочный эффект приращения переменной. Выражение ++a увеличивает значение переменной a и возвращает новое, увеличенное значение. При следующем вычислении этого выражения получится уже другое значение. Различные операторы присваивания также имеют побочные эффекты. Например, выражение a*=2 можно записать в виде a=a*2. Значением выражения является значение a, умноженное на 2, но выражение имеет побочный эффект сохранения нового значения в a. Оператор вызова метода () имеет побочные эффекты, если побочные эффекты есть у вызываемого метода. Некоторые методы, например Math.sqrt(), просто вычисляют и возвращают значение без каких-либо побочных эффектов. Однако обычно методы все-таки имеют побочные эффекты. И наконец, оператор new имеет существенный побочный эффект, выраженный в создании нового объекта.

Порядок вычислений

Вычисляя выражение, интерпретатор Java выполняет различные операции в последовательности, заданной круглыми скобоками, а также приоритетом и ассоциативностью операторов. Однако перед началом операции интерпретатор вычисляет операнды оператора (исключение составляют операторы &&, || и ?:, которые не всегда вычисляют все свои операнды). Интерпретатор всегда вычисляет операнды слева направо. Это имеет значение, если какой-либо из операндов является выражением с побочными эффектами. В качестве примера рассмотрим следующий код:

```
int a = 2;
int v = ++a + ++a * ++a;
```

Хотя умножение выполняется перед сложением, первыми вычисляются операнды оператора +. Таким образом, выражение равно 3+4*5, или 23.

Арифметические операторы

Поскольку большинство программ работает, в первую очередь, с числами, наиболее часто используются операторы, выполняющие арифметические операции. Арифметические операторы можно задавать с целыми числами, числами с плавающей точкой и даже с символами (то есть их можно применять с любым примитивным типом данных, кроме boolean). Арифметические операции с плавающей точкой применяются, если один из операндов является числом с плавающей точкой; в противном слу-

чае задействуется целочисленная арифметика. Это важно, так как арифметика для чисел с плавающей точкой отличается от целочисленной арифметики – например, по способу деления и способу обработки переполнения и округления. Существуют следующие арифметические операторы:

Сложение (+)

Оператор + складывает два числа. Вскоре вы убедитесь, что оператор + можно применять для сцепления строк. Если один из операндов сложения является строкой, то другой операнд также преобразуется в строку. Если вы хотите сочетать сложение со сцеплением, убедитесь в наличии круглых скобок. Например:

```
System.out.println("Total: " + 3 + 4);       // Отображает "Total: 34", не 7!
```

Вычитание (–)

Если оператор является бинарным, то он вычитает второй операнд из первого. Например, 7–3 равняется 4. Оператор «–» может выполнять унарную инверсию.

Умножение (*)

Оператор * умножает два числа. Например, 7*3 равняется 21.

Деление (/)

Оператор / делит первый операнд на второй. Деление целого числа на целое число дает целое число, а возможный остаток теряется. Однако если один из операндов является числом с плавающей точкой, то в результате деления получается число с плавающей точкой. Для целых чисел деление на нуль генерирует исключение `ArithmeticException`. В то же время при вычислениях, вовлекающих числа с плавающей точкой, деление на нуль просто дает бесконечность либо `NaN`:

```
7/3          // Равно 2
7/3.0f       // Равно 2.333333f
7/0          // Генерируется ArithmeticException
7/0.0        // Равно плюс бесконечности
0.0/0.0      // Равно NaN
```

Взятие по модулю (%)

Оператор % вычисляет первый операнд по модулю второго (то есть он возвращает целый остаток от деления первого операнда на второй). Например, 7%3 дает 1. Знак результата совпадает со знаком первого операнда. Хотя оператор взятия по модулю обычно используют с целыми числами, он также подходит для значений с плавающей точкой. Например, 4.3%2.1 равняется 0.1. Вычисление значений по модулю нуль для целых чисел приводит к `ArithmeticException`. Для значений с плавающей точкой любое значение по модулю 0.0 и бесконечность по любому модулю дают `NaN`.

Унарный минус (–)

Если «–» используется как унарный оператор перед отдельным операндом, он выполняет унарную инверсию знака. Другими словами, он переводит положительное значение в эквивалентное ему отрицательное, и наоборот.

Оператор сцепления строк

Кроме сложения чисел, оператор + и родственный оператор += могут сцеплять, или соединять, строки. Если один из складываемых операндов является строкой, оператор преобразовывает другой операнд в строку. Например:

```
// Отображает "Quotient: 2.3333333"
System.out.println("Quotient: " + 7/3.0f);
```

Не забывайте заключать операции сложения в круглые скобки, если сочетаете их со сцеплением строк. В противном случае оператор сложения будет интерпретирован как оператор сцепления.

В интерпретатор Java встроены строковые преобразования для всех примитивных типов данных. Объект можно преобразовать в строку посредством метода `toString()`. В некоторых классах определены специальные методы `toString()`, чтобы с их помощью легко преобразовывать объекты данного класса в строки. Массив преобразуется в строку при вызове встроенного метода `toString()`, который, к сожалению, не возвращает удобное строковое представление содержимого массива.

Операторы инкремента и декремента

Оператор ++ увеличивает на единицу операнд, который является переменной, элементом массива или полем объекта. Поведение данного оператора зависит от его положения относительно операнда. Если оператор находится перед операндом, то он называется оператором *префиксной формы* инкремента (*pre-increment*). Он увеличивает значение операнда на единицу и возвращает вычисленное значение. Если же оператор находится после операнда, то он называется оператором *постфиксной формы* инкремента (*post-increment*). Такой оператор увеличивает значение операнда на единицу, но возвращает значение операнда до увеличения.

Например, в следующем коде обе величины `i` и `j` получают значение 2:

```
i = 1;
j = ++i;
```

Однако в следующем примере `i` получает значение 2, а `j` – значение 1:

```
i = 1;
j = i++;
```

Аналогично, оператор -- уменьшает на единицу числовой операнд, который является переменной, элементом массива или полем объекта. Как и в случае оператора ++, поведение оператора -- зависит от его положения относительно операнда. Находясь перед операндом, он уменьшает значение операнда на единицу и возвращает полученное значение. Находясь после операнда, он уменьшает значение операнда на единицу, но возвращает первоначальное значение.

Выражения `x++` и `x--` эквивалентны выражениям `x=x+1` и `x=x-1` соответственно; `x` вычисляется один раз, за исключением случаев использования операторов инкремента и декремента. Если `x` является выражением с побочными эффектами, это существенно меняет дело. Например, следующие два выражения не идентичны:

```
a[i++]++;              // Увеличивает элемент массива
a[i++] = a[i++] + 1;   // Прибавляет единицу к одному элементу массива, а сохраняет его в другом
```

Чаще всего эти операторы – как в префиксной, так и в постфиксной форме – применяются для увеличения и уменьшения значения счетчика цикла.

Операторы сравнения

Операторы сравнения состоят из операторов равенства, которые проверяют равенство или неравенство значений, и операторов отношения, используемых с упорядоченными типами (числами и символами) при проверке соотношения больше/мень-

ше. Операторы обоих типов возвращают значение типа boolean, поэтому их обычно используют с условными операторами if и циклами while и for для выбора ветви или проверки условия выполнения цикла. Например:

```
if (o != null) ...;      // Оператор «не равно»
while(i < a.length) ...; // Оператор «меньше чем»
```

В Java предусмотрены следующие операторы равенства:

Равно (==)

Оператор == возвращает true (истина), если оба его операнда равны; если нет, то возвращается false (ложь). В случае примитивных операндов он проверяет идентичность самих значений операндов, однако в случае операндов ссылочных типов проверяется, ссылаются ли операнды на один и тот же объект или массив. Другими словами, оператор не проверяет равенство двух разных объектов или массивов. Посредством этого оператора не удастся проверить равенство двух различных строк.

Если оператор == сравнивает два числовых или символьных операнда различных типов, то до начала сравнения более узкий операнд преобразуется к типу более широкого. Например, при сравнении short и float величина типа short преобразуется во float до начала сравнения. Для чисел с плавающей точкой специальное отрицательное нулевое значение считается равным обычному положительному нулевому значению. Кроме того, специальное значение NaN (нечисловое) не равно ни одному другому числу, включая само себя. Чтобы проверить, является ли значение с плавающей точкой значением NaN, используйте метод Float.isNan() или Double.isNan().

Не равно (!=)

Оператор != прямо противоположен оператору ==. Он возвращает true, если два примитивных операнда имеют разные значения либо если два ссылочных операнда относятся к различным объектам или массивам. В противном случае он возвращает false.

Операторы отношения можно использовать с числами и символами, но нельзя применять со значениями типа boolean, объектами или массивами, так как данные типы не упорядочены. В Java предусмотрены следующие операторы отношения:

Меньше (<)

Возвращает true, если первый операнд меньше второго.

Меньше или равно (<=)

Возвращает true, если первый операнд меньше или равен второму.

Больше (>)

Возвращает true, если первый операнд больше второго.

Больше или равно (>=)

Возвращает true, если первый операнд больше или равен второму.

Булевы операторы

Как вы уже знаете, операторы сравнения сравнивают операнды и возвращают значение типа boolean. Величины такого типа часто используются в операторах ветвления и цикла. Чтобы выбор ветви и проверка цикла по условиям стали полезнее простого сравнения, можно задействовать *булевы* (или *логические*) операторы для объединения нескольких выражений сравнения в одно, более сложное выражение. Для логических операторов нужны операнды со значениями типа boolean. Эти операторы также возвращают значения типа boolean. Существуют такие логические операторы:

Условное И (`&&`)

Данный оператор выполняет логическую операцию И над операндами. Он возвращает `true` тогда и только тогда, когда оба операнда истинны. Если один или оба операнда ложны, он возвращает `false`. Например:

```
if (x < 10 && y > 3) ... // Если оба сравнения истинны
```

Данный оператор (и все другие логические операторы кроме унарного оператора `!`) имеет меньший приоритет, чем операторы сравнения. Таким образом, вышеприведенная запись кода вполне допустима. Однако некоторые программисты прибегают к помощи круглых скобок, чтобы явно обозначить порядок вычислений:

```
if ((x < 10) && (y > 3)) ...
```

Следует выбрать тот стиль, который вы считаете более удобным для чтения.

Этот оператор называется условным И, потому что он не всегда оценивает второй операнд. Если первый операнд равен `false`, значение выражения также будет `false`, каким бы ни было значение второго операнда. Поэтому, для большей эффективности, интерпретатор Java пропускает анализ второго операнда. В выражениях с побочными эффектами этот оператор следует применять осторожно, так как нет гарантии, что будет вычислен второй операнд. С другой стороны, этот оператор позволяет писать такие выражения:

```
if (data != null && i < data.length && data[i] != -1) ...
```

Второе и третье сравнения в данном выражении могут привести к ошибкам, если первое или второе сравнение возвращают `false`. К счастью, это не проблема, так как поведение оператора `&&` условно.

Условное ИЛИ (`||`)

Данный оператор выполняет логическую операцию ИЛИ на двух операндах типа `boolean`. Он возвращает `true`, если один или оба операнда истинны. Если оба операнда ложны, он возвращает `false`. Подобно оператору `&&`, оператор `||` не всегда вычисляет второй операнд. Если первый операнд равен `true`, значение выражения тоже будет `true`, каким бы ни было значение второго операнда. В этом случае оператор просто пропускает второй операнд.

Логическое НЕ (`!`)

Этот унарный оператор меняет `boolean`-значение операнда. Он возвращает `false`, если применяется к `true`-значению, и `true`, если задано `false`-значение. Данный оператор можно использовать в таких выражениях:

```
if (!found) …             // found является булевой переменной, объявленной где-то ранее
while(!c.isEmpty()) …     // Метод isEmpty() возвращает булево значение
```

Так как оператор `!` является унарным, он имеет высокий приоритет, и зачастую его нужно заключать в круглые скобки:

```
if (!(x > y && y > z))
```

Логическое И (&)

С операндами типа `boolean` поведение оператора `&` аналогично поведению оператора `&&`, но он всегда вычисляет оба операнда, каким бы ни было значение первого операнда. Однако данный оператор практически всегда используют с целыми числами как побитовый оператор, а большинство Java-программистов даже не догадываются о том, что его можно применять и с операндами типа `boolean`.

Логическое ИЛИ (`|`)

Данный оператор выполняет логическую операцию ИЛИ над двумя операндами типа `boolean`. Он аналогичен оператору `||`, но всегда вычисляет оба операнда, даже если первый операнд является `true`. Оператор `|` почти всегда используется как побитовый оператор для целых чисел, а с операндами типа `boolean` его применяют достаточно редко.

Логическое исключающее ИЛИ (`^`)

Для операндов типа `boolean` данный оператор вычисляет исключающее ИЛИ. Он возвращает `true`, если только один из двух операндов истинен. Другими словами, он возвращает `false`, если оба операнда ложны либо истинны. В отличие от операторов `&&` и `||` он всегда вычисляет оба операнда. Оператор `^` намного чаще применяется как побитовый оператор для целых чисел. С операндами типа `boolean` поведение данного оператора аналогично поведению оператора `!=`.

Побитовые операторы и операторы сдвига

Побитовые операторы и операторы сдвига являются операторами низкого уровня, манипулирующими отдельными битами целого числа. Чаще всего побитовые операторы применяются для проверки и установки отдельных флаговых битов. Чтобы понять их поведение, необходимо понимать двоичные числа (с основанием 2) и формат дополнения, используемый для представления отрицательных чисел. Эти операторы нельзя использовать с массивами, объектами, операндами типа `boolean` или с плавающей точкой. Как было описано в предыдущем разделе, с операндами типа `boolean` операторы `&`, `|` и `^` выполняют другие операции.

Если один из аргументов побитового оператора имеет значение `long`, то в результате тоже получится `long`. Если нет, то результат получается `int`. Если левый операнд оператора сдвига имеет значение `long`, то в результате тоже будет `long`. Если нет, то в результате получится `int`. Существуют такие операторы:

Побитовое дополнение (`~`)

Унарный оператор `~` является оператором побитового дополнения, или побитового НЕ. Он инвертирует каждый бит своего единственного операнда, преобразовывая единицы в нули и нули в единицы. Например:

```
byte b = ~12;           // ~00001100 ==> 11110011 или -13 (десятичная форма)
flags = flags & ~f;     // Очистить флаг f во множестве флагов
```

Побитовое И (`&`)

Этот оператор объединяет два целых операнда посредством логической операции И с их отдельными битами. В результате биты устанавливаются только там, где соответствующие биты установлены в обоих операндах. Например:

```
10 & 7                  // 00001010 & 00000111 ==> 00000010 или 2
if ((flags & f) != 0)   // Проверить наличие флага f
```

Для операндов типа `boolean` оператор `&` является редко используемым логическим оператором И, который описан выше.

Побитовое ИЛИ (`|`)

Этот оператор объединяет два целых операнда посредством логической операции ИЛИ с их отдельными битами. В результате биты устанавливаются только там, где соответствующие биты установлены в одном или обоих операндах. Если оба

соответствующих бита в операндах равны нулю, то результат содержит нулевой бит. Например:

```
10 | 7                  // 00001010 | 00000111 ==> 00001111 или 15
flags = flags | f;      // Установить флаг f
```

Для операндов типа `boolean` оператор | является редко используемым логическим оператором ИЛИ, который описан выше.

Побитовое исключающее ИЛИ (^)

Этот оператор объединяет два целых операнда посредством логической операции исключающего ИЛИ с их отдельными битами. В результате биты установлены только там, где соответствующие биты в двух операндах различны. Если оба бита – нули или единицы, то результат будет содержать нулевой бит. Например:

```
10 ^ 7                  // 00001010 ^ 00000111 ==> 00001101 или 13
```

Для операндов типа `boolean` оператор ^ является редко используемым логическим оператором исключающего ИЛИ.

Сдвиг влево (<<)

Оператор << сдвигает биты левого операнда влево на количество позиций, обозначенное правым операндом. Старшие биты левого операнда теряются, а справа добавляются нулевые биты. Сдвиг целого числа влево на `n` позиций равносилен умножению этого числа на 2^n. Например:

```
10 << 1   // 00001010 << 1 = 00010100 = 20 = 10*2
7 << 3    // 00000111 << 3 = 00111000 = 56 = 7*8
-1 << 2   // 0xFFFFFFFF << 2 = 0xFFFFFFFC = - 4 = -1*4
```

Если левый операнд представляет тип `long`, то значение правого операнда должно находиться в диапазоне между 0 и 63. В противном случае левый операнд считается `int`, а значение правого операнда должно располагаться между 0 и 31.

Сдвиг вправо со знаком (>>)

Оператор >> сдвигает биты левого операнда вправо на количество позиций, обозначенное правым операндом. Младшие биты левого операнда сдвигаются «наружу» и теряются. Старшие биты, сдвигаемые «внутрь», идентичны изначальному старшему биту левого операнда. Другими словами, нули сдвигаются к старшим битам, если левый операнд является положительным. И наоборот, единицы сдвигаются внутрь, если левый операнд – отрицательный. Данный прием называется *расширением знака*. Он применяется для сохранения знака левого операнда. Например:

```
10 >> 1       // 00001010 >> 1 = 00000101 = 5 = 10/2
27 >> 3       // 00011011 >> 3 = 00000011 = 3 = 27/8
-50 >> 2      // 11001110 >> 2 = 11110011 = -13 != - 50/4
```

Если левый операнд положительный, а правый имеет значение `n`, то оператор >> равносилен целочисленному делению на 2^n.

Сдвиг вправо без знака (>>>)

Этот оператор аналогичен оператору >>, но он всегда сдвигает нули к старшим битам, каким бы ни был знак левого операнда. Данный прием называется *дополнением нулями*. Его применяют, если левый операнд интерпретируется как значение без знака (несмотря на то что в Java все типы со знаком). Например:

```
0xff >>> 4    // 11111111 >>> 4 = 00001111 = 15 = 255/16
-50 >>> 2     // 0xFFFFFFCE >>> 2 = 0x3FFFFF33 = 107371811
```

Операторы присваивания

Операторы присваивания сохраняют, или присваивают, значение какой-либо переменной. Левый операнд должен быть локальной переменной, элементом массива или полем объекта. Справа может находиться любое значение типа, совместимого с переменной. Выражение присваивания равно значению, присвоенному переменной. Однако, что более существенно, выражение имеет побочный эффект фактического выполнения присваивания. В отличие от всех остальных бинарных операторов, операторы присваивания обратно ассоциативны, то есть присваивания в `a=b=c` выполняются справа налево: `a=(b=c)`.

Основным оператором присваивания является `=`. Не путайте его с оператором равенства `==`. Чтобы различать эти два оператора, следует читать `=` как «присвоено значение».

Кроме простого оператора присваивания в Java также определены 11 других операторов, которые сочетают присваивание с 5 арифметическими и 6 побитовыми операторами и операторами сдвига. Например, оператор `+=` считывает значение левой переменной, прибавляет к нему значение правого операнда, сохраняет сумму в левой переменной (побочный эффект) и возвращает сумму как значение выражения. Таким образом, выражение `x+=2` практически эквивалентно `x=x+2`. Разница между ними только в том, что при использовании оператора `+=` левый операнд вычисляется только один раз. Если данный операнд имеет побочный эффект, то это очень существенно. Например, два следующих выражения не равнозначны:

```
a[i++] += 2;
a[i++] = a[i++] + 2;
```

Общая форма комбинированных операторов присваивания:

```
var op= value
```

Это эквивалентно следующей форме (если только `var` не имеет побочных эффектов):

```
var = var op value
```

Доступны такие операторы:

```
+= -= *= /= %=          // Арифметические операторы плюс присваивание
&= |= ^=                // Побитовые операторы плюс присваивание
<<= >>= >>>=            // Операторы сдвига плюс присваивание
```

Наиболее часто применяются операторы `+=` и `-=`, хотя `&=` и `|=` можно задействовать в работе с `boolean`-флагами. Например:

```
i += 2;                 // Увеличение счетчика цикла на 2
c -= 5;                 // Уменьшение счетчика на 5
flags |= f;             // Установить флаг f в целочисленном множестве флагов
flags &= ~f;            // Очистить флаг f в целочисленном множестве флагов
```

Условный оператор

Условный оператор `?:` является тернарным (три операнда) оператором, унаследованным из языка C. Он позволяет внедрять условие в само выражение. Его можно представить как версию оператора `if/else`. Знак вопроса (`?`) разделяет первый и второй

операнды условного оператора, а двоеточие (:) – второй и третий. Первый операнд должен иметь boolean-значение. Второй и третий операнды могут быть любого типа, но они должны приводиться к одному и тому же типу.

Вначале условный оператор вычисляет первый операнд. Если он истинен, оператор вычисляет второй операнд, а результат представляет как значение выражения. В противном случае, если первый операнд ложен, условный оператор вычисляет и возвращает третий операнд. Условный оператор никогда не вычисляет оба операнда (второй и третий), поэтому следует осторожно использовать выражения с побочными эффектами. Ниже приведены примеры оператора:

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

Примечание: приоритет оператора ?: ниже, чем всех других операторов, кроме операторов присваивания. Таким образом, операнды этого оператора обычно не нужно заключать в круглые скобки. Однако большинству программистов легче читать условные выражения, когда первый операнд заключен в круглые скобки – особенно если учесть, что и в условном операторе if условное выражение всегда заключено в круглые скобки.

Оператор instanceof

Для оператора instanceof левым операндом должно быть значение объекта или массива, а правым операндом – имя ссылочного типа. Он возвращает true, если объект или массив является *экземпляром* указанного типа, иначе возвращается false. Оператор instanceof всегда возвращает false, если левый операнд является null. Если выражение instanceof равно true, можно без риска присваивать левый операнд переменной типа правого операнда.

Оператор instanceof можно использовать только с типами и значениями массивов и объектов, а не с примитивными типами и значениями. Далее в этой главе типы объектов и массивов рассмотрены более подробно. Ниже приведены примеры использования instanceof:

```
"string" instanceof String    // Истинно: все строки являются экземплярами String
"" instanceof Object          // Истинно: строки также являются экземплярами Object
null instanceof String        // Ложно: null не является чьим-либо экземпляром

Object o = new int[] {1,2,3};
o instanceof int[]            // Истинно: значение массива является int-массивом
o instanceof byte[]           // Ложно: значение массива не является byte-массивом
o instanceof Object           // Истинно: все массивы являются экземплярами Object

// Используйте instanceof, чтобы убедиться в безопасности приведения объекта
if (object instanceof Point) {
  Point p = (Point) object;
}
```

Специальные операторы

В Java определены 5 языковых конструкций, которые иногда считаются операторами, а иногда – просто частью основного синтаксиса языка. Эти операторы перечислены в табл. 2.5, чтобы показать их приоритет по отношению к другим операторам. Далее эти языковые конструкции будут описаны более подробно; здесь они изложены вкратце, чтобы вы могли распознать их в примерах кода:

Доступ к члену объекта (`.`)

Объектом (*object*) называется множество данных и методы, работающие с этими данными, а поля данных и методы объекта являются его членами (members). Оператор точка (`.`) обеспечивает доступ к этим членам. Если `o` – это выражение, представляющее собой объектную ссылку, а `f` является именем поля объекта, то `o.f` равно значению данного поля. Если `m` – имя метода, то `o.m` относится к этому методу и позволяет вызывать его при помощи оператора `()`, описанного ниже.

Доступ к элементу массива (`[]`)

Массивом (*array*) называется нумерованный список значений. Можно обратиться к каждому элементу массива через его номер, или индекс. Оператор `[]` позволяет обращаться к отдельным элементам массива. Если `b` является массивом, а `i` – выражением с типом `int`, то выражение `b[i]` ссылается на один из элементов `b`. В отличие от других операторов, работающих с целыми значениями, данный оператор ограничивает типы значения индекса до `int` и более узких.

Вызов метода (`()`)

Метод (*method*) представляет собой именованный фрагмент Java-кода, который можно запускать, или *вызывать*, сопровождая имя метода несколькими выражениями, заключенными в круглые скобки и разделенными запятыми (выражений может и не быть). Значения этих выражений являются *аргументами* метода. Метод обрабатывает аргументы и может возвращать значение, которое становится значением выражения вызова метода. Если `o.m` является методом, не ожидающим аргументов, то его можно вызвать при помощи `o.m()`. Например, если метод требует наличия трех аргументов, то его можно вызвать выражением `o.m(x,y,z)`. Перед вызовом метода интерпретатор Java вычисляет каждый передаваемый методу аргумент. Эти выражения обязательно вычисляются слева направо, что имеет значение только в случае побочных эффектов какого-либо из аргументов.

Создание объекта (`new`)

В Java объекты (и массивы) создаются при помощи оператора `new`, за которым следует тип создаваемого объекта и заключенный в круглые скобки список аргументов, передаваемый *конструктору* объектов. Конструктором называется особый метод, который инициализирует вновь созданный объект. Таким образом, синтаксис создания объекта похож на синтаксис вызова метода. Например:

```
new ArrayList();
new Point(1,2);
```

Преобразование или приведение типа (`()`)

Как вы уже могли убедиться, круглые скобки можно применять как оператор сужающего преобразования, или приведения типа. Первым операндом этого оператора является тип, к которому выполняется приведение; он заключается в скобки. Второй операнд представляет собой преобразуемое значение; он следует после скобок. Например:

```
(byte) 28              // Целый литерал приводится к типу byte
(int) (x + 3.14f)      // Значение суммы с плавающей точкой приводится к целому
(String)h.get(k)       // Базовый объект приводится к более узкому типу
```

Операторы-инструкции

Оператором-инструкцией (*statement*) называется единичная команда, выполняемая интерпретатором Java.[1] По умолчанию интерпретатор Java последовательно выполняет операторы в том порядке, в котором они записаны. Однако большинство операторов-инструкций, определенных в Java, являются операторами управления потоком. Например, условные операторы и операторы цикла изменяют порядок выполнения по умолчанию на строго определенный. В табл. 2.6 приведены операторы-инструкции, определенные в Java.

Таблица 2.6. Операторы-инструкции Java

Оператор	Назначение	Синтаксис

выражение	побочные эффекты	`var = expr;`
		`expr++;`
		`method();`
		`new Type();`

составной	сгруппировать операторы	`{ statements }`
пустой	ничего не делать	`;`
с меткой	назвать оператор	`label : statement`
переменная	объявить переменную	`[final] type name [= value] [, name [= value]] ...;`
`if`	условный	`if ( expr ) statement [ else statement]`

`switch`	условный	`switch ( expr ) {`
		`  [ case expr : statements ] ...`
		`  [ default: statements ]`
		`}`

`while`	цикл	`while ( expr ) statement`
`do`	цикл	`do statement while ( expr );`
`for`	упрощенный цикл	`for ( init ; test ; increment ) statement`
`break`	выйти из блока	`break [ label ] ;`
`continue`	возобновить цикл	`continue [ label ] ;`
`return`	закончить метод	`return [ expr ] ;`
`synchronized`	критическая секция	`synchronized ( expr ) { statements }`
`throw`	сгенерировать исключение	`throw expr ;`

[1] В английском языке для обозначения обычных операторов, которые рассматривались выше в этой главе, и инструкций существуют два различных термина: operator и statement соответственно. Однако в русском языке оба этих термина переводятся как «оператор». В заголовке этого раздела мы специально сделали акцент на различии этих терминов в английском языке, но далее для краткости будем использовать термин «оператор» как для операторов, так и для операторов-инструкций. – *Примеч. перев.*

Оператор	Назначение	Синтаксис
try	обработать исключение	try { *statements* } [catch (*type name*) { *statements* }] ... [finally { *statements* }]
assert	проверить инвариант	assert *invariant* [: *error*] ; (Java 1.4 и последующие версии)

Операторы-выражения

Как уже упоминалось в этой главе, определенные типы выражений в Java имеют побочные эффекты. Другими словами, они не просто несут какое-то значение, а определенным образом влияют на состояние самой программы. Любое выражение можно использовать в качестве оператора, если поставить после него точку с запятой. Присваивание, инкременты и декременты, вызов метода и создание объекта являются разрешенными типами операторов. Например:

```
a = 1;                              // Присваивание
x *= 2;                             // Присваивание посредством операции
i++;                                // Постфиксная форма инкремента
--c;                                // Префиксная форма декремента
System.out.println("statement"); // Вызов метода
```

Составные операторы

Составным оператором (*compound statement*) называется любое количество операторов любого типа, заключенных в фигурные скобки. Составной оператор можно применить в любой части программы, где согласно синтаксису Java необходим оператор:

```
for(int i = 0; i < 10; i++) {
  a[i]++;                           // Тело цикла является составным оператором.
  b[i]--;                           // Он состоит из двух операторов-выражений,
}                                   // заключенных в фигурные скобки.
```

Пустой оператор

Пустой оператор (*empty statement*) в Java обозначается точкой с запятой. Пустой оператор ничего не делает, однако иногда такой синтаксис бывает полезен. Например, его можно применять для указания пустого тела цикла for:

```
for(int i = 0; i < 10; a[i++]++) // Увеличить элементы массива
  /* empty */;                    // Тело цикла - пустой оператор
```

Оператор с меткой

Оператор с меткой (*labeled statement*) – это оператор, которому было дано имя, записанное перед оператором и отделенное от него двоеточием. Метки используются операторами break и continue. Например:

```
rowLoop: for(int r = 0; r < rows.length; r++) {          // Цикл с меткой
  colLoop: for(int c = 0; c < columns.length; c++) {     // Еще один
break rowLoop;                                           // Использование метки
  }
}
```

Оператор объявления локальной переменной

Локальная переменная (*local variable*), обычно просто называемая переменной, является символическим именем места хранения значения, определяемым внутри метода или составного оператора. Перед использованием переменные нужно объявлять при помощи оператора объявления переменной. Поскольку язык Java строго типизирован, объявление переменной определяет тип переменной, а хранимые в ней значения могут быть только указанного типа.

Самая простая форма объявления переменной определяет тип и имя переменной:

```
int counter;
String s;
```

Объявление переменной также может содержать *инициализатор* – выражение, определяющее начальное значение переменной. Например:

```
int i = 0;
String s = readLine();
int[] data = {x+1, x+2, x+3};          // Инициализаторы массива описаны далее
```

Компилятор Java не позволяет использовать неинициализированную локальную переменную, поэтому обычно удобнее сочетать объявление переменной и ее инициализацию в едином операторе. Выражение-инициализатор не обязательно должно быть литералом или константным выражением, чтобы компилятор мог его вычислить; оно может быть выражением произвольной сложности со значением, вычисляемым при выполнении программы.

В одном операторе объявления переменной можно объявлять и инициализировать несколько переменных, если это переменные одного типа. Имена переменных и инициализаторы отделяются друг от друга запятыми (инициализаторов может не быть):

```
int i, j, k;
float x = 1.0, y = 1.0;
String question = "Really Quit?", response;
```

В Java 1.1 и последующих версиях операторы объявления переменной могут начинаться с ключевого слова `final`. Данный модификатор указывает, что после того как переменной было присвоено начальное значение, оно уже не может быть изменено:

```
final String greeting = getLocalLanguageGreeting();
```

Программистам на языке C следует обратить внимание на то, что в Java операторы объявления переменной могут находиться в любой части кода; они не ограничены началом метода или блоком программы. Объявления локальных переменных также можно включать в инициализацию цикла `for`, о чем будет подробно рассказано позже.

Локальные переменные можно использовать только внутри метода или блока программы, в котором они определены. Это называется *контекстом* (*scope*), или *лексическим контекстом* (*lexical scope*):

```
void method() {                    // Базовый метод
  int i = 0;                       // Объявить i
  while (i < 10) {                 // i здесь в контексте
    int j = 0;                     // Объявить j; здесь начинается контекст j
    i++;                           // i здесь в контексте; увеличить ее
  }                                // j больше не в контексте; ее нельзя больше использовать
  System.out.println(i);           // i все еще в контексте
}                                  // Контекст i заканчивается здесь
```

Оператор if/else

Оператор `if` является основным оператором управления, который позволяет Java принимать решения или, точнее, выполнять операторы по условию. Оператор `if` содержит выражение и оператор. Если выражение равно `true`, то интерпретатор выполняет данный оператор. Однако если выражение равно `false`, то оператор пропускается. Например:

```
if (username == null)              // Если имя пользователя равно null, то
  username = "John Doe";           // определить его
```

Согласно синтаксису оператора `if`, выражение необходимо заключать в круглые скобки, хотя они и кажутся избыточными.

Как уже упоминалось, группа операторов, заключенная в фигурные скобки, сама является оператором, поэтому можно записать такой оператор `if`:

```
if ((address == null) || (address.equals(""))) {
  address = "[undefined]";
  System.out.println("WARNING: адрес не указан.");
}
```

Оператор `if` может содержать необязательное ключевое слово `else`, за которым следует второй операнд. При записи оператора в таком виде вычисляется выражение, а затем, если результат равен `true`, выполняется первый оператор. Иначе выполняется второй оператор. Например:

```
if (username != null) System.out.println("Привет " + username);
else {
  username = askQuestion("Как тебя зовут?");
  System.out.println("Привет " + username + ". Welcome!");
}
```

При использовании вложенных операторов `if/else` необходимо убедиться в том, что оператору `else` соответствует подходящий оператор `if`. Рассмотрим следующие строки:

```
if (i == j)
  if (j == k) System.out.println("i равно k");
else
  System.out.println("i не равно j");     // ОШИБКА!!
```

В этом примере внутренний оператор `if` образует единый оператор, выполнение которого зависит от внешнего оператора `if`. К сожалению, не ясно (если связи не определяются отступами), к какому оператору `if` относится данный оператор `else`. В представленном примере отступы сбивают с толку. Согласно правилу, такой оператор `else` связан с ближайшим к нему оператором `if`. С правильно установленным отступом этот код выглядит следующим образом:

```
if (i == j)
  if (j == k)
    System.out.println("i равно k");
  else
    System.out.println("i не равно j");  // ОШИБКА!!
```

Это допустимый код, но он явно отличается от кода, который хотел записать программист. При работе с операторами `if` нужно использовать фигурные скобки. Это облегчит чтение кода. Лучше писать код таким образом:

```
if (i == j) {
  if (j == k)
    System.out.println("i равно k");
} else {
  System.out.println("i не равно j");
}
```

Оператор else if

Оператор `if/else` используют при проверке состояния и при выборе одного из двух операторов или блоков программы. Но что делать, если нужно выбирать из нескольких блоков? В таком случае обычно применяется оператор `else if`, который на самом деле является идиоматическим вариантом стандартного оператора `if/else`, а не новой синтаксической конструкцией. Он выглядит так:

```
if (n == 1) {
    // Выполнить блок кода №1
}
else if (n == 2) {
    // Выполнить блок кода №2
}
else if (n == 3) {
    // Выполнить блок кода №3
}
else {
    // Если ни одно условие не выполнилось, выполнить блок №4
}
```

В этом коде нет ничего особенного. Это просто серия операторов `if`, каждый из которых является частью выражения `else` предыдущего оператора. Предпочтительнее и удобнее использовать идиому `else if`, а не полную вложенную форму:

```
if (n == 1) {
       // Выполнить блок кода №1
}
else {
  if (n == 2) {
       // Выполнить блок кода №2
  }
  else {
    if (n == 3) {
       // Выполнить блок кода №3
    }
    else {
       // Если ни одно условие не выполнилось, выполнить блок №4
    }
  }
}
```

Оператор switch

Оператор `if` выполняет разветвление потока выполняемой программы. Как показано в предыдущем разделе, для многоальтернативного ветвления можно использовать несколько операторов `if`. Однако это не всегда лучший вариант, особенно если все ветви зависят от значения одной переменной. В этом случае было бы обременительно проверять значение одной и той же переменной в нескольких операторах `if`.

Лучше использовать оператор switch, унаследованный из языка программирования C. Хотя синтаксис данного оператора и не так элегантен, как остальная часть Java, грубая практичность конструкции делает его полезным. Если вы не знакомы с самим оператором switch, вы можете знать по крайней мере базовую концепцию, известную как вычисляемая таблица переходов. Оператор switch состоит из целочисленного выражения и тела, которое содержит различные пронумерованные точки входа. Выражение вычисляется, и управление переходит на точку входа, определенную полученным значением. Например, следующий оператор switch эквивалентен повторяющимся операторам if и else/if, представленным в предыдущем подразделе:

```
switch(n) {
      case 1:                         // Начать здесь, если n == 1
        // Выполнить блок кода №1
        break;                        // Остановиться здесь
      case 2:                         // Начать здесь, если n == 2
        // Выполнить блок кода №2
        break;                        // Остановиться здесь
      case 3:                         // Начать здесь, если n == 3
        // Выполнить блок кода №3
        break;                        // Остановиться здесь
      default:                        // Если ни одно условие не выполнилось,
        // Выполнить блок кода №4
        break;                        // Остановиться здесь
}
```

Как видно из примера, различные точки входа в операторе switch отмечены либо ключевым словом case со следующим за ним целым числом и двоеточием, либо особым ключевым словом default и двоеточием. При выполнении оператора switch интерпретатор вычисляет значение выражения в круглых скобках, а затем ищет метку case, соответствующую полученному значению. Если интерпретатор находит метку, он начинает выполнять блок программы с первого оператора после метки case. Если интерпретатор не находит метку case с соответствующим значением, он начинает выполнение блока с первого оператора после специальной метки default:. Или, если нет метки default:, интерпретатор полностью пропускает тело оператора switch.

Обратите внимание на использование ключевого слова break в конце каждого case в предыдущем коде. Оператор break описан далее в этой главе; в данном случае он заставляет интерпретатор покинуть оператор switch. Метки case определяют только *начальную точку* нужного кода. Отдельные варианты не являются независимыми блоками программы и не содержат никакой неявной точки окончания. Поэтому при помощи оператора break или другого подходящего оператора нужно явно определить окончание каждого варианта. Если нет оператора break, оператор switch начинает выполнять код с первого оператора после соответствующей метки case и продолжает выполнять операторы, пока не достигнет конца блока. Иногда удобно писать код с последовательным переходом от одной метки case к другой, однако в 99% случаев вам пришлось бы завершать каждый раздел case и default оператором, приводящим к завершению выполнения оператора switch. Обычно в таких случаях применяют оператор break, но также подходят return и throw.

Оператор switch может содержать более одной метки case для одного и того же оператора. Рассмотрим такой оператор switch для следующего метода:

```
boolean parseYesOrNoResponse(char response) {
  switch(response) {
    case 'y':
```

```
      case 'Y': return true;
      case 'n':
      case 'N': return false;
      default:
        throw
          new IllegalArgumentException("Ответ должен быть Y или N");
    }
  }
```

Есть несколько важных ограничений для оператора `switch` и его меток `case`. Во-первых, выражение, связанное с оператором `switch`, должно содержать значение типа `byte`, `char`, `short` или `int`. Типы с плавающей точкой и `boolean` не поддерживаются. То же самое относится к `long`, хотя `long` является целым типом. Во-вторых, значение, ассоциируемое с каждой меткой `case`, должно быть постоянным значением или выражением, которое может вычислить компилятор. Например, метка `case` не может содержать выражение, вычисляемое во время выполнения – с переменными и вызовами методов. В-третьих, значения меток `case` должны соответствовать типу данных выражения `switch`. И наконец, не разрешается создавать две и более метки `case` с одинаковым значением или больше одной метки `default`.

Оператор while

Так же как оператор `if` является основным оператором управления, который позволяет Java принимать решения, оператор `while` представляет собой основной оператор для выполнения повторяющихся действий. У него следующая синтаксическая структура:

```
while (expression)
  statement
```

Оператор `while` сначала вычисляет выражение (*expression*). Если оно равно `false`, то интерпретатор пропускает связанный с циклом оператор (*statement*) и переходит к следующему оператору программы. Если выражение равно `true`, то выполняется оператор тела цикла (*statement*), и выражение (*expression*) вычисляется повторно. И снова, если значение выражения равно `false`, то интерпретатор переходит к следующему оператору программы, иначе он заново выполняет данный оператор. Цикл продолжается, пока выражение не станет равно `false` (то есть пока оно равно `true`), после чего оператор `while` завершается, а интерпретатор переходит к следующему оператору. Можно создать бесконечный цикл с синтаксисом `while(true)`.

Ниже приведен пример цикла `while`, который отображает числа от 0 до 9:

```
int count = 0;
while (count < 10) {
  System.out.println(count);
  count++;
}
```

Как видно из примера, сначала переменная `count` равна нулю, а при каждом выполнении цикла она увеличивается на единицу. Как только цикл выполнен десять раз, выражение принимает значение `false` (то есть `count` уже не меньше 10), оператор `while` завершается, а интерпретатор Java переходит к следующему оператору программы. Большинство циклов имеют переменную счетчика, подобную `count`. В качестве счетчиков цикла обычно применяются переменные с такими именами, как `i`, `j` и `k`, хотя следует выбирать более информативные имена, если они облегчат понимание кода.

Оператор do

Цикл do очень похож на цикл while, за исключением того, что выражение цикла проверяется в конце цикла, а не в начале. Это значит, что тело цикла всегда выполняется как минимум один раз. Синтаксис цикла таков:

```
do
  statement
while ( expression ) ;
```

Следует выделить несколько различий между циклом do и обычным циклом while. Во-первых, цикл do требует двух ключевых слов: do для обозначения начала и while для обозначения конца цикла и введения его условия. Кроме того, в отличие от цикла while, цикл do завершается точкой с запятой. Это объясняется тем, что в конце цикла do стоит условие цикла, а не просто фигурная скобка, обозначающая конец тела цикла. Следующий цикл do отображает тот же результат, что и цикл while, рассмотренный выше:

```
int count = 0;
do {
  System.out.println(count);
  count++;
} while(count < 10);
```

Цикл do используется не так часто, как его собрат while. Так происходит потому, что на практике редко возникает ситуация, когда вам необходимо выполнять цикл не менее одного раза.

Оператор for

Зачастую оператор for позволяет организовать цикл более удобно, чем операторы while и do. Оператор for использует типичный шаблон цикла. Большинство циклов содержат счетчик или какую-то переменную состояния, которая задается перед началом цикла, проверяется на необходимость выполнения тела цикла, а затем в конце тела цикла увеличивается на единицу или обновляется каким-либо образом перед новым вычислением контрольного выражения. Инициализация, проверка и обновление – три ключевых этапа обработки переменной цикла, а оператор for непосредственно включает эти этапы в синтаксис цикла:

```
for(initialize ; test ; update)
    statement
```

Этот цикл for, по существу, эквивалентен следующему циклу while:[1]

```
initialize;
while(test) {
  statement;
  update;
}
```

Если выражения *initialize*, *test* и *update* поставить в начале цикла *for*, то будет намного легче понять работу цикла и предотвратить такие ошибки, как пропущенная инициализация или обновление переменной цикла. Интерпретатор отбрасывает зна-

[1] Как будет видно при рассмотрении оператора continue, данный цикл while не в полной мере аналогичен циклу for.

чения выражений *initialize* и *update*. Таким образом, чтобы принести какую-либо пользу, эти выражения должны иметь побочные эффекты. Обычно *initialize* является выражением присваивания, а *update* выступает инкрементом, декрементом или другим выражением присваивания.

Следующий цикл `for`, как и предыдущие циклы `while` и `do`, выводит числа от 0 до 9:

```
int count;
for(count = 0 ; count < 10 ; count++)
  System.out.println(count);
```

Вся важная информация о переменной цикла расположена в одной строке, что дает ясное представление о том, как цикл будет выполняться. Кроме того, выражение обновления в самом операторе `for` уменьшает тело цикла до одного оператора; не нужно даже использовать фигурные скобки в блоке операторов.

Цикл `for` поддерживает дополнительный синтаксис, что еще больше упрощает его использование. Поскольку большинство циклов применяют свои переменные только внутри цикла, цикл `for` допускает, чтобы выражение *initialize* было полным объявлением переменной. Таким образом, переменная ограничена телом цикла и невидима извне. Например:

```
for(int count = 0 ; count < 10 ; count++)
  System.out.println(count);
```

К тому же синтаксис цикла `for` не ограничивается циклами с единственной переменной. Как в выражении *initialize*, так и в выражении *update* можно использовать запятую для разделения нескольких выражений инициализации и обновления. Например:

```
for(int i = 0, j = 10; i < 10; i++, j--)
  sum += i * j;
```

Хотя все предыдущие примеры использовали счетчики, циклы `for` не ограничены такими конструкциями. Например, цикл `for` можно задействовать для прохождения по элементам связного списка:

```
for(Node n = listHead; n != null; n = n.nextNode())
  process(n);
```

Выражения *initialize*, *test* и *update* цикла *for* не обязательны; необходима лишь точка с запятой для разделения выражений. Если пропущено выражение *test*, то считается, что его значение равно `true`. Таким образом, можно написать бесконечный цикл: `for(;;)`.

Оператор break

Оператор `break` заставляет интерпретатор Java сразу перейти к окончанию составного оператора, в который входит этот оператор `break`. Мы уже рассмотрели применение оператора `break` с оператором `switch`. Оператор `break` чаще всего записывают в виде ключевого слова `break` и точки с запятой:

```
break;
```

В таком виде он заставляет интерпретатор Java выйти из ближайшего цикла `while`, `do`, `for` или оператора `switch`. Например:

```
for(int i = 0; i < data.length; i++) {       // Цикл по массиву data
  if(data[i] == target) {                    // Если мы нашли то, что искали,
      index = i;                             // запомним, где мы это нашли,
      break;                                 // и остановим поиск!
```

```
    }
}   // После выполнения break интерпретатор Java переходит сюда.
```

После оператора `break` может стоять метка составного оператора. В таком виде оператор `break` заставляет интерпретатор Java немедленно выйти из указанного блока, который может быть любым оператором, а не только оператором `switch` или циклом. Например:

```
testfornull: if(data != null) {              // Если массив определен,
  for(int row = 0; row < numrows; row++) {   // цикл по строкам,
    for(int col = 0; col < numcols; col++) { // затем по столбцам.
      if(data[row][col] == null)             // Если в массиве нет данных,
      break testfornull;                     // считаем, что массив не определен.
  }
}   // После выполнения break testfornull интерпретатор Java переходит сюда.
```

Оператор continue

Если оператор `break` прерывает цикл, то оператор `continue` завершает текущую итерацию цикла и начинает новую. Оператор `continue`, как с меткой, так и без нее, можно использовать только внутри цикла `while`, `do` или `for`. Оператор `continue` без метки заставляет ближайший цикл начать новую итерацию. Оператор `continue` с меткой, то есть с именем окружающего цикла, заставляет этот цикл начать новую итерацию. Например:

```
for(int i = 0; i < data.length; i++) {      // Цикл по массиву data.
  if(data[i] == -1)                         // Если значение отсутствует,
    continue;                               // начать следующую итерацию.
  process(data[i]);                         // Обработать значение.
}
```

В каждом из циклов `while`, `do` и `for` оператор `continue` начинает новую итерацию поразному:

- В случае цикла `while` интерпретатор Java просто возвращается в начало цикла, еще раз проверяет условие и, если оно равно `true`, снова выполняет тело цикла.
- В случае цикла `do` интерпретатор Java переходит в конец цикла, где он проверяет условие цикла, чтобы определить, нужно ли выполнять следующую итерацию цикла.
- Для цикла `for` интерпретатор переходит в начало цикла, где он сначала вычисляет выражение `update`, а затем выражение `test`, определяя необходимость повторного выполнения цикла. Таким образом, поведение цикла `for` с оператором `continue` отличается от поведения «эквивалентного ему» цикла `while`, рассмотренного ранее; `update` вычисляется только в цикле `for`, но не вычисляется в цикле `while`.

Оператор return

Оператор `return` предписывает интерпретатору Java остановить выполнение текущего метода. Если метод возвращает значение, оператор `return` сопровождается некоторым выражением. Значение этого выражения становится возвращаемым значением метода. Например, следующий метод вычисляет и возвращает квадратный корень числа:

```
double square(double x) {              // Метод для вычисления квадрата x
  return x * x;                        // Вычислить и вернуть значение
}
```

Некоторые методы объявляются как `void`, так как они не возвращают никакого значения. Интерпретатор Java выполняет такие методы путем последовательного выполнения операторов до тех пор, пока он не достигнет конца метода. Выполнив последний оператор, интерпретатор выполняет возврат. Однако иногда метод `void` должен завершиться, не дожидаясь выполнения последнего оператора. В этом случае можно использовать оператор `return` без выражения. Например, следующий метод выводит, но не возвращает квадратный корень аргумента. Если аргумент является отрицательным числом, при возврате из метода ничего не выводится:

```
void printSquareRoot(double x) {              // Метод для печати квадратного корня x
  if(x < 0) return;                           // Если x отрицательный, вернуться
  System.out.println(Math.sqrt(x));           // Напечатать квадратный корень x
}                                             // Конец метода: неявный возврат
```

Оператор synchronized

Поскольку Java является многопоточной системой, нужно постоянно следить, чтобы несколько потоков не модифицировали объект одновременно, разрушая состояние объекта. Секции кода, которые нельзя выполнять одновременно, называются *критическими секциями*. Для защиты таких секций в Java предусмотрен оператор `synchronized`. Его синтаксис таков:

```
synchronized ( expression ) {
  statements
}
```

expression — это выражение, значением которого является объект или массив. Операторы *statements* составляют код критической секции и должны обрамляться фигурными скобками. Перед выполнением критической секции интерпретатор Java получает исключающую блокировку объекта или массива, определенного выражением *expression*. Он сохраняет блокировку до конца выполнения критической секции, а затем освобождает ее. Пока объект заблокирован одним потоком, никакой другой поток не имеет к нему доступа. Следовательно, никакой другой поток не сможет выполнить эту или любую другую критическую секцию с блокировкой данного объекта. Если поток не получает доступа для выполнения критической секции сразу, он просто ждет ее разблокировки.

Оператор `synchronized` нужен лишь тогда, когда программа создает несколько потоков, совместно использующих данные. Не нужно защищать структуру данных оператором `synchronized`, если с ней взаимодействует только один поток. Оператор `synchronized` необходим в таком коде:

```
public static void SortIntArray(int[] a) {
  // Сортировать массив a. Сортировка выполняется внутри оператора synchronized, поэтому
  // никакой другой поток не сможет изменить элементы массива, пока мы его сортируем
  synchronized (a) {
  // Здесь выполняется сортировка
  }
}
```

Ключевое слово `synchronized` может выступать как модификатор. В такой форме оно применяется чаще, чем в форме оператора. Если ключевое слово `synchronized` используется в методе, оно указывает на то, что весь метод является критической секцией. Перед выполнением метода класса `synchronized` (статического метода) Java получает исключающую блокировку этого класса. Для метода экземпляра `synchronized` Java

получает исключающую блокировку экземпляра класса (методы класса и экземпляра рассмотрены в главе 3).

Оператор throw

Исключением (*exception*) называется сигнал о возникновении какой-либо исключительной ситуации или ошибки. *Генерировать* (*throw*) исключение значит сигнализировать об исключительном состоянии. *Перехватить* (*catch*) исключение значит обработать его, т. е. предпринять действия по восстановлению нормального хода программы.

Для генерации исключения в Java используется оператор `throw`:

```
throw expression;
```

Выражение (*expression*) должно возвращать объект исключения, который описывает возникшее исключение или ошибку. Более подробно типы исключений будут рассмотрены позже, а пока вам нужно знать, что исключение представлено объектом. Ниже приведен пример кода, генерирующий исключение:

```
public static double factorial(int x) {
  if(x < 0)
    throw new IllegalArgumentException("x должен быть >= 0");
  double fact;
  for(fact=1.0; x > 1; fact *= x, x--) /* пусто */;
  // Обратите внимание на использование пустого оператора
  return fact;
}
```

При переходе к оператору `throw` интерпретатор сразу прекращает обычное выполнение программы и ищет обработчик исключения, который мог бы перехватить, или обработать, исключение. Обработчики исключений записываются посредством оператора `try/catch/finally`, описанного в следующем подразделе. Сначала интерпретатор Java ищет обработчик исключений в окружающем блоке программы. Если интерпретатор его находит, он прерывает этот блок и выполняет связанный с ним код обработки исключений. Выполнив данный код, интерпретатор продолжает выполнение с оператора, следующего сразу за кодом обработки.

Если окружающий блок программы не содержит необходимый обработчик исключений, интерпретатор переходит к следующему, более объемлющему блоку метода. Это продолжается до тех пор, пока не будет найден обработчик. Если метод не содержит обработчик исключений, который сможет обработать исключение, сгенерированное оператором `throw`, то интерпретатор прерывает текущий метод и возвращается к оператору вызова. Затем интерпретатор ищет обработчик исключений в блоках программы вызываемого метода. Таким образом, исключения распространяются по лексической структуре методов Java и по стеку вызовов интерпретатора Java. Если исключение не перехвачено, оно распространяется вплоть до метода `main()` программы. Если оно не обрабатывается в данном методе, интерпретатор Java выводит сообщение об ошибке и распечатку стека, где указано место возникновения исключения, а затем прекращает выполнение программы.

Типы исключений

В Java исключение является объектом. Тип этого объекта – `java.lang.Throwable` или (чаще всего) один из подклассов `Throwable`, более точно описывающий тип возникшего исключения.[1] `Throwable` содержит два стандартных подкласса: `java.lang.Error` и `ja-`

[1] До сих пор мы еще не говорили о подклассах; они подробно описаны в главе 3.

va.lang.Exception. Исключения подкласса Error обычно указывают на неустранимые проблемы: например, на виртуальной машине закончилась память или файл класса разрушен и не доступен для чтения. Перехватывать и исправлять исключения такого типа можно, но так поступают очень редко. С другой стороны, исключения подкласса Exception возникают в менее серьезных ситуациях. Такие исключения можно ловить и обрабатывать. Например, java.io.EOFException, сигнализирующее о конце файла, и java.lang.ArrayIndexOutOfBoundsException, указывающее на попытку программы прочитать данные за пределами массива. В этой книге понятие «исключение» определяет любой объект исключения, независимо от типа исключения (Exception или Error).

Поскольку исключение является объектом, оно содержит данные, а его класс определяет методы работы с этими данными. Класс Throwable и все его подклассы включают поле типа String, которое сохраняет удобочитаемое сообщение об ошибке с описанием исключительной ситуации. Оно инициализируется при создании объекта исключения и доступно для чтения посредством метода getMessage(). Большинство исключений содержат только это сообщение, но в некоторые из них добавлены дополнительные данные. Например, в java.io.InterruptedIOException добавлено поле bytesTransferred, указывающее количество введенных и выданных байтов на момент возникновения исключительной ситуации.

Объявление исключений

Схема обработки исключительных ситуаций в Java различает не только классы Error и Exception, но также проверяемпроверяемые и непроверяемые исключения. Любой объект исключения с типом Error является непроверяемым. Любой объект исключения Exception проверяемый, если только это не подкласс java.lang.RuntimeException, иначе он непроверяемый (RuntimeException является подклассом Exception). Это объясняется тем, что фактически каждый метод может сгенерировать непроверяемое исключение в любой момент. Например, невозможно предугадать OutOfMemoryError. Кроме того, любой метод, использующий объекты или массивы, может сгенерировать NullPointerException, если передать ему аргумент null. С другой стороны, проверяемые исключения возникают только в особых, вполне определенных ситуациях. Например, прежде чем прочитать данные из файла, нужно по крайней мере предусмотреть возможность генерации исключения FileNotFoundException, если указанный файл не найден.

Правила работы с проверяемым исключениями в Java отличаются от правил работы с непроверяемыми. При написании метода, генерирующего проверяемое исключение, нужно объявить исключение в сигнатуре метода посредством оператора throws. Такой тип исключений является проверяемым, поскольку компилятор Java проверяет их наличие в сигнатурах метода и выводит ошибку компиляции при их отсутствии. Метод factorial() генерирует исключение java.lang.IllegalArgumentException. Это подкласс исключения RuntimeException, то есть непроверяемое исключение, которое не нужно объявлять в секции throws (хотя и можно, для обеспечения ясности).

Даже если вы никогда не генерируете исключение, бывают случаи, когда вы обязаны объявить исключение в секции throws. Если используемый вами метод вызывает метод, генерирующий проверяемое исключение, нужно либо ввести код обработки для этого исключения, либо объявить при помощи throws, что ваш метод тоже может генерировать это исключение.

Каким образом можно узнать, что вызываемый метод может генерировать проверяемое исключение? Можно просмотреть сигнатуру этого метода. Если это невозможно, то компилятор Java известит (оповещением об ошибке компиляции) о том, что вы должны обработать или объявить исключения вызванного метода. Следующий метод

считывает первую строку текста из названного файла. Он использует методы, которые могут генерировать различные типы объектов java.io.IOException, поэтому исключение объявлено в секции throws:

```
public static String readFirstLine(String filename) throws IOException {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  return in.readLine();
}
```

Более подробно объявления и сигнатуры методов обсуждаются далее в этой главе.

Оператор try/catch/finally

Оператор try/catch/finally представляет собой механизм обработки исключений, применяемый в Java. Секция try данного оператора определяет блок программы, в котором будут обрабатываться исключения. После данного блока try следует несколько операторов catch, каждый из которых является блоком операторов обработки определенного типа исключений (операторов catch может не быть). После операторов catch может стоять блок finally, содержащий код очистки, который обязательно будет выполнен вне зависимости от того, что произойдет в блоке try. Операторы catch и finally не обязательны, но каждый блок try должен содержать хотя бы один из них. Все блоки try, catch и finally должны обрамляться фигурными скобками. Они являются обязательной частью синтаксиса и не могут быть опущены, даже если данная секция содержит только один оператор.

На примере следующего кода показан синтаксис и назначение оператора try/catch/finally:

```
try {
  // Обычно данный код выполняется сверху вниз без каких-либо осложнений.
  // Но иногда он может сгенерировать исключение - либо непосредственно
  // с помощью оператора throw, либо косвенно, путем вызова метода, генерирующего исключение.
}
catch (SomeException e1) {
  // Данный блок содержит операторы обработки объекта исключения типа SomeException
  // либо подкласса этого типа. Операторы данного блока могут обращаться к этому
  // объекту исключения по имени e1.
}
catch (AnotherException e2) {
  // Данный блок содержит операторы, обрабатывающие объект исключения типа AnotherException
  // либо подкласса этого типа. Операторы данного блока могут обращаться к этому объекту
  // исключения по имени e2.
}
finally {
  // Данный блок содержит операторы, которые всегда выполняются
  // после выхода из оператора try независимо от того, выходим мы из него сами или нет:
  // 1) обычно по достижении конца блока;
  // 2) из-за оператора break, continue или return;
  // 3) при наличии исключения, обрабатываемого вышестоящим оператором
  // catch, или
  // 4) при наличии не перехваченного и не обработанного исключения.
  // Однако если оператор try вызывает System.exit(),
  // интерпретатор выходит до выполнения оператора finally.
}
```

try

Секция try просто определяет блок программы, который либо содержит обрабатываемые исключения, либо нуждается в выполнении особого кода очистки при его завершении по любой причине. Сам по себе оператор try не осуществляет никаких операций; обработку исключений и операции по очистке выполняют операторы catch и finally.

catch

После блока try может следовать несколько операторов catch (их может и не быть), определяющих код обработки различных типов исключений. В объявлении каждой секции catch присутствует единственный аргумент, который определяет тип обрабатываемых этой секцией исключений, а также предоставляет оператору имя для обращения к объекту исключения, обрабатываемого в данный момент. Способ записи типа и имени исключения, обрабатываемого секцией catch, в точности соответствуют способу записи типа и имени аргумента метода, за исключением того, что в операторе catch аргументом должен быть объект типа Throwable или одного из его подклассов.

После генерации исключения интерпретатор Java ищет секцию catch с аргументом того же типа, что и объект исключения или родительский класс данного типа. Интерпретатор выполняет первую найденную им секцию catch. Код внутри блока catch должен предпринять все необходимые действия для ликвидации последствий исключительной ситуации. Например, если это исключение типа java.io.FileNotFoundException, можно обработать его, попросив пользователя проверить орфографию написанного кода и попробовать еще раз. Не обязательно вводить оператор catch для каждого возможного исключения; иногда правильнее позволить исключению распространиться выше и быть перехваченным вызывающим методом. В других случаях, таких как ошибка программиста, о которой сигнализирует NullPointerException, лучше совсем не перехватывать исключение, а позволить ему распространяться – тогда интерпретатор Java завершит выполнение кода с трассировкой стека и сообщением об ошибке.

finally

Секция finally обычно используется для освобождения ресурсов, использованных в секции try (например, чтобы закрыть файл или сетевые соединения). Важно отметить полезную особенность этого оператора: при выполнении любой части блока try оператор finally обязательно выполняется, независимо от результата выполнения кода в блоке try. В действительности без выполнения оператора finally оператор try можно выполнить только путем вызова метода System.exit(), который останавливает работу интерпретатора Java.

При обычных условиях поток выполнения достигает конца блока try, а затем переходит к блоку finally, который выполняет всю необходимую очистку. Если управление выходит из блока вследствие выполнения оператора return, continue или break, блок finally выполняется перед передачей управления в новую точку назначения.

Если исключение возникает в блоке try и существует связанный блок catch обработки исключения, управление сначала переходит к блоку catch, а затем к блоку finally. Если не существует локального блока catch для обработки исключения, управление сначала переходит к блоку finally, а затем распространяется до ближайшего охватывающего оператора catch, способного обработать исключение.

Если сам блок finally передает управление посредством оператора return, continue, break, throw или путем вызова метода, генерирующего исключение, текущая передача управления остается незавершенной, и выполняется новая передача. Например,

если в секции finally генерируется исключение, оно замещает текущее исключение, находящееся в процессе обработки. Если оператор finally выполняет оператор return, то происходит обычный возврат из метода, даже если было сгенерировано исключение, которое еще не обработано.

Операторы try и finally можно использовать вместе, не прибегая к исключениям или каким-либо операторам catch. В данном случае блок finally является обычным кодом очистки, который обязательно выполняется вне зависимости от наличия операторов break, continue или return внутри оператора try.

При предыдущем обсуждении операторов for и continue мы убедились, что нельзя просто заменить цикл for на цикл while, поскольку поведение оператора continue в цикле for немного отличается от его поведения в цикле while. Оператор finally позволяет написать цикл while, обрабатывающий оператор continue по аналогии с циклом for. Рассмотрим обобщенный цикл for:

```
for( initialize ; test ; update )
    statement
```

Следующий цикл while ведет себя аналогично, даже если блок statement содержит оператор continue:

```
initialize;
while( test ) {
  try { statement }
  finally { update ; }
}
```

Однако следует заметить, что при помещении секции update в блок finally реакция данного цикла while на операторы break будет отлична от реакции цикла for.

Оператор assert

Оператор assert используется для документирования и проверки предположений (assumptions), сформулированных при разработке кода Java. Данный оператор появился в Java 1.4 и не может использоваться в предыдущих версиях языка. *Утверждение* (*assertion*) состоит из ключевого слова assert и следующего за ним булевого выражения, которое по преположению программиста всегда должно возвращать true. По умолчанию утверждения не активизированы, и оператор assert фактически бездействует. Можно активизировать утверждения в качестве средств отладки и тестирования, и тогда оператор assert начнет вычислять выражение. Если выражение действительно true, оператор assert ничего не предпринимает. Однако если результатом вычисления выражения является false, то утверждение неверно, и оператор assert генерирует исключение java.lang.AssertionError.

Оператор assert может содержать необязательное второе выражение, отделяемое от первого двоеточием. Если активизированы утверждения, а первое выражение равно false, то значение второго выражения интерпретируется как код ошибки или как сообщение об ошибке и передается конструктору AssertionError(). Полный синтаксис оператора выглядит следующим образом:

```
assert assertion ;
```

или:

```
assert assertion : errorcode ;
```

Утверждение должно быть булевым выражением. Обычно это означает, что оно содержит оператор сравнения или активизирует метод с булевым значением.

Компилирование утверждений

Поскольку оператор `assert` был добавлен в версию Java 1.4, и до Java 1.4 слово `assert` не было зарезервированным, введение этого оператора может привести к нарушению правильной работы кода, использующего идентификатор `assert`. Поэтому компилятор *javac* не распознает оператор `assert` по умолчанию. Код Java, использующий оператор `assert`, можно скомпилировать только посредством аргумента командной строки `-source 1.4`. Например:

```
javac -source 1.4 ClassWithAssertions.java
```

Если не указан аргумент `-source 1.4`, компилятор *javac* позволит использовать `assert` в качестве идентификатора. Если найден идентификатор `assert`, он выводит предупреждение о несовместимости. В последующих версиях опция командной строки больше не понадобится, а оператор `assert` может распознаваться по умолчанию, поэтому нужно прекратить использование кода, задействующего оператор `assert` в качестве идентификатора.

Активизация утверждений

Операторы `assert` должны использоваться только для предположений, которые всегда истинны. Нет смысла тестировать предположения при каждом выполнении кода, поэтому по умолчанию утверждения блокируются, а операторы `assert` не действуют. Однако код с утверждениями остается скомпилированным в файлах классов, и его всегда можно активизировать для тестирования, диагностики и отладки. При помощи аргументов командной строки, задаваемых для интерпретатора Java, можно активизировать либо все утверждения, либо выборочные. Аргумент `-ea` активизирует утверждения для всех классов, кроме системных. Чтобы активизировать утверждения для системных классов, нужно задать `-esa`. Чтобы активизировать утверждения для определенного класса, нужно ввести `-ea`, двоеточие и имя класса:

```
java -ea:com.example.sorters.MergeSort com.example.sorters.Test
```

Чтобы активизировать утверждения для всех классов пакета и всех его подпакетов, за аргументом `-ea` должно следовать двоеточие, имя пакета и три точки:

```
java -ea:com.example.sorters... com.example.sorters.Test
```

Аналогичным образом можно заблокировать утверждения при помощи аргумента `-da`. Например, чтобы активизировать утверждения пакета и затем заблокировать их в определенном классе или подпакете, следует ввести:

```
java -ea:com.example.sorters...
-da:com.example.sorters.QuickSort java
-ea:com.example.sorters...
-da:com.example.sorters.plugins...
```

Если вы предпочитаете подробные аргументы командной строки, вместо `-esa` можно указать `-enableassertions` и `-disableassertions`.

Версия Java 1.4 добавила в класс `java.lang.ClassLoader` методы активизации и блокировки утверждений для классов, загружаемых посредством этого `ClassLoader`. Вам могут пригодиться данные методы при включении утверждений в специализированном загрузчике классов. См. `ClassLoader` в главе 11.

Применение утверждений

Поскольку по умолчанию утверждения заблокированы и бездействуют, их можно свободно применять для документирования предположений. Такое применение оператора довольно непривычно, но со временем вы узнаете о других вариантах применения `assert`. Например, предположим, что вы уверены, будто в создаваемом методе переменная `x` будет равна либо `0`, либо `1`. Не применяя утверждения, вы можете написать такой оператор `if`:

```
if(x == 0) {
  ...
} else {    // x равен 1
  ...
}
```

Комментарием к этому коду является ваше неформальное утверждение, что в теле оператора `else` переменная `x` всегда будет равна `1`.

Теперь предположим, что ваш код был изменен, и `x` может принимать значения, отличные от **0** и **1**. Комментарий и предположение к этому коду больше недействительны. Это может стать причиной нераспознаваемой или трудно локализируемой ошибки. Решением данной проблемы будет преобразование комментария в оператор `assert`. Код принимает следующий вид:

```
if(x == 0) {
  ...
} else {
  assert x == 1 : x      // x должен равняться 0 или 1
  ...
}
```

Теперь, если `x` каким-либо образом получает непредвиденное значение, генерируется `AssertionError`. Это позволяет сразу распознать ошибку и легко выявить ее местонахождение. Более того, второе выражение в операторе `assert` (после двоеточия) содержит непредвиденное значение `x` – сообщение об ошибке в объекте `AssertionError`. Единственное предназначение данного сообщения – предоставить конечному пользователю информацию не только о провале утверждения, но и о причине его провала.

Похожим образом можно применять операторы `switch`. Если оператор `switch` не содержит оператор `default`, следует сделать предположение о множестве возможных значений для выражения `switch`. Если вы уверены, что другие значения невозможны, вы можете ввести в документ оператор `assert` для подтверждения данного факта. Например:

```
switch(x) {
  case -1: return LESS;
  case 0: return EQUALS;
  case 1: return GREATER;
  default: assert false:x; // Сгенерировать AssertionError, если x не равен -1, 0 или 1.
}
```

Заметьте, что оператор `assert false;` всегда приводит к ошибке. Этот «тупиковый» оператор пригоден в ситуации, когда вы уверены, что никогда не достигнете данного оператора.

Также оператор `assert` широко применяют для выяснения соответствия всех значений передаваемых методу аргументов, то есть для введения предусловий метода. Например:

```
private static Object[] subArray(Object[] a, int x, int y) {
  // Предусловие: x должен быть <= y
  assert x <= y : "subArray: x > y";
  // Теперь создаем и возвращаем подмассив a...
}
```

Заметьте, что данный метод является закрытым. Программист использовал оператор `assert` для документирования предусловия метода `subArray()` и для утверждения, что все методы, вызывающие данный закрытый метод, действительно выполняют это предусловие. Данное утверждение возможно, поскольку программист управляет всеми методами, использующими `subArray()`. Его также можно проверить путем активизации утверждений во время тестирования кода. Однако при тестировании кода с заблокированными утверждениями аргументы метода не тестируются при каждом его вызове. Заметьте также, что программист не использовал оператор `assert`, чтобы проверить, что `a` не равно нулю, а аргументы `x` и `y` являются допустимыми индексами данного массива. Java всегда проверяет эти неявные предусловия во время выполнения программы и в случае ошибки возбуждает непроверяемые исключения `NullPointerException` или `ArrayIndexOutOfBoundsException`, поэтому для них утверждение не требуется.

Обратите внимание, что оператор `assert` не подходит для введения предусловий в открытых методах. Открытый метод можно вызвать из любой точки программы. Программист не может заранее утверждать, что активизация метода прошла правильно. Грубо говоря, открытый API должен явно проверять аргументы и вводить предусловия при каждом его вызове, независимо от состояния утверждений.

Соответственно, оператор `assert` можно использовать для проверки инвариантности класса. Предположим, вы хотите создать класс, содержащий список объектов, которые можно вводить и удалять, но этот список всегда остается упорядоченным. Вы можете утвердить этот инвариант, написав метод тестирования упорядоченности списка, а затем при помощи оператора `assert` активизировать его в конце каждого метода модификации списка. Например:

```
public void insert(Object o) {
  ...                          // Здесь вводится объект
  assert isSorted();           // Здесь проверяется инвариантность класса
}
```

При написании кода, который должен быть «потокобезопасным» (thread-safe), вы должны получить блокировку (при помощи метода или оператора `synchronized`) в тех местах, где это необходимо. Обычно в данной ситуации оператор `assert` проверяет текущий поток на наличие необходимой блокировки:

```
assert Thread.holdsLock(data);
```

Метод `Thread.holdsLock()` был добавлен в Java 1.4 главным образом для использования с оператором `assert`.

Для эффективного применения утверждений следует избегать некоторых действий: во-первых, нужно помнить, что ваши программы будут выполняться как с активизированными, так и с заблокированными утверждениями. Это значит, что следует проследить, чтобы выражения утверждения не содержали побочных эффектов. Иначе поведение вашего кода будет зависеть от состояния утверждений (активизированы или заблокированы). Конечно, это правило имеет несколько исключений. Например, если метод содержит два оператора `assert`, то первый из них может содержать побочный эффект, который влияет только на второе утверждение. Еще один способ приме-

нения побочных эффектов в утверждениях – идиома, определяющая активизированность утверждения (что вряд ли соответствует назначению вашего кода):

```
boolean assertions = false; // Если утверждения включены
assert assertions = true;   // Данный оператор assert никогда не
                            // выдает отказ, но имеет побочный эффект.
```

Примечание: выражение в операторе `assert` является присваиванием, а не сравнением. Значение выражения присваивания всегда будет равно присваиваемому значению, поэтому это выражение всегда будет `true`, а утверждение никогда не выдает отказ. Поскольку выражение присваивания является частью оператора `assert`, переменная `assertions` принимает значение `true`, только если активизированы утверждения.

Второе правило при работе с оператором `assert` является дополнением к правилу избежания побочных эффектов: не следует перехватывать `AssertionError` (если только вы не делаете это на самом верхнем уровне, что позволит более наглядно отобразить ошибку). Сгенерированное исключение `AssertionError` указывает, что одно из предположений программиста не выполнилось. Другими словами, код используется не в том контексте, для которого был предназначен, и не следует ожидать, что он будет работать верно. Короче говоря, нет смысла продолжать работу после возникновения исключения `AssertionError` и не стоит пытаться его перехватывать.

Методы

Метод – это именованная последовательность операторов Java, вызываемая другим кодом Java. При вызове метода ему передается несколько значений, известных как аргументы (их может и не быть). Метод выполняет некоторые вычисления и возвращает значение (если это необходимо). Вызов метода является выражением, вычисляемым интерпретатором Java. Однако поскольку вызов метода может приводить к побочным эффектам, его также можно применять в качестве оператора-выражения.

Вы уже знаете, как определить тело метода; это произвольная последовательность операторов, заключенная в фигурные скобки. Больший интерес представляет сигнатура метода. Сигнатура определяет:[1]

- Имя метода
- Количество, порядок, тип и имена параметров, используемых методом
- Тип значения, возвращаемого методом
- Проверяемые исключения, которые может генерировать метод (в сигнатуре могут также перечисляться непроверяемые исключения, хотя это необязательно)
- Различные модификаторы метода, предоставляющие дополнительную информацию о методе

В сигнатуре метода определена вся информация, необходимая для вызова метода. Она называется *спецификацией* (*specification*) метода и определяет его API. Справочный раздел данной книги в основном представляет собой список сигнатур для всех открытых методов и классов платформы Java. Чтобы обращаться к справочному разделу данной книги, вы должны знать правила чтения сигнатуры метода. А чтобы пи-

[1] В спецификации языка Java понятие «сигнатура» имеет техническое значение, которое немного отличается от используемого в этой книге. Здесь применяется менее официальное определение сигнатуры метода.

сать программы на Java, вы должны уметь определять собственные методы, каждый из которых начинается с сигнатуры метода.

Сигнатура метода выглядит следующим образом:

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

За сигнатурой (спецификацией метода) следует тело метода (реализация метода), то есть последовательность операторов Java, заключенных в фигурные скобки. В некоторых случаях (описанных в главе 3) реализация опускается, а тело метода заменяется точкой с запятой.

Ниже представлено несколько примеров определений методов. Тело метода в них опущено, что позволяет сосредоточить все внимание на сигнатурах:

```
public static void main(String[] args) { ... }
public final synchronized int indexOf(Object element, int startIndex) { ... }
double distanceFromOrigin() { ... }
static double squareRoot(double x) throws IllegalArgumentException { ... }
protected abstract String readText(File f, String encoding)
        throws FileNotFoundException, UnsupportedEncodingException;
```

Модификатором (*modifiers*) называется несколько специальных модифицирующих ключевых слов, отделяемых друг от друга пробелом (ключевых слов может и не быть). Например, метод можно объявить с модификаторами `public` и `static`. Другими разрешенными модификаторами являются `abstract`, `final`, `native`, `private`, `protected` и `synchronized`. Пока значения этих модификаторов не важны; они описаны в главе 3.

Тип (*type*) в сигнатуре метода определяет возвращаемый тип метода. Если метод возвращает значение, то в *type* записывается имя примитивного типа, массива или класса. Если метод не возвращает значение, тип должен быть `void`. *Конструктором* (*constructor*) является особый вид метода для инициализации новых созданных объектов. В главе 3 вы увидите, что конструкторы определяются так же, как и методы, но их сигнатуры не содержат описанной выше спецификации типа.

Имя метода следует после спецификации его модификаторов и типа. Имена методов, так же как и имена переменных, являются идентификаторами Java. Как и все идентификаторы Java, они могут содержать буквы любого языка, представленного в наборе Unicode. Разрешается (а иногда и полезно) определять более одного метода с одинаковым именем, поскольку каждая версия метода содержит отличный от других список параметров. Определение нескольких методов с одинаковым именем называется *перегрузкой методов* (*method overloading*). Примером перегруженного метода является уже многократно встречавшийся нам метод `System.out.println()`. Один метод с данным именем отображает строку, а другие методы с таким же именем выдают значения различных примитивных типов. На основании типа аргумента, передаваемого методу, компилятор Java определяет метод для вызова.

При определении метода за именем метода всегда следует список его параметров, заключенный в скобки. Этот список параметров определяет аргументы, передаваемые методу (их может и не быть). Каждая спецификация параметра (если параметры есть) содержит тип и имя. При наличии нескольких параметров спецификации отделяются друг от друга запятой. При вызове метода передаваемые ему значения аргументов должны соответствовать количеству, типу и порядку параметров, определенных в строке сигнатуры данного метода. Передаваемые значения не обязательно должны точно соответствовать типу, определенному в сигнатуре, но должны преобразовываться в такой тип без использования приведения. Программистам, пишущим на C и C++, следует обратить внимание на следующую особенность: если метод

Java не принимает никаких аргументов, его список параметров записывается в виде(), а не (void).

Оператор throws, рассмотренный ранее, завершает сигнатуру метода. Если метод применяет оператор throw для генерации проверяемого исключения или если он вызывает какой-либо другой метод, генерирующий проверяемое исключение, и не перехватывает или не обрабатывает данное исключение, то метод должен объявить, что он может сгенерировать это исключение. Если метод может генерировать одно или несколько проверяемых исключений, то после списка аргументов ставится ключевое слово throws, за которым следует имя класса или классов генерируемых им исключений. Если метод не генерирует никаких исключений, ключевое слово throws не указывается. Если метод генерирует исключения нескольких типов, имена классов исключений должны отделяться друг от друга запятыми.

Классы и объекты

Мы уже рассмотрели операторы, выражения и методы, и теперь, наконец, можно обсудить классы. *Класс* (*class*) – это именованная совокупность полей, содержащих значения данных и методы для работы с этими значениями. Некоторые классы также содержат вложенные внутренние классы. Классы – это основные структурные элементы всех программ Java. Не определив класс, нельзя написать Java-код. Все операторы Java находятся внутри методов, а все методы определены внутри классов.

Классы не просто являются одним из уровней структуры синтаксиса Java. Подобно тому как клетка является наименьшей природной единицей, способной жить и размножаться, так и класс представляет собой наименьшую самостоятельную единицу Java-кода. Компилятор и интерпретатор Java не распознают фрагменты кода, меньшие чем класс. Класс является основной единицей выполнения в Java, что придает ему особое значение. В действительности в Java определен еще один структурный компонент под названием *интерфейс* (*interface*), который мало чем отличается от класса. Разницу между классами и интерфейсами мы рассмотрим в главе 3, а пока будем использовать понятие «класс» для обозначения как класса, так и интерфейса.

Классы важны еще по одной причине: каждый новый класс определяет новый тип данных. Например, можно определить класс с именем Point, представляющий точку в декартовой двухмерной системе координат. Этот класс может определять поля (типа double), содержащие координаты точки X и Y и методы манипулирования точкой. Класс Point является новым типом данных.

При обсуждении типов данных важно различать сам тип данных и значения, которые он представляет. char является типом данных: он представляет символы Unicode. А значение char представляет собой отдельный символ. Класс является типом данных; значение класса называется *объектом* (*object*). Мы используем имя класса, поскольку каждый класс определяет тип (или сорт, или вид, или класс) объектов. Класс Point является типом данных, представляющим точки X,Y, в то время как объект Point представляет одну конкретную точку X,Y. Как вы, наверное, догадались, классы и их объекты тесно связаны между собой. В следующих разделах мы рассмотрим и те и другие.

Определение класса

Ниже представлено возможное определение рассматриваемого класса Point:

```
/**
  Представляет точку в декартовом пространстве (x,y)
*/
public class Point {
  public double x, y;                          // Координаты точки
  public Point(double x, double y) {           // Конструктор, в котором
    this.x = x; this.y = y;                    // инициализируются поля
  }
  public double distanceFromOrigin() {         // Метод, который оперирует
    return Math.sqrt(x*x + y*y);               // полями x и y
  }
}
```

Данное определение класса сохраняется в файле *Point.java* и компилируется в файл *Point.class*. После этого оно становится доступно для использования Java-программами и другими классами. Определение класса приведено здесь для полноты представления, поэтому не ждите, что вы поймете все детали прямо сейчас; определению классов посвящена бо́льшая часть главы 3. И все же стоит обратить особое внимание на первую (незакомментированную) строку определения класса. Так же как первая строка определения метода (его сигнатуры) определяет API метода, так и данная строка определяет базовый API класса (более подробно эта тема рассматривается в следующей главе).

Помните, что вам не обязательно определять каждый класс, который вы применяете в Java-программе. Платформа Java состоит из более чем 1500 предопределенных классов, которые доступны на каждом компьютере, где запущена Java.

Создание объекта

Теперь, когда класс `Point` определен как новый тип данных, мы можем объявить переменную, содержащую объект `Point`, с помощью следующей строки:

```
Point p;
```

Однако при объявлении переменной для хранения объекта `Point` сам объект не создается. Чтобы создать объект, нужно применить оператор `new`. За этим ключевым словом следует указать класс объекта (то есть его тип) и необязательный список аргументов, обрамленный круглыми скобками. Эти аргументы передаются конструктору класса, который инициализирует внутренние поля нового объекта:

```
// Создать объект Point с координатами (2,-3.5) и сохранить его в переменной p
Point p = new Point(2.0, -3.5);
// Создать также несколько других объектов
Date d = new Date(); // Объект Date, представляющий текущее время
Vector list = new Vector(); // Объект Vector для хранения списка объектов
```

Ключевое слово `new` является наиболее типичным способом создания объектов. Все же следует упомянуть еще несколько способов. Во-первых, существует несколько важных классов, для которых в Java специально определен точный синтаксис создания объектов этих типов (мы рассмотрим их более подробно в следующем разделе). Во-вторых, Java поддерживает механизм динамической загрузки, что позволяет программам загружать классы и создавать экземпляры этих классов во время выполнения. Динамическое создание экземпляров во время выполнения реализуется при помощи методов `newInstance()` класса `java.lang.Class` и конструктора `java.lang.reflect.Constructor`. Наконец, начиная с версии Java 1.1 и далее объекты можно создавать путем десериализации. Другими словами, объект, состояние которого было со-

хранено, или сериализовано (как правило, в файл), можно восстановить с помощью класса `java.io.ObjectInputStream`.

Объекты-литералы

Как уже говорилось, в Java определен специальный синтаксис создания экземпляров двух наиболее важных классов. Первый класс – `String` – представляет текст в виде строки символов. Поскольку общение пользователя с программой происходит в основном в письменной форме, возможность работы с текстовыми строками достаточно важна для любого языка программирования. В некоторых языках строки являются таким же примитивным типом, как целые числа и символы. Однако в Java строки являются объектами, а класс `String` – это тип данных для представления текста.

Поскольку строки являются одним из основных типов данных, Java позволяет включать текст в программу в виде литералов, заключенных в двойные кавычки ("). Например:

```
String name = "David";
System.out.println("Hello, " + name);
```

Не следует путать двойные кавычки, обрамляющие строки-литералы, с одинарными кавычками (или апострофами), которые обрамляют `char`-литералы. Строки-литералы могут содержать любую из управляющих последовательностей, разрешенных в `char`-литералах (см. табл. 2.3). Управляющие последовательности можно успешно применять для вставки символов двойных кавычек в строки-литералы, которые сами заключены в двойные кавычки. Например:

```
String story = "\t\"Как ты можешь противиться этому?\",- спросил он.\n";
```

Строки-литералы не содержат комментарии; кроме того, они могут состоять только из одной строки. Java не поддерживает синтаксис слияния символов, который позволял бы интерпретировать две отдельные строки как одну. Если вам нужно представить длинную строку текста, которая не умещается в одной строке, разбейте ее на отдельные строки-литералы и примените оператор + для слияния литералов. Например:

```
String s = "Это тест аварийной           // Это неправильно; строковые
            системы оповещения";         // литералы не должны разрываться таким образом.

String s = "Это тест аварийной " +       // Делайте так, как показано здесь
           "системы оповещения";
```

Слияние литералов происходит в ходе компилирования программы, а не во время ее выполнения, поэтому вам не стоит беспокоиться об ухудшении производительности.

Вторым классом, поддерживающим собственный специальный синтаксис объекта-литерала, является класс с именем `Class`. `Class` – это самоотносимый (self-referential) тип данных, который представляет все типы данных Java. Помимо типов классов сюда еще входят примитивные типы и массивы. Чтобы непосредственно включить объект `Class` в Java-программу, нужно после имени любого типа данных ввести `.class`. Например:

```
Class typeInt = int.class;
Class typeIntArray = int[].class;
Class typePoint = Point.class;
```

Эта возможность поддерживается начиная с версии Java 1.1.

Зарезервированное слово `null` является специальным литералом, совместимым с любым классом. Он представляет не литеральный объект, а его отсутствие. Например:

```
String s = null;
Point p = null;
```

Наконец, объекты можно непосредственно включать в программу Java с помощью конструкции, известной как анонимный внутренний класс. Более подробно об анонимных классах рассказано в главе 3.

Применение объекта

Поскольку мы уже рассмотрели определение и создание экземпляров классов путем создания объектов, мы можем перейти к синтаксису Java, позволяющему использовать эти объекты. Вспомним, что класс определяет совокупность полей и методов. Каждый объект содержит собственные копии этих полей и имеет доступ к этим методам. При обращении к именованным полям и методам объекта следует применять символ точки (.). Например:

```
Point p = new Point(2, 3);            // Создать объект
double x = p.x;                       // Прочитать значение поля объекта
p.y = p.x * p.x;                      // Установить значение поля объекта
double d = p.distanceFromOrigin();    // Получить доступ к методу
```

Это основной синтаксис объектно-ориентированного программирования в Java; он будет довольно часто встречаться и далее. Обратите особое внимание на выражение `p.distanceFromOrigin()`. В нем компилятору Java дается указание отыскать метод `distanceFromOrigin()`, определенный классом `Point`, и применить этот метод для выполнения вычислений над полями объекта `p`. Более детально эту операцию мы рассмотрим в главе 3.

Массивы

Массивы представляют собой второй вид ссылочных типов в Java.[1] Массив – это упорядоченная совокупность, или пронумерованный список, значений. Это могут быть как примитивные значения, так и объекты или даже другие массивы, однако все значения массива должны принадлежать одному типу. Тип массива идентичен типу содержащихся в нем значений. После типа массива ставятся символы []. Например:

```
byte b;                          // byte – это примитивный тип
byte[] arrayOfBytes;             // byte[] – это массив байтов
byte[][] arrayOfArrayOfBytes;    // byte[][] – другой тип массива байтов
Point[] points;                  // Point[] – это массив объектов Point
```

Java поддерживает еще один синтаксис объявления переменных типа массив, обеспечивающий совместимость с C и C++. Согласно этому синтаксису, одна или несколько пар квадратных скобок следуют за именем переменной, а не за именем типа:

```
byte arrayOfBytes[];             // То же, что и byte[] arrayOfBytes
byte arrayOfArrayOfBytes[][];    // То же, что и byte[][] arrayOfArrayOfBytes
byte[] arrayOfArrayOfBytes[];    // Уф! То же, что и byte[][] arrayOfArrayOfBytes
```

[1] В действительности массивы являются объектами Java, но у них более специализированный синтаксис и поведение, что позволяет рассматривать их отдельно.

Однако зачастую такой синтаксис сбивает с толку, поэтому его следует избегать.

Говоря о классах и объектах, следует отметить отдельные понятия для типа и значений этого типа. В массивах одно понятие «массив» выполняет двойную функцию – имени типа и имени значения. Таким образом, можно рассматривать как тип массива `int[]`, так и сам массив `int` (конкретное значение массива). На практике можно понять по контексту, какое из двух понятий имеется в виду.

Создание массивов

Как и при создании объекта, для создания массива в Java применяют ключевое слово `new`. Однако в отличие от объектов массивы не нужно инициализировать, то есть не нужно передавать список аргументов в круглых скобках. Тем не менее надо определить необходимый размер массива. Например, при создании `byte[]` следует установить количество значений `byte`, которое массив должен содержать. В Java размер массива фиксирован. Созданный массив нельзя увеличить или уменьшить. Желаемый размер создаваемого массива задается неотрицательным целым числом в квадратных скобках.

```
byte[] buffer = new byte[1024];
String[] lines = new String[50];
```

При создании массива с таким синтаксисом все элементы массива автоматически инициализируются значениями по умолчанию. Это `false` для значений `boolean`, `'\u0000'` для значений `char`, `0` для целых значений, `0.0` для значений с плавающей точкой и `null` для объектов или массивов.

Применение массивов

Чтобы получить доступ к отдельным значениям массива, созданного при помощи оператора `new` и синтаксиса с квадратными скобками, нужно снова применить квадратные скобки. Вспомним, что массив является упорядоченной совокупностью значений. Элементы массива пронумерованы последовательно, начиная с 0. Номер элемента массива относится к этому элементу. Номер часто называют *индексом*, а процесс поиска значения массива по номеру иногда называют *индексированием массива*.

Для ссылки на определенный элемент массива нужно после имени массива ввести индекс выбранного элемента в квадратных скобках. Например:

```
String[] responses = new String[2];    // Создать массив из двух строк
responses[0] = "Yes";                  // Установить первый элемент массива
responses[1] = "No";                   // Установить второй элемент массива
// Теперь прочесть эти элементы массива
System.out.println(question + " (" + responses[0] + "/" + responses[1] + " ): ");
```

Зачастую в программах, написанных на C и C++, встречается ошибочный код, пытающийся прочитать или записать элементы массива, номера которых выходят за пределы самого массива. Java этого не позволяет. Каждый раз, когда вы получаете доступ к элементу массива, интерпретатор Java автоматически проверяет допустимость указанного вами индекса. Если указан отрицательный индекс или индекс, больший последнего индекса массива, интерпретатор генерирует исключение типа `ArrayIndexOutOfBoundsException`. Это не позволяет обращаться к несуществующим элементам массива.

Значения индекса массива представлены целыми числами; нельзя индексировать массив значением с плавающей точкой, `boolean`, объектом или другим массивом. Поскольку значения `char` можно преобразовать в значения `int`, для индексирования

массива можно задействовать символы. Хотя long и является целым типом данных, для индексирования значения long применять нельзя. Поначалу это может показаться странным, но если подумать, индекс int поддерживает массивы с более чем двумя миллиардами элементов. Для int[] с таким количеством элементов необходимо 8 Гбайт памяти. Неудивительно, что значения long недопустимы для индексирования массивов.

Кроме установки и чтения элементов массива существует еще одно действие, которое можно выполнять с массивами. Вспомним, что при создании каждого массива нужно указывать количество содержащихся в нем элементов. Данное значение указывает на длину массива; это внутреннее свойство массива. Чтобы узнать длину массива, нужно к имени массива добавить .length:

```
if(errorCode < errorMessages.length)
  System.out.println(errorMessages[errorCode]);
```

Каждый массив содержит поле length, которое определяет количество входящих в него элементов. Обратите внимание на то, что данное поле предназначено только для чтения: из него можно прочитать длину массива, но ему нельзя присвоить никакого значения, а также устанавливать или изменять с его помощью длину массива.

В предыдущем примере индексом массива, заключенным в квадратные скобки, была переменная, а не целый литерал. На самом деле чаще всего массивы применяются вместе с циклами, а именно с циклами for, где они индексируются путем положительного или отрицательного приращения переменной при каждом прохождении цикла:

```
int[] values;      // Допустим, создан массив, который инициализирован в другом месте
int total = 0;     // Сохранить некоторые элементы здесь
for(int i = 0; i < values.length; i++)      // Пройти цикл по элементам массива
  total += values[i];                       // Сложить их
```

В Java первый элемент массива всегда идет под номером 0. Поначалу это вызывает определенные трудности у разработчиков, привыкших к языку программирования, в котором нумерация начинается с 1. Для массива a первым элементом будет a[0], вторым – a[1]. Последний элемент:

```
a[a.length - 1]    // Последний элемент массива a
```

Массивы-литералы

Литерал null, обычно указывающий на отсутствие объекта, можно также использовать для обозначения отсутствия массива. Например:

```
char[] password = null;
```

Кроме литерала null в Java также определен специальный синтаксис, позволяющий непосредственно указывать элементы массива в программе. На самом деле существует два отдельных синтаксиса массивов-литералов. Первый синтаксис можно применять только при объявлении переменной типа массив. Это наиболее типичная синтаксическая структура. С ее помощью создается объект-массив и инициализируются его элементы:

```
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

Эта строка позволяет создать массив, который содержит восемь элементов int, заключенных в фигурные скобки. Обратите внимание, что в этом синтаксисе массива-литерала не нужно применять ключевое слово new и устанавливать тип массива. Тип

неявно присутствует в объявлении переменной, частью которого является инициализатор. Кроме того, в данном синтаксисе не указывается длина массива; она определяется неявно, при подсчете количества элементов, заключенных в фигурные скобки. В данном массиве-литерале следом за закрывающей фигурной скобкой ставится точка с запятой. Это один из тонких моментов синтаксиса Java. Если фигурные скобки разграничивают классы, методы и составные операторы, то точка с запятой после них не ставится. Однако в данном синтаксисе массива-литерала точка с запятой завершает оператор объявления переменной.

Единственное неудобство этого синтаксиса массива-литерала состоит в том, что он функционирует только в случае объявления переменной типа массив. Иногда массив нужно задействовать лишь один раз (например, передать его методу), следовательно, вы не хотите тратить время на присваивание его переменной. В Java 1.1 и последующих версиях предусмотрен синтаксис массива-литерала, который поддерживает анонимные массивы (они не присваиваются переменным и, следовательно, у них нет имен). Такой массив-литерал выглядит следующим образом:

```
// Вызвать метод, передавая анонимный массив с двумя строками
String response = askQuestion("Do you want to quit?",
                              new String[] {"Yes", "No"});

// Вызвать другой метод с анонимным массивом (анонимным объектом)
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
                                               new Point(3,4),
                                               new Point(3,2) });
```

В этом синтаксисе необходимо использовать ключевое слово `new` и определить тип массива, зато не нужно явно указывать его длину.

Важно понимать, что архитектура виртуальной машины Java не поддерживает эффективную инициализацию массива. Другими словами, массивы-литералы создаются и инициализируются во время выполнения программы, а не во время ее компиляции. Рассмотрим следующий массив-литерал:

```
int[] perfectNumbers = {6, 28};
```

Он компилируется в такой байт-код Java:

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

Поэтому если вам нужно разместить в Java-программе много данных, лучше не включать их непосредственно в массив, поскольку компилятору Java придется создавать много байт-кодов инициализации массива, а затем интерпретатору Java нужно будет кропотливо выполнять весь этот код. В таких случаях лучше сохранять данные во внешнем файле и считывать их в программу во время ее выполнения.

Однако тот факт, что Java инициализирует массив во время выполнения программы, имеет важные последствия. Это означает, что элементы массива-литерала являются произвольными выражениями, вычисляемыми во время выполнения программы, а не постоянными выражениями, вычисляемыми компилятором. Например:

```
Point[] points = { circle1.getCenterPoint(), circle2.getCenterPoint() };
```

Многомерные массивы

Как вы уже знаете, тип массива – это тип элементов, сопровождаемый квадратными скобками. Массив `char` обозначается `char[]`, а массив массивов `char` – `char[][]`. Массив называется *многомерным*, если элементы массива сами являются массивами. Прежде чем приступить к работе с многомерными массивами, нужно понять несколько дополнительных деталей.

Допустим, вы хотите представить таблицу умножения в виде многомерного массива:

```
int[][] products;        // Таблица умножения
```

Каждая пара квадратных скобок представляет одно измерение, поэтому данный массив является двухмерным. Чтобы получить доступ к одиночному элементу `int` двухмерного массива, нужно указать два значения индекса, по одному для каждого измерения. Если допустить, что данный массив был задан как таблица умножения, то значение `int`, находящееся в любом элементе, есть произведение этих индексов. Таким образом, значение `products[2][4]` равно 8, а значением `products[3][7]` будет 21.

Новый многомерный массив создается при помощи ключевого слова `new` с указанием размеров обоих измерений массива. Например:

```
int[][] products = new int[10][10];
```

В некоторых языках такой массив создается в виде единого блока из 100 значений `int`. Java поступает иначе. Эта строка кода выполняет три действия:

- Объявляет переменную с именем `products`, которая содержит массив массивов `int`.
- Создает массив из 10 элементов, который должен содержать 10 массивов `int`.
- Создает еще 10 массивов, каждый из которых в свою очередь является массивом `int` из 10 элементов. Присваивает каждый из этих 10 новых массивов элементам исходного массива. По умолчанию каждый элемент `int` каждого из этих 10 новых массивов получает значение 0.

Другими словами, представленная строка кода теперь имеет такой вид:

```
int[][] products = new int[10][];    // Массив, содержащий 10 значений int[]
for(int i = 0; i < 10; i++)          // Пройти цикл 10 раз...
  products[i] = new int[10];         // ...и создать 10 массивов
```

Ключевое слово `new` автоматически выполняет дополнительную инициализацию. Его также можно использовать и с массивами, размерность которых больше двух:

```
float[][][] globalTemperatureData = new float[360][180][100];
```

Если вы применяете оператор `new` в многомерных массивах, вам необязательно указывать все измерения массива – важно задать только крайнее слева измерение или измерения. Например, разрешены такие строки:

```
float[][][] globalTemperatureData = new float[360][][];
float[][][] globalTemperatureData = new float[360][180][];
```

Первая строка создает одномерный массив, где каждый элемент массива может содержать `float[][]`. Вторая строка создает двухмерный массив, где каждый элемент массива является `float[]`. Однако если вы определяете размер только некоторых измерений массива, то они должны быть крайними слева. Следующие строки недопустимы:

```
float[][][] globalTemperatureData = new float[360][][100]; // Ошибка!
float[][][] globalTemperatureData = new float[][180][100]; // Ошибка!
```

Подобно одномерным массивам, многомерные массивы можно инициализировать при помощи массива-литерала. Нужно просто вложить массивы в другие массивы, применяя множество вложенных фигурных скобок. Например, можно объявить, создать и инициализировать таблицу умножения 5*5:

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

Или можно применить синтаксис анонимного инициализатора, если вы не хотите объявлять переменную при использовании многомерного массива:

```
boolean response =
  bilingualQuestion(question, new String[][] {
                                              { "Yes", "No" },
                                              { "Oui", "Non" } } );
```

Создавая многомерный массив посредством ключевого слова new, вы всегда будете получать *прямоугольный* (*rectangular*) массив: все массивы данного измерения будут одного размера. Подобный массив идеален для таких прямоугольных структур данных, как матрица. Однако вы можете использовать не только прямоугольные массивы, поскольку в Java многомерные массивы реализуются как массивы массивов, а не как отдельный прямоугольный блок элементов. Например, поскольку наша таблица умножения симметрична по диагонали от верхнего левого угла до нижнего правого, мы можем представить ту же информацию в непрямоугольном массиве с меньшим количеством элементов:

```
int[][] products = { {0},
                     {0, 1},
                     {0, 2, 4},
                     {0, 3, 6, 9},
                     {0, 4, 8, 12, 16} };
```

Для создания и инициализации многомерных массивов вам часто придется использовать вложенные циклы. Например, можно создать и инициализировать большую треугольную таблицу умножения таким образом:

```
int[][] products = new int[12][];          // Массив 12 массивов int.
for(int row = 0; row < 12; row++) {        // Для каждого элемента массива
  products[row] = new int[row+1];          // назначить массив int.
  for(int col = 0; col < row+1; col++)     // Каждый элемент  int[]
    products[row][col] = row * col;        // инициализировать с помощью операции произведения.
}
```

Ссылочные типы

Мы рассмотрели синтаксис работы с объектами и массивами. Теперь можно вернуться к вопросу о том, почему массивы и классы называются ссылочными типами. Из табл. 2.2 следует, что стандартные размеры каждого примитивного типа Java четко определены, поэтому все примитивные значения можно сохранять в фиксированном объеме памяти (один – восемь байт, в зависимости от типа). Однако массивы и классы являются составными типами; объекты и массивы содержат другие значения, поэтому они не имеют стандартного размера, а также зачастую требуют намного боль-

ше восьми байт памяти. По этой причине Java не управляет напрямую объектами и массивами, а работает со *ссылками* (*references*) на них. Поскольку Java обрабатывает объекты и массивы через ссылки, то классы и массивы являются ссылочными типами. В отличие от них значения примитивных типов обрабатываются напрямую или по значению.

Ссылкой на объект или массив называется значение фиксированного размера, которое каким-либо образом связано с объектом или массивом.[1] Присваивая переменной объект или массив, вы на самом деле устанавливаете в ней ссылку на этот объект или массив. Аналогично, передавая методу объект или массив, в действительности вы передаете методу ссылку на объект или массив, по которой он может оперировать этим объектом или массивом.

Программистам, пишущим на языках C и C++, следует обратить внимание на то, что Java не поддерживает оператор взятия адреса & и операторы разыменования * и ->. В Java примитивные типы всегда обрабатываются только по значению, а объекты и массивы – только по ссылкам: оператор . в Java ближе по значению к оператору -> в языках C и C++, чем к оператору . этих языков. Вы должны также понимать, что в отличие от указателей в языках C и C++, ссылки в Java совершенно непрозрачные: их нельзя получить из целых чисел и преобразовать в целые числа, а также прирастить или уменьшить.

Хотя ссылки и составляют важную часть Java, Java-программы не могут ими манипулировать. Несмотря на это поведение примитивных и ссылочных типов разительно отличается в двух важных областях: в копировании и сравнении значений.

Копирование объектов и массивов

Рассмотрим следующий код, оперирующий примитивным значением `int`:

```
int x = 42;
int y = x;
```

После выполнения этих строк переменная `y` содержит копию значения, хранимого в переменной `x`. Внутри виртуальной машины Java содержатся две независимые копии 32-битного целого числа 42.

Давайте посмотрим, что произойдет, если при выполнении того же базового кода мы используем не примитивный, а ссылочный тип:

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

После выполнения этого кода переменная `q` получает копию ссылки, хранящейся в переменной `p`. Виртуальная машина содержит только одну копию объекта `Point` и две копии ссылок на этот объект. В дальнейшем этот факт может сыграть важную роль. Предположим, что после двух предыдущих строк кода следует такой код:

```
System.out.println(p.x);        // Вывести координату X для p: 1.0
q.x = 13.0;                     // Теперь изменить координату X для q
System.out.println(p.x);        // Вывести p.x снова; в этот раз она равна 13.0
```

[1] Обычно ссылка представляет собой адрес памяти, по которому хранится данный объект или массив. Однако поскольку в Java ссылки непрозрачные и неуправляемые, ссылка является деталью реализации.

Поскольку переменные p и q содержат ссылки на один и тот же объект, изменить объект можно при помощи любой из них, причем эти изменения отразятся и на второй переменной.

Такое поведение присуще не только объектам; так же ведут себя и массивы, что показано в следующем коде:

```
char[] greet = { 'h','e','l','l','o' };   // greet содержит ссылку на массив char[]
cuss = greet;                             // cuss содержит ту же ссылку
cuss[4] = '!';                            // Используйте ссылку, чтобы изменить элемент
System.out.println(greet);                // Печатает "hell!"
```

Такое же различие в поведении примитивных и ссылочных типов проявляется при передаче аргументов методам. Рассмотрим следующий метод:

```
void changePrimitive(int x) {
  while(x > 0)
    System.out.println(x--);
}
```

При вызове этого метода ему передается копия переменной x. Код метода использует x в качестве счетчика цикла и уменьшает его значение до нуля. Это довольно разумное решение, поскольку x является примитивным типом, а метод содержит собственную копию данного значения.

Рассмотрим, что произойдет, если мы изменим метод и параметр станет ссылочного типа:

```
void changeReference(Point p) {
  while(p.x > 0)
    System.out.println(p.x--);
}
```

При вызове данного метода ему передается копия ссылки на объект Point, которую он может использовать для изменения этого объекта Point. Рассмотрим такой пример:

```
Point q = new Point(3.0, 4.5);      // Точка с координатой X = 3
changeReference(q);                 // Отображает 3, 2, 1 и модифицирует Point
System.out.println(q.x);            // Координата X для q теперь равна 0!
```

При вызове метода changeReference() ему передается копия ссылки из переменной q. Теперь переменная q и параметр метода p содержат ссылки на один и тот же объект. Метод может использовать эту ссылку для изменения содержимого объекта. Следует заметить, однако, что он не может изменить содержимое переменной q. Другими словами, метод может изменить объект Point до неузнаваемости, но он не может изменить факт принадлежности переменной q к этому объекту.

Хотя данный раздел называется «Копирование объектов и массивов», мы до сих пор рассматривали только копии ссылок на объекты и массивы, а не сами копии объектов и массивов. Сам объект или массив можно скопировать при помощи метода clone(), наследуемого всеми объектами от java.lang.Object:

```
Point p = new Point(1,2);       // p относится к одному объекту Point
q = (Point) p.clone();          // q относится к копии этого объекта
q.y = 42;                       // Изменить копируемый объект, но не менять исходные данные
int[] data = {1,2,3,4,5};       // Массив int[]
copy  = (int[]) data.clone();   // Копия массива
```

Примечание: чтобы возвращаемое значение метода clone() было соответствующего типа, необходимо выполнить приведение. Вы узнаете причину, дочитав эту главу до конца. Для правильного использования clone() нужно усвоить несколько моментов. Во-первых, не все объекты можно клонировать. Java позволяет клонировать объект, если класс объектов был явно объявлен клонируемым посредством реализации интерфейса Cloneable (мы еще не рассматривали ни интерфейсы, ни их реализацию; эта тема обсуждается в главе 3). Представленное ранее определение Point в действительности не реализует данный интерфейс, поэтому наш тип Point при реализации не клонируется. Однако следует заметить, что массивы клонируемы всегда. При вызове метода clone() для неклонируемого объекта генерируется исключение CloneNotSupportedException, поэтому метод clone() лучше применять внутри блока try, что позволит перехватить это исключение.

Кроме того, нужно знать, что по умолчанию метод clone() создает поверхностную копию объекта или массива. Скопированный объект или массив содержит копии всех примитивных значений и ссылок исходного объекта или массива. Другими словами, любые ссылки из клонируемого объекта или массива копируются, а не клонируются; clone() не делает копии объектов или массивов по их ссылкам рекурсивно. Класс может подменять поверхностную копию на свою версию метода clone(), которая бы явно выполняла более детальное копирование там, где это потребуется. Чтобы понять поверхностное поведение clone(), рассмотрим клонирование двухмерного массива массивов:

```
int[][] data = {{1,2,3}, {4,5}};     // Массив с двумя ссылками int[][]
copy = (int[][]) data.clone();       // Скопировать эти две ссылки в новый массив
copy[0][0] = 99;                     // Также изменится data[0][0]!
copy[1] = new int[] {7,8,9};         // Не изменится data[1]
```

Чтобы сделать детальную копию (deep copy) этого многомерного массива, нужно явно копировать каждое измерение:

```
int[][] data = {{1,2,3}, {4,5}};     // Массив с двумя ссылками int[][]
copy = new int[data.length][];       // Новый массив, содержащий скопированные массивы
for(int i = 0; i < data.length; i++)
  copy[i] = (int[]) data[i].clone();
```

Сравнение объектов и массивов

Мы знаем, что присваивание переменным, передача методам и копирование значений примитивных типов отличаются от манипуляций со значениями ссылочных типов. Также по-разному происходит и сравнение на равенство. Для примитивных значений оператор равенства (==) просто проверяет два значения на идентичность (то есть их биты должны в точности совпадать). Однако, что касается ссылочных типов, оператор == сравнивает ссылки, а не сами объекты или массивы. Другими словами, == проверяет, относятся ли две ссылки к одному и тому же объекту или массиву; он не проверяет содержимое двух объектов или массивов на идентичность. Например:

```
String letter = "o";
String s = "hello";                    // Эти два объекта String
String t = "hell" + letter;            // содержат совершенно одинаковый текст.
if (s == t) System.out.println("equal"); // Но они не равны!
byte[] a = { 1, 2, 3 };                // Массив.
byte[] b = (byte[]) a.clone();         // Копия с идентичным содержимым.
if (a == b) System.out.println("equal"); // Но они не равны!
```

Для ссылочных типов существуют два вида равенства: равенство ссылок и равенство объектов. Важно различать эти два вида. Чтобы не запутаться, можно применять слово «равно» (equals) по отношению к ссылкам и слово «эквивалентно» (equivalent) по отношению к двум различным объектам или массивам с аналогичным содержимым. К сожалению, разработчики Java не использовали такую терминологию, поскольку метод проверки двух объектов на эквивалентность называется `equals()`. Чтобы проверить два объекта на эквивалентность, следует передать один из них методу `equals()` второго объекта:

```
String letter = "o";
String s = "hello";                    // Эти два строковых объекта
String t = "hell" + letter;            // содержат совершенно одинаковый текст.
if (s.equals(t))                       // И метод equals()
  System.out.println("equivalent");    // нам это показывает.
```

Все объекты наследуют метод `equals()` у класса `Object`, но в реализации по умолчанию оператор `==` проверяет на равенство ссылки, а не содержимое. Если вы хотите сравнить объекты какого-либо класса на эквивалентность, такой класс может определить собственную версию метода `equals()`. Наш класс `Point` не определяет такой метод, зато его определяет класс `String`, как показано выше. Вы можете вызвать метод `equals()` из массива. Это аналогично использованию оператора `==`, поскольку массивы всегда по умолчанию наследуют метод `equals()`, который сравнивает ссылки, а не содержимое массива. Начиная с версии Java 1.2, можно сравнивать массивы на эквивалентность при помощи вспомогательного метода `java.util.Arrays.equals()`. Однако при работе с более ранними версиями придется самостоятельно проходить по элементам массива (в цикле) и сравнивать их.

Ссылка null

С ключевым словом `null` мы уже познакомились при обсуждении объектов и массивов. Поскольку мы уже знаем, что такое ссылки, давайте снова вернемся к `null` и вспомним, что `null` представляет собой отсутствие ссылки или особое, ни на что не ссылающееся, значение. `null` является значением по умолчанию для всех ссылочных типов. Значение `null` уникально тем, что его можно присвоить любой переменной любого ссылочного типа.

Терминология: передача по значению

Как уже говорилось, Java обрабатывает массивы и объекты «по ссылкам». Не перепутайте эту фразу с выражением «передать по ссылке».[1] Термин «передать по ссылке» описывает соглашения о вызовах метода, принятых в языках программирования. В языке, поддерживающем передачи по ссылке, значения (даже примитивные) не передаются методу напрямую. Напротив, методу всегда передаются ссылки на значения. Поэтому если метод изменяет параметры, эти изменения отражаются по возвращении метода, даже для примитивных типов.

Это *не* верно для Java, поскольку это язык «с передачей по значению». Однако передаваемое значение ссылочного типа всегда будет ссылкой. И все же это не передача по ссылке. Если бы язык Java был языком с передачей по ссылке, то при передаче методу значения ссылочного типа передавалась бы ссылка на ссылку.

[1] К сожалению, эта разница не была четко определена в предыдущих изданиях данной книги.

Распределение памяти и сборка мусора

Как уже говорилось, объекты и массивы являются составными значениями, содержащими другие значения. Зачастую они требуют значительного объема памяти. Если вы создаете новый объект при помощи ключевого слова `new` или используете в своей программе объект или массив литералов, Java автоматически создает для вас объект, выделяя необходимый объем памяти. От вас в данном случае ничего не требуется.

Кроме того, Java автоматически освобождает эту память для последующего использования, если она больше не нужна. Это освобождение осуществляется в процессе *сборки мусора* (*garbage collection*). Объект становится мусором, если переменные, поля объектов или элементы массивов больше не содержат никаких ссылок на этот объект. Например:

```
Point p = new Point(1,2);          // Создать копию объекта
d = p.distanceFromOrigin();        // Применить ее
p = new Point(2,3);                // Создать новый объект
```

Ссылка на новый объект `Point` замещает ссылку на первый объект, когда интерпретатор Java заканчивает третью строку. На первый объект ссылок не остается, поэтому он становится мусором. На определенном этапе сборщик мусора обнаружит этот факт и освободит память, используемую объектом.

У С-программистов, использующих `malloc()` и `free()` для управления памятью, и программистов, пишущих на C++, которые явно удаляют свои объекты при помощи `delete`, такая передача управления и полное доверие сборщику мусора может вызвать сложности. Такой способ кажется неправдоподобным, но он на самом деле работает! Сборщик мусора может слегка ухудшить производительность и значительно замедлить выполнение Java-программ только во время освобождения памяти. Поскольку сборка мусора заложена в самом языке, значительно сокращаются утечки памяти и связанные с ними ошибки, а производительность программиста практически всегда повышается.

Преобразование ссылочных типов

Ранее в этой главе при обсуждении примитивных типов вы узнали, что значения одного типа можно преобразовывать в значения другого типа. По необходимости интерпретатор Java автоматически выполняет расширяющие преобразования. Однако сужающие преобразования могут привести к потере данных, поэтому интерпретатор выполняет их только при явном приведении к типу.

Java не допускает никаких преобразований из примитивного типа в ссылочный или наоборот. Однако можно выполнять расширяющие и сужающие преобразования над определенными ссылочными типами. Как мы успели убедиться, возможных ссылочных типов бесконечно много. Чтобы разобраться в процессе преобразования этих типов, вы должны понять, что эти типы входят в так называемую *иерархию классов*.

Каждый класс Java *расширяет* другой класс, который является его родительским классом. Класс наследует поля и методы родительского класса, а затем определяет собственные дополнительные поля и методы. Основой иерархии классов Java служит специальный класс `Object`. Он не расширяет никакой другой класс, зато все другие классы Java расширяют `Object` или какой-либо другой класс, одним из родительских классов которого является `Object`. Класс `Object` определяет специальные методы, наследуемые (или переопределяемые) всеми классами. Сюда входят такие описанные ранее методы, как `toString()`, `clone()` и `equals()`.

Как определенный нами ранее класс `Point`, так и предопределеный класс `String` расширяют `Object`. Благодаря этому мы можем утверждать, что все объекты `String` также являются объектами `Object`. Мы также можем сказать, что все объекты `Point` являются объектами `Object`. Впрочем, обратное утверждение будет неверно. Нельзя сказать, что каждый `Object` является `String`, потому что, как мы только что убедились, некоторые объекты `Object` являются объектами `Point`.

Теперь, усвоив базовую иерархию классов, мы можем вернуться к правилам преобразования ссылочного типа:

- Нельзя преобразовать объект в неродственный ему тип. Например, компилятор Java не позволяет преобразовывать `String` в `Point`, даже если вы используете оператор приведения.
- Объект можно преобразовать в тип родительского класса. Это расширяющее преобразование, и приведение для него не требуется. Например, значение `String` можно присвоить переменной типа `Object` или передать методу, который ожидает параметр `Object`. Примечание: на самом деле не выполняется никакое преобразование; объект просто интерпретируется как экземпляр родительского класса.
- Объект можно преобразовать в тип подкласса, но это будет сужающее преобразование, требующее приведения. Компилятор Java позволяет такое преобразование, но во время выполнения программы интерпретатор Java проверяет его обоснованность. Приводить объект к типу подкласса следует лишь в тех случаях, когда по логике вашей программы объект действительно является экземпляром этого подкласса. В противном случае интерпретатор генерирует исключение `ClassCastException`. Например, если переменной типа `Object` присвоить объект `String`, то значение данной переменной можно привести обратно к этому типу `String`:

```
Object o = "string";          // Расширяющее преобразование от String к Object
// Позже в программе...
String s = (String) o;        // Сужающее преобразование от Object к String
```

Массивы являются объектами и следуют некоторым правилам преобразования. Во-первых, любой массив можно преобразовать в значение `Object` путем расширения. Обратно в массив такое значение объекта можно преобразовать при помощи сужения и приведения к типу. Например:

```
Object o = new int[] {1,2,3};     // Расширяющее преобразование от массива к Object
// Позже в программе...
int[] a = (int[]) o;              // Сужающее преобразование обратно к массиву
```

Вы можете преобразовать не только массив в объект, но также и массив в другой тип массива, если «базовыми типами» обоих массивов являются ссылочные типы, которые также можно преобразовывать. Например:

```
// Здесь представлен массив строк
String[] strings = new String[] { "hi", "there" };
// Допускается расширяющее преобразование к CharSequence[],
// потому что String можно расширить до CharSequence.
CharSequence[] sequences = strings;
// Сужающее преобразование обратно к String[] требует приведения
strings = (String[]) sequences;           // Это массив массивов строк
String[][] s = new String[][] { strings };
// Его нельзя преобразовать в CharSequence[], потому что String[]
// нельзя преобразовать в CharSequence; не совпадает количество измерений.
sequences = s;                            // Эта строка не компилируется,
// однако s можно преобразовать в Object или Object[],  поскольку
```

```
// все типы массивов (включая String[] и String[][]) можно преобразовать в Object.
Object[] objects = s;
```

Примечание: данные правила преобразования массивов подходят только для массивов объектов и массивов массивов. Нельзя преобразовать массив примитивного типа в какой-либо тип массива, даже если можно преобразовать примитивные базовые типы:

```
// int[] нельзя преобразовать в double[] несмотря на то, что int можно расширить до double
double[] data = new int[] {1,2,3};          // Данная строка приводит к ошибке компиляции
// Однако следующая строка разрешена, поскольку int[] можно преобразовать в Object
Object[] objects = new int[][] {{1,2},{3,4}};
```

Пакеты и пространство имен Java

Пакет (*package*) представляет собой именованную совокупность классов (и, возможно, подпакетов). Пакеты группируют классы и определяют пространства имен для классов, которые в них входят.

Платформа Java содержит пакеты, имена которых начинаются на `java`, `javax` и `org.omg`. (Кроме того, в пакетах, имена которых начинаются с `javax`, Sun определяет стандартные расширения платформы Java.) Основные классы языка входят в пакет `java.lang`. Различные вспомогательные классы находятся в `java.util`. Классы для ввода и вывода входят в `java.io`, а классы для работы в сети – в `java.net`. Некоторые их этих пакетов содержат подпакеты. Например, `java.lang` содержит два специализированных пакета `java.lang.reflect` и `java.lang.ref`, а `java.util` содержит подпакет `java.util.zip`, который в свою очередь содержит классы для работы с ZIP-архивами.

Каждый класс имеет как простое имя, данное ему в определении, так и полное имя, включающее имя пакета, в который он входит. Например, класс `String` является частью пакета `java.lang`, а его полное имя – `java.lang.String`.

Определение пакета

Чтобы определить пакет для какого-либо класса, следует использовать директиву `package`. Если в Java-коде присутствует ключевое слово `package`, оно должно быть первой лексемой кода в файле Java, то есть должно следовать сразу после комментариев и пробелов. После ключевого слова должно стоять имя требуемого пакета и точка с запятой. Рассмотрим файл кода Java, начинающийся с такой директивы:

```
package com.davidflanagan.jude;
```

Все классы, определяемые этим файлом, входят в пакет `com.davidflanagan.jude`.

Классы, определяемые файлом Java-кода без директивы `package`, входят в пакет без имени, существующий по умолчанию. Как вы узнаете из главы 3, классы одного пакета получают специальный доступ друг к другу. Поэтому, если только вы не пишете простые программы, всегда используйте директиву `package`, чтобы закрыть доступ в ваши классы из совершенно посторонних классов, сохраняемых в пакете без имени.

Импорт классов и пакетов

Класс пакета `p` может ссылаться на любой другой класс в `p` по его простому имени. А поскольку классы пакета `java.lang` являются базовыми для языка Java, любой Java-код может ссылаться на любой класс этого пакета по его простому имени. Значит, всегда можно набирать `String` вместо `java.lang.String`. Однако по умолчанию для всех

других классов нужно указывать полные имена. Поэтому, чтобы применить класс `File` пакета `java.io`, следует набрать `java.io.File`.

Каждый раз явно указывать имена пакетов довольно утомительно, поэтому в Java предусмотрена директива `import`, которая избавит вас от части такой работы. `import` используют для определения классов и пакетов классов, на которые можно ссылаться не только по полностью определенным, но и по простым именам. В файле Java ключевое слово `import` можно применять бесконечное количество раз, однако эти ключевые слова должны располагаться только в начале файла, сразу же после директивы `package`, если она существует. Конечно, между директивой `package` и директивами `import` могут находиться комментарии, но никакой другой код Java здесь размещать нельзя.

Директива `import` бывает двух видов. Чтобы определить отдельный класс, на который можно ссылаться по его простому имени, следом за ключевым словом `import` введите имя класса и точку с запятой:

```
import java.io.File;     // Теперь можно набирать File вместо java.io.File
```

Чтобы импортировать целый пакет классов, введите после `import` имя пакета, символы `.*` и точку с запятой. Например, чтобы использовать класс `File` вместе с несколькими другими классами пакета `java.io`, нужно просто импортировать целый пакет:

```
import java.io.*;      // Теперь простые имена можно применять для всех классов из java.io
```

Этот синтаксис импорта пакета не относится к подпакетам. Даже если я импортирую пакет `java.util`, мне все еще придется обращаться к классу `java.util.zip.ZipInputStream` по его полному имени. Если два класса, импортированные из разных пакетов, называются одинаково, к ним нельзя обращаться по простому имени; во избежание двусмысленности к обоим классам следует обращаться по их полным именам.

Глобально уникальные имена пакетов

Одной из важных функций пакетов является разделение пространства имен Java и предотвращение конфликта имен между классами. Например, классы `java.util.List` и `java.awt.List` можно различить только по именам их пакетов. Однако важно, чтобы различались и имена самих пакетов. Как разработчик Java Sun контролирует все имена пакетов, которые начинаются с `java`, `javax` и `sun`.

Для остальных пакетов Sun предлагает схему именования, правильное следование которой обеспечивает глобально уникальные имена пакетов. В качестве префикса для всех имен пакетов схема предполагает использование интернет-имени вашего домена, причем его элементы располагаются в обратном порядке. Мой веб-сайт – *davidflanagan.com*, поэтому все мои Java-пакеты начинаются с `com.davidflanagan`. Я могу разделять по собственному усмотрению пространство имен ниже `com.davidflanagan`, а поскольку это имя домена принадлежит мне, никакой другой человек или организация, следующие этому правилу, не могут определять имя пакета, которое есть у меня.

Структура файла Java

В этой главе мы прошли от простых к сложным элементам синтаксиса Java, от отдельных символов и лексем к операторам, выражениям и методам, вплоть до классов и пакетов. На практике чаще всего вы будете сталкиваться с такой единицей структуры Java-программы, как файл Java. Файл Java является наименьшей единицей Java-кода, компилируемой компилятором Java. Он состоит из:

- Необязательной директивы `package`

- Нескольких директив `import` (их может не быть)
- Одного или нескольких определений класса

Данные элементы могут перемежаться с комментариями, но они должны идти именно в такой последовательности. Это все, что нужно знать о файле Java. Все операторы Java (кроме директив `package` и `import`, которые не являются полноценными операторами) должны находиться внутри методов, а все методы – внутри определения класса.

Существует еще несколько важных ограничений по файлам Java. Во-первых, каждый файл может содержать максимум один класс, объявленный как `public`. Такой класс предназначен для использования другими классами в других пакетах. Более подробно `public`-классы и связанные с ними модификаторы рассматриваются в главе 3. Данное ограничение по открытым классам относится только к классам верхнего уровня; как вы узнаете из главы 3, класс может содержать любое количество вложенных, или внутренних, классов, объявленных как `public`.

Второе ограничение касается имени файла в Java. Если файл Java содержит класс `public`, то имя файла должно представлять собой имя класса, к которому присоединено расширение *.java*. Например, если `Point` является `public`-классом, его исходный код должен находиться в файле *Point.java*. Даже если ваши классы не являются `public`, полезно определять только по одному классу на файл и давать файлу имя класса.

Как только файл Java скомпилирован, каждый из определяемых им классов компилируется в отдельный файл классов, который содержит байт-код Java, интерпретируемый виртуальной машиной. Файл класса имеет то же имя, что и определяемый им класс, к которому добавляется расширение *.class*. Поэтому если файл *Point.java* определяет класс `Point`, то компилятор Java компилирует его в файл *Point.class*. В большинстве систем файлы классов сохраняются в каталогах, соответствующих именам их пакетов. Следовательно, класс `com.davidflanagan.jude.DataFile` определяется файлом классов *com/davidflanagan/jude/DataFile.class*.

Интерпретатор Java знает местоположение файлов классов в стандартной системе и может правильно загрузить их. Если интерпретатор выполняет программу, которая хочет использовать класс `com.davidflanagan.jude.DataFile`, он знает, что код этого класса находится в каталоге *com/davidflanagan/jude*, и по умолчанию «ищет» в текущем каталоге подкаталог с таким именем. Заставить интерпретатор искать файлы в каком-либо другом каталоге можно при помощи опции `-classpath`. Другой подход – установить переменную среды `CLASSPATH`. Для получения более подробных сведений обратитесь к документации по интерпретатору Java (*java*), представленной в главе 8.

Определение и выполнение Java-программ

Java-программа состоит из нескольких взаимодействующих определений классов. Но не каждый класс или файл Java является программой. Чтобы создать программу, нужно определить класс, содержащий специальный метод со следующей сигнатурой:

```
public static void main(String[] args)
```

Метод `main()` является основной точкой входа в вашу программу. Именно здесь начинает работать интерпретатор Java. Этот метод получает массив строк и не возвращает никакого значения. Интерпретатор Java завершает программу, как только заканчивается `main()` (если только `main()` не создал отдельные потоки; в этом случае интерпретатор ждет, пока не завершатся все эти потоки).

Java-программу можно выполнить, запустив интерпретатор *java* с указанием полного имени класса, содержащего метод `main()`. Заметьте, что следует указать имя класса, а не имя файла класса, содержащего этот класс. Любые дополнительные аргументы, указанные в командной строке, передаются методу `main()` в качестве его параметра `String[]`. Вы также можете применить опцию `-classpath` (или `-cp`), чтобы указать интерпретатору местоположение необходимых программе классов. Рассмотрим следующую команду:

```
% java -classpath /usr/local/Jude com.davidflanagan.jude.Jude datafile.jude
```

java – это команда запуска интерпретатора Java. *-classpath /usr/local/Jude* сообщает интерпретатору местоположение файлов *.class*. `com.davidflanagan.jude.Jude` – это имя выполняемой программы (то есть имя класса, определяющего метод `main()`). И наконец, *datafile.jude* является строкой, передаваемой методу `main()` в качестве единственного элемента массива объектов `String`.

В Java 1.2 существует более простой способ запуска программ. Можно запустить программу, просто указав имя файла JAR, если программа и ее вспомогательные классы (кроме тех, которые составляют платформу Java) были должным образом упакованы в файл архива Java (JAR):

```
% java -jar /usr/local/Jude/jude.jar datafile.jude
```

Некоторые операционные системы автоматически запускают файлы JAR. В этих системах достаточно просто ввести:

```
% /usr/local/Jude/jude.jar datafile.jude
```

Более подробно о запуске интерпретатора рассказано в главе 8.

Различия между языками C и Java

Программистам, пишущим на языках C и C++, большая часть синтаксиса Java, особенно операторы, должна показаться знакомой. Языки Java и C достаточно похожи друг на друга, и программистам, пишущим на C и C++, важно знать, где заканчивается их схожесть. Ниже представлен краткий список некоторых важных отличий между Java и C:

Отсутствие препроцессора

В язык Java не входит препроцессор; не определены никакие аналоги директив `#define`, `#include` и `#ifdef`. Определения констант замещены полями `static final` (например, поле `java.lang.Math.PI`). В Java недоступны макроопределения, но продвинутая технология компиляции и макрокомандный метод программирования сделали их менее необходимыми. В Java не нужна директива `#include`, поскольку в Java нет заголовочных файлов. Файлы классов Java содержат как API класса, так и реализацию класса, а компилятор по необходимости считывает API-информацию из файлов классов. В Java нет никакой условной компиляции, но межплатформенная переносимость Java делает ее практически ненужной.

Отсутствие глобальных переменных

В Java очень ясно определено пространство имен. Пакеты содержат классы, классы содержат поля и методы, а методы содержат локальные переменные. Но в Java нет глобальных переменных, поэтому конфликт между пространствами имен этих переменных невозможен.

Четко определенные диапазоны примитивных типов

В Java четко определены диапазоны значений всех примитивных типов. В языке C диапазоны типов `short`, `int` и `long` зависят от платформы, что ухудшает переносимость.

Отсутствие указателей

В Java классы и массивы являются ссылочными типами, а ссылки на объекты и массивы сродни указателям в языке C. Однако, в отличие от указателей в языках C, ссылки в Java совершенно непрозрачны. Ссылку нельзя преобразовать в примитивный тип, прирастить или уменьшить. Не существует также оператора взятия адреса `&` и оператора разыменования `*` или `->`, а также оператора `sizeof`. Указатели являются общеизвестным источником ошибок. Их отсутствие только упрощает язык и делает Java-программы безопасными и устойчивыми к сбоям.

Сборка мусора

Сборкой мусора занимается виртуальная машина Java, поэтому Java-программистам не нужно явно управлять памятью, используемой всеми объектами и массивами. Эта особенность устраняет целую категорию распространенных ошибок, а также утечку памяти в Java-программах.

Отсутствие оператора goto

Java не поддерживает оператор `goto`. За исключением четко определенных ситуаций, применение `goto` не считается лучшим методом работы. К операторам управления потоком, предлагаемым языком C, в Java добавлены операторы обработки исключений и операторы с меткой `break` и `continue`. Они успешно заменяют `goto`.

Объявления переменных в любой части программы

В языке C локальные переменные нужно объявлять в начале метода или блока, в то время как Java допускает объявления переменных в любом месте метода или блока. Однако большинство программистов предпочитают помещать все объявления переменных в начале метода.

Опережающие ссылки

Компилятор Java умнее компилятора C, поскольку он позволяет вызывать метод перед его определением. Это устраняет необходимость объявлять функции в заголовочном файле перед определением их в файле программы, как это реализовано в C.

Перегрузка метода

Программы Java могут определять несколько методов с одинаковым именем, при условии, что их списки параметров различны.

Отсутствие типов struct и union

Java не поддерживает такие типы языка C, как `struct` и `union`. Тем не менее `class` Java можно считать расширенным `struct`.

Отсутствие перечисляемых типов

Java не поддерживает ключевое слово `enum`, которое определяет в языке C типы, состоящие из фиксированных множеств значений с именами. Это необычно для такого сильно типизированного языка, как Java, однако данное свойство можно смоделировать при помощи объектных констант.

Отсутствие битовых полей

Java не поддерживает возможность языка C определять количество отдельных бит, занятых в полях типа `struct`. В C эта возможность используется нечасто.

Отсутствие typedef

Java не поддерживает ключевое слово `typedef`, определяющее в языке C альтернативные имена для имен типов. Отсутствие указателей в Java упрощает и делает более последовательной схему именования типов, если сравнивать ее с языком C, поэтому в применении `typedef` в Java нет необходимости.

Отсутствие указателей на методы

Язык C позволяет сохранять адрес функции в переменной и передавать этот указатель на функцию другим функциям. С методами Java так не получится, зато подобного результата можно достичь, передав объект, который реализует определенный интерфейс. Кроме того, метод Java можно представить и вызвать посредством объекта `java.lang.reflect.Method`.

Отсутствие списка аргументов переменной длины

Java не позволяет определять методы, получающие переменное количество аргументов, подобно `printf()` в C. В простых ситуациях перегрузка метода позволяет моделировать функции с переменным количеством аргументов, предлагаемые языком C. Кроме того, аргументы можно передавать как `Object[]`, однако универсальной замены такого свойства не существует.

Глава 3

Объектно-ориентированное программирование в Java

Java – это объектно-ориентированный язык программирования. Как обсуждалось ранее, все Java-программы используют объекты, а каждая Java-программа определяется как класс. В предыдущей главе объяснялся базовый синтаксис языка программирования Java, включая типы данных, операторы и выражения, и даже было показано, как определить простой класс и как работать с объектами. Эта глава продолжает объяснение с того места, на котором мы остановились. Здесь излагаются детали объектно-ориентированного программирования в Java.

Если у вас нет опыта объектно-ориентированного программирования (ООП), не волнуйтесь: эта глава не предполагает никакого предыдущего опыта. Если же у вас есть опыт в ООП, будьте осторожны. В разных языках термин «объектно-ориентированный» означает разные вещи. Не стоит думать, что Java работает так же, как ваш любимый объектно-ориентированный язык. В первую очередь это относится к программистам, пишущим на C++. В предыдущей главе мы видели, что между Java и C можно провести близкие аналогии. В то же время подобные аналогии при сравнении Java и C++ невозможны. Java использует концепции объектно-ориентированного программирования на C++ и даже во многих случаях заимствует синтаксис C++, но общего между Java и C++ меньше, чем между Java и C. Не дайте вашему опыту в C++ убаюкать вас ложной схожестью C++ и Java.

Члены класса

Как уже обсуждалось в главе 2, класс – это коллекция данных, хранимых в именованных полях, и кода, организованного в именованные методы, оперирующие этими данными. Поля и методы называются *членами* класса (members). В Java 1.1 и последующих версиях классы могут содержать другие классы. Эти классы-члены, или внутренние классы, предоставляют дополнительную функциональность и обсуждаются далее в этой главе. Сейчас мы рассмотрим только поля и методы. Члены класса делятся на два различных типа: члены класса, связанные непосредственно с классом (статические), и члены экземпляра, связанные с каждым экземпляром класса (то есть с объектом). Не принимая во внимание классы-члены, мы получаем четыре типа членов:

- Поля класса

- Методы класса
- Поля экземпляра
- Методы экземпляра

Определение простого класса `Circle`, приведенное в примере 3.1, содержит члены всех четырех типов.

Пример 3.1. Простой класс и его члены

```
public class Circle {
  // Поле класса
  public static final double PI = 3.14159;          // Полезная константа

  // Метод класса: просто вычисляет значение на основе аргумента
  public static double radiansToDegrees(double rads) {
    return rads * 180 / PI;
  }

  // Поле экземпляра
  public double r;                                  // Радиус окружности

  // Два метода экземпляра: оперируют полями экземпляра
  public double area() {                            // Вычисляет площадь окружности
    return PI * r * r;
  }
  public double circumference() {                   // Вычисляет периметр окружности
    return 2 * PI * r;
  }
}
```

Поля класса

Поля класса связаны с классом, в котором они определены, а не с экземпляром класса. Следующая строка объявляет поле класса:

```
public static final double PI = 3.14159;
```

Эта строка объявляет поле типа `double` с именем `PI` и присваивает ему значение 3.14159. Как вы видите, объявление поля похоже на объявление локальных переменных. Разница, конечно, в том, что переменные объявляются в методах, а поля являются членами класса.

Модификатор `static` говорит о том, что поле является полем класса. Поля класса иногда называют статическими полями из-за модификатора `static`. Модификатор `final` говорит о том, что значение поля не меняется. Так как поле PI является константой, мы объявляем его с модификатором `final`, чтобы его нельзя было изменить. В Java (и других языках) существует соглашение о том, что константы именуются заглавными буквами, поэтому имя нашего поля `PI`, а не `pi`. Определение таких констант – это наиболее частый вариант применения полей класса, то есть модификаторы `static` и `final` часто используются вместе. Как известно, не все поля являются константами. Другими словами, поле может быть объявлено с модификатором `static`, но без модификатора `final`. И наконец, модификатор `public` говорит о том, что поле может быть использовано кем угодно. Это модификатор видимости, и мы обсудим его и связанные с ним модификаторы далее в этой главе.

Ключевым моментом в понимании статического поля является то, что существует только одна копия этого поля. Это поле связано непосредственно с классом, а не с экземпляром класса. Если вы посмотрите на различные методы класса `Circle`, то увидите, что они используют это поле. Внутри класса `Circle` на данное поле можно ссылаться просто по имени `PI`. Однако вне класса нужно указать имена класса и поля для того, чтобы уникально идентифицировать это поле. Методы, не являющиеся частью класса `Circle`, ссылаются на это поле как на `Circle.PI`.

По существу, поле класса является глобальной переменной. Имена полей класса однозначно идентифицируются уникальными именами классов, которые их содержат. Таким образом, в Java нет конфликтов имен, свойственных другим языкам программирования, когда различные участки кода объявляют глобальные переменные с одинаковыми именами.

Методы класса

Как и поля класса, методы класса объявляются с модификатором `static`:

```
public static double radiansToDegrees(double rads) { return rads * 180 / PI; }
```

Эти строки определяют метод класса с именем `radiansToDegrees()`. У него есть единственный параметр типа `double`, и он возвращает значение типа `double`. Тело метода достаточно небольшое; метод выполняет простое вычисление и возвращает результат.

Как и поля класса, методы класса связаны с классом, а не с объектом. При вызове метода класса из кода, который существует вне самого класса, вы должны указать имя класса и имя метода. Например:

```
// Сколько градусов в 2.0 радианах?
double d = Circle.radiansToDegrees(2.0);
```

Если вы хотите вызвать метод класса из самого класса, то вам не нужно указывать имя класса. Впрочем, часто хорошим стилем считается указание имени класса, чтобы было ясно, что вызывается метод класса.

Обратите внимание: в теле нашего метода `Circle.radiansToDegrees()` используется поле класса `PI`. Метод класса может обращаться к любым полям и методам класса, в котором он определен (наравне с полями и методами других классов). Но он не может использовать поля или методы экземпляра, так как методы класса не связаны ни с одним экземпляром класса. Другими словами, несмотря на то что метод `radiansToDegrees()` определен в классе `Circle`, он не использует объекты `Circle`. Поля и методы экземпляра класса связаны с объектами `Circle`, а не с самим классом. Так как метод класса не связан с экземпляром этого класса, он не может использовать методы или поля экземпляра.

Как мы обсуждали ранее, поле класса – это, по сути, глобальная переменная. Точно так же, метод класса – это глобальный метод, или глобальная функция. Несмотря на то что метод `radiansToDegrees()` не оперирует объектами `Circle`, он определен внутри класса `Circle`, потому что это служебный метод, который иногда полезен при работе с окружностями. Во многих не объектно-ориентированных языках программирования все методы или функции являются глобальными. Вы можете писать сложные Java-программы, используя только методы класса. Впрочем, такой подход не является объектно-ориентированным программированием и не использует все преимущества и мощь языка Java. Чтобы программировать «объектно-ориентированно», вы должны добавить поля и методы экземпляра в свой репертуар.

Поля экземпляра

Любое поле, объявленное без модификатора `static`, является полем экземпляра:

```
public double r;          // Радиус окружности
```

Поля экземпляра связаны с экземпляром класса, а не с самим классом. Поэтому каждый объект `Circle`, который мы создаем, имеет собственную копию поля `r` типа `double`. В нашем примере `r` представляет собой радиус окружности. Поэтому каждый объект `Circle` может иметь радиус, независимый от других объектов `Circle`.

Внутри определения класса на поля экземпляра ссылаются по имени. Вы можете увидеть пример этого, посмотрев на тело метода `circumference()` экземпляра. В коде, находящемся за пределами класса, имени метода экземпляра должна предшествовать ссылка на окружающий объект. Например, если у нас есть объект `Circle` с именем `c`, то мы можем ссылаться на поле `r` экземпляра как на `c.r`:

```
Circle c = new Circle();  // Создать новый объект Circle; сохранить его в переменной c
c.r = 2.0;                // Присвоить значение полю экземпляра r
Circle d = new Circle();  // Создать другой объект Circle
d.r = c.r * 2;            // Сделать его вдвое больше
```

Поле экземпляра является ключевым понятием объектно-ориентированного программирования. Поля экземпляра определяют объект; значения этих полей отличают один объект от другого.

Методы экземпляра

Любой метод, объявленный без модификатора `static`, является методом экземпляра. Метод экземпляра работает с экземпляром класса (объектом), а не с самим классом. Именно с применением методов экземпляров объектно-ориентированное программирование становится интересным. Класс `Circle`, описанный ранее, содержит два метода экземпляра, `area()` и `circumference()`, которые вычисляют и возвращают площадь и периметр окружности, представленной данным объектом `Circle`.

Для использования метода экземпляра вне класса, в котором он определен, мы должны предварить имя метода ссылкой на экземпляр класса, в котором определен этот метод. Например:

```
Circle c = new Circle(); // Создать объект Circle; сохранить его в переменной c
c.r = 2.0;               // Установить значение поля экземпляра объекта
double a = c.area();     // Вызвать метод экземпляра объекта
```

Если вы новичок в объектно-ориентированном программировании, то для вас последняя строка кода может выглядеть несколько странно. Я не написал:

```
a = area(c);
```

Вместо этого я написал:

```
a = c.area();
```

Вот почему такой подход называется объектно-ориентированным программированием; внимание фокусируется на объекте, а не на вызове функции. Возможно, это маленькое синтаксическое различие является единственной наиболее важной особенностью объектно-ориентированной концепции.

Особенность данного подхода состоит в том, что нам не нужно передавать аргумент методу `c.area()`. Объект `c`, с которым мы работаем, в этом синтаксисе неявный. Еще

раз просмотрите исходный пример. Вы заметите то же самое в сигнатуре метода area(): у него нет параметров. Теперь посмотрите на тело метода area(): он использует поле r экземпляра. Так как метод area() является частью класса, который определяет это поле экземпляра, он может применять неполное имя r. Понятно, что оно ссылается на радиус любого экземпляра Circle, для которого вызывается метод.

Другая важная особенность, на которую нужно обратить внимание в телах методов area() и circumference(), — они оба используют поле PI класса. Ранее мы видели, что методы класса могут обращаться только к полям и методам класса, но не к полям или методам экземпляра. Методы экземпляра не имеют таких ограничений: они могут использовать любые члены класса вне зависимости от того, объявлены они с модификатором static или нет.

Как работают методы экземпляра

Рассмотрим следующую строку кода:

```
a = c.area();
```

Что здесь происходит? Как метод, не имеющий параметров, может знать, с какими данными ему работать? В действительности метод area() не имеет параметров. Все методы экземпляра реализованы с неявным параметром, который не показан в сигнатуре метода. Этот неявный аргумент называется this; он хранит ссылку на объект, для которого был вызван метод. В нашем примере это объект Circle.

Неявный параметр this не отображается в сигнатуре метода, потому что обычно он не нужен; когда бы метод Java ни пытался получить доступ к полям своего класса, он пытается обратиться к объекту, на который ссылается параметр this. То же самое происходит, когда метод экземпляра вызывает другие методы экземпляра того же класса. Ранее я говорил, что для вызова метода экземпляра следует подготовить ссылку на объект, метод которого нужно вызвать. Когда метод экземпляра вызывается другим методом экземпляра того же класса, вам не нужно указывать объект. В этом случае метод неявно вызывается для объекта this.

Вы можете использовать ключевое слово this явно, когда хотите четко показать, что метод оперирует собственным полем или методом. Например, мы можем переписать метод area() так, чтобы он явно использовал this при ссылке на поля экземпляра:

```
public double area() {
  return Circle.PI * this.r * this.r;
}
```

Этот код также использует имя класса, чтобы явно сослаться на поле PI класса. Это простой метод, поэтому нет необходимости явного указания this. Тем не менее в более сложных случаях явное указание this там, где это строго не требуется, повышает ясность вашего кода.

Есть несколько случаев, в которых ключевое слово this необходимо. Например, когда параметр метода или локальная переменная в методе называется так же, как одно из полей класса, вы должны использовать this для ссылки на поле, поскольку имя поля без явного указания this ссылается на параметр метода или локальную переменную. Например, мы можем добавить следующий метод в класс Circle:

```
public void setRadius(double r) {
  this.r = r;                        // Присвоить значение аргумента (r) полю (this.r)
                                     // Обратите внимание, что мы не можем просто написать r = r
}
```

И наконец, обратите внимание, что методы экземпляра могут использовать ключевое слово this, а методы класса – не могут. Это происходит потому, что методы класса не связаны с объектом.

Методы экземпляра или методы класса?

Методы экземпляра представляют собой одну из ключевых особенностей объектно-ориентированного программирования. Тем не менее это не означает, что вы должны избегать методов класса. Существует много случаев, когда определение методов класса абсолютно оправданно. Например, при работе с классом Circle вы можете обнаружить, что очень часто приходится вычислять площадь окружности с данным радиусом, но вам не хочется утруждать себя созданием отдельного объекта Circle для этих целей. В этом случае метод класса более удобен:

```
public static double area(double r) { return PI * r * r; }
```

Для класса создание нескольких методов с одним и тем же именем является абсолютно легальным до тех пор, пока методы различаются параметрами. Поскольку эта версия метода area() представляет собой метод класса, у него нет явного параметра this и должен быть параметр, указывающий радиус окружности. Этот параметр отличает его от метода экземпляра с таким же именем.

В качестве другого примера, показывающего разницу между методами класса и методами экземпляра, рассмотрим метод с именем bigger(), который сравнивает два объекта Circle и возвращает объект с большим радиусом. Мы можем оформить метод bigger() как метод экземпляра:

```
// Сравнить неявно заданную окружность «this» с окружностью
// «that», переданной явно в качестве аргумента, и вернуть большую.
public Circle bigger(Circle that) {
  if (this.r > that.r) return this;
  else return that;
}
```

Мы также можем реализовать метод bigger() как метод класса:

```
// Сравнить окружности a и b и вернуть ту, у которой радиус больше
public static Circle bigger(Circle a, Circle b) {
  if (a.r > b.r) return a;
  else return b;
}
```

Получив два объекта Circle x и y, для определения большей окружности мы можем использовать либо метод экземпляра, либо метод класса. Однако синтаксис вызова этих методов разный:

```
Circle biggest = x.bigger(y);          // Метод экземпляра: аналогично, y.bigger(x)
Circle biggest = Circle.bigger(x, y); // Статический метод
```

Оба метода работают хорошо, и с точки зрения объектно-ориентированного проектирования ни один из этих методов не считается «более правильным». Метод экземпляра формально является более объектно-ориентированным, но синтаксис его вызова страдает от некоторой асимметрии. В подобном случае выбор между методом класса и методом экземпляра является просто проектным решением. В зависимости от обстоятельств выбор одного или другого метода будет более естественным.

Тайна разгадана

Как мы видели в главах 1 и 2, вывод текста на терминал в Java осуществляется с помощью метода `System.out.println()`. В этих главах не объяснялось, почему у данного метода такое длинное и неуклюжее имя и что в этом имени означают две точки. Теперь, когда вы понимаете, что такое поля класса и экземпляра, а также методы класса и экземпляра, легко понять, что происходит. `System` является классом. У него есть поле класса с именем `out`. Поле `System.out` ссылается на объект. У объекта `System.out` есть метод экземпляра с именем `println()`. Тайна разгадана! Если вы хотите исследовать ее более тщательно, можете обратиться к описанию класса `java.lang.System` в главе 12. Из краткого обзора класса следует, что поле `out` имеет тип `java.io.PrintStream`, о котором можно прочесть в главе 11.

Создание и инициализация объектов

Взгляните еще раз на то, как мы создавали объекты `Circle`:

```
Circle c = new Circle();
```

Что здесь означают скобки? Похоже на то, что мы вызываем метод. Фактически это как раз то, что мы делаем. У каждого Java-класса есть по крайней мере один *конструктор*, который является методом с таким же именем, как и имя класса. Его назначение – выполнить всю необходимую инициализацию нового объекта. Поскольку мы не определили конструктор для нашего класса `Circle`, Java предоставляет конструктор по умолчанию, у которого нет аргументов и который не выполняет никакой особенной инициализации.

Вот как работает конструктор. Оператор `new` создает новый, но не проинициализированный экземпляр класса. Затем вызывается метод-конструктор, в который неявно передается вновь созданный объект (ссылка `this`, как мы видели ранее) и любые аргументы, указанные в скобках. Конструктор может задействовать эти аргументы для выполнения любой необходимой инициализации.

Определение конструктора

Для объектов-окружностей мы можем выполнить некоторую инициализацию, так что давайте определим конструктор. В примере 3.2 представлено новое определение класса `Circle`, которое содержит конструктор, позволяющий указать радиус нового объекта `Circle`. Конструктор также использует ссылку `this` для того, чтобы отличить параметр метода от поля экземпляра с таким же именем.

Пример 3.2. Конструктор класса Circle

```
public class Circle {
  public static final double PI = 3.14159; // Константа
  public double r;                    // Поле экземпляра, в котором хранится радиус окружности
  // Конструктор: инициализирует поле радиуса
  public Circle(double r) {
    this.r = r;
  }
  // Методы экземпляра: вычисляют значение на основе радиуса
  public double circumference() { return 2 * PI * r; }
  public double area() { return PI * r*r; }
}
```

Когда мы полагаемся на конструктор по умолчанию, предоставляемый компилятором, мы вынуждены писать код для явной инициализации радиуса:

```
Circle c = new Circle();
c.r = 0.25;
```

При использовании нового конструктора инициализация является частью создания объекта:

```
Circle c = new Circle(0.25);
```

Вот несколько важных замечаний по поводу именования, объявления и написания конструкторов:

- Имя конструктора всегда совпадает с именем класса.
- В отличие от остальных методов, конструктор объявляется без указания возвращаемого типа, даже без `void`.
- Тело конструктора должно инициализировать объект `this`.
- Конструктор не должен возвращать `this` или любое другое значение.

Определение нескольких конструкторов

Иногда нужно проинициализировать объект различными способами в зависимости от того, как это удобнее сделать в конкретных обстоятельствах. Например, можно проинициализировать радиус окружности, присвоив ему указанное значение или разумное значение по умолчанию. Поскольку у нашего класса `Circle` есть только одно поле экземпляра, то, конечно, вариантов инициализации немного. Но в более сложных классах часто удобнее определить несколько конструкторов. Вот как можно определить два конструктора для класса `Circle`:

```
public Circle() { r = 1.0; }
public Circle(double r) { this.r = r; }
```

Вполне допустимо объявить несколько конструкторов, различающихся списком параметров. По количеству и типу переданных аргументов компилятор определяет, какой конструктор вы хотите использовать. Это просто пример перегрузки, описанной ранее.

Вызов одного конструктора из другого

Ключевое слово `this` можно применять специальным образом, когда у класса есть несколько конструкторов; его можно задействовать для вызова одного из прочих конструкторов того же класса. Другими словами, мы можем переписать два предыдущих конструктора класса `Circle` так:

```
// Это базовый конструктор: инициализация радиуса
public Circle(double r) { this.r = r; }
// Этот конструктор использует this() для вызова конструктора, приведенного выше
public Circle() { this(1.0); }
```

Синтаксис `this()` — это способ вызова одного из прочих конструкторов класса. Конечно, вызываемый конструктор определяется по количеству и типу аргументов. Эта техника полезна, когда несколько конструкторов используют один и тот же фрагмент кода и нужно избежать его повторения. Разумеется, этот пример был бы более показательным, если бы версия конструктора `Circle()` с одним параметром выполняла больше действий по инициализации, чем выполняет на самом деле.

Есть важное ограничение на использование `this()`: этот вызов должен быть первым оператором конструктора. Конечно, за ним может следовать любая дополнительная инициализация, которая необходима для данной версии конструктора. Причиной подобного ограничения является автоматический вызов конструкторов родительского класса, который мы рассмотрим далее в этой главе.

Значения полей по умолчанию и инициализаторы

Не каждое поле класса требует инициализации. В отличие от локальных переменных, у которых нет значений по умолчанию и которые нельзя применять до явной инициализации, полям класса автоматически присваиваются значения по умолчанию, показанные в табл. 2.2. По существу, каждому полю базового типа по умолчанию присваивается `false` или нуль – в зависимости от типа. По умолчанию все ссылочные поля устанавливаются в `null`. Эти значения по умолчанию гарантированы Java. Если для поля необходимо значение по умолчанию, вы можете просто положиться на Java и не выполнять явную инициализацию поля. Инициализация по умолчанию применяется к полям экземпляра и к полям класса.

Как мы видели, синтаксис объявления поля класса во многом схож с синтаксисом объявления локальной переменной. Знак равенства и присваиваемое значение могут следовать за объявлением поля класса и за объявлением поля экземпляра:

```
public static final double PI = 3.14159;
public double r = 1.0;
```

Как мы обсуждали ранее, объявление переменной – это оператор, находящийся в методе Java; инициализация переменной происходит во время выполнения этого оператора. Однако объявление поля не является частью метода, поэтому не может быть выполнено как оператор. Вместо этого компилятор Java автоматически генерирует код инициализации полей экземпляра и помещает его в конструктор или конструкторы класса. Код инициализации вставляется в конструктор в том порядке, в каком поля встречаются в исходном коде. Это означает, что инициализатор поля может использовать значения полей, объявленных до него. Рассмотрим следующий код, в котором показан конструктор и два поля экземпляра некоего класса:

```
public class TestClass {
    ...
  public int len = 10;
  public int[] table = new int[len];
  public TestClass() {
    for(int i = 0; i < len; i++) table[i] = i;
  }
// Остальное пропущено
    ...
}
```

В данном случае код, сгенерированный для конструктора, эквивалентен следующему коду:

```
public TestClass() {
   len = 10;
   table = new int[len];
   for(int i = 0; i < len; i++) table[i] = i;
}
```

Если конструктор начинается вызовом `this()` другого конструктора, то код инициализации поля не добавляется в первый конструктор. Вместо этого инициализация происходит в конструкторе, вызванном выражением `this()`.

Итак, если поля экземпляра инициализируются в конструкторах, то где инициализируются поля класса? Эти поля связаны с классом. Даже если не создано ни одного экземпляра класса, они должны быть проинициализированы до того, как будет вызван конструктор. Для этого компилятор Java автоматически генерирует метод инициализации каждого класса. Поля класса инициализируются в теле этого метода, который вызывается перед первым использованием класса (зачастую, когда класс впервые загружается).[1] Как и выражения инициализации полей экземпляра, выражения инициализации полей класса вставляются в метод инициализации класса в том порядке, в котором поля появляются в исходном коде. Это означает, что выражение инициализации для поля класса может задействовать поля класса, объявленные до него. Метод инициализации класса представляет собой внутренний метод, скрытый от Java-программистов. Если вы дезассемблируете байт-код файла класса Java, вы увидите код инициализации класса в методе с именем `<clinit>`.

Блоки инициализаторов

Мы видели, что объект можно инициализировать с помощью выражений инициализации для полей и произвольного кода в конструкторе. У класса есть метод инициализации, который похож на конструктор, но мы не можем явно определить тело этого метода, как в случае с конструктором. Тем не менее Java позволяет написать произвольный код для инициализации полей класса с помощью конструкции, называемой *статическим инициализатором*. Статический инициализатор – это просто ключевое слово `static`, за которым следует блок кода в фигурных скобках. Статический инициализатор может появляться в объявлении класса в любом месте, в котором может быть расположено объявление поля или метода. Например, рассмотрим следующий код, который выполняет нетривиальную инициализацию двух полей класса:

```
// Мы можем нарисовать контур окружности, используя тригонометрические функции
// Они медленные, поэтому заранее вычислим набор значений
public class TrigCircle {
  // Это массивы и их инициализаторы
  private static final int NUMPTS = 500;
  private static double sines[] = new double[NUMPTS];
  private static double cosines[] = new double[NUMPTS];
  // Это статический инициализатор, заполняющий массив
  static {
    double x = 0.0;
    double delta_x = (Circle.PI/ 2)/(NUMPTS-1);
    for(int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
      sines[i] = Math.sin(x);
      cosines[i] = Math.cos(x);
    }
  }
// Остаток класса пропущен
    ...
}
```

[1] В действительности можно написать инициализатор класса для класса C, который вызывает метод другого класса, создающий экземпляр класса C. В этом запутанном рекурсивном случае экземпляр класса C создается перед тем, как класс будет полностью инициализирован. Впрочем, эта ситуация не является обычной в ежедневной практике.

У класса может быть любое количество статических инициализаторов. В методе инициализации класса тело каждого блока инициализаторов объединено с выражениями инициализации статических полей. Статический инициализатор похож на метод класса тем, что не может использовать ключевое слово `this`, поля экземпляра и методы экземпляра.

В Java 1.1 и последующих версиях у классов также могут быть инициализаторы экземпляра. Инициализатор экземпляра похож на статический инициализатор, за исключением того, что он инициализирует объект, а не класс. У класса может быть произвольное количество инициализаторов экземпляра и они могут появляться в любом месте, в котором определяются поля и методы. Тело каждого инициализатора экземпляра вставляется в начало каждого конструктора класса вместе с выражениями инициализации полей. Инициализатор экземпляра похож на статический инициализатор, но он не использует ключевое слово `static`. Другими словами, инициализатор экземпляра – это просто блок Java-кода, заключенный в фигурные скобки.

Инициализаторы экземпляра могут инициализировать массивы или другие поля, для которых требуется комплексная инициализация. Иногда это полезно, потому что код инициализации размещается сразу за полем, а не отделяется в конструкторе. Например:

```
private static final int NUMPTS = 100;
private int[] data = new int[NUMPTS];
{ for(int i = 0; i < NUMPTS; i++) data[i] = i;}
```

Впрочем, на практике инициализаторы экземпляра используются редко. Инициализаторы экземпляров были введены в Java для поддержки анонимных внутренних классов, и это их основное предназначение (мы обсудим анонимные внутренние классы позже в этой главе).

Подготовка к уничтожению и уничтожение объектов

Теперь, когда мы увидели, как в Java создаются и инициализируются новые объекты, нам нужно изучить другую точку жизненного цикла объекта и исследовать, как объекты подготавливаются к *уничтожению* (*finalization*) и уничтожаются. Подготовка к уничтожению – это процесс, противоположный инициализации.

Как я упомянул ранее, память, занимаемая объектом, автоматически очищается, когда объект больше не нужен. Это выполняется с помощью процесса, называемого *сборкой мусора* (*garbage collection*). Сборка мусора – это не новая техника; многие годы она применялась в таких языках, как Lisp. К ней нужно привыкнуть тем программистам, которые работали с C и C++, где для высвобождения памяти нужно вызывать функцию `free()` или оператор `delete`. Тот факт, что вам не нужно помнить о необходимости уничтожения каждого созданного объекта, является одной из особенностей Java, которая делает работу с этим языком приятной. Программы, написанные на Java, менее подвержены ошибкам, чем программы, написанные на языках, не поддерживающих автоматическую сборку мусора.

Сборка мусора

Интерпретатор Java точно знает, под какие объекты и массивы он выделил память. Кроме того, он может определить, какие локальные переменные ссылаются на те или иные объекты и массивы, а также какие объекты и массивы ссылаются на другие объекты и массивы. Поэтому интерпретатор имеет возможность определить,

когда на созданный объект больше не ссылается ни один другой объект и ни одна переменная. Если интерпретатор находит подобный объект, он знает, что может его уничтожить. Именно так он и поступает. Кроме того, сборщик мусора способен выявлять и уничтожать объекты, которые ссылаются друг на друга, но на которые не ссылается ни один другой активный объект. Все подобные циклы (замкнутые контуры) также освобождаются.

Сборщик мусора Java запускается как поток с низким приоритетом, поэтому он выполняет большую часть своей работы, когда ничего не происходит, например во время ожидания ввода данных. И лишь если объем доступной памяти чрезвычайно низок, сборщик мусора работает во время выполнения высокоприоритетных заданий (то есть это единственный случай, когда сборщик мусора замедляет работу системы). Такое бывает не слишком часто, потому что низкоприоритетный поток выполняет очистку в фоновом режиме.

Эта схема может показаться медленной и расточительной по отношению к памяти. В действительности современные сборщики мусора на удивление эффективны. Сборка мусора никогда не будет столь же эффективной, как хорошо реализованное явное выделение и высвобождение памяти. Но она делает программирование намного более легким и менее подверженным ошибкам. А для реальных программ быстрота разработки, отсутствие ошибок и простота сопровождения более важны, нежели скорость работы или эффективность распределения памяти.

Утечка памяти в Java

Тот факт, что Java поддерживает сборку мусора, значительно уменьшает количество ошибок, называемых *утечкой памяти*. Утечка памяти происходит, когда память выделяется и более не высвобождается. На первый взгляд может показаться, что сборка мусора предотвращает утечку памяти, поскольку освобождаются все неиспользуемые объекты. Утечка памяти все еще может произойти в Java, если остается действующая (но не используемая) ссылка на неиспользуемый объект. Например, когда метод выполняется долго (или бесконечно), локальные переменные этого метода могут хранить ссылки на объекты гораздо дольше, чем это необходимо. Следующий код показывает подобную ситуацию:

```
public static void main(String args[]) {
  int big_array[] = new int[100000];

  // Провести вычисления с большим массивом и получить результат.
  int result = compute(big_array);

  // Нам больше не нужен big_array. Он будет удален сборщиком мусора, когда на него не
  // останется ссылок. Поскольку big_array – это локальная переменная, то она ссылается на
  // массив до тех пор, пока метод не завершится. Но этот метод не завершается. Поэтому нам
  // нужно явно избавиться от этой ссылки, чтобы сборщик мусора узнал о том, что
  // можно освободить массив.
  big_array = null;
  // Вечный цикл, обработка ввода пользователя
  for(;;) handle_input(result);
}
```

Кроме того, утечка памяти может произойти, когда вы используете хеш-таблицу или подобную структуру данных для связи одного объекта с другим. Даже если ни один объект больше не нужен, связь остается в таблице, а объект не будет уничтожен до тех

пор, пока не будет очищена сама таблица. Если время жизни таблицы значительно превышает время жизни объектов, это может привести к утечке памяти.

Подготовка к уничтожению объектов

Финализатор (*finalizer*) является противоположностью конструктора в Java. В то время как конструктор выполняет инициализацию объекта, финализатор подготавливает объект к уничтожению. Сборщик мусора автоматически освобождает память, используемую объектами, но объект может содержать другие типы ресурсов, отличные от памяти, например открытые файлы и сетевые соединения. Сборщик мусора не может освободить за вас эти ресурсы, поэтому вам следует написать финализаторы для всех объектов, которым нужно выполнять закрытие файлов, разрыв сетевых соединений, удаление временных файлов и т. д.

Финализатор является методом экземпляра, который не принимает входных параметров и не возвращает результат. У класса может быть только один финализатор, который должен называться `finalize()`.[1] Финализатор может инициировать любое исключение или ошибку, но когда финализатор автоматически вызывается сборщиком мусора, то игнорируется любое исключение или ошибка, которые он может инициировать, а выполнение метода завершается. Обычно финализатор объявляется с модификатором `protected` (который еще не обсуждался), но он может быть объявлен и с модификатором `public`. Вот пример финализатора:

```
protected void finalize() throws Throwable {
  // Вызвать финализатор родительского класса
  // Мы еще не обсуждали родительские классы и этот синтаксис
  super.finalize();

  // Удалить временный файл, который мы использовали
  // Если файл не найден или tempfile равно null, то может
  // возникнуть исключение, но оно игнорируется.
  tempfile.delete();
}
```

Вот несколько важных моментов, касающихся финализаторов:

- Если у объекта есть финализатор, то он вызывается, когда объект больше не используется (или становится недоступным). Это происходит перед тем, как сборщик мусора освободит объект.
- Java не гарантирует, что сборщик мусора отработает или объекты будут удаляться в том же порядке, в котором создавались. Поэтому Java не гарантирует точного времени запуска финализатора (неизвестно, будет ли он запущен вообще). Java не обеспечивает определенный порядок вызова финализаторов и не задает поток, в котором будет вызван финализатор.
- Интерпретатор Java может завершить свою работу без сборки мусора, поэтому некоторые финализаторы могут быть никогда не вызваны. В этом случае все занятые ресурсы обычно высвобождаются операционной системой. В Java 1.1 метод `runFinalizersOnExit()` класса `Runtime` может заставить виртуальную машину вызвать финализаторы перед выходом. К сожалению, этот метод может вызвать взаимобло-

[1] Программисты на C++ должны обратить внимание на следующее: несмотря на то что конструктор Java называется так же, как конструктор C++, деструкторы в Java и C++ называются по-разному (finalizer и destructor). Как мы увидим, финализаторы действуют не так, как деструкторы C++.

кировку. По сути, он небезопасен, поэтому его не применяют начиная с Java 1.2. В Java 1.3 в класс Runtime был добавлен метод addShutdownHook(), который может безопасно выполнить код перед тем, как интерпретатор Java завершит работу.

- Сразу же после вызова финализатора объект не высвобождается, потому что финализатор может воскресить объект, сохранив где-нибудь ссылку this. Таким образом, на объект уже ссылаются, поэтому после вызова метода finalize() сборщик мусора должен еще раз определить, что объект не используется, прежде чем попытаться его высвободить. Впрочем, даже если объект воскрешен, финализатор не вызывается более одного раза. Воскрешение объекта никогда не было полезным – это просто странный поворот процесса уничтожения объекта. Начиная с Java 1.2 с помощью класса java.lang.ref.PhantomReference можно реализовать альтернативный финализатор, не допускающий воскрешение.

На практике метод finalize() редко востребован на уровне приложений. Финализаторы больше полезны при написании Java-классов, которые представляют собой интерфейсы к платформо-зависимому коду с вызовом native-методов. В этом случае «родные» методы могут выделять память или другие ресурсы, которые не контролируются сборщиком мусора Java и должны быть освобождены методом native finalize().

Хотя Java поддерживает инициализацию классов и экземпляров через статические инициализаторы и конструкторы, подготавливать к уничтожению можно только экземпляры. Оригинальная спецификация Java содержит метод classFinalize(), который может подготовить класс к уничтожению, когда он больше не используется и выгружается из виртуальной машины. Эта особенность никогда не была реализована. Поскольку она была признана ненужной, подготовка класса к уничтожению была удалена из спецификации языка.

Подклассы и наследование

Класс Circle, определенный ранее, является простым классом, который различает окружности только по их радиусам. Предположим, что вместо этого мы хотим представить окружность, у которой есть и размер, и местоположение. Например, окружность радиусом 1.0 с центром в точке (0, 0) в декартовых координатах отличается от окружности радиусом 1.0 и центром в точке (1, 2). Нам нужен новый класс, который мы назовем PlaneCircle. Мы хотим добавить возможность работы с положением окружности, не утратив существующей функциональности класса Circle. Для этого определим класс PlaneCircle, производный от класса Circle. Он наследует поля и методы родительского класса Circle. Возможность добавлять функциональность, создавая подкласс или расширяя исходный, является центральной в парадигме объектно-ориентированного программирования.

Расширение класса

В примере 3.3 показана реализация PlaneCircle как подкласса Circle.

Пример 3.3. Расширение класса Circle

```
public class PlaneCircle extends Circle {
  // Мы автоматически наследуем поля и методы класса Circle, поэтому нам нужно только добавить
  // новые детали.
  // Новые поля экземпляра для хранения координат окружности
  public double cx, cy;
```

```
    // Новый конструктор для инициализации новых полей
    // Он использует специальный синтаксис для вызова конструктора Circle()
    public PlaneCircle(double r, double x, double y) {
      super(r);          // Вызвать конструктор родительского класса, Circle()
      this.cx = x;       // Проинициализировать поле cx экземпляра
      this.cy = y;       // Проинициализировать поле cy экземпляра
    }

    // Методы area() и circumference() унаследованы от Circle
    // Новый метод экземпляра, который проверяет, находится ли точка внутри окружности
    // Заметим, что он использует унаследованное поле r экземпляра
    public boolean isInside(double x, double y) {
      double dx = x - cx, dy = y - cy;            // Расстояние от центра
      double distance = Math.sqrt(dx*dx + dy*dy); // Теорема Пифагора
      return (distance < r);                      // Вернуть true или false
    }
  }
```

Обратите внимание на использование ключевого слова `extends` в первой строке. Оно говорит Java о том, что класс `PlaneCircle` расширяет класс `Circle` или является его подклассом. Таким образом, он наследует поля и методы этого класса.[1] Определение метода `isInside()` демонстрирует наследование полей; данный метод использует поле `r`, определенное в классе `Circle`, так, как будто оно было определено в самом классе `PlaneCircle`. Класс `PlaneCircle` наследует методы `Circle`. Поэтому, если у нас есть объект `PlaneCircle`, на который ссылается переменная `pc`, то можно написать:

```
double ratio = pc.circumference() / pc.area();
```

Эта строка отработает так, как будто методы `area()` и `circumference()` были определены в самом классе `PlaneCircle`.

Другой особенностью подклассов является то, что каждый объект `PlaneCircle` также является абсолютно легальным объектом `Circle`. Поэтому если переменная `pc` ссылается на объект `PlaneCircle`, то мы можем присвоить ее переменной типа `Circle` и забыть все, что касается дополнительных возможностей:

```
PlaneCircle pc = new PlaneCircle(1.0, 0.0, 0.0);     // Исходная окружность
Circle c = pc;               // Присвоение значения переменной типа Circle без приведения типа
```

Присвоение объекта `PlaneCircle` переменной типа `Circle` может быть выполнено без приведения типа. Как мы обсуждали ранее, это расширяющее преобразование, которое всегда разрешено. Значение, хранимое в переменной `c` типа `Circle`, представляет собой полноценный объект `PlaneCircle`, но компилятор не может знать этого точно, поэтому он не позволяет выполнять обратное преобразование без приведения типа:

```
// Для сужающего преобразования необходимо приведение типа и проверка виртуальной машиной
// во время выполнения
PlaneCircle pc2 = (PlaneCircle) c;
boolean origininside = ((PlaneCircle) c).isInside(0.0, 0.0);
```

[1] Программисты, пишущие на C++, должны обратить внимание на то, что ключевое слово `extends` в Java эквивалентно «`:`» в C++; оба ключевых слова показывают, что класс является подклассом.

Классы с модификатором final

Когда класс объявлен с модификатором `final`, это означает, что класс не может быть расширен. Класс `java.lang.System` является примером такого класса. Объявление класса с модификатором `final` предотвращает нежелательное расширение класса и позволяет компилятору оптимизировать вызов методов класса. Более подробно эта тема будет рассмотрена далее в этой главе, когда речь пойдет о замещении методов.

Родительские классы, Object и иерархия классов

В нашем примере класс `PlaneCircle` является подклассом класса `Circle`. Можно сказать, что класс `Circle` – это родительский класс для `PlaneCircle`. Родительский класс указывается в операторе `extends`:

```
public class PlaneCircle extends Circle { ... }
```

У каждого определяемого вами класса есть родительский класс. Если вы не указали родительский класс в операторе `extends`, то его родительским классом будет класс `java.lang.Object`. Класс `Object` уникален по двум причинам:

- Это единственный класс в Java, у которого нет родителя.
- Все классы Java наследуют методы класса `Object`.

Так как у каждого класса есть родитель, то классы в Java образуют иерархию классов, которая может быть представлена в виде дерева с классом `Object` в качестве корня. На рис. 3.1 представлена диаграмма иерархии классов, включающая наши классы `Circle` и `PlaneCircle` наряду с некоторыми стандартными классами из Java API.

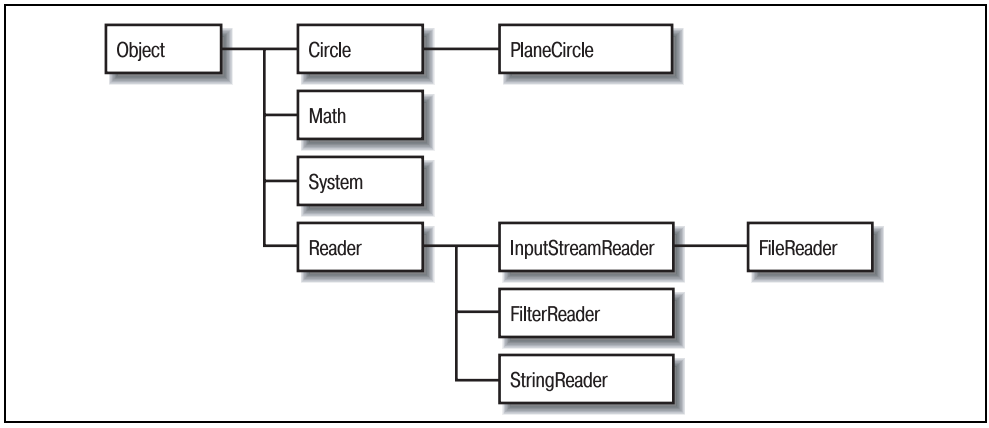


Рис. 3.1. *Диаграмма иерархии классов*

Конструкторы подклассов

Посмотрите еще раз на конструктор класса `PlaneCircle()`:

```
public PlaneCircle(double r, double x, double y) {
  super(r);    // Вызвать конструктор родительского класса, Circle()
  this.cx = x; // Проинициализировать поле cx экземпляра
  this.cy = y; // Проинициализировать поле cy экземпляра
}
```

Этот конструктор явно инициализирует поля `cx` и `cy`, объявленные в классе `PlaneCircle`, но в инициализации унаследованных полей класса он полагается на конструктор родительского класса `Circle()`. Для его вызова наш конструктор применяет метод `super()`. `super` — это зарезервированное слово в Java. Одним из вариантов его использования является вызов конструктора родительского класса из конструктора подкласса. Такой подход аналогичен использованию метода `this()` для вызова одного конструктора класса из другого конструктора того же класса. На применение метода `super()` для вызова конструктора родительского класса налагаются те же ограничения, что и на использование метода `this()`:

- Метод `super()` можно применять только в конструкторах.
- Вызов конструктора родительского класса должен быть первым оператором конструктора. Он должен происходить перед объявлением локальных переменных.

Аргументы, переданные в `super()`, должны соответствовать параметрам конструктора родительского класса. Если родительский класс определяет несколько конструкторов, метод `super()` можно применять для вызова любого из них в зависимости от переданных аргументов.

Цепочки конструкторов и конструктор по умолчанию

Java гарантирует, что конструктор класса будет вызываться при каждом создании экземпляра класса. Конструктор будет вызываться всякий раз при создании экземпляра подкласса. Чтобы гарантировать второе утверждение, Java обеспечивает порядок, согласно которому каждый конструктор будет вызывать родительский конструктор. Поэтому, если первый оператор в конструкторе не вызывает другой конструктор с помощью `this()` или `super()`, то Java неявно вставляет вызов `super()`, то есть вызывает родительский конструктор без аргументов. Если у родительского класса нет конструктора без аргументов, то подобный неявный вызов приводит к ошибке компиляции.

Рассмотрим, что происходит, когда мы создаем новый экземпляр класса `PlaneCircle`. Во-первых, вызывается конструктор класса `PlaneCircle`. Этот конструктор явно вызывает конструктор класса `Circle` с помощью оператора `super(r)`, а тот в свою очередь с помощью `super()` неявно вызывает конструктор родительского класса `Object`. Сначала выполняются операторы конструктора класса `Object`, а затем операторы конструктора `Circle()`. И наконец, когда вызов `super(r)` завершен, выполняются оставшиеся операторы конструктора `PlaneCircle()`.

Все это означает, что вызовы конструкторов объединяются в цепочку; при каждом создании объекта вызывается последовательность конструкторов: конструктор подкласса, конструктор родительского класса и далее вверх по иерархии классов до конструктора класса `Object`. Так как конструктор родительского класса всегда вызывается первым оператором конструктора подкласса, то операторы конструктора класса `Object` всегда выполняются первыми. Затем выполняются операторы конструктора подкласса и далее вниз по иерархии классов вплоть до конструктора класса, объект которого создается. Здесь есть важное следствие: когда вызван конструктор, он может положиться на то, что поля родительского класса уже проинициализированы.

Конструктор по умолчанию

В предыдущем описании образования цепочек конструкторов был упущен один важный момент. Если конструктор не вызывает конструктор родительского класса, то Java делает это неявно. А если класс объявлен без конструктора? В этом случае Java неявно добавляет конструктор. Конструктор по умолчанию не делает ничего, кроме

вызова родительского конструктора. Например, если мы не объявим конструктор для класса PlaneCircle, то Java неявно вставит следующий конструктор:

```
public PlaneCircle() { super(); }
```

Если в родительском классе Circle не объявлен конструктор без аргументов, то вызов super() в конструкторе класса PlaneCircle(), вставленном автоматически, вызовет ошибку компиляции. В общем, если класс не определяет конструктор без аргументов, то все его подклассы должны определять конструкторы, которые явно вызывают конструктор родительского класса с необходимыми аргументами.

Если класс не объявляет ни одного конструктора, то для него по умолчанию создается конструктор без аргументов. Классы, объявленные с указанием модификатора public, получают конструкторы с модификатором public. Все остальные классы получают конструктор по умолчанию, который объявляется без каких бы то ни было модификаторов видимости (понятие «видимость» объясняется позже в этой главе). Если вы создаете класс с модификатором public, экземпляры которого нельзя создавать вне класса, то вы должны объявить хотя бы один конструктор без модификатора public. Тогда удастся избежать добавления конструктора по умолчанию с модификатором public. Классы, которые вообще не должны иметь ни одного экземпляра (например, java.lang.Math или java.lang.System), должны определять конструктор с модификатором private. Такой конструктор нельзя вызвать вне класса, но данная схема позволяет избежать добавления конструктора по умолчанию.

Цепочки финализаторов?

Вы можете предположить, что поскольку Java сцепляет вызовы конструкторов, то автоматически сцепляются и вызовы финализаторов для объекта. Другими словами, вы можете предположить, что финализатор класса автоматически вызывает финализатор родительского класса и т. д. В действительности Java такого *не* делает. Когда вы пишете метод finalize(), вы должны явно вызвать финализатор родительского класса. Это необходимо, даже если вы знаете, что у родительского класса нет финализатора, потому что финализатор может быть добавлен в будущие реализации класса.

Как мы видели в примере финализатора, представленного ранее в этой главе, можно вызвать метод родительского класса, используя специальный синтаксис с ключевым словом super:

```
// Вызвать финализатор родительского класса
super.finalize();
```

Мы обсудим этот синтаксис более подробно при рассмотрении замещения методов. На практике необходимость применения финализаторов и, соответственно, сцепления финализаторов, возникает редко.

Затенение полей родительского класса

Представим, что наш PlaneCircle должен знать расстояние между центром окружности и началом координат (0, 0). Мы можем добавить поле экземпляра для хранения этого значения:

```
public double r;
```

Добавим следующую строку для вычисления значения поля:

```
this.r = Math.sqrt(cx*cx + cy*cy);      // Теорема Пифагора
```

Но подождите, это новое поле r называется так же, как поле радиуса r в родительском классе Circle. В таких случаях мы говорим, что поле r класса PlaneCircle *затеняет* (shadows) поле r родительского класса Circle. (Конечно, это искусственный пример: в действительности новое поле должно быть названо distanceFromOrigin. Вы должны избегать подобных ситуаций, хотя поля подклассов иногда затеняют поля родительских классов.)

В новом определении класса PlaneCircle оба выражения — r и this.r — ссылаются на поле класса PlaneCircle. Как тогда мы можем сослаться на поле r класса Circle, которое хранит радиус окружности? Для этого существует специальный синтаксис, использующий ключевое слово super:

```
r                   // Ссылается на поле класса PlaneCircle
this.r              // Ссылается на поле класса PlaneCircle
super.r             // Ссылается на поле класса Circle
```

Другой способ сослаться на затененное поле — привести this (или любой экземпляр класса) к соответствующему родительскому классу и обратиться к полю:

```
((Circle) this).r   // Ссылается на поле r класса Circle
```

Эта техника приведения полезна, когда вам нужно сослаться на затененное поле, определенное в классе, который не является непосредственным родительским классом. Предположим, что все классы A, B и C определяют поле x и что класс C — родитель для класса B, который в свою очередь является родителем для класса A. Тогда в методах класса C вы можете ссылаться на эти поля:

```
x                   // Поле x класса C
this.x              // Поле x класса C
super.x             // Поле x класса B
((B)this).x         // Поле x класса B
((A)this).x         // Поле x класса A
super.super.x       // Неправильно; не ссылается на поле x класса A
```

Вы не можете ссылаться на затененное поле x в родителе родителя с помощью вызова super.super.x. Это неправильный синтаксис.

Точно так же, если есть экземпляр c класса C, то вы можете ссылаться на три поля x:

```
c.x                 // Поле x класса C
((B)c).x            // Поле x класса B
((A)c).x            // Поле x класса A
```

До настоящего времени мы обсуждали поля экземпляров. Поля класса также могут быть затенены. Вы можете использовать тот же синтаксис super для того, чтобы сослаться на значение этого поля, но это никогда не нужно, поскольку вы всегда можете сослаться на поле класса, указав перед ним имя необходимого класса. Положим, что автор класса PlaneCircle решил, что поле Circle.PI недостаточно точно отражает число π. Он может определить собственное поле класса PI:

```
public static final double PI = 3.14159265358979323846;
```

Теперь в коде класса PlaneCircle можно задействовать более точное значение, написав PI или PlaneCircle.PI. Можно также сослаться на старое, менее точное значение, написав super.PI или Circle.PI. Обратите внимание, что методы area() и circumference(), унаследованные классом PlaneCircle, определены в классе Circle. Именно поэтому они используют значение Circle.PI даже несмотря на то, что это поле затенено полем PlaneCircle.PI.

Замещение методов родительского класса

Когда класс определяет метод экземпляра с таким же именем, типом возвращаемого значения и параметрами, что и метод его родительского класса, то данный метод *замещает* (overrides) метод родительского класса. Когда этот метод выполняется для экземпляра класса, то вызывается новое определение, а не старое определение родительского класса.

Замещение методов – это важная и полезная техника объектно-ориентированного программирования. Класс `PlaneCircle` не замещает методов, определенных в классе `Circle`. Предположим, что мы определили подкласс `Circle` с именем `Ellipse`.[1]

В этом случае для класса `Ellipse` важно переопределить методы `area()` и `circumference()` класса `Circle`, поскольку формулы, применяемые для вычисления площади и периметра окружности, не работают для эллипса.

Следующее обсуждение замещения методов касается только методов экземпляра. Методы класса ведут себя несколько по-другому, и сказать о них можно не так много. Как и поля, методы класса можно затенить методами подкласса, но они не могут быть замещены. Как я заметил ранее в этой главе, полезно предварять вызов метода именем класса, в котором определен этот метод. Если рассматривать имя класса как часть имени метода, то эти два метода будут иметь разные имена, поэтому в действительности ничего не затенено. Метод класса не может затенять метод экземпляра.

Перед тем как продолжить обсуждение замещения методов, вы должны понять разницу между замещением и перегрузкой методов. Как мы обсуждали в главе 2, перегрузка методов – это определение нескольких методов (в одном и том же классе) с одинаковыми именами, но разными параметрами. Эта практика сильно отличается от замещения методов, поэтому не путайте их.

Замещение – это не затенение

Несмотря на то что во многих случаях Java одинаково работает с полями и методами, замещение методов совсем не похоже на затенение полей. Вы можете сослаться на затененное поле, приведя тип объекта к необходимому родительскому классу, но с помощью этой техники вы не сможете вызвать замещенный метод экземпляра. Следующий код иллюстрирует это важное различие:

```
class A {                           // Определить класс с именем A
  int i = 1;                        // Поле экземпляра
  int f() { return i; }             // Метод экземпляра
  static char g() { return 'A'; }   // Метод класса
}

class B extends A {                 // Определить подкласс класса A
  int i = 2;                        // Затеняет поле i класса A
  int f() { return -i; }            // Замещает метод f экземпляра класса A
  static char g() { return 'B'; }   // Затеняет метод g() класса A
}

public class OverrideTest {
```

[1] Борцы за чистоту математики могут возразить, что все окружности являются эллипсами и что класс `Ellipse` должен быть родительским классом для класса `Circle`. Однако прагматичный инженер оспорит это утверждение, сказав, что окружности могут быть представлены меньшим количеством полей экземпляра, поэтому класс `Circle` не стоит обременять наследованием ненужных полей класса `Ellipse`. В любом случае это полезный пример.

```
  public static void main(String args[]) {
    B b = new B();                    // Создать новый объект типа B
    System.out.println(b.i);          // Ссылается на B.i; выводит 2
    System.out.println(b.f());        // Ссылается на B.f(); выводит -2
    System.out.println(b.g());        // Ссылается на B.g(); выводит B
    System.out.println(B.g());        // Так лучше вызывать B.g()

    A a = (A) b;                      // Привести b к экземпляру класса A
    System.out.println(a.i);          // Теперь ссылается на A.i; выводит 1
    System.out.println(a.f());        // Все еще ссылается на B.f(); выводит -2
    System.out.println(a.g());        // Ссылается на A.g(); выводит A
    System.out.println(A.g());        // Так лучше вызывать A.g()
  }
}
```

Если разница между замещением методов и затенением полей остается неясной, то непродолжительные размышления все расставят на свои места. Предположим, у нас есть объекты `Circle` и `Ellipse`. Для отслеживания окружностей и эллипсов мы используем массив типа `Circle[]`. Это возможно, поскольку класс `Ellipse` является подклассом `Circle`, а все объекты класса `Ellipse` – это легальные объекты класса `Circle`. Когда мы перебираем элементы этого массива, нам не нужно знать или заботиться о том, что представляет собой элемент – экземпляр класса `Circle` или экземпляр класса `Ellipse`. Что нам действительно очень важно, так это вычисление правильного значения при вызове метода `area()` для любого элемента массива. Другими словами, мы не хотим использовать формулу для вычисления площади окружности, когда объект в действительности является эллипсом! В таком контексте неудивительно, что Java обрабатывает замещение методов не так, как затенение полей.

Динамический поиск методов

Если у нас есть массив `Circle[]`, в котором хранятся объекты `Circle` и `Ellipse`, то как компилятор определяет, какой метод вызывать для каждого элемента массива – метод `area()` класса `Circle` или класса `Ellipse`? В действительности компилятор этого не знает, потому что не может знать. Тем не менее компилятору известно о своем незнании. Он создает код, который использует динамический поиск методов в процессе выполнения.

Когда интерпретатор выполняет код, он находит соответствующий метод `area()`, который нужно вызвать для каждого объекта в массиве. То есть когда интерпретатор разбирает выражение `o.area()`, он проверяет действительный тип объекта, на который ссылается переменная `o`, и находит метод `area()`, соответствующий этому типу. Он не просто использует метод `area()`, статически связанный с переменной `o`, а проводит поиск. Процесс динамического поиска методов иногда называют виртуальным вызовом методов.[1]

Методы с модификатором final и поиск статических методов

Виртуальный вызов методов быстр, но обычный вызов методов еще быстрее, так как не требует динамического поиска во время выполнения. К счастью, существует ряд условий, при которых Java не нуждается в динамическом поиске методов. В частнос-

[1] Программисты, пишущие на C++, должны обратить внимание, что в C++ динамический поиск методов представлен функциями `virtual`. Важное отличие между Java и C++ состоит в том, что в Java нет ключевого слова `virtual`. В Java методы являются виртуальными по умолчанию.

ти, если метод объявлен с модификатором final, то это означает, что объявление метода окончательное; он не может быть замещен ни в одном из подклассов. Если метод не может быть замещен, то компилятор знает, что существует только одна версия этого метода и динамический поиск метода не нужен.[1]

Кроме того, все методы класса с модификатором final неявно объявлены точно с таким же модификатором и не могут быть замещены. Как мы увидим далее в этой главе, методы с модификатором private не наследуются подклассами, поэтому они не могут быть замещены (то есть все методы с модификатором private неявно объявляются с модификатором final). И наконец, методы класса ведут себя как поля (то есть они могут быть затенены в подклассе, но их нельзя подменить). Таким образом, все методы, которые объявлены с модификатором final, и методы, объявленные как final, private или static, вызываются без динамического поиска методов. Эти методы являются кандидатами на встраивание во время работы компилятора JIT (just-in-time) или похожего инструмента оптимизации.

Вызов замещенных методов

Мы видели важную разницу между замещением методов и затенением полей. Тем не менее синтаксис вызова замещенных методов очень похож на синтаксис доступа к затененным полям: в обоих случаях применяется ключевое слово super. Следующий код это иллюстрирует:

```
class A {
  int i = 1;                  // Поле экземпляра, затененное подклассом B
  int f() { return i; }       // Метод экземпляра, замещенный подклассом B
}

class B extends A {
  int i;                      // Это поле затеняет поле i класса A
  int f() {                   // Этот метод замещает метод f() в классе A
    i = super.i + 1;          // Так он может получить значение A.i
    return super.f() + i;     // Так он может вызвать метод A.f()
  }
}
```

Вспомните, что при использовании ключевого слова super для ссылки на затененное поле вы делаете то же самое, что и при приведении ссылки this к родительскому классу и получении доступа к его полям. Однако использование super для вызова замещенных методов отнюдь не то же самое, что приведение this. Другими словами, в предыдущем коде выражение super.f() отличается от выражения ((A)this).f().

Когда интерпретатор вызывает метод экземпляра с помощью синтаксиса super, то динамический поиск выполняется в модифицированной форме. Первый шаг такой же, как и при обычном поиске, – определение действительного класса объекта, метод которого вызван. Обычно динамический поиск соответствующего определения метода начинается с этого класса. Когда метод вызывается с помощью синтаксиса super, поиск начинается с родительского класса. Если родительский класс непосредственно реализует данный метод, то вызывается эта версия метода. Если родительский класс наследует метод, то вызывается унаследованная версия метода.

Обратите внимание, что ключевое слово super вызывает замещенную версию метода. Предположим, у класса A есть подкласс B, у которого есть подкласс C, и все три класса

[1] В этом смысле модификатор final противоположен модификатору virtual в C++. В Java все методы, объявленные без модификатора final, по умолчанию являются виртуальными.

определяют один и тот же метод `f()`. Тогда метод `C.f()` может вызвать метод `B.f()`, непосредственно замещенный с помощью выражения `super.f()`. Но в методе `C.f()` нет способа напрямую вызвать метод `A.f()`: в Java синтаксис `super.super.f()` запрещен. Конечно, если метод `C.f()` вызывает метод `B.f()`, то резонно полагать, что метод `B.f()` может вызвать метод `A.f()`. Такой вид связывания относительно часто применяется при работе с замещенными методами: это способ расширения функциональности метода без изменения самого метода. Мы видели эту технику в примере метода `finalize()`, представленном ранее в этой главе: данный метод вызывал метод `super.finalize()` для вызова родительского финализатора.

Не путайте применение ключевого слова `super` для вызова замещенного метода с вызовом `super()`, используемым в конструкторах для вызова родительских конструкторов. Эти методы применяют одно и то же ключевое слово, но два совершенно разных синтаксиса. В действительности вы можете использовать `super` для вызова замещенного метода в любом месте замещающего класса, а `super()` можно применять только для вызова родительского конструктора, указав его в первом операторе.

Также важно помнить, что `super` можно применять только для вызова замещенного метода из замещающего класса. Если задан объект `e` класса `Ellipse`, то в программе, применяющей этот объект (с использованием или без использования синтаксиса `super`), не существует способа вызвать метод `area()`, определенный в классе `Circle`.

Я уже объяснял, что методы класса могут затенять методы родительского класса, но не могут их замещать. При вызове методов класса предпочтительнее указывать имя класса перед именем метода. В противном случае вы можете использовать синтаксис `super` для вызова затененных методов класса, как если бы вы вызывали метод экземпляра или ссылались на затененное поле.

Сокрытие данных и инкапсуляция

Мы начали эту главу с описания класса, представив его как «коллекцию данных и методов». Одной из важных объектно-ориентированных техник, которую мы еще не обсуждали, является сокрытие данных внутри класса и получение доступа к ним с помощью методов. Эта техника известна как *инкапсуляция* (*encapsulation*), поскольку она безопасно изолирует данные (и внутренние методы) внутри «капсулы», или класса, где доступ к ним могут получать только доверенные пользователи (то есть методы класса).

Зачем это может вам понадобиться? Самая важная причина – сокрытие деталей внутренней реализации вашего класса. Если работа с данным классом не связана с погружением в эти детали, то вы можете изменять реализацию, не беспокоясь о том, что нарушите работоспособность кода, который использует этот класс.

Другая причина для инкапсуляции – это желание защитить ваш класс от случайной или намеренной глупости. Класс часто содержит несколько взаимосвязанных полей, значения которых должны быть согласованы. Если вы позволите программисту (включая самого себя) непосредственно манипулировать этими полями, то он сможет изменить одно поле без изменения связанных полей, тем самым сделав класс противоречивым. Если вместо этого он вызовет метод для изменения поля, этот метод выполнит все необходимое для согласования. Точно так же, если класс объявляет определенные методы для внутреннего использования, то сокрытие этих методов предотвращает их использование извне.

Вот другой взгляд на инкапсуляцию: когда все данные класса скрыты, методы определяют единственный возможный способ работы с экземпляром класса. Тщательно

протестировав и отладив методы, вы можете быть уверены в том, что класс будет работать как ожидалось. С другой стороны, если всеми полями класса можно будет манипулировать напрямую, то количество вариантов, которые нужно будет протестировать, станет неуправляемым.

Кроме того, существуют другие причины скрывать поля и методы класса:

- Внутренние поля и методы, видимые извне, просто увеличивают неразбериху в API. Минимизация количества видимых полей делает ваш класс более аккуратным и, следовательно, более понятным и удобным.
- Если поле или метод видны пользователю класса, то вы должны его задокументировать. Берегите собственное время и усилия – скрывайте поля и методы вместо того, чтобы их документировать.

Контроль доступа

Все поля и методы класса можно применять внутри тела самого класса. Java определяет правила контроля доступа, которые ограничивают использование членов класса вне этого класса. В примерах данной главы вы увидите, как используется модификатор `public` в объявлениях полей и методов. Ключевые слова `public`, `protected` и `private` являются модификаторами контроля доступа; они определят правила доступа к полям и методам.

Доступ к пакетам

Пакет всегда доступен для кода, определенного внутри пакета. Доступность данного пакета коду из других пакетов зависит от того, как пакет развернут на сервере. Например, когда файлы классов, входящих в пакет, хранятся в каталоге, то для доступа к пакету пользователь должен иметь права на чтение каталога и файлов. Доступ к пакетам не является сам по себе частью языка Java. Вместо этого контроль доступа обычно выполняется на уровне классов и членов классов.

Доступ к классам

По умолчанию классы верхнего уровня доступны в том пакете, в котором они определены. Впрочем, если класс верхнего уровня объявлен как `public`, то он доступен везде (или везде, где доступен сам пакет). Мы ограничили это утверждение классами верхнего уровня, потому что классы могут быть объявлены как члены других классов. Так как эти внутренние классы являются членами класса, то они подчиняются правилам контроля доступа к членам класса.

Доступ к членам класса

Как я уже говорил, члены класса всегда доступны внутри тела класса. По умолчанию члены класса также доступны в пакете, в котором класс определен. Это означает, что классы, размещенные в одном пакете, должны доверять друг другу в деталях своей внутренней реализации. Уровень доступа по умолчанию часто называют *пакетным доступом*. Это только один из четырех возможных уровней доступа. Другие три уровня доступа определяются модификаторами `public`, `protected` и `private`. Вот пример кода, использующего модификаторы:

```
public class Laundromat {              // Этот класс можно использовать.
  private Laundry[] dirty;             // Это внутреннее поле нельзя использовать,
  public void wash() { ... }           // но можно применять открытые методы
  public void dry() { ... }            // для работы с внутренним полем.
}
```

Вот правила доступа к членам класса:

- Если член класса объявлен с модификатором `public`, то это означает, что он доступен в любом месте, в котором доступен сам класс. Это наименее ограниченный тип контроля доступа.
- Если член класса объявлен с модификатором `private`, то он нигде не доступен, кроме самого класса. Это наиболее ограниченный тип контроля доступа.
- Если член класса объявлен с модификатором `protected`, то он доступен всем классам внутри пакета (так же как и в случае контроля по умолчанию) и в теле любых подклассов этого класса, вне зависимости от пакета, в котором определен подкласс. Этот тип доступа накладывает более сильные ограничения, чем тип доступа `public`, но менее сильные, чем пакетный доступ.
- Если член класса объявлен без модификаторов, то он получает пакетный тип доступа по умолчанию: он доступен в классах, определенных в том же пакете, но не доступен вне пакета.

Тип доступа `protected` требует пояснения. Предположим, что в классе `A` объявлено `protected`-поле `x`, а этот класс является родительским классом для класса `B`, который объявлен в другом пакете (последнее утверждение важно). Класс `B` наследует `protected`-поле `x` и получает доступ к этому полю в текущем экземпляре класса `B` и любых других экземплярах, на которые может ссылаться код. Это не означает, что код класса `B` может читать защищенное поле произвольного экземпляра класса `A`! Если объект представляет собой экземпляр класса `A`, но не является экземпляром класса B, то эти поля не наследуются классом `B`, а код класса `B` не может их прочесть.

Контроль доступа и наследование

Согласно спецификации Java, подкласс наследует все доступные ему поля и методы экземпляра родительского класса. Если подкласс определен в том же пакете, что и родительский класс, то он наследует все поля и методы экземпляра, не объявленные как `private`. Если подкласс определен в другом пакете, то он наследует все `protected`- и `public`-поля и методы экземпляра. Поля и методы `private` никогда не наследуются, так же как и поля и методы класса. И наконец, конструкторы не наследуются; они сцепляются, как было объяснено ранее в этой главе.

Утверждение, согласно которому подкласс не наследует недоступные поля и методы своего родительского класса, может сбить с толку. Может показаться, что если вы создаете экземпляр подкласса, то для `private`-полей, объявленных в родительском классе, не выделяется память. Впрочем, это не было целью утверждения. На практике каждый экземпляр подкласса включает в себя полный экземпляр родительского класса, в том числе все недоступные поля и методы. Просто все дело в терминологии. Так как недоступные поля не могут использоваться в подклассе, то мы говорим, что они не наследуются. Ранее в этой главе я утверждал, что члены класса всегда доступны в теле класса. Если это утверждение применить ко всем членам класса, включая унаследованные, то мы можем сказать, что «унаследованные члены класса» включают в себя только доступные члены. Представим это определение следующим образом:

- Класс наследует *все* поля и методы экземпляра (но не конструкторы) своего родительского класса.
- Из тела класса всегда можно получить доступ ко всем полям и методам, которые класс объявляет сам. Кроме того, можно обратиться к *доступным* полям и методам, унаследованным от родительского класса.

Доступ к членам класса: сводка

Таблица 3.1 обобщает правила доступа к членам класса.

Таблица 3.1. Доступность членов класса

	Видимость членов класса			
Доступен для	**public**	**protected**	**package**	**private**
Определяющий класс	Да	Да	Да	Да
Класс в том же пакете	Да	Да	Да	Нет
Подкласс в другом пакете	Да	Да	Нет	Нет
Не подкласс в другом пакете	Да	Нет	Нет	Нет

Вот несколько простых правил по использованию модификаторов видимости:

- Используйте модификатор `public` только для методов и констант, формирующих открытый API класса. Некоторые важные или часто используемые поля также могут быть `public`, но чаще всего поля объявляются без модификатора `public` и инкапсулируются с помощью `public`-методов доступа.
- Используйте модификатор `protected` для полей и методов, которые не нужны большинству программистов, использующих класс, но которые могут заинтересовать программистов, создающих подклассы в другом пакете. Обратите внимание, что `protected`-члены класса формально являются частью API, экспортируемого классом. Они должны быть задокументированы. Их изменение может привести к нарушению кода, который их применяет.
- Используйте пакетную видимость по умолчанию для полей и классов, которые являются деталями внутренней реализации, но применяются взаимодействующими классами в этом же пакете. Вы не получите реальных преимуществ от использования пакетной видимости, пока не начнете применять директиву `package` для группировки взаимодействующих классов в пакет.
- Указывайте модификатор `private` для полей и методов, которые используются только внутри класса и должны быть скрыты в любом другом месте.

Если вы не сомневаетесь в том, что лучше использовать — `protected`, пакетную видимость или `private`, — лучше выбрать избыточное ограничение доступа к членам класса. Вы всегда сможете смягчить ограничения доступа в будущих версиях класса, если это необходимо. Обратный подход не является хорошим выбором, поскольку усиление ограничений доступа — это не обратно-совместимое изменение.

Методы доступа к данным

В примере класса `Circle` мы объявили радиус окружности как `public`-поле. Класс `Circle` является классом, в котором поле намеренно сделано открытым; это достаточно простой класс, в нем нет зависимостей между полями. С другой стороны, текущая реализация класса позволяет установить отрицательное значение радиуса у объекта `Circle`, а окружностей с отрицательным радиусом просто не существует. Так как радиус хранится в `public`-поле, любой программист может установить произвольное значение этого поля вне зависимости от того, насколько это логично. Единственное решение — запретить программисту прямой доступ к полю и определить `public`-метод, который предоставит косвенный доступ. Предоставление `public`-методов чтения и записи — это не то же самое, что объявление поля `public`. Ключевое отличие состоит в том, что метод может выполнить проверку на ошибки.

Пример 3.4 показывает другую реализацию класса Circle, позволяющую избежать появления окружностей с отрицательным радиусом. Эта версия класса Circle объявляет поле r как protected и определяет методы доступа с именами getRadius() и setRadius(), выполняющие операции чтения и записи значения поля с одновременной проверкой на отрицательное значение радиуса. Так как поле r объявлено protected, то оно непосредственно (и более эффективно) доступно подклассам.

Пример 3.4. Класс Circle, использующий сокрытие данных и инкапсуляцию

```
package shapes;                          // Указать пакет для класса

public class Circle {                    // Класс все еще public
  // Это важная константа, так что мы оставим ее public
  public static final double PI = 3.14159;

  protected double r;                    // Радиус скрыт, но виден подклассам

  // Метод для ограничения радиуса
  // Это детали реализации, которые могут быть интересны подклассам
  protected void checkRadius(double radius) {
    if (radius < 0.0)
      throw new IllegalArgumentException("радиус не может быть отрицательным.");
  }

// Конструктор
  public Circle(double r) {
    checkRadius(r);
    this.r = r;
  }

  // Открытые методы доступа
  public double getRadius() { return r; }
  public void setRadius(double r) {
    checkRadius(r);
    this.r = r;
  }

  // Методы для работы с полем экземпляра
  public double area() { return PI * r * r; }
  public double circumference() { return 2 * PI * r; }
}
```

Мы определили класс Circle в пакете с именем shapes. Так как поле r является protected-полем, то любой другой класс в пакете shapes имеет прямой доступ к этому полю и может задавать его значение. Здесь мы полагаем, что все классы в пакете shapes были написаны одним автором или группой тесно взаимодействующих авторов. Предположим также, что все эти классы доверяют друг другу в том, что они не нарушат уровня доступа к деталям реализации друг друга.

И наконец, код, ограничивающий возможность присвоения отрицательного значения радиуса, помещен в protected-метод checkRadius(). Пользователи класса Circle не могут непосредственно вызвать этот метод, а подклассы могут его вызвать и даже заменить, если захотят установить новое ограничение на радиус.

Обратите особое внимание на методы getRadius() и setRadius() в примере 3.4. В Java принято начинать названия методов доступа к данным префиксами «get» и «set». Если метод возвращает значение типа boolean, то метод доступа «get» может быть заменен на эквивалентный метод, начинающийся с «is». Например, метод доступа к bo-

olean-полю с именем readable обычно называют isReadable(), а не getReadable(). В программных соглашениях компонентной модели JavaBeans, описанной в главе 6, *свойством* (*property*) называют скрытое поле с одним или несколькими методами доступа к данным, имя которого начинается с «get», «is» или «set». Интересным способом изучить сложный класс является анализ его свойств. Свойства – это обычное понятие для AWT и Swing API, которые описаны в книге «Java Foundation Classes in a Nutshell» (O'Reilly).

Абстрактные классы и методы

В примере 3.4 мы объявили наш класс Circle частью пакета shapes. Предположим, нужно определить несколько подобных классов-форм: Rectangle, Square, Ellipse, Triangle и т. д. У наших классов может быть два базовых метода: area() и circumference(). Теперь для упрощения работы с массивами форм полезно предусмотреть, чтобы все наши формы имели общего родителя – класс Shape. Если мы выстроим иерархию классов подобным образом, то любая фигура, вне зависимости от того, какую конкретно форму она представляет, может быть присвоена переменной, полю или элементу массива типа Shape. Мы хотим, чтобы класс Shape инкапсулировал особенности, присущие всем нашим фигурам, например методы area() и circumference(). Но наш общий класс Shape в действительности не представляет ни одной реальной фигуры, поэтому он не может определять полезных реализаций методов. Java обрабатывает подобные ситуации с помощью *абстрактных методов*.[1]

Java позволяет определить метод без реализации, объявив его с модификатором abstract. У абстрактного метода нет тела; у него есть только заголовок, заканчивающийся точкой с запятой. Вот правила для абстрактных методов и абстрактных классов, которые их содержат:

- Любой класс с абстрактным методом автоматически становится абстрактным и должен быть объявлен как abstract.
- Нельзя создавать ни одного экземпляра абстрактного класса.
- Экземпляры подклассов абстрактного класса, можно создавать только в том случае, если все методы, объявленные как abstract, замещены и реализованы (то есть имеют тело). Такие классы часто называют *реальными*, чтобы подчеркнуть тот факт, что они не абстрактные.
- Если подкласс абстрактного класса, не реализует всех методов, объявленных как abstract, то этот подкласс сам является абстрактным.
- Методы, объявленные как static, private и final, не могут быть объявлены как abstract, поскольку такие методы не могут быть замещены подклассами. Точно так же класс, объявленный как final, не может содержать методов, объявленных как abstract.
- Класс может быть объявлен как abstract, даже если он не содержит ни одного абстрактного метода. Такое объявление означает, что реализация класса все еще не закончена и класс будет служить родителем для одного или нескольких

[1] Абстрактный метод в Java похож на чистую виртуальную функцию в C++ (то есть виртуальную функцию, объявленную как = 0). В C++ класс, содержащий виртуальную функцию, называют абстрактным классом, и ни одного экземпляра такого класса не может быть создано. То же самое верно и для классов Java, которые содержат абстрактные методы.

подклассов, которые завершат реализацию. Экземпляр такого класса не может быть создан.

Есть одна важная особенность этих правил. Если мы определим класс Shape с методами area() и circumference(), объявленными как abstract, то любой подкласс должен будет реализовать данные методы с тем, чтобы можно было создавать экземпляры этого класса. Другими словами, у каждого объекта Shape обязательно есть реализации этих методов. В примере 3.5 показано, как это работает. В нем объявлен абстрактный класс Shape с двумя реальными подклассами.

Пример 3.5. Абстрактный класс и реальные подклассы

```
public abstract class Shape {
  public abstract double area();          // Абстрактный метод: обратите внимание, что
  public abstract double circumference(); // вместо тела стоит точка с запятой.
}

class Circle extends Shape {
  public static final double PI = 3.14159265358979323846;
  protected double r;                       // Данные экземпляра
  public Circle(double r) { this.r = r; }  // Конструктор
  public double getRadius() { return r; }  // Метод доступа
  // Реализация абстрактных методов.
  public double area() { return PI*r*r; }
  public double circumference() { return 2*PI*r; }
}

class Rectangle extends Shape {
  protected double w, h;                    // Данные экземпляра
  public Rectangle(double w, double h) {   // Конструктор
    this.w = w; this.h = h;
  }
  public double getWidth() { return w; }   // Метод доступа
  public double getHeight() { return h; }  // Еще один метод доступа
  // Реализация абстрактных методов.
  public double area() { return w*h; }
  public double circumference() { return 2*(w + h); }
}
```

В конце каждого абстрактного метода класса Shape после скобок стоит точка с запятой. У таких методов нет фигурных скобок и не определено тело метода. Используя классы, определенные в примере 3.5, мы можем написать следующий код:

```
Shape[] shapes = new Shape[3];            // Создать массив для хранения фигур
shapes[0] = new Circle(2.0);              // Заполнить массив
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
double total_area = 0;
for(int i = 0; i < shapes.length; i++)
total_area += shapes[i].area();           // Вычислить площадь фигур
```

Здесь есть два важных замечания:

- Подклассы Shape могут быть присвоены элементам массива типа Shape. Приведение типа не требуется. Это еще один пример расширяющего приведения ссылочных типов (обсуждалось в главе 2).

- Вы можете вызывать методы `area()` и `circumference()` для любых объектов `Shape`, несмотря на то что сам класс `Shape` не определяет тела этих методов. В этом случае вызываемый метод находится динамически, так что площадь окружности вычисляется методом, определенным в классе `Circle`, а площадь прямоугольника – методом, определенным в классе `Rectangle`.

Интерфейсы

Расширим наш пакет `shapes`. Предположим, что теперь мы хотим реализовать несколько фигур, которые знают не только свой размер, но и позиции своих центров в декартовой системе координат. Один из способов создать такие фигуры – определить абстрактный класс `CenteredShape`, а затем реализовать нужные подклассы, например `CenteredCircle`, `CenteredRectangle` и т. д.

Но мы также хотим, чтобы позиционируемые классы поддерживали ранее определенные методы `area()` и `circumference()` без повторной реализации этих методов. Например, мы хотим определить класс `CenteredCircle` как подкласс `Circle`, чтобы он унаследовал методы `area()` и `circumference()`. Но у класса в Java может быть только один непосредственный родительский класс. Если класс `CenteredCircle` расширяет класс `Circle`, то он не может расширять абстрактный класс `CenteredShape`![1]

В Java решение этой проблемы сводится к созданию интерфейсов. Хотя Java-класс может расширять только один родительский класс, он способен *реализовать* (implement) несколько интерфейсов.

Определение интерфейса

Интерфейс – это ссылочный тип, очень похожий на класс. Почти все, что вы уже прочли в этой книге о классах, применимо и к интерфейсам. Определение интерфейса во многом похоже на определение абстрактного класса, за исключением того, что слова `abstract` и `class` заменяются на слово `interface`. Когда вы определяете интерфейс, вы создаете новый ссылочный тип, как при создании класса. В соответствии со своим названием интерфейс предоставляет API к определенной функциональности. Интерфейс не определяет реализации этого API. Есть несколько ограничений, накладываемых на члены интерфейса:

- Интерфейс не содержит реализации чего бы то ни было. Все методы интерфейса являются абстрактными, даже если модификатор `abstract` опущен. У методов интерфейса нет реализации; вместо тела метода стоит точка с запятой. Так как интерфейс содержит только абстрактные методы, а методы класса не могут быть абстрактными, то методы интерфейса всегда являются методами экземпляра.
- Интерфейс определяет открытый API. Все методы интерфейса неявно объявлены как `public`, даже если модификатор `public` опущен. Определение `protected-` или `private`-методов в интерфейсе недопустимо.
- Хотя класс определяет данные и методы их обработки, интерфейс не может определить поля экземпляра. Поля являются деталями реализации, а интерфейс не определяет никакой реализации. В определении интерфейса могут появляться только константы, объявленные с модификаторами `static` и `final`.

---

[1] C++ позволяет классам иметь несколько родителей с помощью техники, известной как множественное наследование. Множественное наследование сильно усложняет язык; Java поддерживает более элегантное решение.

- Нельзя создать ни одного экземпляра интерфейса, поэтому у него нет конструктора.

В примере 3.6 представлено определение интерфейса с именем `Centered`. Этот интерфейс определяет методы, которые подкласс класса `Shape`, должен реализовать, зная координаты `x` и `y` своего центра.

Пример 3.6. Определение интерфейса

```
public interface Centered {
  public void setCenter(double x, double y);
  public double getCenterX();
  public double getCenterY();
}
```

Реализация интерфейса

Подобно тому как класс использует ключевое слово `extends` для указания родительского класса, он может применить ключевое слово `implements` для указания интерфейсов, которые он реализует. `implements` — это ключевое слово Java, которое может появиться в объявлении класса за секцией `extends`. За `implements` должно следовать имя или имена интерфейсов, реализуемых классом. Имена разделяются запятыми.

Когда класс объявляет интерфейс в своей секции `implements`, он сообщает о предоставлении реализации (то есть тела) для каждого метода этого интерфейса. Если класс реализует интерфейс, но не предоставляет реализации для всех его методов, то он наследует нереализованные методы как абстрактные и сам должен быть объявлен как `abstract`. Если класс реализует более одного интерфейса, то он должен предоставить реализацию каждого метода каждого интерфейса либо он должен быть объявлен как `abstract`.

В примере 3.7 определен класс `CenteredRectangle`, который расширяет наш класс `Rectangle` и реализует интерфейс `Centered`, представленный в примере 3.6.

Пример 3.7. Реализация интерфейса

```
public class CenteredRectangle extends Rectangle implements Centered {
  // Новые поля экземпляра
  private double cx, cy;

  // Конструктор
  public CenteredRectangle(double cx,  double cy, double w, double h) {
    super(w, h);
    this.cx = cx;
    this.cy = cy;
  }

  // Мы наследуем все методы Rectangle, но должны
  // предоставить реализацию всех методов Centered.
  public void setCenter(double x, double y) { cx = x; cy = y; }
  public double getCenterX() { return cx; }
  public double getCenterY() { return cy; }
}
```

Как я отметил ранее, в определении интерфейса могут появляться константы. Любой класс, реализующий интерфейс, наследует эти константы и может использовать их так, словно они были определены непосредственно в классе. Нет необходимости ставить перед константами имя интерфейса или как-то их реализовывать. Когда у вас есть набор констант, используемых несколькими классами (например, номер порта и

другие протокольные константы, применяемые клиентом и сервером), то удобнее объявить их в интерфейсе, у которого нет методов. Затем любой класс, которому нужны эти константы, должен только объявить о том, что он реализует соответствующий интерфейс. `java.io.ObjectStreamConstants` является таким интерфейсом.

Использование интерфейсов

Предположим, что мы реализуем `CenteredCircle` и `CenteredSquare` так же, как реализовали `CenteredRectangle` в примере 3.7. Поскольку каждый класс расширяет класс `Shape`, то с экземплярами этих классов можно работать как с классом `Shape`. Каждый класс реализует интерфейс `Centered`, поэтому с экземплярами классов можно работать как с экземплярами данного типа. Следующий код демонстрирует обе техники:

```
Shape[] shapes = new Shape[3];                    // Создать массив для хранения фигур

// Создать несколько позиционируемых фигур и сохранить их в Shape[]. Приведение типов не
// требуется: здесь используется расширяющее преобразование
shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);
shapes[1] = new CenteredSquare(2.5, 2, 3);
shapes[2] = new CenteredRectangle(2.3, 4.5, 3, 4);

// Вычислить среднюю площадь фигур и среднее расстояние от центра
double totalArea = 0;
double totalDistance = 0;
for(int i = 0; i < shapes.length; i++) {
  totalArea += shapes[i].area();              // Вычислить площадь фигур
  if (shapes[i] instanceof Centered) {        // Фигура является экземпляром Centered
    // Обратите внимание на то, что требуется приведение Shape
    // к Centered (не нужно приведение CenteredSquare к Centered).
    Centered c = (Centered) shapes[i];        // Присвоить Centered переменной
    double cx = c.getCenterX();               // Получить координаты центра
    double cy = c.getCenterY();               // Вычислить расстояние от центра
    totalDistance += Math.sqrt(cx*cx + cy*cy);
  }
}
System.out.println("Средняя площадь: " + totalArea/shapes.length);
System.out.println("Среднее расстояние: " + totalDistance/shapes.length);
```

Этот пример демонстрирует, что, подобно классам, интерфейсы являются типами данных. Когда класс реализует интерфейс, экземпляры этого класса могут быть присвоены переменным с типом интерфейс. Не воспринимайте данный пример так, словно вы должны присвоить объект `CenteredRectangle` переменной `Centered` перед вызовом метода `setCenter()` или переменной `Shape` перед вызовом `area()`. Интерфейс `CenteredRectangle` определяет метод `setCenter()` и наследует метод `area()` от родителя класса `Rectangle`, поэтому вы всегда можете вызвать эти методы.

Когда нужно применять интерфейсы

При определении абстрактного типа (например, `Shape`), у которого будет несколько подтипов (например, `Circle`, `Rectangle`, `Square`), вы можете оказаться перед выбором между интерфейсами и абстрактными классами. Так как у них похожие возможности, то не всегда ясно, что лучше использовать.

Интерфейс полезен тем, что его может реализовать любой класс, даже если этот класс расширяет другой родительский класс. Но интерфейс – это спецификация API; он не

содержит реализации. Если у интерфейса есть много методов, то вам может наскучить снова и снова реализовывать эти методы, особенно если большая часть реализации дублируется в каждом классе.

С другой стороны, класс, расширяющий абстрактный класс, не может расширять другой класс. В некоторых ситуациях это вызывает трудности при проектировании. Впрочем, если класс не является полностью абстрактным, он может содержать частичную реализацию, которую могут применять подклассы. В некоторых случаях множество подклассов может рассчитывать на реализации методов по умолчанию, предоставленные абстрактным классом.

Другое важное отличие между интерфейсами и абстрактными классами состоит в обеспечении совместимости. Если вы объявили интерфейс как часть открытого API, а затем добавили в интерфейс новый метод, то вы нарушили целостность всех классов, которые реализуют предыдущую версию интерфейса. Если вы используете абстрактный класс, вы можете благополучно добавить в этот класс неабстрактный метод без необходимости модификации существующих классов, расширяющих абстрактный класс.

В некоторых ситуациях интерфейс или абстрактный класс являются правильным выбором. В других случаях типовой шаблон подразумевает использование обоих вариантов. Во-первых, определите тип как полностью абстрактный интерфейс. Затем создайте абстрактный класс, который реализует интерфейс и предоставляет полезные предопределенные реализации подклассам. Например:

```
// Вот простой интерфейс. Он представляет собой фигуру, умещающуюся внутри ограничивающего
// прямоугольника. Любой класс, желающий выступать в роли RectangularShape,
// может реализовать все эти методы.
public interface RectangularShape {
  public void setSize(double width, double height);
  public void setPosition(double x, double y);
  public void translate(double dx, double dy);
  public double area();
  public boolean isInside();
}

// Вот частичная реализация этого интерфейса. Она будет полезна многим другим реализациям
public abstract class AbstractRectangularShape implements RectangularShape {
  // Позиция и размер фигуры
  protected double x, y, w, h;

  // Реализация по умолчанию некоторых методов интерфейса
  public void setSize(double width, double height) { w = width; h = height; }
  public void setPosition(double x, double y) { this.x = x; this.y = y; }
  public void translate (double dx, double dy) { x += dx; y += dy; }
}
```

Реализация нескольких интерфейсов

Предположим, нам нужен объект-фигура, который можно позиционировать не только относительно центра, но и относительно левого верхнего угла. Допустим, нам нужна фигура, которую можно масштабировать в большую и меньшую сторону. Несмотря на то что класс может расширять только один родительский класс, он может реализовать любое количество интерфейсов. Полагая, что мы определили соответствующие интерфейсы `UpperRightCornered` и `Scalable`, объявим класс таким образом:

```
public class SuperDuperSquare extends Shape
       implements Centered, UpperRightCornered, Scalable {
  // Члены класса опущены
}
```

Когда класс реализует более одного интерфейса, это просто означает, что он должен предоставить реализации для всех абстрактных методов всех интерфейсов.

Расширение интерфейсов

У интерфейсов могут быть подынтерфейсы (subinterfaces) подобно тому, как у классов могут быть подклассы. Подынтерфейс наследует все абстрактные методы и константы родительского интерфейса и может определять новые абстрактные методы и константы. Есть одно важное отличие интерфейсов от классов: у интерфейса секция `extends` может содержать более одного родительского интерфейса. Например, вот некоторые интерфейсы, расширяющие другие интерфейсы:

```
public interface Positionable extends Centered {
  public void setUpperRightCorner(double x, double y);
  public double getUpperRightX();
  public double getUpperRightY();
}
public interface Transformable extends Scalable, Translatable, Rotatable {}
public interface SuperShape extends Positionable, Transformable {}
```

Интерфейс, который расширяет более одного интерфейса, наследует все абстрактные методы и константы от каждого родителя и может определять собственные дополнительные абстрактные методы и константы. Класс, реализующий такой интерфейс, должен реализовать абстрактные методы, определенные непосредственно в интерфейсе, наряду с абстрактными методами, унаследованными от родительских интерфейсов.

Интерфейсы-маркеры

Иногда полезно определить пустой интерфейс. Класс может реализовать этот интерфейс, указав его в секции `implements`. При этом нет необходимости реализовывать методы. Любой экземпляр класса становится экземпляром интерфейса. С помощью оператора `instanceof` Java-код может проверить, является ли объект экземпляром интерфейса. Таким образом, эта техника полезна для предоставления дополнительной информации об объекте. Интерфейс `Cloneable` из пакета `java.lang` является примером *интерфейса-маркера* (*marker interface*). Он не определяет методов, но идентифицирует класс, внутреннее состояние которого можно клонировать методом `clone()` класса `Object`. Начиная с Java 1.1 интерфейс `java.io.Serializable` является еще одним примером интерфейса-маркера. Пусть дан произвольный объект. Наличие у него работающего метода `clone()` можно определить с помощью следующего кода:

```
Object o;           // Проинициализирован в другом месте
Object copy;
if (o instanceof Cloneable) copy = o.clone();
   else copy = null;
```

Внутренние классы: обзор

Все классы и интерфейсы, которые мы видели в этой главе, являлись классами верхнего уровня (то есть они являлись непосредственными членами пакетов и не были

вложены в другие классы). В Java 1.1 появились четыре новых типа классов, именуемых *внутренними классами* (*inner classes*), которые могут быть определены в Java-программе. Внутренние классы являются элегантным и мощным инструментом языка Java. Эти четыре типа классов представлены ниже:

Статические классы-члены (static member classes)

Статический класс-член – это класс (или интерфейс), определенный как `static`-член другого класса. Метод с модификатором `static` называют методом класса, поэтому внутренний класс такого типа можно назвать «классом класса», но подобная терминология была бы очень запутанной. Статические классы-члены ведут себя почти так же, как обычные классы верхнего уровня, за исключением того, что они могут получать доступ к `static`-членам класса, в котором они содержатся. Интерфейсы могут быть определены как статические члены классов.

Классы-члены (member classes)

Класс-член определяется как член окружающего класса, но он объявляется без модификатора `static`. Этот тип внутренних классов является аналогом методов и полей экземпляра. Экземпляр класса-члена всегда связан с экземпляром окружающего класса, а код класса-члена имеет доступ ко всем полям и методам (как статическим, так и нестатическим) его класса. Есть несколько особенностей синтаксиса языка Java, проявляющихся при работе с экземпляром класса, содержащего класс-член. Интерфейсы могут быть определены только как статические члены, но не как нестатические.

Локальные классы (local classes)

Локальный класс – это класс, определенный в блоке Java-кода. Как и локальная переменная, локальный класс виден только внутри блока. Несмотря на то что локальные классы не являются классами-членами, они объявлены внутри внешнего класса, поэтому им присущи многие особенности классов-членов. Кроме того, локальные классы могут работать с любыми `final`-переменными или параметрами, которые доступны в блоке, определяющем класс. Интерфейсы нельзя определить локально.

Анонимные классы (anonymous classes)

Анонимный класс – это вид локального класса без имени; он комбинирует синтаксис для объявления класса с синтаксисом для присвоения значения объекту. Объявление локального класса – это оператор Java, а определение анонимного класса (и присвоение значения) – это выражение Java, поэтому такое определение может появляться только как часть еще большего выражения, например вызова метода. Интерфейсы нельзя определить анонимно.

Программисты, пишущие на Java, не пришли к единому мнению по поводу именования различных типов внутренних классов. Поэтому вы можете обнаружить, что в разных ситуациях их называют по-разному. В частности, статические классы-члены иногда называют «вложенными классами верхнего уровня», а термин «вложенные классы» может ссылаться на все типы внутренних классов. Термин «внутренние классы» перегружен. В одних случаях он ссылается на классы-члены, в других – на классы-члены, локальные классы и анонимные классы, но не на статические классы-члены. В этой книге я использую термин «внутренний класс» для обозначения любого класса, отличного от стандартного класса верхнего уровня, а имена, приведенные выше, – для ссылки на отдельные типы внутренних классов.

Статические классы-члены

Статический класс-член (или интерфейс) больше всего похож на обычный класс верхнего уровня (или интерфейс). Впрочем, для удобства он вложен в другой класс (или интерфейс). В примере 3.8 представлен вспомогательный интерфейс, определенный как статический член окружающего класса. Пример также показывает, как интерфейс используется этим классом и внешними классами. Обратите внимание на применение иерархического имени во внешнем классе.

Пример 3.8. Определение и использование статического интерфейса-члена

```
// Класс, реализующий стек, как связный список
public class LinkedStack {
  // Этот статический интерфейс-член определяет связывание объектов
  public static interface Linkable {
    public Linkable getNext();
    public void setNext(Linkable node);
  }

  // Начало списка - это объект Linkable
  Linkable head;

  // Тела методов опущены
  public void push(Linkable node) { ... }
  public Object pop() { ... }
}

// Этот класс реализует статический интерфейс-член
class LinkableInteger implements LinkedStack.Linkable {
  // Это данные и конструктор узла
  int i;
  public LinkableInteger(int i) { this.i = i; }

  // Это данные и методы, необходимые для реализации интерфейса
  LinkedStack.Linkable next;
  public LinkedStack.Linkable getNext() { return next; }
  public void setNext(LinkedStack.Linkable node) { next = node; }
}
```

Особенности статических классов-членов

Статический класс-член или интерфейс-член определен как `static`-член окружающего класса, что делает его аналогом поля и метода класса, которые так же объявлены как `static`. Как и метод класса, статический класс-член не связан ни с одним экземпляром окружающего класса (то есть нет объекта `this`). Тем не менее статический класс-член имеет доступ ко всем `static`-членам окружающего класса, включая любые другие статические классы-члены и интерфейсы-члены. Статический класс-член может использовать любой статический член без указания имени окружающего класса.

Статический класс-член имеет доступ ко всем статическим членам окружающего класса, включая `private`-члены. Обратное тоже верно: методы окружающего класса имеют доступ ко всем членам статического класса-члена, включая `private`-члены этого класса.

Поскольку статические классы-члены сами являются членами класса, то статический класс-член может быть объявлен с модификаторами доступа. Эти модификаторы имеют одно и то же значение для статических классов-членов и для остальных чле-

нов класса. В примере 3.8 интерфейс `Linkable` объявлен как `public`, поэтому он может быть реализован любым классом, который нужно сохранить в `LinkedStack`.

Ограничения на статические классы-члены

Статический класс-член не может называться так же, как называется любой из окружающих классов. Кроме того, статические класс-член и интерфейс-член могут быть объявлены только в классах верхнего уровня и других статических классах-членах и интерфейсах-членах. В действительности это часть более общего запрета на использование `static`-членов любого вида в классах-членах, локальных и анонимных классах.

Новый синтаксис для статических классов-членов

В коде, расположенном вне окружающего класса, на статический класс-член или интерфейс-член ссылаются по имени внешнего класса с последующим добавлением имени внутреннего класса (например, `LinkedStack.Linkable`). Для импорта статических классов-членов можно применять директиву `import`:

```
import LinkedStack.Linkable; // Импортировать указанный внутренний класс
import LinkedStack.*;        // Импортировать все внутренние классы LinkedStack
```

Импортировать внутренние классы не рекомендуется, потому что данная операция скрывает факт того, что класс тесно связан с содержащим его классом.

Классы-члены

Класс-член – это класс, который объявлен как нестатический член окружающего класса. Если статический класс-член является аналогом поля или метода класса, то класс-член является аналогом поля или метода экземпляра. В примере 3.9 показано, как класс-член может быть объявлен и задействовован. Этот пример расширяет предыдущий пример с `LinkedStack`, позволяя просмотреть все элементы стека с помощью метода `enumerate()`, который возвращает реализацию интерфейса `java.util.Enumeration`. Реализация этого интерфейса определена как класс-член.

Пример 3.9. Перечисление, реализованное как класс-член

```
public class LinkedStack {
  // Наш статический интерфейс-член; тело опущено...
  public static interface Linkable { ... }

  // Начало списка
  private Linkable head;

  // Тела методов здесь опущены
  public void push(Linkable node) { ... }
  public Linkable pop() { ... }

  // Данный метод возвращает объект Enumeration для этого LinkedStack
  public java.util.Enumeration enumerate() { return new Enumerator(); }

  // Это реализация интерфейса Enumeration, определенная как класс-член.
  protected class Enumerator implements java.util.Enumeration {
    Linkable current;
    // Конструктор использует private-поле head окружающего класса
    public Enumerator() { current = head; }
    public boolean hasMoreElements() { return (current != null); }
```

```
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  }
}
```

Обратите внимание на то, как класс `Enumerator` размещен в классе `LinkedStack`. Поскольку класс `Enumerator` является вспомогательным классом и используется только в классе `LinkedStack`, то код получится действительно элегантным, если объявить `Enumerator` близко к тому месту, где он применяется окружающим классом.

Особенности классов-членов

Подобно полям и методам экземпляра, каждый класс-член связан с экземпляром класса, внутри которого он определен (то есть каждый экземпляр класса-члена связан с экземпляром окружающего класса). Это означает, что код класса-члена имеет доступ ко всем полям и методам экземпляра (так же как и к статическим членам) окружающего класса, включая все члены, объявленные как `private`.

Эта ключевая особенность продемонстрирована в примере 3.9. Рассмотрим тело конструктора `LinkedStack.Enumerator()` еще раз:

```
current = head;
```

Эта строка кода устанавливает поле `current` внутреннего класса в значение поля `head` окружающего класса. Код работает точно так же несмотря на то, что поле `head` объявлено как `private`-поле в окружающем классе.

У класса-члена, как и любого члена класса, может быть установлен один из трех уровней видимости: `public`, `protected` или `private`. Если ни один из этих модификаторов не указан, то по умолчанию применяется пакетная видимость. В примере 3.9 класс `Enumerator` объявлен как `protected`, поэтому он не доступен коду в другом пакете, использующему класс `LinkedStack`, но доступен любому подклассу класса `LinkedStack`.

Ограничения на классы-члены

Существует три важных ограничения на классы-члены:

- Класс-член не может иметь имя, совпадающее с именем окружащего класса или пакета. Это важно помнить. Правило не распространяется ни на поля, ни на методы.
- Класс-член не может иметь полей, методов или классов, объявленных как `static` (за исключением полей-констант, объявленных как `static` и `final`). Статические поля, методы и классы являются конструкциями верхнего уровня, которые не связаны с конкретными объектами, в то время как каждый класс-член связан с экземпляром окружающего класса. Определение `static`-члена верхнего уровня в классе-члене низкого уровня вносит путаницу и является плохим стилем программирования, поэтому вы должны определять все статические члены в классе верхнего уровня, статическом классе-члене или интерфейсе-члене.
- Интерфейсы не могут быть определены как классы-члены. Ни одного экземпляра интерфейса не может быть создано, поэтому не с чем связывать экземпляр окру-

жающего класса. Если вы определили интерфейс как член класса, то интерфейс неявно является статическим, что делает его статическим классом-членом.

Новый синтаксис классов-членов

Самой важной особенностью классов-членов является возможность доступа к полям и методам экземпляра окружающего объекта. Мы это видели в конструкторе `LinkedStack.Enumerator()` в примере 3.9:

```
public Enumerator() { current = head; }
```

В этом примере `head` является полем класса `LinkedStack`, и мы можем присвоить этот объект полю `current` класса `Enumerator`. Этот код работает, но что если мы хотим сделать эти ссылки явными? Проверим следующий код:

```
public Enumerator() { this.current = this.head; }
```

Этот код не компилируется. С `this.current` все в порядке; это явная ссылка на поле `current` во вновь созданном объекте `Enumerator`. Источником проблемы является выражение `this.head`, которое ссылается на поле с именем `head` объекта `Enumerator`. Поскольку такого поля нет, компилятор генерирует ошибку. Для решения этой проблемы Java определяет специальный синтаксис для явной ссылки на окружающий экземпляр объекта `this`. Поэтому, если мы хотим определенности в нашем конструкторе, то можно использовать следующий синтаксис:

```
public Enumerator() { this.current = LinkedStack.this.head; }
```

Общий синтаксис выглядит как `classname.this`, где `classname` – это имя окружающего класса. Обратите внимание, что класс-член может сам содержать классы-члены любой степени вложенности. Поскольку ни один класс-член не может иметь такое же имя, как и окружающий класс, то использование имени включающего класса перед `this` является прекрасным способом сослаться на любой окружающий экземпляр. Этот синтаксис необходим только в том случае, если вы ссылаетесь на член окружающего класса, скрытый членом класса-члена с таким же именем.

Доступ к родительским членам окружающего класса

Когда класс затеняет или замещает члены родительского класса, для ссылки на скрытый член можно применить ключевое слово `super`. Кроме того, данное ключевое слово можно задействовать для работы с классами-членами. В тех редких случаях, когда вам нужно сослаться на затененное поле `f` или на замещенный метод `m` родителя окружающего класса `C`, используйте следующие выражения:

```
C.super.f
C.super.m()
```

Этот синтаксис не был реализован в компиляторах Java 1.1, но работает правильно в Java 1.2.

Указание окружающего интерфейса

Как мы уже видели, каждый экземпляр класса-члена связан с экземпляром окружающего класса. Рассмотрим еще раз объявление метода `enumerate()` в примере 3.9:

```
public Enumeration enumerate() {  return new Enumerator(); }
```

Когда конструктор класса-члена вызывается подобным образом, новый экземпляр класса-члена автоматически связывается с объектом `this`. Это то, что вы ожидаете,

и как раз то, что хотите получить в большинстве случаев. Впрочем, иногда вы хотите явно указать окружающий экземпляр при создании класса-члена. Для этого оператор new нужно предварить ссылкой на окружающий экземпляр. Следовательно, метод enumerate(), представленный выше, является краткой записью следующего кода:

```
public Enumeration enumerate() { return this.new Enumerator(); }
```

Предположим, что мы не определяли метод enumerate() класса LinkedStack. В этом случае код получения объекта Enumerator для данного объекта LinkedStack может выглядеть так:

```
LinkedStack stack = new LinkedStack();     // Создать пустой стек
Enumeration e = stack.new Enumerator();    // Создать для него объект Enumeration
```

Окружающий экземпляр неявно указывает имя окружающего класса; явное указание окружающего класса является синтаксической ошибкой:

```
Enumeration e = stack.new LinkedStack.Enumerator(); // Синтаксическая ошибка
```

Есть еще одна особенность синтаксиса Java, благодаря которой можно явно указать включающий экземпляр для класса-члена. Но перед рассмотрением этого синтаксиса отметим, что вам вряд ли придется его использовать. Это один из патологических случаев, когда подобный синтаксис существует в языке наряду со всеми остальными элегантными особенностями внутренних классов.

Как бы странно это ни показалось, но класс верхнего уровня может расширять класс-член. Это означает, что у подкласса нет окружающего экземпляра, но у его родителя есть. Когда конструктор подкласса вызывает родительский конструктор, он должен указать окружающий экземпляр. Для этого после окружающего экземпляра ставится точка и ключевое слово super. Если бы мы не объявили наш класс Enumerator как protected-член класса LinkedStack, мы могли бы его унаследовать. Мы можем написать следующий код:

```
// Класс верхнего уровня, расширяющий класс-член
class SpecialEnumerator extends LinkedStack.Enumerator {
   // Конструктор должен явно указать окружающий экземпляр
   // при вызове конструктора родительского класса.
   public SpecialEnumerator(LinkedStack s) { s.super(); }
   // Остаток класса опущен...
}
```

Область видимости и наследование в классах-членах

Мы заметили, что класс верхнего уровня может расширять класс-член. С введением классов-членов появляются две отдельные иерархии, к которым может принадлежать любой класс. Первая – это *иерархия классов*, от родителя к подклассу, определяющая поля и методы, наследуемые классом-членом. Вторая – это *иерархия вложенности*, от окружающего класса к вложенному классу, определяющая набор полей и методов, который входит в область видимости класса-члена (и, соответственно, этот набор ему доступен).

Эти две иерархии полностью отличаются друг от друга; важно, чтобы вы их не путали. Следует избегать конфликтов имен, когда поля или методы родительского класса носят такие же имена, как поля или методы окружающего класса. В случае конфликта имен унаследованное поле или метод имеют более высокий приоритет, чем поле или метод окружающего класса. Такое поведение логично: когда класс наследует поле или метод, то поле или метод становится частью класса. Поэтому унаследованные по-

ля и методы в области видимости класса, который их наследует, имеют преимущество перед полями и методами с такими же именами во внешней области видимости.

Хороший способ предотвратить конфликт между иерархией классов и иерархией вложенности – избегать глубокой вложенности. Если классы имеют более двух уровней вложенности, то проблем будет больше, чем возможных выгод. Более того, если у класса глубокая иерархия (то есть у него много родительских классов), лучше определить его как класс верхнего уровня, нежели как класс-член.

Локальные классы

В отличие от классов-членов, *локальный класс* объявляется в блоке Java-кода. Обычно локальный класс определяется в методе, но он также может быть определен в статическом инициализаторе или инициализаторе экземпляра класса. Поскольку все блоки Java-кода находятся внутри определения класса, то все локальные классы вложены в окружающие классы. По этой причине локальные классы имеют много общего с классами-членами. Но чаще их рассматривают как отдельный вид внутренних классов. Локальный класс имеет примерно то же отношение к классу-члену, как локальная переменная – к переменной экземпляра класса.

Определяющей характеристикой локального класса является локальность для блока кода. Как и локальная переменная, локальный класс действителен только в области видимости окружающего блока. Если класс-член используется только внутри одного метода окружающего класса, то можно определить его как локальный класс, а не как класс-член. В примере 3.10 показано, каким образом можно изменить метод `enumerate()` класса `LinkedStack`, чтобы он определял `Enumerator` как локальный класс, а не как класс-член. За счет этого мы переместим определение класса ближе к тому месту, где он используется, и, следовательно, код будет более понятным. Для краткости в примере 3.10 показан только метод `enumerate()`, а не весь класс `LinkedStack`, который его содержит.

Пример 3.10. Определение и использование локального класса

```
// Этот метод создает и возвращает объект Enumeration
public java.util.Enumeration enumerate() {

// Вот определение Enumerator как локального класса
  class Enumerator implements java.util.Enumeration {
    Linkable current;
    public Enumerator() { current = head; }
    public boolean hasMoreElements() { return (current != null); }
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  }
  // Теперь вернем экземпляр класса Enumerator, который только что определили
  return new Enumerator();
}
```

Особенности локальных классов

Локальные классы имеют следующие интересные особенности:

- Подобно классам-членам, локальные классы связаны с окружающим экземпляром и имеют доступ ко всем членам, включая `private`-члены содержащего класса.
- Помимо того что они имеют доступ к полям окружающего класса, локальные классы могут обращаться ко всем локальным переменным, параметрам метода или параметрам-исключениям, которые находятся в области видимости локального метода и объявлены как `final`.

Ограничения локальных классов

Существуют следующие ограничения локальных классов:

- Локальный класс виден только внутри блока, который его определяет; его нельзя применять вне этого блока.
- Локальный класс нельзя объявить как `public`, `protected`, `private` или `static`. Эти модификаторы используются только для членов класса; они недоступны для объявления локальных переменных или классов.
- Как и классы-члены, и по тем же причинам, локальные классы не могут содержать `static`-поля, методы или классы. Единственное исключение составляют константы, объявленные как `static` и `final`.
- Интерфейсы нельзя определить локально.
- Подобно классу-члену, локальный класс не может называться так же, как и окружающий его класс.
- Как было отмечено ранее, локальный класс может использовать локальные переменные, параметры методов и даже параметры-исключения, находящиеся в его области видимости, но только те из них, которые объявлены как `final`. Это объясняется тем, что время жизни локального класса может значительно превышать время выполнения метода, в котором класс определен. По этой причине локальный класс должен иметь свои внутренние копии всех локальных переменных, которые он использует (эти копии автоматически создаются компилятором). Единственный способ обеспечить идентичность значений локальной переменной и ее копии – объявить локальную переменную как `final`.

Новый синтаксис для локальных классов

В Java 1.0 только поля, методы и классы могли быть объявлены как `final`. Добавление локальных классов в Java 1.1 потребовало послаблений в использовании модификатора `final`. Теперь он может применяться с локальными переменными, параметрами методов и даже исключениями-параметрами оператора `catch`. Значение модификатора `final` осталось тем же самым: после того как локальной переменной было присвоено значение, это значение уже не может быть изменено.

Экземпляры локальных классов, как и экземпляры классов-членов, имеют окружающий экземпляр, который неявно передается всем конструкторам локальных классов. Локальные классы могут применять такой же синтаксис `this`, какой используют классы-члены для явной ссылки на член окружающего класса. Так как локальные классы не видимы вне блока, в котором они определены, то нет необходимости использовать синтаксис `new` и `super`, применяемый классами-членами для явного указания окружающего экземпляра.

Область видимости локальных классов

При обсуждении классов-членов мы видели, что класс-член имеет доступ к любым членам, унаследованным от родительского класса, и ко всем членам, определенным окружающим классом. То же самое верно и для локальных классов, но локальные классы также имеют доступ к локальным переменным и параметрам `final`. Следующий код иллюстрирует поля и методы, которые доступны локальному классу:

```
class A { protected char a = 'a'; }

class B { protected char b = 'b'; }

public class C extends A {
  private char c = 'c';          // Закрытые поля доступны локальному классу
  public static char d = 'd';
  public void createLocalObject(final char e)
  {
    final char f = 'f';
    int i = 0;                   // i не final; не может использоваться локальными классами
    class Local extends B
    {
      char g = 'g';
      public void printVars()
      {
        // Все эти поля и переменные доступны данному классу
        System.out.println(g);   // (this.g) g - поле этого класса
        System.out.println(f);   // f - локальная переменная final
        System.out.println(e);   // e - локальный параметр final
        System.out.println(d);   // (C.this.d) d - поле окружающего класса
        System.out.println(c);   // (C.this.c) c - поле окружающего класса
        System.out.println(b);   // b - наследуется этим классом
        System.out.println(a);   // a - наследуется окружающим классом
      }
    }
    Local l = new Local();       // Создать экземпляр локального класса
      l.printVars();             // и вызвать его метод printVars().
  }
}
```

Область видимости локальных классов и локальных переменных

Локальная переменная определяется в блоке кода, который задает ее область видимости. Локальная переменная перестает существовать вне области видимости. Java – это язык с *лексической областью видимости*. Это означает, что область видимости зависит от того, как написан исходный код. Любой код внутри фигурных скобок, обрамляющих границы блока, может использовать локальные переменные, определенные в этом блоке.[1]

Лексическая область видимости просто определяет сегмент исходного кода, внутри которого может применяться переменная. Обычно область видимости принято считать временной областью, то есть локальная переменная существует с момента, когда

[1] Этот раздел охватывает углубленные материалы; читатели, впервые читающие этот раздел, могут пропустить его и вернуться к нему позже.

интерпретатор Java начал выполнение некоторого блока, и до момента, когда интерпретатор вышел из этого блока. Обычно именно так нужно представлять локальные переменные и область их видимости.

Введение в язык локальных классов портит картину, так как локальные классы могут использовать локальные переменные, а период жизни экземпляров локальных классов может превышать период, в течение которого интерпретатор выполняет блок кода. Другими словами, если вы создаете экземпляр локального класса, то он не пропадает автоматически, когда интерпретатор завершает выполнение блока, создавшего этот класс. Эту особенность демонстрирует следующий код:

```
public class Weird {
  // Статический интерфейс-член, используемый ниже
  public static interface IntHolder { public int getValue(); }
  public static void main(String[] args) {
    IntHolder[] holders = new IntHolder[10];
    // Массив для хранения 10 объектов
    for(int i = 0; i < 10; i++) {              // Заполнить массив
      final int fi = i;                        // локальная переменная final
      class MyIntHolder implements IntHolder { // Локальный класс
        public int getValue() { return fi; }   // использует переменную final
      }
      holders[i] = new MyIntHolder();          // Присвоить значение локальному классу
    }
    // Локальный класс теперь вне области видимости, поэтому мы не можем к нему обратиться.
    // Но в массиве у нас есть 10 действующих экземпляров этого класса.
    // Локальная переменная fi вне области нашей видимости, но она все еще в области видимости
    // метода getValue() каждого из 10 объектов. Поэтому мы можем вызвать метод getValue() для
    // каждого объекта и вывести значения. Выводятся цифры от 0 до 9.
    for(int i = 0; i < 10; i++) System.out.println(holders[i].getValue());
  }
}
```

Поведение предыдущей программы может вызвать удивление. Помните, что лексическая область видимости методов локального класса никак не связана со временем входа и выхода интерпретатора из блока, определяющего локальный класс. Можно сказать и по-другому: каждый экземпляр локального класса содержит автоматически созданную копию каждой локальной переменной `final`, которую он использует. Поэтому у него есть собственная приватная копия области видимости, которая существовала при создании класса.

Анонимные классы

Анонимный класс – это локальный класс без имени. Анонимный класс определяется и инициализируется в едином выражении с помощью оператора `new`. Несмотря на то что определение локального класса – это оператор в блоке Java-кода, определение анонимного класса представляет собой выражение. Это означает, что его можно записать как часть большего выражения, например вызова метода. Когда локальный класс используется всего один раз, можно применить синтаксис анонимного класса, который позволяет совместить определение и использование класса.

В примере 3.11 представлен класс `Enumeration`, реализованный как анонимный класс в методе `enumerate()` класса `LinkedStack`. Сравните его с тем же классом, реализованным в виде локального класса.

Пример 3.11. Перечисление, реализованное через анонимный класс

```
public java.util.Enumeration enumerate() {

  // Анонимный класс, определенный как часть оператора return
  return new java.util.Enumeration() {
    Linkable current;
    { current = head; }       // Заменить конструктор инициализатором экземпляра
    public boolean hasMoreElements() { return (current != null); }
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  }; // Обратите внимание на точку с запятой. Этот символ завершает оператор return
}
```

Один из общих способов использования анонимного класса – это предоставление простой реализации класса адаптера. *Класс адаптера* – это класс, который определяет код, вызываемый другим объектом. Возьмем, например, метод `list()` класса `java.io.File`. Этот метод возвращает список файлов в каталоге. Перед возвращением списка он передает имя каждого файла объекту `FilenameFilter`, который вы должны предоставить. Объект `FilenameFilter` пропускает или не пропускает файл. Когда вы реализуете интерфейс `FilenameFilter`, вы определяете класс адаптера для использования с методом `File.list()`. Поскольку тело такого класса обычно короткое, то легче объявить класс адаптера как анонимный класс. Вот как вы можете определить класс `FilenameFilter`, который будет возвращать список только тех файлов, которые имеют расширение *.java*:

```
File f = new File("/src"); // Каталог для просмотра

// Теперь вызовем метод list() с единственным аргументом FilenameFilter
// Определим и проинициализируем анонимную реализацию
// FilenameFilter как часть выражения, вызывающего метод.
String[] filelist = f.list(new FilenameFilter() {
  public boolean accept(File f, String s) { return s.endsWith(".java"); }
}); // Не забудьте скобку и точку с запятой в конце вызова метода!
```

Как видите, синтаксис определения анонимного класса и создания экземпляра этого класса использует ключевое слово `new`, за которым следует имя класса и его определение в фигурных скобках. Если имя, следующее за ключевым словом `new`, есть имя класса, то анонимный класс является подклассом этого класса. Если имя, следующее за ключевым словом `new`, представляет собой интерфейс (как в двух предыдущих примерах), то анонимный класс реализует этот интерфейс и расширяет класс `Object`. Данный синтаксис не позволяет указать секции `extends`, `implements` или имя класса.

Так как у анонимного класса нет имени, то в теле класса нельзя определить его конструктор. Это одно из основных ограничений анонимных классов. Любые аргументы, которые вы укажете в круглых скобках, стоящих за именем родительского класса в определении анонимного класса, неявно передаются конструктору родительского класса. Чаще всего анонимные классы применяются для расширения родительских классов простыми классами, которые не требуют аргументов конструктора, поэтому скобки в определении анонимного класса зачастую пусты. В предыдущем примере каждый анонимный класс реализовывал интерфейс и расширял класс

`Object`. Поскольку конструктор `Object()` не принимает аргументов, то в этих примерах скобки были пусты.

Особенности анонимных классов

Самая элегантная особенность анонимных классов заключается в том, что они позволяют определить короткий класс как раз в том месте, где это необходимо. Кроме того, у анонимных классов лаконичный синтаксис, что уменьшает путаницу в коде.

Ограничения на анонимные классы

Так как анонимные классы представляют один из типов локальных классов, анонимные классы и локальные классы несут одинаковые ограничения. Анонимный класс не может определять статические поля, методы или классы, кроме констант `static final`. Интерфейс не может быть объявлен анонимно, потому что нет способа реализовать интерфейс без имени. Так же как и локальные классы, анонимные классы не могут быть `public`, `private`, `protected` или `static`.

Поскольку у анонимного класса нет имени, то невозможно определить конструктор анонимного класса. Если вашему классу нужен конструктор, вы должны задействовать локальный класс. Впрочем, зачастую в качестве замены конструктора можно применять инициализатор экземпляра. В действительности инициализатор экземпляра был введен в язык именно для этой цели.

Синтаксис определения анонимных классов сочетает определение и инициализацию. Поэтому применение анонимных классов вместо локальных классов не подходит в случае, когда вам нужно создавать более одного экземпляра класса при каждом выполнении окружающего блока.

Новый синтаксис анонимных классов

Мы уже видели примеры синтаксиса определения анонимного класса. Представим его более формально:

```
new class-name ( [ argument-list ] ) { class-body }
```

или:

```
new interface-name () { class-body }
```

Как я уже упоминал, инициализаторы экземпляра – это еще один специальный раздел синтаксиса Java, который был введен для поддержки анонимных классов. Как уже говорилось в этой главе, инициализатор экземпляра – это блок кода инициализации, содержащийся в фигурных скобках внутри описания класса. Содержимое инициализатора экземпляра для класса автоматически вставляется во все конструкторы класса, включая автоматически создаваемый конструктор по умолчанию. Анонимный класс не может определить конструктор, поэтому он получает конструктор по умолчанию. Используя инициализатор экземпляра, вы можете обойти ограничение, запрещающее определить конструктор для анонимного класса.

Когда применять анонимные классы

Как мы уже обсуждали, анонимный класс ведет себя как локальный класс, отличаясь от него лишь синтаксисом определения и инициализации. Когда вам нужно выбрать между анонимным и локальным классом, решение сводится к выбору стиля.

Следует применять тот синтаксис, который делает код более понятным. Выбирайте анонимный класс вместо локального, если:

- У класса очень маленькое тело.
- Нужен только один экземпляр класса.
- Класс используется сразу же после его определения.
- Имя класса не сделает ваш код яснее.

Отступы и форматирование анонимных классов

Общие соглашения по структурированию и форматированию, с которыми мы знакомы по языкам с блочной структурой, например Java и C, начинают нарушаться, когда мы помещаем определение анонимного класса в обычное выражение. Основываясь на своем опыте работы с внутренними классами, инженеры Sun рекомендуют следующие правила форматирования:

- Открывающая фигурная скобка не должна быть единственным символом в строке; по возможности она должна следовать за закрывающей скобкой оператора `new`. Точно так же оператор `new` следует располагать в той же строке, где находится присвоение или другое выражение, частью которого он является.
- Тело анонимного класса должно быть сдвинуто относительно начала строки, содержащей ключевое слово `new`.
- Закрывающая фигурная скобка также не должна быть единственной в строке; за ней должны следовать любые лексемы, необходимые для завершения выражения. Зачастую это точка с запятой или закрывающая скобка с точкой с запятой. Такая пунктуация уведомляет читателя о том, что это не просто блок кода. Кроме того, распознать анонимные классы становится легче.

Как работают внутренние классы

Предыдущие главы пояснили особенности и поведение внутренних классов различных типов. Собственно говоря, этих знаний о внутренних классах вам достаточно. Впрочем, некоторым программистам будет легче освоить внутренние классы, если они поймут, как внутренние классы реализованы.

Внутренние классы были впервые представлены в Java 1.1. Несмотря на огромные изменения в языке Java, введение внутренних классов не повлекло за собой изменения в виртуальной машине и формате файлов Java-классов. С точки зрения интерпретатора Java внутренних классов вообще не существует: все классы являются классами верхнего уровня. Чтобы заставить внутренние классы вести себя так, словно они действительно объявлены внутри другого класса, компилятору Java приходится добавлять к генерируемому классу скрытые поля, методы и аргументы конструктора. Вы можете воспользоваться декомпилятором *javap*, чтобы декомпилировать некоторые файлы классов для внутренних классов. Тогда вы увидите, на какие трюки идет компилятор, чтобы заставить работать внутренние классы (более подробно декомпилятор *javap* рассмотрен в главе 8).

Реализация статических классов-членов

Вернемся к рассмотрению класса `LinkedStack` (пример 3.8), который определял статический интерфейс-член с именем `Linkable`. Когда вы компилируете этот класс, ком-

пилятор в действительности генерирует два файла класса. Первый – с именем *LinkedStack.class*, как и ожидалось. Второй называется *LinkedStack$Linkable.class*. Символ $ в имени автоматически добавляется компилятором Java. Этот файл содержит реализацию статического интерфейса-члена.

Как мы обсуждали ранее, статический класс-член может обращаться к `static`-членам окружающего класса. В таких случаях компилятор автоматически добавляет имя окружающего класса перед выражением доступа к члену. Статический класс-член даже имеет доступ к полям `private static` окружающего класса. Поскольку статический класс-член скомпилирован как обычный класс верхнего уровня, то нет способа напрямую обратиться к `private`-членам окружающего класса. Поэтому, если статический класс-член использует `private`-члены окружающего класса (или наоборот), то компилятор автоматически генерирует не-`private`-методы доступа и конвертирует выражение, использующее `private`-члены, в выражение, применяющее сгенерированные методы. Эти методы получают пакетный доступ. Такого доступа достаточно, поскольку класс-член и окружающий класс обязательно находятся в одном пакете.

Реализация классов-членов

Реализация класса-члена во многом похожа на реализацию статического класса-члена. Он компилируется в отдельный файл класса верхнего уровня. Компилятор выполняет различные манипуляции с кодом, чтобы члены классов при доступе друг к другу работали корректно.

Самое важное отличие классов-членов от статических классов-членов состоит в том, что каждый экземпляр класса-члена связан с экземпляром окружающего класса. Для этого компилятор определяет искусственное поле с именем `this$0` в каждом классе-члене. Данное поле применяется для хранения ссылки на окружающий экземпляр. Каждый конструктор класса-члена получает дополнительный параметр, инициализирующий это поле. Каждый раз при вызове конструктора класса-члена компилятор автоматически передает ссылку на окружающий класс в этом дополнительном параметре.

Как мы видели, класс-член, как и любой другой член класса, может быть объявлен как `public`, `protected` или `private` либо он может иметь пакетную видимость. Впрочем, для поддержки классов-членов в виртуальную машину Java ничего не было добавлено. Классы-члены компилируются в обычные файлы классов, такие же как классы верхнего уровня, но классы верхнего уровня могут иметь только общий или пакетный доступ. Поэтому с точки зрения интерпретатора Java классы-члены имеют только общую или пакетную видимость. Это означает, что класс-член, объявленный как `protected`, считается открытым классом, а класс-член, объявленный как `private`, в действительности имеет пакетную видимость. Это не значит, что вы не должны объявлять класс-член как `protected` или `private`. Несмотря на то что интерпретатор не может обработать эти модификаторы, они сохраняются в файле класса. Это позволяет любому совместимому компилятору Java обработать эти модификаторы и избежать непредусмотренного доступа к классам-членам.

Реализация локальных и анонимных классов

Локальный класс может ссылаться на поля и методы окружающего класса по тем же причинам, что и класс-член; компилятор передает в конструктор скрытую ссылку на окружающий класс, которая сохраняется в `private`-поле. Кроме того, подобно классам-членам, локальные классы могут применять `private`-поля и методы окружающих классов, так как компилятор вставляет все необходимые методы доступа.

Локальные классы отличает от классов-членов возможность ссылаться на локальные переменные в области видимости, в которой они объявлены. Ключевое ограничение на эту возможность заключается в том, что локальные классы могут ссылаться только на локальные переменные и параметры, объявленные как `final`. Причина этому становится видимой при реализации. Локальный класс может применять локальные переменные, потому что компилятор автоматически отдает классу `private`-поле экземпляра для хранения копии каждой локальной переменной, используемой классом. Кроме того, компилятор добавляет скрытые параметры в каждый конструктор локального класса для инициализации автоматически созданных `private`-полей. Поэтому локальный класс не обращается к локальным переменным, а работает с собственными их копиями. Единственный способ заставить этот код работать правильно – объявить локальные переменные как `final`, чтобы они не менялись. Получив такую гарантию, локальный класс может быть уверен в том, что его внутренние копии переменных всегда синхронизированы с настоящими локальными переменными.

Поскольку у анонимных классов нет имен, вы можете поинтересоваться, как называются представляющие их файлы классов. Это детали реализации, но для имен анонимных классов компилятор Java от Sun использует номера. Если вы скомпилируете код, представленный в примере 3.11, вы увидите, что компилятор создал файл с именем, похожим на *LinkedStack$1.class*. Это файл анонимного класса.

Модификаторы: сводка

Как мы уже видели, классы, интерфейсы и их члены могут быть объявлены с одним или несколькими *модификаторами* – такими ключевыми словами, как `public`, `static` и `final`. Эта глава представляет модификаторы доступа `public`, `protected` и `private`, а также модификаторы `abstract`, `final` и `static`. В дополнение к этим шести модификаторам Java определяет еще пять менее используемых модификаторов. В табл. 3.2 приведены модификаторы Java. Таблица объясняет, какие типы Java-конструкций они могут менять и что именно они делают.

Таблица 3.2. Модификаторы Java

Модификатор	Используется в	Значение
`abstract`	Класс	Класс содержит нереализованные методы и не может порождать объекты.
	Интерфейс	Все интерфейсы являются абстрактными. Модификатор не обязателен при объявлении интерфейса.
	Метод	Не существует тела для метода; оно есть только у подклассов. Сразу за сигнатурой следует точка с запятой. Окружающий класс также должен быть абстрактным.
`final`	Класс	Класс не может иметь подклассов.
	Метод	Метод не может быть замещен (и не находится динамически).
	Поле	Значение поля не может быть изменено после компиляции.
	Переменная	Значения локальной переменной, параметра метода или исключения-параметра не могут быть изменены (Java 1.1 и последующие версии). Полезно для локальных классов.
`native`	Метод	Реализация метода зависит от платформы (зачастую на C). Нет тела; за сигнатурой следует точка с запятой.
None (пакет)	Класс	Не открытый класс; доступен только внутри своего пакета.
	Интерфейс	Не открытый интерфейс; доступен только в том же пакете.

Модификатор	Используется в	Значение
	Член	Член, не являющийся `private`, `protected` или `public`, имеет пакетную видимость и доступен только внутри пакета.
`private`	Член	Член доступен только внутри класса, в котором он определен.
`protected`	Член	Член доступен только внутри класса, в котором он определен, и внутри подклассов.
`public`	Класс	Класс доступен в любом месте, где доступен его пакет.
	Интерфейс	Интерфейс доступен в любом месте его пакета.
	Член	Член доступен везде, где доступен его класс.
`strictfp`	Класс	Все методы этого класса неявно объявлены с модификатором `strictfp` (Java 1.2 и последующие версии).
	Метод	Все вычисления с плавающей точкой, выполняемые методом, должны проводиться в строгом соответствии со стандартом IEEE 754. В особенности все значения, включая промежуточные результаты, должны выражаться как IEEE `float` или `double`. Запрещается применение расширенной точности или диапазона вещественного формата (Java 1.2 и последующие версии). Этот модификатор применяется редко.
`static`	Класс	Внутренний класс, объявленный как `static`, является классом верхнего уровня, не связанным с окружающим классом (Java 1.1 и последующие версии).
	Метод	Статический метод является методом класса. Ему не передается неявная ссылка `this` на объект. Вызов через имя класса.
	Поле	Статическое поле является полем класса. Вне зависимости от количества созданных экземпляров класса может существовать только один экземпляр поля. Доступно через имя класса.
	Инициализатор	Инициализатор выполняется при загрузке класса, вне зависимости от того, когда создается экземпляр класса.
`synchronized`	Метод	Метод выполняет неатомарные изменения класса или экземпляра, поэтому нужно позаботиться о том, чтобы два потока не изменяли класс или экземпляр одновременно. Перед вызовом статического метода необходимо блокировать класс, а перед вызовом нестатического метода – экземпляр.
`transient`	Поле	Поле не является сохраняемой частью объекта и не должно сериализоваться вместе с объектом. Применяется при сериализации объектов; см. `java.io.ObjectOutputStream`.
`volatile`	Поле	Поле доступно для несинхронизируемых потоков, поэтому оптимизация не нужна. Иногда этот модификатор может применяться как альтернатива `synchronized`. Используется редко.

Особенности C++, отсутствующие в Java

По ходу этой главы я в сносках обращал внимание на сходства и отличия между Java и C++. Java в достаточной степени разделяет концепции и особенности C++, чтобы переход программистов, пишущих на C++, к языку Java был более легким. Тем не менее есть несколько особенностей C++, которые не имеют аналогий в Java. В целом, язык Java не перенял особенности C++, которые усложняют язык.

C++ поддерживает множественное наследование реализаций методов от нескольких родительских классов. Хотя эта особенность кажется полезной, она добавляет в язык

много сложностей. Разработчики Java избежали этих трудностей, добавив в язык интерфейсы. Поэтому класс в Java может наследовать реализации методов только от одного родительского класса, а объявления методов – от любого количества интерфейсов.

C++ поддерживает шаблоны, позволяющие, к примеру, реализовать класс `Stack`, а затем проинициализировать его как `Stack<int>` или `Stack<double>`, чтобы получить два разных типа: стек целых значений и стек вещественных значений. Java не предоставляет такую возможность, но работа в этом направлении уже ведется, поэтому такая возможнось будет добавлена стандартизованным образом. Боле того, поскольку каждый класс Java является подклассом класса `Object`, то каждый объект можно привести к экземпляру класса `Object`. Поэтому в Java зачастую достаточно определить структуру данных (например, класс `Stack`), которая оперирует значениями `Object`; при необходимости эти объекты можно привести к их исходному типу.

C++ позволяет определять операторы, которые выполняют любые операции над экземплярами ваших классов. В действительности вы можете расширить синтаксис языка. Эта интересная особенность, называемая перегрузкой операторов, позволяет создавать элегантные примеры. Впрочем, на практике код становится трудным для понимания. После долгих дебатов разработчики языка Java решили отказаться от перегрузки операторов. Обратите внимание, что оператор `+`, используемый в Java для конкатенации строк, сильно напоминает перегрузку операторов.

C++ позволяет определять функции преобразования для класса, которые автоматически вызывают соответствующий конструктор, когда переменной этого класса присваивается значение. Это просто сокращение синтаксиса, похожее на замещение оператора присваивания; оно не включено в Java.

По умолчанию в C++ объекты передаются по значению; вы должны использовать `&` для указания переменной или аргумента функции, передаваемого по ссылке. В Java все объекты передаются по ссылке, поэтому в подобном синтаксисе нет необходимости.

Глава 4

Java-платформа

Главы 2 и 3 содержат документацию по языку программирования Java. В этой главе угол обзора меняется. Здесь рассматривается Java-платформа – обширная коллекция предопределенных классов, доступных любой Java-программе вне зависимости от системы, на которой программа запущена. Классы Java-платформы собраны в связные группы, называемые *пакетами*. Эта глава начинается с обзора пакетов Java-платформы, которые описаны в данной книге. Затем на коротких примерах мы продемонстрируем работу с наиболее полезными классами из этих пакетов. Большинство примеров – это только фрагменты кода, а не полные программы, которые можно скомпилировать и запустить. За полноценными практическими примерами обращайтесь к книге «Java Examples in a Nutshell» (O'Reilly).[1] Эта книга прекрасно дополняет данную главу.

Обзор Java-платформы

Таблица 4.1 содержит перечень ключевых пакетов Java-платформы, описанных в этой книге.

Таблица 4.1. Ключевые пакеты Java-платформы

Пакет	Описание
`java.beans`	Компонентная модель JavaBeans для многократно используемых, встраиваемых программных компонентов.
`java.beans.beancontext`	Дополнительные классы, описывающие контекстные объекты компонента, которые хранят и предоставляют сервисы для JavaBeans-объектов, которые их содержат.
`java.io`	Классы и интерфейсы, предназначенные для ввода/вывода. Несмотря на то что некоторые классы в этом модуле предназначены для непосредственной работы с файлами, большинство классов работает с файлами через потоки байтов или символов.
`java.lang`	Базовые классы языка, такие как `String`, `Math`, `System`, `Thread` и `Exception`.
`java.lang.ref`	Классы, определяющие «слабые» (weak) ссылки на объекты (или ссылочные классы – *Примеч. перев*). Эти ссылки не препятствуют сборщику мусора в удалении связанных с ними объектов.

[1] Д. Флэнаган «Java в примерах. Справочник». – Пер. с англ. – СПб: Символ-Плюс, 2003.

Таблица 4.1 (продолжение)

Пакет	Описание
java.lang.reflect	Классы и интерфейсы, позволяющие Java-программам самим получать информацию о конструкторах, методах и полях классов.
java.math	Небольшой пакет, содержащий классы для выполнения арифметических операций с целыми и вещественными числами произвольной точности.
java.net	Классы и интерфейсы для связи с другими системами по сети.
java.nio	Буферные классы для работы с новым API ввода/вывода.
java.nio.channels	Классы и интерфейсы каналов и селекторов для высокопроизводительных неблокирующих операций ввода/вывода.
java.nio.charset	Перекодировщики символов для конвертации строк Unicode в байты и обратного преобразования.
java.security	Классы и интерфейсы для контроля прав доступа и аутентификации. Пакет поддерживает шифрование сообщений и цифровые подписи.
java.security.acl	Пакет, поддерживающий списки прав доступа. Начиная с Java 1.2 этот пакет не используется.
java.security.cert	Классы и интерфейсы для работы с сертификатами открытых ключей.
java.security.interfaces	Интерфейсы, используемые для DSA- и RSA-шифрования с открытым ключом.
java.security.spec	Классы и интерфейсы для прозрачного представления ключей и параметров, используемых при шифровании с открытым ключом.
java.text	Классы и интерфейсы для работы с текстом в интернациональных (internationalized) приложениях.
java.util	Разнообразные служебные классы, включающие мощную систему работы с коллекциями объектов (collections framework).
java.util.jar	Классы для чтения и записи JAR-файлов.
java.util.logging	Гибкое средство регистрации.
java.util.prefs	API для чтения и записи настроек пользователя и системы.
java.util.regex	Сопоставление текстовых паттернов с помощью регулярных выражений.
java.util.zip	Классы для чтения и записи ZIP-файлов.
javax.crypto	Классы и интерфейсы для шифрования и дешифрования данных.
javax.crypto.interfaces	Интерфейсы, представляющие открытые/закрытые ключи Диффи-Хелмана (Diffie-Hellman), которые применяются в протоколе согласования ключей Диффи-Хелмана.
javax.crypto.spec	Классы, определяющие прозрачное представление ключей и параметров, которые применяются при шифровании.
javax.net	Определяет фабрику классов для создания обычных и серверных сокетов. Позволяет создавать сокеты, чьи типы отличны от типов, заданных по умолчанию.
javax.net.ssl	Классы для создания шифрованных сетевых соединений с использованием SSL.

Пакет	Описание
javax.security.auth	Пакет верхнего уровня для JAAS API (аутентификация и авторизация).
javax.security.auth.callback	Классы, обеспечивающие связь модуля регистрации (login module) низкого уровня и пользователя через пользовательский интерфейс.
javax.security.auth.kerberos	Служебные классы для поддержки аутентификации с использованием протокола Kerberos.
javax.security.auth.login	LoginContext и другие связанные с ним классы для аутентификации пользователя.
javax.security.auth.spi	Определяет интерфейс LoginModule, который реализован посредством подключаемых модулей аутентификации пользователя.
javax.security.auth.x500	Служебный класс, предоставляющий информацию сертификата X.500.
javax.xml.parsers	API высокого уровня для разбора XML-документов с использованием подключаемых DOM- и SAX-парсеров (parser).
javax.xml.transform	API высокого уровня для преобразования XML-документов с использованием подключаемых средств преобразования XSLT и для конвертации XML-документов между потоками, деревьями DOM и событиями SAX.
javax.xml.transform.dom	Классы преобразования XML для DOM.
javax.xml.transform.sax	Классы преобразования XML для SAX.
javax.xml.transform.stream	Классы преобразования XML для XML-потоков.
org.ietf.jgss	Java-реализация API базовых сервисов безопасности (Generic Security Services), которая определяет единый API для базовых механизмов безопасности, таких как Kerberos.
org.w3c.dom	Интерфейсы, определенные консорциумом W3C (World Wide Web Consortium) для представления XML-документа в виде дерева DOM.
org.xml.sax	Классы и интерфейсы для разбора XML-документа с использованием API, основанного на событиях SAX (Simple API for XML).
org.xml.sax.ext	Классы, расширяющие функциональность SAX API.
org.xml.sax.helpers	Служебные классы для SAX API.

Таблица 4.1 содержит не все пакеты Java-платформы, а только те, которые описаны в этой книге. Кроме того, в таблицу не включены несколько пакетов spi, которые описаны в книге, но представляют интерес только для поставщиков услуг низкого уровня. Java также определяет несколько пакетов для работы с графикой и программирования графического интерфейса пользователя, а также для выполнения распределенных вычислений и вычислений уровня предприятия. Пакеты графики и графического интерфейса пользователя – это пакеты java.awt и javax.swing со всем множеством их подпакетов. Эти пакеты вместе с пакетом java.applet описаны в книге «Java Foundation Classes in a Nutshell» (O'Reilly). Java-пакеты уровня предприятия (enterprise packages) включают в себя пакеты java.rmi, java.sql, javax.jndi, org.omg.CORBA и org.omg.CosNaming со всеми подпакетами. Эти пакеты, как и несколько стандартных расширений Java-платформы, описаны в «Java Enterprise in a Nutshell» (O'Reilly).

Строки и символы

Текстовые строки – это фундаментальный и наиболее часто используемый тип данных. В Java, как известно, строки не относятся к примитивным типам данных, подобно `char`, `int` и `float`. Напротив, строки представлены классом `java.lang.String`, который определяет много полезных методов для работы с ними. Объекты `String` неизменны: с момента создания объекта `String` нет возможности изменить текстовую строку, которую он представляет. Таким образом, каждый метод, оперирующий строками, обычно возвращает новый объект `String`, который хранит измененную строку.

Этот код показывает некоторые базовые операции, которые вы можете выполнять со строками:

```
// Создание строки
String s = "Now";                // Объекты String имеют специальный синтаксис
String t = s + " is the time."; // Конкатенация строк выполняется с помощью оператора +
String t1 = s + " " + 23.4;      // Оператор + приводит значения переменных других типов
                                 // к типу String
t1 = String.valueOf('c');        // Получить строку, соответствующую значению символа
t1 = String.valueOf(42);         // Получить целое или другое значение в текстовом виде
t1 = object.toString();          // Конвертировать объекты в строки с помощью метода toString()

// Длина строки
int len = t.length();            // Количество символов в строке: 16

// Подстрока строки
String sub = t.substring(4);     // Возвращает символы с 4-го и до конца строки: "is the time."
sub = t.substring(4, 6);         // Возвращает 4-й и 5-й символы: "is"
sub = t.substring(0, 3);         // Возвращает символы с 0-го по 2-й: "Now"
sub = t.substring(x, y);         // Возвращает символы между позициями x и y
int numchars = sub.length();     // Длина подстроки всегда (y-x)

// Выделить символы из строки
char c = t.charAt(2);            // Возвращает 3-й символ строки t: w
char[] ca = t.toCharArray();     // Конвертирует строку в массив символов
t.getChars(0, 3, ca, 1);         // Помещает первые 3 символа строки t в ca[1]-ca[3]

// Изменение регистра
String caps = t.toUpperCase();  // Конвертирует в верхний регистр
String lower = t.toLowerCase(); // Конвертирует в нижний регистр

// Сравнение строк
boolean b1 = t.equals("hello");        // Возвращает false: строки не равны
boolean b2 = t.equalsIgnoreCase(caps); // Сравнение без различения прописных и строчных
                                       // букв: true
boolean b3 = t.startsWith("Now");      // Возвращает true
boolean b4 = t.endsWith("time.");      // Возвращает true
int r1 = s.compareTo("Pow");       // Возвращает значение меньше нуля, если s идет перед "Pow"
int r2 = s.compareTo("Now");       // Возвращает 0: строки равны
int r3 = s.compareTo("Mow");       // Возвращает значение больше нуля, если s идет после "Mow"
r1 = s.compareToIgnoreCase("pow"); // Возвращает значение меньше нуля
                                   // (Java 1.2 и последующие версии)

// Поиск символов и подстрок
int pos = t.indexOf('i');          // Позиция первого найденного символа 'i': 4
pos = t.indexOf('i', pos+1);       // Позиция следующего символа 'i': 12
```

```
pos = t.indexOf('i', pos+1);       // Возвращает -1, если символов 'i' больше не найдено
pos = t.lastIndexOf('i');          // Позиция последнего символа 'i' в строке: 12
pos = t.lastIndexOf('i', pos-1);   // Ищет символ 'i' в обратном направлении,
                                   // начиная с 11-го символа
pos = t.indexOf("is");             // Ищет подстроку: возвращает 4
pos = t.indexOf("is", pos+1);      // Подстрока встречается только один раз: возвращает -1
pos = t.lastIndexOf("the ");       // Ищет подстроку в обратном направлении
String noun = t.substring(pos+4);  // Выделить слово, следующее за "the"

// Заменить все вхождения одного символа на другой символ
String exclaim = t.replace('.', '!');    // Работает только с символами, но не со строками

// Удалить лишние пробелы в начале и в конце строки
String noextraspaces = t.trim();

// Получить уникальные экземпляры строк с помощью intern()
String s1 = s.intern();              // Возвращает s1, равное s
String s2 = "Now".intern();          // Возвращает s2, равное "Now"
boolean equals = (s1 == s2);    // Теперь можно проверять равенство с помощью оператора ==
```

Класс Character

Как вы знаете, отдельные символы представлены в Java базовым типом `char`**. Java-платформа также определяет класс** `Character`**, который содержит полезные методы для проверки типа символа и для преобразования регистра символа. Например:**

```
char[] text;         // Массив символов, инициализированный в другом месте
int p = 0;           // Наша текущая позиция в массиве символов
// Пропустить ведущие пробелы
while((p < text.length) && Character.isWhitespace(text[p])) p++;
// Перевести первое слово в верхний регистр
while((p < text.length) && Character.isLetter(text[p])) {
  text[p] = Character.toUpperCase(text[p]);
  p++;
}
```

Класс StringBuffer

Так как объект `String` **неизменен, вы не можете манипулировать символами объекта** `String`**, которому уже присвоено значение. Если это необходимо, применяйте вместо класса** `String` **класс** `java.lang.StringBuffer`**:**

```
// Создать StringBuffer из строки
StringBuffer b = new StringBuffer("Mow");

// Получить и установить отдельные символы StringBuffer
char c = b.charAt(0);      // Возвращает 'M': так же как String.charAt()
b.setCharAt(0, 'N');       // b содержит "Now": со строками такая операция недопустима!

// Добавление к StringBuffer
b.append(' ');             // Добавляет символ
b.append("is the time."); // Добавляет строку
b.append(23);              // Добавляет целое число или значение другого типа

// Вставляет строки или другие типы в StringBuffer
b.insert(6, "n't");        // Теперь b содержит: "Now isn't the time.23"
```

```
// Заменить цепочку символов на строку (Java 1.2 и последующие версии)
b.replace(4, 9, "is");     // Теперь снова "Now is the time.23"

// Удалить символы
b.delete(16, 18);          // Удалить цепочку: "Now is the time"
b.deleteCharAt(2);         // Удалить 2-й: "No is the time"
b.setLength(5);            // Обрезать, установив длину: "No is"

// Другие полезные функции
b.reverse();               // Зеркально отобразить символы: "si oN"
String s = b.toString();   // Конвертировать обратно в неизменную строку
s = b.substring(1,2);      // Или выделить подстроку: "i"
b.setLength(0);            // Очистить буфер; теперь он готов для повторного использования
```

Интерфейс CharSequence

В Java 1.4 классы `String` и `StringBuffer` реализуют новый интерфейс `java.lang.CharSequence`, который является стандартным интерфейсом для получения длины и выделения символов и подстрок из последовательностей символов, доступных для чтения. Этот интерфейс также реализован классом `java.nio.CharBuffer`, который является частью нового API ввода/вывода, впервые появившегося в Java 1.4. Интерфейс `CharSequence` позволяет выполнять простые операции над символьными строками вне зависимости от низкоуровневой реализации этих строк. Например:

```
/**
 * Возвращает префикс последовательности CharSequence, который начинается с 1-го символа
 * последовательности и заканчивается (включительно) 1-м вхождением символа c в
 * последовательности. Возвращает null, если c не найден. s может иметь следующий
 * тип: String, StringBuffer или java.nio.CharBuffer.
 */
public static CharSequence prefix(CharSequence s, char c) {
  int numChars = s.length();          // Получить длину последовательности
  for(int i = 0; i < numChars; i++) { // Перебрать все символы в последовательности
    if (s.charAt(i) == c)             // Если найден символ c, то
      return s.subSequence(0,i+1);    // вернуть найденный префикс
  }
  return null;                        // В противном случае вернуть null
}
```

Паттерны и регулярные выражения

В Java 1.4 и последующих версиях вы можете сопоставлять текстовые паттерны с помощью регулярных выражений. Поддержка регулярных выражений предоставляется классами `Pattern` и `Matcher` пакета `java.util.regex`, но класс `String` определяет несколько удобных методов, упрощающих использование регулярных выражений. Регулярные выражения применяют четкую комплексную грамматику для описания символьных паттернов. В реализации Java используется такой же синтаксис регулярных выражений, как и в языке программирования Perl. Краткое описание синтаксиса представлено в главе 17 в разделе, посвященном классу `java.util.regex.Pattern`. За более подробной информацией следует обратиться к хорошей книге по программированию на Perl. Полное руководство по регулярным выражениям Perl содержится в книге «Mastering Regular Expressions» (O'Reilly).[1]

[1] Дж. Фридл «Регулярные выражения». – Пер. с англ. – СПб: Питер, 2003.

Простейший метод объекта `String`, принимающий регулярное выражение в качестве аргумента, – это `matches()`. Он возвращает `true`, если строка соответствует паттерну, определенному регулярным выражением:

```
// Эта строка является регулярным выражением, описывающим паттерн типичного предложения.
// В синтаксисе регулярных выражений Perl она представляет собой строку, начинающуюся с
// заглавной буквы и заканчивающуюся точкой, знаком вопроса или восклицательным знаком.
String pattern = "^[A-Z].*[\\.?!]$";
String s = "Java is fun!";
s.matches(pattern); // Строка соответствует паттерну, поэтому метод вернет true
```

Метод `matches()` возвращает `true`, только если вся строка соответствует паттерну. Программисты, пищущие на языке Perl, должны отметить, что такой алгоритм отличается от поведения Perl, согласно которому совпадение означает лишь соответствие части строки паттерну. Чтобы определить, вся ли строка или только ее часть соответствует паттерну, просто измените регулярное выражение так, чтобы были допустимы произвольные символы перед паттерном и после него. В следующем коде символы регулярного выражения .* соответствуют любому количеству произвольных символов:

```
s.matches(".*\\bJava\\b.*"); // True, если s содержит слово «Java» в любом месте.
                             // Символ b означает границу слова
```

Если вы уже знакомы с синтаксисом регулярных выражений Perl, то вы знаете, что он предполагает свободное использование символов обратной косой черты для ввода некоторых служебных символов. В Perl регулярные выражения являются примитивами языка, а их синтаксис – его частью. Однако в Java регулярные выражения описываются строками и обычно вставляются в программы в виде строковых литералов. Синтаксис строковых литералов Java также использует символ обратной косой черты как блокирующий символ (escape character), поэтому для включения символа обратной косой черты в регулярное выражение вы должны использовать два таких символа. Таким образом, в Java-коде можно часто увидеть двойные символы обратной косой черты в регулярных выражениях.

Помимо сопоставления текстовых паттернов, регулярные выражения можно применять для операций поиска и замены. Методы `replaceFirst()` и `replaceAll()` находят первую или все подстроки, соответствующие данному паттерну, заменяют их на введенную строку и возвращают новую строку, содержащую результат замены. Например, с помощью этого кода можно удостовериться, что слово «Java» в строке `s` пишется с большой буквы:

```
s.replaceAll("(?i)\\bjava\\b", // Паттерн: слово "java"; не чувствительно к регистру
             "Java");          // Замена произведена правильно
```

Строка замены, переданная методам `replaceAll()` и `replaceFirst()`, должна быть простым строковым литералом; она также может включать ссылки на текст, который соответствует подвыражениям, содержащимся в скобках (parenthesized subexpressions) внутри паттерна. Эти ссылки обозначаются знаком доллара, за которым следует номер подвыражения. (Если вы не знакомы с подвыражениями, содержащими скобки в регулярных выражениях, то обратитесь к описанию класса `java.util.regex.Pattern`). Например, для поиска таких слов, как «JavaBean», «JavaScript», «JavaOS» и «JavaVM» (но не «Java» или «Javanese»), и для замены префикса «Java» на символ «J» без изменения суффикса (оставшейся части) вы можете применить следующий код:

```
s.replaceAll("\\bJava([A-Z]\\w+)",   // Паттерн
             "J$1");                 // За J следует суффикс, который  соответствует подвыражению
                                     // в скобках: [A-Z]\\w+
```

Еще один новый метод класса `String` в Java 1.4, который использует регулярные выражения – это метод `split()`. Он возвращает массив подстрок строки, изолированных разделителями, которые соответствуют определенному паттерну. Чтобы получить массив слов строки, разделенной любым количеством пробелов, знаков табуляции или символов новой строки, выполните следующий код:

```
String sentence = "This is a\n\ttwo-line sentence";
String[] words = sentence.split("[ \t\n\r]+");
```

Необязательный второй аргумент указывает максимальное количество элементов в возвращаемом массиве.

Методы `matches()`, `replaceFirst()`, `replaceAll()` и `split()` удобны, когда регулярное выражение нужно применить только один раз. Если вы хотите использовать регулярные выражения для многократной проверки соответствия, то лучше обратиться к классам `Pattern` и `Matcher` пакета `java.util.regex`. Прежде всего, с помощью метода `Pattern.compile()` создайте объект `Pattern`, представляющий собой регулярное выражение. Другая причина для использования класса `Pattern` вместо вспомогательных методов класса `String` заключается в том, что метод `Pattern.compile()` позволяет установить такие флаги, как `Pattern.CASE_INSENSITIVE`, которые глобально влияют на процесс обработки паттерна. Обратите внимание, что метод `compile()` может вызвать исключение `PatternSyntaxException`, если вы передадите ему неверную строку регулярного выражения. Это исключение также вызывается различными методами для работы с `String`. Класс `Pattern` определяет метод `split()`, который похож на метод `String.split()`. Для всех остальных проверок вы должны создать объект `Matcher` с помощью метода `matcher()` и передать ему текст для проверки:

```
import java.util.regex.*;

Pattern javaword = Pattern.compile("\\bJava(\\w*)", Pattern.CASE_INSENSITIVE);
Matcher m = javaword.matcher(sentence);
boolean match = m.matches();               // True, если текст точно соответствует паттерну
```

После того как вы получили объект `Matcher`, строку можно сравнить с паттерном различными способами. Один из наиболее изощренных способов – найти все подстроки, соответствующие паттерну:

```
String text = "Java is fun; JavaScript is funny.";
m.reset(text); // Начать сравнение новой строки
// Перебрать все найденные подстроки и вывести подробности каждого соответствия
while(m.find()) {
  System.out.println("Найдено `" + m.group(0) + "` в позиции " + m.start(0));
  if (m.start(1) < m.end(1)) System.out.println("Суффикс " + m.group(1));
}
```

Более подробно класс `Matcher` описан в главе 17.

Сравнение строк

Методы `compareTo()` и `equals()` класса `String` позволяют сравнивать строки. Метод `compareTo()` сравнивает строки на основе порядка символов, принятого в кодировке Unicode, а метод `equals()` определяет равенство строк как точное посимвольное соответствие. Однако эти методы не всегда полезны. В некоторых языках порядок символов,

задаваемый стандартом Unicode, не соответствует лексикографическому порядку, используемому при сортировке строк по алфавиту. Например, в испанском языке буквы «ch» рассматриваются как одна буква, расположенная между «c» и «d». При сравнении строк, предназначенных для показа пользователю, в интернациональных приложениях вы должны применять класс `java.text.Collator`:

```
import java.text.*;

// Сравнить две строки. Результат зависит от того, где программа запущена
// Значения, возвращаемые методом Collator.compare(), соответствуют результатам
// метода String.compareTo()
Collator c = Collator.getInstance();      // Получить объект Collator для текущих региональных
                                          // параметров (locale)
int result = c.compare("chica", "coche"); // Использовать его для сравнения двух строк
```

Класс StringTokenizer

Существует несколько Java-классов, которые оперируют строками и символами. Одним из таких классов является `java.util.StringTokenizer`, который можно применять для разбиения текстовой строки на составляющие ее слова:

```
String s = "Now is the time";
java.util.StringTokenizer st = new java.util.StringTokenizer(s);
while(st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

Вы даже можете задействовать этот класс для разбиения строки на слова, разделенные символами, отличными от пробелов:

```
String s = "a:b:c:d";
java.util.StringTokenizer st = new java.util.StringTokenizer(s, ":");
```

Числа и математика

Для представления чисел в Java есть примитивные типы `byte`, `short`, `int`, `long`, `float` и `double`. Пакет `java.lang` включает соответствующие классы `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`, каждый из которых является подклассом класса `Number`. Эти классы можно применять как интерфейсы к примитивным числовым типам данных, которые также определяют некоторые полезные константы:

```
// Общие граничные константы для классов Integer, Long и Character
Byte.MIN_VALUE         // Наименьшее (наименьшее отрицательное) значение byte
Byte.MAX_VALUE         // Наибольшее значение byte
Short.MIN_VALUE        // Наименьшее отрицательное значение short
Short.MAX_VALUE        // Наибольшее значение short

// Общие граничные константы для типов с плавающей точкой
// Double также определяет эти константы

Float.MIN_VALUE        // Наименьшее (ближайшее к нулю) положительное значение float
Float.MAX_VALUE        // Наибольшее положительное значение float

// Другие полезные константы
Math.PI                // 3.14159265358979323846
Math.E                 // 2.7182818284590452354
```

Преобразование чисел в строки и обратно

Java-программа, оперирующая числами, должна откуда-то брать входные данные. Зачастую такие программы читают текстовое представление чисел и переводят его в числовое представление. Разнообразные подклассы класса `Number` определяют полезные методы преобразования:

```
String s = "-42";
byte b = Byte.parseByte(s);              // s как byte
short sh = Short.parseShort(s);          // s как short
int i = Integer.parseInt(s);             // s как int
long l = Long.parseLong(s);              // s как long
float f = Float.parseFloat(s);           // s как float (Java 1.2 и последующие версии)
f = Float.valueOf(s).floatValue();       // s как float (до Java 1.2)
double d = Double.parseDouble(s);        // s как double (Java 1.2 и последующие версии)
d = Double.valueOf(s).doubleValue();     // s как double (до Java 1.2)

// Операции преобразования целых чисел работают с различными системами счисления
byte b = Byte.parseByte("1011", 2);      // 1011 в двоичной системе - это 11 в десятичной
short sh = Short.parseShort("ff", 16);   // ff в шестнадцатеричной системе - это 255
                                         // в десятичной

// Метод valueOf() может работать с произвольными системами
// счисления с основанием от 2 до 36
int i = Integer.valueOf("egg", 17).intValue(); // Основание 17!

// Метод decode() работает с восьмеричной, десятичной или
// шестнадцатеричной системой в зависимости от префикса
short sh = Short.decode("0377").byteValue();   // Ведущий символ 0 означает основание 8
int i = Integer.decode("0xff").shortValue();   // Ведущие символы 0x означают основание 16
long l = Long.decode("255").intValue();        // Остальные цифры означают основание 10

// Класс Integer может конвертировать числа в строки
String decimal = Integer.toString(42);
String binary = Integer.toBinaryString(42);
String octal = Integer.toOctalString(42);
String hex = Integer.toHexString(42);
String base36 = Integer.toString(42, 36);
```

Форматирование чисел

В разных странах числовые значения отображаются по-разному. Например, многие европейские языки используют запятую для разделения целой и дробной частей чисел с плавающей точкой (вместо точки). При отображении денежных значений разница в форматировании может быть еще больше. Поэтому при конвертации чисел в строки лучше всего применять класс `java.text.NumberFormat`, который позволяет учитывать региональные параметры (locale):

```
import java.text.*;
// Используйте NumberFormat для форматирования и
// разбора чисел с учетом региональных параметров
NumberFormat nf = NumberFormat.getNumberInstance(); // Получить NumberFormat
System.out.println(nf.format(9876543.21));          // Форматировать число с учетом региона
try {
  Number n = nf.parse("1.234.567,89");          // Разобрать строку в соответствии с регионом
```

```
} catch (ParseException e) { /* Обработать исключение */ }

// Денежные значения иногда форматируются иначе, чем другие числа
NumberFormat moneyFmt = NumberFormat.getCurrencyInstance();
System.out.println(moneyFmt.format(1234.56));          // Выводит $1,234.56 в США
```

Математические функции

Класс `Math` определяет несколько методов, которые наряду с другими методами позволяют выполнять тригонометрические, логарифмические, экспоненциальные операции и операции округления. Этот класс полезен при работе с числами с плавающей точкой. Для тригонометрических функций углы выражаются в радианах. Логарифмические и экспоненциальные функции применяют основание *e*, а не 10. Вот несколько примеров:

```
double d = Math.toRadians(27);        // Переводит 27 градусов в радианы
d = Math.cos(d);                      // Вычисляет косинус
d = Math.sqrt(d);                     // Вычисляет квадратный корень
d = Math.log(d);                      // Вычисляет натуральный логарифм
d = Math.exp(d);                      // Выполняет обратную операцию: e в степени d
d = Math.pow(10, d);                  // Возводит 10 в степень
d = Math.atan(d);                     // Вычисляет арктангенс
d = Math.toDegrees(d);                // Конвертирует обратно в градусы
double up = Math.ceil(d);             // Округляет до большего целого
double down = Math.floor(d);          // Округляет до меньшего целого
long nearest = Math.round(d);         // Округляет до ближайшего целого
```

Случайные числа

Класс `Math` также определяет простейший метод для генерации псевдослучайных чисел, но класс `java.util.Random` более гибок. Если вам нужно *по-настоящему* случайное псевдослучайное число, вы можете применить класс `java.security.SecureRandom`:

```
// Простой генератор случайных чисел
double r = Math.random();             // Возвращает d в интервале 0.0 <= d < 1.0

// Создать новый объект Random на основе текущего времени
java.util.Random generator = new java.util.Random(System.currentTimeMillis());
double d = generator.nextDouble();   // 0.0 <= d < 1.0
float f = generator.nextFloat();     // 0.0 <= f < 1.0
long l = generator.nextLong();       // Выбрать из всего интервала long
int i = generator.nextInt();         // Выбрать из всего интервала
i = generator.nextInt(limit);        // 0 <= i < limit (Java 1.2 и последующие версии)
boolean b = generator.nextBoolean(); // true или false (Java 1.2 и последующие версии)
d = generator.nextGaussian();        // Среднее значение: 0.0; стандартное отклонение: 1.0
byte[] randomBytes = new byte[128];
generator.nextBytes(randomBytes);    // Заполнить массив случайными байтами

// Для криптостойких случайных чисел применяйте подкласс SecureRandom
java.security.SecureRandom generator2 = new java.security.SecureRandom();
// Генератор сгенерирует собственное 16-байтное начальное
// значение; занимает *много* времени
generator2.setSeed(generator2.generateSeed(16));  // Дополнительное случайное
                                                  // 16-байтное значение
// Затем использовать SecureRandom, подобно другим генераторам
generator2.nextBytes(randomBytes);                // Еще сгенерировать случайные байты
```

Большие числа

Пакет `java.math` содержит классы `BigInteger` и `BigDecimal`, которые позволяют работать с целыми и вещественными числами произвольного размера и точности. Например:

```
import java.math.*;

// Посчитать факториал 1000
BigInteger total = BigInteger.valueOf(1);
for(int i = 2; i <= 1000; i++)
  total = total.multiply(BigInteger.valueOf(i));
System.out.println(total.toString());
```

В Java 1.4 класс `BigInteger` имеет метод, позволяющий генерировать случайные большие числа, которые полезны во многих криптографических приложениях:

```
BigInteger prime = BigInteger.probablePrime(1024,     // 1024 бит
                                generator2);          // generator2 взят из предыдущего примера
```

Дата и время

Java использует несколько различных классов для работы с датой и временем. Класс `java.util.Date` представляет момент времени (с точностью до миллисекунды). Этот класс есть не что иное, как обертка (wrapper) вокруг значения `long`, которое содержит количество миллисекунд, прошедших с полуночи 1 января 1970 года (время GMT). Вот два способа определить текущее время:

```
long t0 = System.currentTimeMillis();      // Текущее время в миллисекундах
java.util.Date now = new java.util.Date(); // По сути то же самое
long t1 = now.getTime();                   // Конвертировать Date в long
```

Класс `Date` имеет несколько интересных методов, но почти все они уже не используются из-за наличия методов классов `java.util.Calendar` и `java.text.DateFormat`.

Форматирование дат с помощью DateFormat

Чтобы напечатать дату или время, задействуйте класс `DateFormat`, который автоматически использует соглашения по форматированию даты и времени, зависящие от региональных параметров. `DateFormat` правильно работает даже при региональных установках, использующих календарь, отличный от текущего (Григорианского) календаря, который применяется во всем мире:

```
import java.util.Date;
import java.text.*;

// Отобразить сегодняшнюю дату с использованием формата
// по умолчанию для текущего региона
DateFormat defaultDate = DateFormat.getDateInstance();
System.out.println(defaultDate.format(new Date()));

// Отобразить текущее время с использованием короткого формата для текущего региона
DateFormat shortTime = DateFormat.getTimeInstance(DateFormat.SHORT);
System.out.println(shortTime.format(new Date()));

// Отобразить дату и время с использованием длинного формата для обоих значений
DateFormat longTimestamp =
```

```
    DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL);
System.out.println(longTimestamp.format(new Date()));

// Использовать SimpleDateFormat для определения собственного паттерна форматирования
// См. синтаксис в описании java.text.SimpleDateFormat
DateFormat myformat = new SimpleDateFormat("yyyy.MM.dd");
System.out.println(myformat.format(new Date()));
try {                          // DateFormat может анализировать и даты
   Date leapday = myformat.parse("2000.02.29");
} catch (ParseException e) { /* Обработать исключение */ }
```

Арифметические действия с датами и класс Calendar

Класс `Date` и миллисекундное представление даты, получаемое с его помощью, позволяют проводить только очень простые арифметические операции:

```
long now = System.currentTimeMillis();          // Текущее время
long anHourFromNow = now + (60 * 60 * 1000);    // Добавить 3 600 000 миллисекунд
```

Для выполнения более сложных арифметических операций с датой и временем и манипулирования датами в человеческом представлении применяйте класс `java.util.Calendar`:

```
import java.util.*;

// Получить Calendar для текущих региональных установок и часового пояса
Calendar cal = Calendar.getInstance();

// Вычислить текущий день года
cal.setTime(new Date());                                // Присвоить текущее время
int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);          // Какой это день года?

// Каким днем недели был 29 февраля 2000 года?
cal.set(2000, Calendar.FEBRUARY, 29);                   // Установить поля года, месяца и дня
int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);          // Получить день недели

// Какой день месяца был в 3-й четверг мая 2001 года?
cal.set(Calendar.YEAR, 2001);                           // Установить год
cal.set(Calendar.MONTH, Calendar.MAY);                  // Установить месяц
cal.set(Calendar.DAY_OF_WEEK, Calendar.THURSDAY);       // Установить день недели
cal.set(Calendar.DAY_OF_WEEK_IN_MONTH, 3);              // Установить неделю
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);        // Получить день в месяце

// Получить объект Date, представляющий дату через 30 дней после текущей
Date today = new Date();                                // Текущая дата
cal.setTime(today);                                     // Установить в объекте Calendar
cal.add(Calendar.DATE, 30);                             // Добавить 30 дней
Date expiration = cal.getTime();                        // Получить итоговую дату
```

Массивы

Класс `java.lang.System` определяет метод `arraycopy()`, который полезен при копировании заданных элементов одного массива в заданную позицию другого. Второй массив должен быть того же типа, что и первый; это может быть тот же самый массив:

```
char[] text = "Now is the time".toCharArray();
```

```
char[] copy = new char[100];
// Скопировать 10 символов, начиная с 4-го элемента массива
// text в массив copy, начиная с copy[0]
System.arraycopy(text, 4, copy, 0, 10);

// Переместить элементы массива text, чтобы освободить место для вставок элементов
System.arraycopy(copy, 3, copy, 6, 7);
```

В Java 1.2 и последующих версиях класс `java.util.Arrays` определяет полезные методы для манипулирования массивами, включая методы сортировки и поиска в массиве:

```
import java.util.Arrays;

int[] intarray = new int[] { 10, 5, 7, -3 };       // Массив целых чисел
Arrays.sort(intarray);                             // Сортировать его
int pos = Arrays.binarySearch(intarray, 7);        // Значение 7 найдено по индексу 2
pos = Arrays.binarySearch(intarray, 12);           // Не найдено: функция вернула
                                                   // отрицательный результат

// Массивы объектов так же могут быть отсортированы, и по ним можно выполнить поиск
String[] strarray = new String[] { "now", "is", "the", "time" };
Arrays.sort(strarray);                  // { "is", "now", "the", "time" }

// Arrays.equals() сравнивает все элементы двух массивов
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone);       // Да, они равны

// Arrays.fill() инициализирует элементы массива
byte[] data = new byte[100];                 // Пустой массив; элементы обнулены
Arrays.fill(data, (byte) -1);                // Установить их все в -1
Arrays.fill(data, 5, 10, (byte) -2);         // Установить элементы 5, 6, 7, 8, 9 в -2
```

С массивами в Java можно работать так же, как и с объектами. Пусть есть объект `o`. Выясним, является ли этот объект массивом и если да, то какого типа:

```
Class type = o.getClass();
if (type.isArray()) {
  Class elementType = type.getComponentType();
}
```

Коллекции

В Java инструментарий для работы с коллекциями – это набор важных служебных классов и интерфейсов, расположенных в пакете `java.util`. Существует два фундаментальных типа коллекций. Класс `Collection` – это группа объектов, а `Map` – набор отображений (mappings), или связей между объектами. `Set` – это коллекция, которая не содержит дубликатов, а `List` – коллекция, в которой элементы упорядочены. `Collection`, `Set`, `List`, `Map`, `SortedSet` и `SortedMap` – это интерфейсы, но пакет `java.util` также определяет и несколько конкретных реализаций, таких как списки, основанные на массивах и связных списках, а также классы `Map` и `Set`, основанные на хеш-таблицах или двоичных деревьях (полный перечень представлен в описании пакета `java.util`). Другие важные интерфейсы – это `Iterator` и `ListIterator`, которые позволяют перебирать объекты в коллекции. Инструментарий для работы с коллекциями – новый. Он появился в Java 1.2, а в предыдущих версиях можно применять классы `Vector` и `Hashtable`, которые примерно соответствуют `ArrayList` и `HashMap`.

В Java 1.4 инструментарий коллекций расширился с добавлением интерфейса RandomAccess, который представлен реализациями класса List, поддерживающими эффективный случайный доступ (то есть интерфейс реализован классами Vector и ArrayList, но не классом LinkedList). Кроме того, в Java 1.4 появились классы LinkedHashMap и LinkedHashSet, которые являются соответственно ассоциативным массивом и набором, реализованными с помощью хеш-таблиц и сохраняющими порядок вставки элементов. Наконец добавлен класс IdentityHashMap, являющийся реализацией Map на основе хеш-таблицы. Для сравнения объектов он использует оператор «==», а не метод equals().

Следующий код демонстрирует создание и выполнение базовых операций над Set, List и Map:

```
import java.util.*;
Set s = new HashSet();                  // Реализация, основанная на хеш-таблице
s.add("test");                          // Добавляет строковый объект в набор s
boolean b = s.contains("test2");        // Проверка, содержит ли набор объект
s.remove("test");                       // Удалить объект из набора
Set ss = new TreeSet();                 // TreeSet реализует SortedSet
ss.add("b");                            // Добавить несколько элементов
ss.add("a");
// Перебрать и вывести все элементы
for(Iterator i = ss.iterator(); i.hasNext();)
  System.out.println(i.next());

List l = new LinkedList();              // LinkedList реализует  двусвязный список
l = new ArrayList();                    // Обычно ArrayList более удобен
Vector v = new Vector();                // Vector - это альтернатива в Java 1.1/1.0
l.addAll(ss);                           // Добавить несколько элементов
l.addAll(1, ss);                        // Вставить элементы в позицию 1
Object o = l.get(1);                    // Получить второй элемент
l.set(3, "new element");                // Установить 4-й элемент
l.add("test");                          // Добавить новый элемент в конец
l.add(0, "test2");                      // Добавить новый элемент в начало
l.remove(1);                            // Удалить 2-й элемент
l.remove("a");                          // Удалить элемент «a»
l.removeAll(ss);                        // Удалить элементы
if (!l.isEmpty())                       // Если список не пуст,
  System.out.println(l.size());         // вывести количество его элементов
boolean b1 = l.contains("a");           // Содержит ли он значение?
boolean b2 = l.containsAll(ss);         // Содержит ли он все значения?
List sublist = l.subList(1,3);          // Подсписок из 2-го и 3-го элементов
Object[] elements = l.toArray();        // Конвертировать в массив
l.clear();                              // Удалить все элементы

Map m = new HashMap();                  // Аналог Hashtable из Java 1.1/1.0
m.put("key", new Integer(42));          // Связать объект-значение с объектом-ключом
Object value = m.get("key");            // Найти значение, ассоциированное с ключом
m.remove("key");                        // Удалить ассоциацию
keys = m.keySet();                      // Получить набор ключей
```

Преобразование в массив и обратное преобразование

Массивы объектов и коллекции служат одним целям. Есть возможность конвертировать их друг в друга:

```
Object[] members = set.toArray();       // Получить набор элементов как массив
Object[] items = list.toArray();        // Получить список элементов как массив
```

```
Object[] keys = map.keySet().toArray();      // Получить набор объектов-ключей как массив
Object[] values = map.values().toArray();    // Получить набор объектов-значений как массив

List l = Arrays.asList(a);                   // Просмотреть массив как неизменный список
List l = new ArrayList(Arrays.asList(a));    // Создать изменяемую копию
```

Служебные методы в классе Collections

Подобно тому как класс `java.util.Arrays` определяет методы для работы с массивами, так и класс `java.util.Collections` определяет методы работы с коллекциями. Наиболее полезны методы сортировки и поиска элементов коллекции:

```
Collections.sort(list);
int pos = Collections.binarySearch(list, "key");  // Сначала нужно отсортировать список
```

Вот несколько других интересных методов:

```
Collections.copy(list1, list2);              // Копирует list2 в list1, перезаписывая list1
Collections.fill(list, o);                   // Заполняет список объектами o
Collections.max(c);                          // Находит самый большой элемент в коллекции
Collections.min(c);                          // Находит самый маленький элемент в коллекции

Collections.reverse(list);                   // Переворачивает коллекцию
Collections.shuffle(list);                   // Перемешивает список

Set s = Collections.singleton(o);           // Возвращает неизменный набор с одним элементом o
List ul = Collections.unmodifiableList(list); // Неизменная обертка для списка
Map sm = Collections.synchronizedMap(map);    // Синхронизируемая обертка для map
```

Типы, отражение и динамическая загрузка

Класс `java.lang.Class` представляет типы данных в Java. Наряду с классами пакета `java.lang.reflect`, он дает Java-программисту возможность выполнять интроспекцию (или самоотражение); Java-класс может посмотреть сам на себя или на другой класс и выяснить его родительский класс, методы, определяемые этим классом и т. д.

Объекты Class

В Java существует несколько способов получить объект `Class`:

```
// Получить Class объекта o
Class c = o.getClass();

// Получить объект Class для примитивных типов с разными предопределенными константами
c = Void.TYPE;          // Специальный тип «пустое возвращаемое значение»
c = Byte.TYPE;          // объект Class, представляющий Byte
c = Integer.TYPE;       // объект Class, представляющий Integer
c = Double.TYPE;        // и т. д.; см. также Short, Character, Long, Float

// Выразить литерал класса как имя класса с последующим ".class"
c = int.class;          // То же, что и Integer.TYPE
c = String.class;       // То же, что и "dummystring".getClass()
c = byte[].class;       // Тип массива байт
c = Class[][].class;    // Тип массива массивов объектов Class
```

Отражение на Class

С объектом `Class` можно выполнить несколько операций отражения:

```
import java.lang.reflect.*;
Object o;                          // Неизвестный объект для исследования
Class c = o.getClass();            // Получить его тип

// Если это массив, то определить его базовый тип
while (c.isArray()) c = c.getComponentType();

// Если c не примитивный тип, то вывести его иерархию классов
if (!c.isPrimitive()) {
  for(Class s = c; s != null; s = s.getSuperclass())
    System.out.println(s.getName() + " extends");
}

// Попробовать создать новый экземпляр c. Для этого нужен конструктор без аргументов
Object newobj = null;
try { newobj = c.newInstance(); } catch (Exception e) {
  // Обработать InstantiationException, IllegalAccessException
}

// Проверить есть ли у класса метод setText, принимающий String
// Если да, то вызвать его с соответствующим аргументом
try {
  Method m = c.getMethod("setText", new Class[] { String.class});
  m.invoke(newobj, new Object[] { "My Label" });
} catch(Exception e) { /* Обработать исключения */ }
```

Динамическая загрузка классов

`Class` также предоставляет простой механизм для динамической загрузки классов в Java. Тем не менее для более полного контроля над динамической загрузкой классов нужно использовать объект `java.lang.ClassLoader`. Обычно это `java.net.URLClassLoader`. Данная техника полезна, когда вы хотите загрузить класс, имя которого прописано в конфигурационном файле, а не жестко закодировано в программе:

```
// Загрузить класс, имя которого прописано в конфигурационном файле
String classname =                      // Найти имя класса
config.getProperty("filterclass",       // Имя свойства
"com.davidflanagan.filters.Default");   // Значение по умолчанию

try {
  Class c = Class.forName(classname);   // Динамически загрузить класс
  Object o = c.newInstance();           // Динамически проинициализировать его
} catch (Exception e) { /* Обработать исключения */ }
```

Предыдущий код работает, если класс, который нужно загрузить, расположен в одном из каталогов, представленных в переменной путей к классам. Если это не так, вы можете создать собственный объект `ClassLoader` для загрузки необходимого класса из каталога (или по URL):

```
import java.net.*;
String classdir = config.getProperty("filterDirectory"); // Найти путь к классу
try {
  ClassLoader loader = new URLClassLoader(new URL[] { new URL(classdir) });
  Class c = loader.loadClass(classname);
} catch (Exception e) { /* Обработать исключения */ }
```

Динамические прокси

В Java 1.3 в пакет java.lang.reflect был добавлен класс Proxy и интерфейс InvocationHandler. Proxy – это мощный, но нечасто используемый класс, который позволяет динамически создавать новые классы или экземпляры классов, реализующие один или несколько интерфейсов. Кроме того, он перенаправляет вызовы методов интерфейса объекту InvocationHandler.

Потоки

Java позволяет легко объявлять несколько потоков выполнения внутри одной программы. java.lang.Thread – это фундаментальный класс потока в Java API. Есть два способа объявить поток. Первый – породить свой класс от класса Thread, заменить его метод run(), а затем проинициализировать созданный подкласс. Другой способ – объявить класс, реализующий интерфейс Runnable (то есть объявить метод run()), а затем передать экземпляр этого объекта конструктору Thread(). В любом случае результатом будет объект Thread, в котором метод run() является телом потока. Когда вы вызываете метод start() объекта Thread, интерпретатор создает новый поток и выполняет его метод run(). Новый поток продолжает работу до тех пор, пока выполнение метод run() не закончится. В этот момент поток перестает существовать. Между тем выполнение исходного потока продолжается с оператора, следующего за методом start(). Следующий код это показывает:

```
final List list; // Некоторый длинный, несортированный список
                 // объектов; инициализирован в другом месте

/** Класс Thread для сортировки списка в фоновом режиме */
class BackgroundSorter extends Thread {
  List l;
  public BackgroundSorter(List l) { this.l = l; }     // Конструктор
  public void run() { Collections.sort(l); }          // Тело потока
}

// Создать поток BackgroundSorter
Thread sorter = new BackgroundSorter(list);
// Начать выполнение; новый поток выполняет метод run(), а основной поток продолжает
// выполняться вне зависимости от того, какой оператор следует далее
sorter.start();

// Еще один способ создать похожий поток
Thread t = new Thread(new Runnable() {                // Создать новый поток
  public void run() { Collections.sort(list); }       // для сортировки списка объектов
});
t.start();            // Начать выполнение
```

Приоритеты потоков

Потоки могут выполняться с разными уровнями приоритетов. Поток с определенным приоритетом обычно не выполняется до тех пор, пока ожидают выполнения потоки с более высокими приоритетами. Ниже приведен код, который можно использовать при работе с приоритетами потоков:

```
// Понизить приоритет потока t до уровня ниже нормального
t.setPriority(Thread.NORM_PRIORITY-1);
```

```
// Установить приоритет потока ниже приоритета текущего потока
t.setPriority(Thread.currentThread().getPriority() - 1);

// Потоки, которые не приостанавливаются для выполнения операций
// ввода/вывода, должны явно освобождать процессор, чтобы предоставить другим
// потокам с таким же приоритетом возможность выполнения.
Thread t = new Thread(new Runnable() {
  public void run() {
    for(int i = 0; i < data.length; i++) { // Перебрать все элементы данных
      process(data[i]);                    // Обработать их,
      if ((i % 10) == 0)                   // но после каждых 10-ти итераций
        Thread.yield();                    // приостанавливаться, чтобы выполнялись
                                           // другие потоки.
    }
  }
});
```

Перевод потока в режим ожидания

Зачастую потоки применяются для выполнения повторяющихся задач через фиксированные интервалы времени. Это особенно актуально при программировании графики, которое вовлекает анимацию или похожие функции. Ключевой момент – перевести поток в режим ожидания (`sleep`) или остановить его выполнение на определенный период времени. Эту задачу выполняет статический метод `Thread.sleep()`:

```
public class Clock extends Thread {
  java.text.DateFormat f =  // Как отформатировать время для этого региона (locale)
    java.text.DateFormat.getTimeInstance(java.text.DateFormat.MEDIUM);
  volatile boolean keepRunning = true;

  public Clock() {          // Конструктор
    setDaemon(true);        // Поток-демон: интерпретатор, который может
                            // выйти (exit) во время его выполнения
    start();                // Этот поток запускает сам себя на выполнение
  }

  public void run() {       // Тело потока.Этот поток выполняется до тех пор,
    while(keepRunning) {    // пока его не попросят остановиться
      String time = f.format(new java.util.Date()); // Текущее время
      System.out.println(time);                     // Вывести время
      try { Thread.sleep(1000); }                   // Подождать 1000 миллисекунд
      catch (InterruptedException e) {}             // Проигнорировать это исключение
  }
}

// Попросить поток остановиться
  public void pleaseStop() { keepRunning = false; }
}
```

Обратите внимание на метод `pleaseStop()`. Вы можете принудительно прервать выполнение потока, вызвав метод `stop()`, но этот метод не рекомендуется, так как после принудительного завершения могут остаться неактуальные объекты, с которыми работал поток. Если вам нужно прервать выполнение потока, то вы можете определить метод, например `pleaseStop()`, который остановит поток контролируемым способом.

Таймеры

В Java 1.3 классы `java.util.Timer` **и** `java.util.TimerTask` **позволяют еще легче выполнять повторяющиеся задачи. Ниже приведен код, которые ведет себя практически так же, как и предыдущий класс** `Clock`**:**

```
import java.util.*;

// Как отформатировать время для текущего региона
java.text.DateFormat timeFmt =
  java.text.DateFormat.getTimeInstance(java.text.DateFormat.MEDIUM);
// Определить задачу, отображающую время
TimerTask displayTime = new TimerTask() {
  public void run() {
    System.out.println(timeFmt.format(new Date()));
  }
};
// Создать объект-таймер для выполнения задачи (и, возможно, других задач)
Timer timer = new Timer();
// Теперь сообщим ему, что нужно запускать задачу каждые 1000 миллисекунд,
// начиная с текущего момента
Timer.schedule(displayTime, 0, 1000);

// Остановить выполнение задачи
displayTime.cancel();
```

Ожидание окончания выполнения потока

Иногда один поток нужно остановить и дождаться окончания выполнения другого потока. Этого можно добиться с помощью метода `join()`**:**

```
List list; // Длинный список объектов, которые нужно
           // отсортировать; инициализируется в другом месте.

// Определить поток для сортировки списка: его приоритет более
// низкий, поэтому поток выполняется, только если текущий поток
// ожидает ввода/вывода, а затем запускает поток на выполнение
Thread sorter = new BackgroundSorter(list);                 // Определено ранее
sorter.setPriority(Thread.currentThread.getPriority()-1); // Более низкий приоритет
sorter.start();                                             // Начать сортировку

// Тем временем в основном потоке прочитаем данные из файла
byte[] data = readData();                                   // Метод определен в другом месте

// Для продолжения нам нужен отсортированный список, поэтому мы должны дождаться завершения
// потока, выполняющего сортировку, если он еще не завершился
try { sorter.join(); } catch(InterruptedException e) {}
```

Синхронизация потоков

При использовании нескольких потоков вы должны быть очень осторожны, если позволяете более чем одному потоку работать с одними и теми же структурами данных. Подумайте, что произойдет, если один поток попытается перебрать все элементы списка, пока другой поток сортирует эти элементы. Избежать эту проблему позволяет так называемая *синхронизация потоков*, являющаяся одной из центральных задач многопоточных вычислений. Для предотвращения ситуации, когда два потока

получают доступ к одному и тому же объекту в одно и то же время, применяется техника, требующая от потока блокировать объект перед тем, как поток сможет изменять его. Пока один поток блокирует объект, другой поток, которому также требуется заблокировать этот объект, вынужден ждать до тех пор, пока первый поток завершит работу и снимет блокировку. Каждый объект Java имеет встроенную возможность блокировки.

Самый простой способ, позволяющий безопасно работать с объектами в многопоточной среде – объявить все потенциально опасные методы с модификатором `synchronized`. Поток должен заблокировать объект перед тем, как сможет вызвать метод с модификатором `synchronized`. Этот модификатор означает, что никакой другой поток не сможет вызвать ни один метод с модификатором `synchronized` в это же время. Если статический метод объявлен как `synchronized`, то поток должен заблокировать класс, а далее все происходит так же, как и с объектами. Чтобы выполнить выборочную блокировку, вы можете пометить блоки кода модификатором `synchronized`, позволяющим заблокировать объект на короткий период времени:

```
// Этот метод переставляет элементы двух массивов в защищенном блоке (synchronized)
public static void swap(Object[] array, int index1, int index2) {
  synchronized(array) {
    Object tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
  }
}

// Реализации классов Collection, Set, List и Map в пакете java.util не имеют
// синхронизированных методов (за исключением старых реализаций Vector и Hashtable).
// При работе с несколькими потоками вы можете получить
// защищенные интерфейсные объекты (wrapper objects)
List synclist = Collections.synchronizedList(list);
Map syncmap = Collections.synchronizedMap(map);
```

Взаимоблокировка

При синхронизации потоков вы должны быть осторожны, чтобы не допустить *взаимоблокировок* (*deadlock*), когда два потока ожидают снятия блокировки каждым из них. Так как ни один поток не может ни продолжать выполнение, ни снять блокировку, то они оба останавливаются:

```
// Когда два потока пытаются заблокировать два объекта, то может
// возникнуть взаимоблокировка, если только они не
// запрашивают блокировку всегда в одном и том же порядке.
final Object resource1 = new Object();      // Здесь два объекта для
final Object resource2 = new Object();      // блокирования
Thread t1 = new Thread(new Runnable() {     // Заблокировать resource1, а затем resource2
  public void run() {
    synchronized(resource1) {
      synchronized(resource2) { compute(); }
    }
  }
});

Thread t2 = new Thread(new Runnable() {     // Заблокировать resource2, а затем resource1
  public void run() {
    synchronized(resource2) {
```

```
        synchronized(resource1) { compute(); }
      }
    }
  });

  t1.start();  // Блокирует resource1
  t2.start();  // Блокирует resource2, и теперь ни один из потоков не может продолжить
               // выполнение!
```

Координирование потоков с помощью методов wait() и notify()

Иногда потоку нужно остановиться и дождаться, пока не произойдет какое-нибудь событие, сигнализирующее о том, что можно продолжать выполнение. Это делается с помощью методов `wait()` и `notify()`. Однако эти методы принадлежат не классу `Thread`, а классу `Object`. Так как каждый объект в Java может быть заблокирован, то каждый объект может поддерживать список ждущих потоков. Когда поток вызывает метод `wait()` объекта, все блокировки, вызванные потоком, временно снимаются, поток добавляется в список ждущих потоков этого объекта и останавливается. Когда другой поток вызывает метод `notify()` того же объекта, объект будит один из ждущих потоков и позволяет ему продолжить выполнение:

```
/**
* Очередь. Один поток вызывает метод push() для добавления в очередь. Другой вызывает pop()
* для извлечения объекта из очереди. Ели нет данных, то pop() ожидает, пока они не
* появятся, используя методы wait()/notify(). Методы wait() и
* notify() должны применяться в синхронизированном методе или блоке.
*/
import java.util.*;

public class Queue {
LinkedList q = new LinkedList();          // Здесь хранятся объекты
  public synchronized void push(Object o) {
    q.add(o);                             // Добавить объект в конец списка
    this.notify();                        // Сообщить ожидающему потоку, что данные готовы
  }

  public synchronized Object pop() {
    while(q.size() == 0) {
      try { this.wait(); }
      catch (InterruptedException e) { /* Проигнорировать исключения */}
      }
    return q.remove(0);
  }
}
```

Прерывание выполнения потока

В примере, иллюстрирующем методы `sleep()`, `join()` и `wait()`, можно заметить, что вызовы каждого метода заключены в блок `try...catch`, перехватывающий исключение `InterruptedException`. Это необходимо потому, что метод `interrupt()` позволяет одному потоку прервать выполнение другого. Прерывание потока не означает прекращения его работы; оно просто будит поток, находящийся в заблокированном состоянии.

Если для потока, который не заблокирован, вызван метод `interrupt()`, то поток продолжает выполнение, но устанавливается его «статус блокировки», означающий, что получен запрос на прерывание. Поток может проверить свой статус, вызвав статический метод `Thread.interrupted()`, который возвращает `true`, если поток был прерван, и очищает этот статус (побочный эффект). Один поток может проверить статус другого потока, вызвав метод класса `isInterrupted()`, который запрашивает статус, но не очищает его.

Если поток вызывает методы `sleep()`, `join()` или `wait()`, когда статус прерывания установлен, то он не блокируется, а сразу вызывает исключение `InterruptedException` (побочный эффект – очищение статуса). Если для потока, который уже заблокирован вызовами `sleep()`, `join()` или `wait()`, вызван метод `interrupt()`, то поток становится незаблокированным и вызывается исключение `InterruptedException`.

Один из наиболее частых вариантов блокирования потока – выполнение операций ввода/вывода; зачастую поток приостанавливает выполнение в ожидании данных от файловой системы или из сети (`java.io`, `java.net` и `java.nio` API, применяемые при выполнении операций ввода/вывода, обсуждаются далее в этой главе). К сожалению, метод `interrupt()` не будит поток, заблокированный методами ввода/вывода пакета `java.io`. Это один из недостатков пакета `java.io`, который устраняется в новом API ввода/вывода (пакет `java.nio`). Если поток прерван, когда он заблокирован операцией ввода/вывода по любому из каналов, реализованных в `java.nio.channels.InterruptibleChannel`, то канал закрывается. Устанавливается статус прерывания потока и поток засыпает, вызвав исключение `java.nio.channels.ClosedByInterruptException`. Та же ситуация происходит, если поток пытается вызвать блокирующий метод ввода/вывода, когда его статус прерывания активен. Если поток прерван, когда он заблокирован выполнением метода `select()` класса `java.nio.channels.Selector` (или если он вызывает `select()`, когда его статус прерывания установлен), то вызов метода разблокирует поток (или никогда не запустит его на выполнение) и немедленно вернет управление. В этом случае не генерируется никаких исключений; прерванный поток просыпается и метод `select()` возвращает управление.

Файлы и каталоги

Класс `java.io.File` представляет собой файл или каталог и определяет несколько важных методов для работы с файлами и каталогами. Обратите внимание, что ни один метод не позволяет прочитать содержимое файла; это работа класса `java.io.FileInputStream`, который представляет один из множества типов потоков ввода/вывода, применяемых в Java и описанных в следующей главе. Вот несколько примеров того, что можно делать с файлами:

```
import java.io.*;
import java.util.*;

// Получить название домашнего каталога пользователя и представить его как объект File
File homedir = new File(System.getProperty("user.home"));
// Создать объект File для представления файла в этом каталоге
File f = new File(homedir, ".configfile");

// Выяснить размер файла и дату последнего изменения
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());
```

```
// Если файл существует, не является каталогом и доступен для
// чтения, то переместить его во вновь созданный каталог.
if (f.exists() && f.isFile() && f.canRead()) {          // Проверить конфигурационный файл
  File configdir = new File(homedir, ".configdir");     // Новый каталог для
                                                        // конфигурационных файлов
  configdir.mkdir();                                    // Создать этот каталог
  f.renameTo(new File(configdir, ".config"));           // Переместить в него файл
}

// Получить список всех файлов в домашнем каталоге
String[] allfiles = homedir.list();

// Получить список всех файлов, имеющих суффикс «.java»
String[] sourcecode = homedir.list(new FilenameFilter() {
  public boolean accept(File d, String name) {return name.endsWith(".java"); }
});
```

Начиная с Java 1.2 класс `File` **предоставляет дополнительную функциональность:**

```
// Получить список корневых каталогов для всех файловых систем; в Windows мы получим
// File-объекты для всех букв дисков (Java 1.2 и последующие версии).
File[] rootdirs = File.listRoots();

// Атомарно создать lock-файл, а затем удалить его (Java 1.2 и последующие версии)
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
  // Мы удачно создали файл, поэтому нужно что-нибудь сделать
   ...
  // Затем удалим файл
  lock.delete();
}
else {
  // Мы не создали файл; кто-то другой установил блокировку
  System.err.println("Can't create lock file; exiting.");
  System.exit(1);
}

// Создать временный файл, чтобы использовать его во время
// обработки (Java 1.2 и последующие версии)
File temp = File.createTempFile("app", ".tmp");   // Префикс и суффикс файла

// Убедитесь, что файл удален после того, как вы закончили
// работу с ним (Java 1.2 и последующие версии)
temp.deleteOnExit();
```

Класс RandomAccessFile

Пакет `java.io` **также определяет класс** `RandomAccessFile`**, который позволяет читать двоичные данные из любого места в файле. Это может быть полезно в некоторых ситуациях, но большинство приложений читают файлы последовательно, применяя классы потоков, описанные в следующей главе. Ниже приведен пример использования класса** `RandomAccessFile`**:**

```
// Открыть файл с доступом на чтение/запись ("rw")
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100);                        // Перейти к байту номер 100
```

```
byte[] data = new byte[100];        // Создать буфер для хранения данных
f.read(data);                       // Прочитать 100 байт из файла
int i = f.readInt();                // Прочитать четырехбайтное целое число из файла
f.seek(100);                        // Вернутся назад к байту номер 100
f.writeInt(i);                      // Сначала записать целое,
f.write(data);                      // а затем 100 байт
f.close();                          // Закрыть файл, когда работа с ним завершена
```

Потоки ввода/вывода

Пакет java.io определяет большое количество классов для потоковых, или последовательных, чтения и записи данных. Классы InputStream и OutputStream предназначены для чтения и записи потоков байтов, а классы Reader и Writer – для чтения и записи потоков символов. Потоки могут быть вложенными. Это означает, что вы можете читать символы из объекта FilterReader, который читает и обрабатывает символы из базового потока Reader. Поток Reader может читать байты из объекта InputStream и конвертировать их в символы.

Чтение ввода с консоли

С потоками можно выполнять несколько типичных операций. Одна из них – это чтение входных строк, введенных пользователем в консоли:

```
import java.io.*;

BufferedReader console = new BufferedReader(newInputStreamReader(System.in));
System.out.print("Как вас зовут: ");
String name = null;
try {
  name = console.readLine();
}
catch (IOException e) { name = "<" + e + ">";}   // Этого никогда не должно произойти
System.out.println("Hello " + name);
```

Чтение строк из текстового файла

Чтение строк текста из файла – это похожая операция. Следующий код читает весь текстовый файл:

```
String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  String line;
  while((line = in.readLine()) != null) { // Прочитать строку,
                                          // проверить не достигнут ли конец файла
    System.out.println(line);             // Вывести строку
}
  in.close();                   // Всегда закрывайте поток, когда закончили работу с ним
}
catch (IOException e) {
  // Здесь обработать исключение FileNotFoundException и т. д.
}
```

Запись текста в файл

На протяжении всей книги вы видели применение метода `System.out.println()` для отображения текста в консоли. `System.out` просто ссылается на поток вывода. Подобным образом вы можете выводить текст в любой поток вывода. Следующий код показывает, как вывести текст в файл:

```
try {
  File f = new File(homedir, ".config");
  PrintWriter out = new PrintWriter(new FileWriter(f));
  out.println("## Автоматически сгенерированный файл конфигурации. НЕ РЕДАКТИРОВАТЬ!");
  out.close();       // Мы закончили запись
}
catch (IOException e) { /* Обработать исключения */ }
```

Чтение двоичного файла

Как известно, не все файлы содержат текст. Следующие строки кода работают с файлом, как с потоком байтов, и читают байты в большой массив:

```
try {
  File f;                           // Файл для чтения; инициализируется в другом месте
  int filesize = (int) f.length();  // Сохранить размер файла в переменной
  byte[] data = new byte[filesize]; // Создать массив достаточного размера
  // Создать поток для чтения файла
  DataInputStream in = new DataInputStream(new FileInputStream(f));
  in.readFully(data);               // Прочитать содержимое файла в массив
  in.close();
}
catch (IOException e) { /* Обработать исключения */ }
```

Сжатие данных

Различные пакеты Java-платформы определяют специализированные потоковые классы, оперирующие потоковыми данными. Следующий код показывает, как применять потоковые классы из пакета `java.util.zip` для подсчета контрольной суммы данных и последующего сжатия данных при записи в файл:

```
import java.io.*;
import java.util.zip.*;

try {
  File f;      // Файл для записи; инициализируется в другом месте
  byte[] data; // Данные для записи; инициализируются в другом месте
  Checksum check = new Adler32(); // Объект для подсчета контрольной суммы

  // Создать поток, который пишет байты в файл f
  FileOutputStream fos = new FileOutputStream(f);
  // Создать поток для сжатия байтов и записи их в fos
  GZIPOutputStream gzos = new GZIPOutputStream(fos);
  // Создать поток, подсчитывающий контрольную сумму байтов и записывающий их в gzos
  CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

  cos.write(data);             // Теперь запишем данные во вложенный поток
  cos.close();                 // Закрыть цепочку вложенных потоков
  long sum = check.getValue(); // Получить подсчитанную контрольную сумму
}
catch (IOException e) { /* Обработать исключения */ }
```

Чтение ZIP-файлов

Пакет `java.util.zip` также содержит класс `ZipFile`, позволяющий получать доступ к произвольным элементам ZIP-архива и читать их через поток:

```
import java.io.*;
import java.util.zip.*;

String filename;     // Файл для чтения; инициализируется в другом месте
String entryname;    // Элемент для чтения из ZIP-файла; инициализируется в другом месте
ZipFile zipfile = new ZipFile(filename);                // Открыть ZIP-файл
ZipEntry entry = zipfile.getEntry(entryname);           // Получить один элемент
InputStream in = zipfile.getInputStream(entry);         // Поток для чтения элемента
BufferedInputStream bis = new BufferedInputStream(in);  // Повышает эффективность
// Теперь читаем байты из bis...
// Вывести содержимое ZIP-файла
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
  ZipEntry zipentry = (ZipEntry) e.nextElement();
  System.out.println(zipentry.getName());
}
```

Вычисление профилей сообщений (Message Digests)

Если вам нужно вычислить криптостойкую контрольную сумму (известную как профиль сообщения), применяйте потоковые классы пакета `java.security`. Например:

```
import java.io.*;
import java.security.*;
import java.util.*;

File f; // Файл для чтения и источник для вычисления профиля;
        // инициализируется в другом месте
List text = new ArrayList(); // Здесь мы будем хранить строки текста

// Получить объект, который может вычислить профиль сообщения SHA
MessageDigest digester = MessageDigest.getInstance("SHA");
// Поток для чтения байтов из файла f
FileInputStream fis = new FileInputStream(f);
// Поток для чтения байтов из fis и вычисления профиля сообщения SHA
DigestInputStream dis = new DigestInputStream(fis, digester);
// Поток для чтения байтов из dis и конвертации их в символы
InputStreamReader isr = new InputStreamReader(dis);
// Поток, который может читать по одной строке за один раз
BufferedReader br = new BufferedReader(isr);
// Теперь прочитаем строки
for(String line; (line = br.readLine()) != null; text.add(line));
// Закрыть потоки
br.close();
// Получить профиль сообщения
byte[] digest = digester.digest();
```

Потоковое чтение данных из массива и запись в массив

До настоящего времени мы применяли множество потоковых классов для работы с потоковыми данными, но данные в конечном счете приходят из файлов или записы-

ваются в консоль. Пакет java.io определяет другие потоковые классы, которые могут читать данные из массивов байтов или строк текста и записывать их обратно:

```
import java.io.*;

// Создать поток, который использует массив байтов как место назначения
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");             // Записать строку в байтах
out.writeDouble(Math.PI);          // Записать вещественное значение в байтах
byte[] data = baos.toByteArray(); // Получить записанный массив байтов
out.close();                       // Закрыть потоки

// Создать поток для чтения символов из строки
Reader in = new StringReader("Now is the time!");
// Читать символы из потока
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

Подобным образом работают классы ByteArrayInputStream, StringWriter, CharArrayReader и CharArrayWriter.

Передача данных между потоками с использованием каналов (pipes)

Классы PipedInputStream, PipedOutputStream и их символьные эквиваленты, PipedReader и PipedWriter, представляют собой другой интересный набор потоков, определенный в java.io. Эти потоки записывают байты в поток PipedOutputStream или символы в канал PipedWriter, а другой поток читает байты или символы из соответствующих PipedInputStream или PipedReader:

```
// Пара соединенных канальных потоков ввода/вывода формируют канал. Один поток пишет
// байты в PipedOutputStream, а другой читает их из соответствующего PipedInputStream.
// Для символов применяется PipedWriter/PipedReader.
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new PipedInputStream(writeEndOfPipe);

// Этот поток читает байты из канала и игнорирует их
Thread devnull = new Thread(new Runnable() {
  public void run() {
    try { while(readEndOfPipe.read() != -1);}
    catch (IOException e) {}             // Игнорировать их
  }
});
devnull.start();
```

Сериализация

Одной из наиболее важных особенностей пакета java.io является возможность *сериализовать* (*serialize*) объект: преобразование объекта в поток байтов с последующим восстановлением (deserialization) копии исходного объекта. Следующий код показывает, как использовать сериализацию для записи объектов в файл и последующего чтения:

```
Object o; // Объект для сериализации; должен реализовывать интерфейс Serializable
File f;   // Файл для записи
```

```
try {
  // Сериализовать объект
  ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(f));
  oos.writeObject(o);
  oos.close();

  // Прочитать объект
  ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f));
  Object copy = ois.readObject();
  ois.close();
}
catch (IOException e) { /* Обработать исключения ввода/вывода */ }
catch (ClassNotFoundException cnfe) {
  /* Метод readObject() может генерировать это исключение*/
}
```

Предыдущий пример сериализует объект в файл. Помните, что вы можете сериализовать объект в поток любого типа. Таким образом, можно записывать объекты в массив байтов, а затем читать их из массива, создавая точную копию объекта. Вы можете записать объект в сжатый поток или даже в поток, соединенный по сети с другой программой!

Постоянство JavaBeans

Java 1.4 представляет новый механизм сериализации, необходимый при использовании компонентов JavaBeans. Сериализация `java.io` сохраняет состояние внутренних полей объекта. С другой стороны, постоянство (persistence) `java.beans` сохраняет состояние как последовательность вызовов открытых методов, определенных в классе. Так как постоянство основано на открытом API, а не на внутреннем состоянии, то механизм постоянства JavaBeans дает возможность взаимодействовать разным реализациям одного и того же API. Он позволяет более надежно обрабатывать ситуации несоответствия версий и служит удобным долгосрочным хранилищем сериализованных объектов.

Компонент, его наследники или другие объекты сериализуются с помощью класса `java.beans.XMLEncoder` и могут быть восстановлены с помощью `java.beans.XMLDecoder`. Эти классы записывают данные в определенные потоки и читают из потоков, но сами по себе не являются потоковыми классами. Вот пример того, как вы можете закодировать компонент:

```
// Создать JavaBean и задать некоторые его свойства
javax.swing.JFrame bean = new javax.swing.JFrame("PersistBean");
bean.setSize(300, 300);
// Сохранить его в файл bean.xml
BufferedOutputStream out =                // Создать выходной поток
    new BufferedOutputStream(new FileOutputStream("bean.xml"));
XMLEncoder encoder = new XMLEncoder(out); // Создать XMLEncoder для потока
encoder.writeObject(bean);                // Сериализовать bean
encoder.close();                          // Закрыть XMLEncoder и поток
```

А вот пример восстановления сериализованного компонента:

```
BufferedInputStream in =                  // Создать входной поток
    new BufferedInputStream(new FileInputStream("bean.xml"));
XMLDecoder decoder = new XMLDecoder(in);  // Создать объект для восстановления
Object b = decoder.readObject();          // Восстановить компонент
```

```
decoder.close();                        // Закрыть все
bean = (javax.swing.JFrame) b;          // Привести тип компонента к нужному типу
bean.setVisible(true);                  // Применять его
```

Работа в сети

Пакет java.net определяет несколько классов, которые позволяют на удивление легко создавать сетевые приложения. Далее приведены разнообразные примеры.

Работа в сети с классом URL

Самый простой сетевой класс – это URL. Он представляет собой универсальный указатель на информационный ресурс (uniform resource locator). Различные реализации Java могут поддерживать разные наборы URL-протоколов. Как минимум, можно рассчитывать на поддержку протоколов http://, ftp:// и file://. В Java 1.4 безопасный HTTP поддерживается протоколом https://. Вот несколько способов применения класса URL:

```
import java.net.*;
import java.io.*;

// Создать объект URL
URL url=null, url2=null, url3=null;
try {
  url = new URL("http://www.oreilly.com");        // Абсолютный URL
  url2 = new URL(url, "catalog/books/javanut4/"); // Относительный URL
  url3 = new URL("http:", "www.oreilly.com", "index.html");
} catch (MalformedURLException e) { /* Проигнорировать */ }

// Прочитать содержимое URL из входного потока
InputStream in = url.openStream();
// Для более полного контроля над процессом чтения применяйте объект URLConnection
object URLConnection conn = url.openConnection();

// Теперь получим немного информации об URL
String type = conn.getContentType();
String encoding = conn.getContentEncoding();
java.util.Date lastModified = new java.util.Date(conn.getLastModified());
int len = conn.getContentLength();

// Если необходимо, прочтем содержимое URL из потока
InputStream in = conn.getInputStream();
```

Работа с сокетами

Иногда нужен более жесткий контроль над сетевым приложением, нежели возможности контроля, предоставляемые класс URL. В таком случае для непосредственной связи с сервером можно использовать класс Socket. Например:

```
import java.net.*;
import java.io.*;

// Здесь приведена простая клиентская программа, которая соединяется с веб-сервером,
// запрашивает документ и читает его
String hostname = "java.oreilly.com";      // Сервер, с которым соединяемся
```

```
int port = 80;                                  // Стандартный порт для HTTP
String filename = "/index.html";                // Файл, который нужно прочитать с сервера
Socket s = new Socket(hostname, port);          // Соединиться с сервером

// Получить потоки ввода/вывода, которые мы можем задействовать для общения с сервером
InputStream sin = s.getInputStream();
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sin));
OutputStream sout = s.getOutputStream();
PrintWriter toServer = new PrintWriter(new OutputStreamWriter(sout));

// Запросить файл с сервера, используя протокол HTTP
toServer.print("GET " + filename + " HTTP/1.0\r\n\r\n");
toServer.flush();

// Прочитать ответ сервера, предполагая, что это текстовый файл, и вывести его
for(String l = null; (l = fromServer.readLine()) != null; )
  System.out.println(l);

// Закрыть все по окончании работы
toServer.close();
fromServer.close();
s.close();
```

Защищенные сокеты и SSL

В Java 1.4 расширение защищенных сокетов для Java (Java Secure Socket Extension или JSSE) было добавлено в ядро Java-платформы, а именно в пакеты `javax.net` и `javax.net.ssl`.[1] Этот API позволяет шифровать сетевое соединение через сокеты, использующие протокол SSL (Secure Socket Layer, также известный как TLS). SSL широко применяется в Интернете. Он является основой безопасных сетевых коммуникаций с использованием протокола `https://`. В Java 1.4 и последующих версиях протокол `https://` можно применять совместно с классом `URL` для безопасной загрузки документов с серверов, поддерживающих SSL.

Как и все Java API, имеющие отношение к безопасности, JSSE является полностью конфигурируемым. Он контролируется на нижнем уровне, позволяя проводить самую тонкую настройку и управлять установлением соединения через сокет SSL. Пакеты `javax.net` и `javax.net.ssl` достаточно сложны, но на практике для установления безопасного соединения с серверами вам пригодятся всего несколько классов. Следующая программа – это вариант предыдущего кода, который задействует HTTPS вместо HTTP для безопасной передачи содержимого запрошенного URL:

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;
import java.security.cert.*;

/**
* Получить документ с удаленного сервера, используя HTTPS. Вызов:
*  java HttpsDownload <hostname> <filename>
**/
```

[1] Ранняя версия JSSE, использующая разные имена пакетов, доступна в форме отдельно загружаемого модуля по адресу *http://java.sun.com/products/jsse/*. Эту версию можно применять в Java 1.2 или Java 1.3.

```
public class HttpsDownload {
  public static void main(String[] args) throws IOException {
    // Получить объект SocketFactory для создания SSL-сокета
    SSLSocketFactory factory =
           (SSLSocketFactory)SSLSocketFactory.getDefault();

    // Использовать фабрику для создания безопасного сокета, соединенного с HTTPS-портом
    // заданного сервера.
    SSLSocket sslsock=  (SSLSocket)factory.createSocket(args[0], // Сервер
                                                        443); // HTTPS-порт

    // Получить сертификат, предоставляемый сервером
    SSLSession session = sslsock.getSession();
    X509Certificate cert;
    try { cert = (X509Certificate) session.getPeerCertificates()[0]; }
    catch(SSLPeerUnverifiedException e) {
      // Если сертификат не найден или недействителен.
      System.err.println(session.getPeerHost() +
                     " не предоставил действительный сертификат.");
      return;
    }

    // Отобразить детальную информацию о сертификате
    System.out.println(session.getPeerHost() +
                     " предоставил сертификат, принадлежащий:");
    System.out.println("\t[" + cert.getSubjectDN().getName() + "]");
    System.out.println("Сертификат содержит действительную сигнатуру:");
    System.out.println("\t[" + cert.getIssuerDN().getName() + "]");

    // Если пользователь не доверяет сертификату, то прервать процесс
    System.out.print("Доверяете ли вы этому сертификату (y/n)? ");
    System.out.flush();
    BufferedReader console =
      new BufferedReader(new InputStreamReader(System.in));
    if (Character.toLowerCase(console.readLine().charAt(0)) != 'y') return;

    // Теперь задействуйте безопасный сокет, как будто вы  работаете с обычным сокетом.
    // Во-первых, отправьте обычный HTTP-запрос через SSL-сокет
    PrintWriter out = new PrintWriter(sslsock.getOutputStream());
    out.print("GET " + args[1] + " HTTP/ 1.0\r\n\r\n");
    out.flush();

    // Далее прочитать ответ сервера и вывести его на консоль
    BufferedReader in =
      new BufferedReader(new InputStreamReader(sslsock.getInputStream()));
    String line;
    while((line = in.readLine()) != null) System.out.println(line);

    // Наконец, закрыть сокет
    sslsock.close();
  }
}
```

Серверы

Клиентское приложение применяет Socket для связи с сервером. Сервер выполняет то же самое: он использует объект Socket для связи с каждым клиентом. Однако сервер выполняет дополнительную задачу – он должен распознать и принять запросы клиентов на соединение. Эту задачу выполняет класс ServerSocket. Следующий код демонстрирует работу с классом ServerSocket. Код реализует простой HTTP-сервер, который отвечает на все запросы, отсылая обратно содержимое HTTP-запроса. Подобный макет сервера полезен при отладке HTTP-клиентов:

```
import java.io.*;
import java.net.*;

public class HttpMirror {
public static void main(String[] args) {
  try {
    int port = Integer.parseInt(args[0]);        // Порт для прослушивания
    ServerSocket ss = new ServerSocket(port);    // Создать сокет для прослушивания
    for(;;) {                                    // Вечный цикл
      Socket client = ss.accept();               // Ждать соединения
      ClientThread t = new ClientThread(client); // Поток для его обработки
      t.start();                                 // Запустить поток на выполнение
    }                                            // Следующий запрос
  }
  catch (Exception e) {
    System.err.println(e.getMessage());
    System.err.println("Вызов: java HttpMirror <port>;");
  }
}

static class ClientThread extends Thread {
  Socket client;
  ClientThread(Socket client) { this.client = client; }
  public void run() {
    try {
      // Получить поток для общения с клиентом
      BufferedReader in =
        new BufferedReader(new InputStreamReader(client.getInputStream()));
      PrintWriter out =
        new PrintWriter(new OutputStreamWriter(client.getOutputStream()));

      // Отправить клиенту заголовок HTTP-ответа
      out.print("HTTP/1.0 200\r\nContent-Type: text/plain\r\n\r\n");
      // Прочитать HTTP-запрос и отправить его обратно.
      // Остановиться, когда прочитана пустая строка, означающая конец запроса
      // и его заголовка.
      String line;
      while((line = in.readLine()) != null) {
        if (line.length() == 0) break;
        out.println(line);
      }

      out.close();
      in.close();
      client.close();
    }
```

```
      catch (IOException e) { /* Игнорировать исключения */ }
    }
  }
}
```

С помощью JSSE код этого сервера может быть изменен, чтобы добавить поддержку SSL-соединений. Обеспечить безопасность сервера труднее, чем безопасность клиента, поскольку сервер должен иметь сертификат, предоставляемый клиенту. Серверная часть JSSE здесь не продемонстрирована.

Дейтаграммы

Классы URL и Socket работают на верхнем уровне сетевого соединения, основанного на потоках. Тем не менее настройка и сопровождение потока выполняются на сетевом уровне. Иногда вам нужен низкоуровневый доступ, чтобы увеличить скорость прохождения пакетов по сети, но вы не хотите заботиться о поддержании потока. Кроме того, если вам не нужна гарантия того, что данные попадут куда нужно или что пакеты данных прибудут в том порядке, в котором вы их отправляли, то обратите внимание на классы DatagramSocket и DatagramPacket:

```
import java.net.*;

// Отправить сообщение на другой компьютер через дейтаграммы
try {
  String hostname = "host.example.com"; // Адрес компьютера, на который отправляется сообщение
  InetAddress address =                 // Получить IP-адрес по имени через DNS
    InetAddress.getByName(hostname);
  int port = 1234;                      // Порт, с которым соединяемся
  String message = "Орел приземлился."; // Сообщение
  byte[] data = message.getBytes();     // Конвертировать строку в байты
  DatagramSocket s = new DatagramSocket(); // Сокет, через который отправляем сообщение
  DatagramPacket p =                    // Создать пакет для отправки
    new DatagramPacket(data, data.length, address, port);
  s.send(p);                            // Отправить!
  s.close();                            // Всегда закрываем сокет по завершении работы
}
catch (UnknownHostException e) {}       // Генерируется методом InetAddress.getByName()
catch (SocketException e) {}            // Генерируется оператором new DatagramSocket()
catch (java.io.IOException e) {}        // Генерируется методом DatagramSocket.send()

// Здесь пример того, как другой компьютер может получить дейтаграмму
try {
  byte[] buffer = new byte[4096];             // Буфер для хранения данных
  DatagramSocket s = new DatagramSocket(1234); // Принимающий сокет
  DatagramPacket p =
    new DatagramPacket(buffer, buffer.length); // Получаемый пакет
  s.receive(p);                               // Дождаться получения пакета
  String msg =                                // Конвертировать байты из
    new String(buffer, 0, p.getLength());     // пакета обратно в строку.
  s.close(); // Всегда закрываем сокет
}
catch (SocketException e) {}                 // Генерируется оператором new DatagramSocket()
catch (java.io.IOException e) {}             // Генерируется методом DatagramSocket.receive()
```

Свойства и предпочтения

Класс `java.util.Properties` **является подклассом класса** `java.util.Hashtable` **— старого класса коллекций, предшествующего API коллекциям в Java 1.2. Объект** `Properties` **обеспечивает соответствие строковых ключей и строковых значений. Он определяет методы, которые позволяют записывать свойства в простой текстовый файл и читать их. Такая возможность делает класс** `Properties` **идеальным средством для работы с конфигурационными файлами и файлами настроек пользователя. Класс** `Properties` **также применяется для системных свойств, возвращаемых методом** `System.getProperty()`**:**

```
import java.util.*;
import java.io.*;

// Обратите внимание: многие из этих системных свойств могут сгенерировать исключение
// безопасности, если будут вызываться из ненадежного кода, например апплета.
String homedir = System.getProperty("user.home"); // Получить одно системное свойство
Properties sysprops = System.getProperties();     // Получить все системные свойства

// Вывести имена всех определенных системных свойств
for(Enumeration e = sysprops.propertyNames(); e.hasMoreElements();)
  System.out.println(e.nextElement());

sysprops.list(System.out);   // Вот еще более легкий способ получить список системных свойств

// Прочитать свойства из конфигурационного файла
Properties options = new Properties();                    // Пустой список свойств
File configfile = new File(homedir, ".config");           // Конфигурационный файл
try {
  options.load(new FileInputStream(configfile));          // Загрузить свойства из файла
} catch (IOException e) { /* Обработать исключения */ }

// Получить значение свойства "color". Если не определено, указать
// значение по умолчанию "gray"
String color = options.getProperty("color", "gray");

// Установить значение свойства "color" в "green"
options.setProperty("color", "green");

// Сохранить содержимое объекта Properties в файле
try {
  options.store(new FileOutputStream(configfile),       // Выходной поток
                "MyApp Config File");                   // Текст комментария заголовка файла
} catch (IOException e) { /* Обработать исключения */ }
```

Предпочтения

Java 1.4 представляет новый API предпочтений (Preferences API), который специально приспособлен для работы с пользовательскими и системными настройками. Для подобных целей он полезнее `Properties`**. API определен в пакете** `java.util.prefs`**. Ключевой класс в этом пакете —** `Preferences`**. Вы можете получить объект** `Preferences` **с пользовательскими предпочтениями, вызвав статический метод** `Preferences.userNodeForPackage()`**. Объект, содержащий системные предпочтения, можно получить с помощью метода** `Preferences.systemNodeForPackage()`**. Оба метода принимают объект** `java.lang.Class` **в качестве единственного аргумента и возвращают объект** `Preferences`**, который совместно используется всеми классами в данном пакете. Это означает, что имена используемых параметров должны быть уникальны внутри пакета. Получив**

объект Preferences, вызовите метод get() для получения строкового значения предпочтения по имени или методы, возвращающие значения других типов, например getInt(), getBoolean() и getByteArray(). Обратите внимание, что для получения значения предпочтения всем методам должно быть передано значение по умолчанию. Это значение возвращается, если предпочтение с запрашиваемым именем не зарегистрировано либо файл или база данных, хранящие предпочтения, не доступны. Типичный вариант использования Preferences показан ниже:

```
package com.davidflanagan.editor;
import java.util.prefs.Preferences;

public class TextEditor {
  // Поля, которые нужно инициализировать из значений предпочтений
  public int width;          // Ширина экрана в символах
  public String dictionary; // Имя словаря для проверки орфографии

  public void initPrefs() {
    // Получить объект Preferences для предпочтений пользователя  и системы в текущем пакете
    Preferences userprefs = Preferences.userNodeForPackage(TextEditor.class);
    Preferences sysprefs = Preferences.systemNodeForPackage(TextEditor.class);

    // Найти значения предпочтений. Обратите внимание, что
    // всегда передаются значения по умолчанию.
    width = userprefs.getInt("width", 80);
    // Найти пользовательское значение, используя системное
    // значение как значение по умолчанию.
    dictionary = userprefs.get("dictionary",
                               sysprefs.get("dictionary",
                                            "default_dictionary"));
  }
}
```

В дополнение к методу get(), применяемому для получения значений предпочтений, соответствующий метод put() используется для установки значений именованных предпочтений:

```
// Пользователь ввел новое предпочтение; сохраним его
userprefs.putBoolean("autosave", false);
```

Если вашему приложению нужно, чтобы его проинформировали, когда при выполнении приложения изменится пользовательское или системное предпочтение, то оно может зарегистрировать PreferenceChangeListener методом addPreferenceChangeListener(). Объект Preferences может экспортировать имена и значения своих предпочтений в XML-файл и читать параметры из такого же файла (см. методы importPreferences(), exportNode() и exportSubtree() пакета java.util.pref.Preferences в главе 17). Объекты Preferences обычно представлены в иерархии, соответствующей иерархии имен пакетов. Методы для навигации по иерархии существуют, но обычно не используются приложениями.

Протоколирование

Другой новой возможностью Java 1.4 является API протоколирования (Logging API), определенный в пакете java.util.logging. Обычно разработчики приложений используют объект Logger с именем, соответствующим имени класса или пакета приложения, для регистрации сообщений на любом из семи уровней (см. класс Level в

предметном указателе). Эти сообщения могут информировать об ошибках и предупреждать о чем-либо. Они предоставляют информацию об интересующих событиях в жизненном цикле приложения. Они могут включать отладочную информацию или даже отслеживать выполнение важных методов программы.

Системный администратор или конечный пользователь приложения отвечает за настройку конфигурационного файла, который определяет направление передачи сообщений (консоль, файл, сетевой сокет или их комбинация), их формат (простой текст или XML-документ) и уровень протоколирования (сообщения с уровнем серьезности ниже указанного игнорируются и не влияют на процесс выполнения приложения). Уровень протоколирования может быть задан отдельно для каждого именованного объекта `Logger`. На уровне конечного пользователя такая конфигурируемость означает, что вы можете писать программы, выводящие диагностические сообщения. Обычно эти сообщения игнорируются, но они могут регистрироваться при разработке приложения или появлении проблем в приложении, установленном у пользователя. Протоколирование очень важно для таких приложений, как серверы, которые работают без участия человека и не имеют графического интерфейса пользователя.

В большинстве приложений использовать API протоколирования достаточно просто. Получите именованный объект `Logger`. Для этого вызовите метод `Logger.getLogger()` и передайте ему имя пакета или класса приложения. Затем примените один из методов экземпляра `Logger` для протоколирования сообщений. Самый простой путь – выбирать имена, соответствующие различным уровням серьезности событий, например `severe()`, `warning()` и `info()`:

```
import java.util.logging.*;

// Получить объект Logger с именем текущего пакета
Logger logger = Logger.getLogger("com.davidflanagan.servers.pop");
logger.info("Starting server.");      // Зарегистрировать информационное сообщение
ServerSocket ss;                      // Что-нибудь выполнить
try { ss = new ServerSocket(110); }
catch(Exception e) {                  // Зарегистрировать исключение
  logger.log(Level.SEVERE, "Не могу открыть порт 110", e);      // Комплексное сообщение
  logger.warning("Выходим");                                    // Простое предупреждение
  return;
}
logger.fine("получен серверный сокет");   // Низкоуровневое отладочное сообщение
```

Новый API ввода/вывода

Java 1.4 представляет абсолютно новый API для высокопроизводительного, неблокирующего ввода/вывода и работы с сетью. Главным образом этот API состоит из трех новых пакетов. Пакет `java.nio` определяет класс `Buffer`, который используется для хранения последовательности байтов или значений других базовых типов. Пакет `java.nio.channels` определяет *каналы*, через которые можно передавать данные между буфером и источником данных или приемником, таким как файл или сетевой сокет. Кроме того, этот пакет содержит важные классы, используемые для неблокирующего ввода/вывода. И наконец, пакет `java.nio.charset` содержит классы для эффективной конвертации буфера байтов в буфер символов. Следующие подразделы содержат примеры использования этих пакетов и примеры выполнения специфичных задач ввода/вывода с помощью нового API ввода/вывода.

Простые операции с буфером

Пакет `java.nio` включает абстрактный класс `Buffer`, который определяет общие операции над буферами. Этот пакет также определяет подклассы для разных типов данных, таких как `ByteBuffer`, `CharBuffer` и `IntBuffer`. Описание классов `Buffer` и `ByteBuffer`, представленное в главе 14, содержит подробную информацию об этих классах и их разнообразных методах. Следующий код иллюстрирует типичную последовательность операций с объектом `ByteBuffer`. Классы буферов для других типов данных имеют похожие методы.

```
import java.nio.*;

// У буферов нет открытых конструкторов. Вместо этого под них выделяется память.
ByteBuffer b = ByteBuffer.allocate(4096);
// Создать буфер размером 4096 байт
// Или попробовать получить буфер от операционной системы нижнего уровня
ByteBuffer buf2 = ByteBuffer.allocateDirect(65536);
// Вот другой способ получить буфер: «обертываем» массив
byte[] data;                // Полагаем, что массив создан и проинициализирован в другом месте
ByteBuffer buf3 = ByteBuffer.wrap(data);    // Создать буфер,использующий массив
// Можно создать «буфер просмотра» для просмотра байтов, приведя их к другому типу
buf3.order(ByteOrder.BIG_ENDIAN);           // Указать порядок байтов для буфера
IntBuffer ib = buf3.asIntBuffer();          // Просмотреть байты как целые числа

// Теперь сохраним какие-нибудь данные в буфере
b.put(data);                    // Скопируем байты из массива в текущую позицию буфера
b.put((byte)42);                // Сохраним другой байт в новой текущей позиции
b.put(0, (byte)9);              // Перезапишем первый байт в буфере. Позиция не изменяется.
b.order(ByteOrder.BIG_ENDIAN); // Установим порядок байтов
b.putChar('x');                 // Сохраним два байта символа Unicode в буфере
b.putInt(0xcafebabe);           // Сохраним четыре байта целого числа в буфере

// Вот несколько методов для получения простейшей информации о буфере
int capacity = b.capacity();   // Сколько байт может хранить буфер? (4096)
int position = b.position();   // Какой байт будет записан или прочитан следующим?
// Лимит буфера указывает, сколько байт может хранить буфер. При записи в буфер это значение
// должно быть размером буфера. При чтении – количеством байт, записанных ранее.
int limit = b.limit();         // Сколько нужно использовать?
int remaining = b.remaining(); // Сколько осталось? Вернуть крайнюю позицию.
boolean more=b.hasRemaining(); // Проверить, есть ли еще место в буфере

// Позиция и лимит также могут быть установлены соответствующими методами
// Предположим, вы хотите прочитать байты, записанные в буфер
b.limit(b.position());         // Присвоить лимиту текущую позицию
b.position(0);                 // Установить лимит в 0; приступить к чтению с начала

// Вместо двух предыдущих шагов вы обычно применяете удобный метод
b.flip();   // Присвоить лимиту позицию, а позицию установить в 0; приготовиться к чтению
b.rewind(); // Установить позицию в 0; не менять лимит; приготовиться к повторному чтению
b.clear();  // Установить позицию в 0, а лимиту присвоить размер; приготовиться к записи

// Предполагаем, что вы вызвали flip(); вы можете начать чтение байтов из буфера
buf2.put(b);       // Прочитать все байты из b и поместить их в buf2
b.rewind();            // Вернуться к началу b для повторного чтения
byte b0 = b.get();     // Прочитать первый байт; увеличить позицию на 1
byte b1 = b.get();     // Прочитать второй байт; увеличить позицию на 1
```

```
byte[] fourbytes = new byte[4];
b.get(fourbytes);       // Прочитать четыре байта; увеличить позицию на 4
byte b9 = b.get(9);     // Прочитать 10-й байт без изменения позиции
int i = b.getInt();     // Прочитать четыре байта; увеличить позицию на 4

// Отбросить уже прочитанные байты; сдвинуть оставшиеся к началу буфера; присвоить позиции
// новый лимит, а лимиту – размер; приготовить буфер для записи байтов.
b.compact();
```

Обратите внимание, что многие методы возвращают объект того же типа, с которым оперируют. Это сделано для того, чтобы вызовы методов можно было выстраивать в цепочки:

```
ByteBuffer bb = ByteBuffer.allocate(32).order(ByteOrder.BIG_ENDIAN).putInt(1234);
```

Многие методы в пакете `java.nio` и его подпакетах возвращают текущий объект, чтобы методы можно было вызывать по цепочке. Обратите внимание, что использование подобных вызовов – это вопрос выбора стиля, который не оказывает существенного влияния на производительность.

Класс `ByteBuffer` является самым важным буферным классом. Другой часто используемый класс – `CharBuffer`. Объект `CharBuffer` можно создать из строки и конвертировать в строку. `CharBuffer` реализует новый интерфейс `java.lang.CharSequence`. Это означает, что в определенных приложениях его можно применять как `String` или `StringBuffer` (например, для проверки соответствия паттерну с помощью регулярных выражений).

```
// Создать из строки буфер CharBuffer «только для чтения»
CharBuffer cb = CharBuffer.wrap("Эта строка содержит данные для CharBuffer");
String s = cb.toString(); // Конвертировать в String с помощью метода toString()
System.out.println(cb);   // или положиться на неявный вызов метода toString().
char c = cb.charAt(0);    // Использовать методы CharSequence для получения символов
char d = cb.get(1);       // или применить CharBuffer для абсолютного чтения.
// Относительное чтение, при котором считываются символы и инкрементируется текущая позиция
// Обратите внимание, что для CharSequence или при конвертации CharBuffer в String
// используются только символы между позицией и лимитом
char e = cb.get();
```

Байты в `ByteBuffer` часто конвертируются в символы `CharBuffer`, и наоборот. Мы увидим, как это делается, когда будем рассматривать пакет `java.nio.charset`.

Простые операции с каналами

Сами по себе буферы не так уж полезны – нет ничего особенного в хранении байтов только для того, чтобы прочесть их. Обычно буферы применяют вместе с каналами: ваша программа сохраняет байты в буфер, затем передает его каналу, который считывает байты из буфера и записывает их в файл, сетевой сокет или другой приемник. Или наоборот, ваша программа передает буфер каналу, который читает байты из файла, сокета или другого источника и сохраняет эти байты в буфере, из которого они могут быть получены программой. Пакет `java.nio.channels` определяет несколько классов, представляющих файл, сокет, дейтаграмму или канал (pipe). Далее в этой главе мы увидим примеры использования этих классов. Впрочем, следующий код опирается на различные возможности интерфейсов каналов, определенных в `java.nio.channels`, и должен работать с любым объектом `Channel`:

```
Channel c; // Объект, реализующий интерфейс Channel; инициализирован в другом месте
if (c.isOpen()) c.close();          // Это методы, определенные классом Channel
```

```
// Методы read() и write() определены интерфейсами ReadableByteChannel и WritableByteChannel.
ReadableByteChannel source;         // Инициализирован в другом месте
WritableByteChannel destination;    // Инициализирован в другом месте
ByteBuffer buffer = ByteBuffer.allocateDirect(16384); // Буфер низкого уровня размером 16 Кбайт

// Вот базовый цикл для чтения байтов из канала-источника и записи их в канал-приемник.
// Выполняется до тех пор, пока в источнике не останется байтов для чтения, а в буфере -
// байтов для записи.
while(source.read(buffer) != -1 || buffer.position() > 0) {
  // Перевернуть буфер: присвоить лимиту позицию, а позицию установить в 0.
  // Это подготовит буфер к чтению, которое выполняется методом *write* канала).
  buffer.flip();
  // Записать в приемник несколько или все байты, находящиеся в буфере
  destination.write(buffer);
  // Отбросить уже записанные байты, скопировав оставшиеся в начало буфера.
  // Установить позицию в лимит, а лимит в размер, подготовив буфер к записи
  // (выполняется методом *read* канала).
  buffer.compact();
}

// Не забыть закрыть каналы
source.close();
destination.close();
```

В дополнение к интерфейсам `ReadableByteChannel` и `WritableByteChannel`, проиллюстрированным в предыдущем коде, пакет `java.nio.channels` определяет несколько других интерфейсов. `ByteChannel` просто расширяет интерфейсы для чтения и записи без добавления новых методов. Это полезная особенность каналов, которые поддерживают чтение и запись. `GatheringByteChannel` — это расширение `WritableByteChannel`. Он определяет методы `write()`, *собирающие* байты из нескольких буферов и записывающие их. Подобным образом, `ScatteringByteChannel` — это расширение `ReadableByteChannel`. Он определяет методы `read()`, читающие байты из канала и *разбрасывающие*, или распределяющие, их по нескольким буферам. Собирающие и разбрасывающие методы `write()`и `read()` полезны при работе с сетевыми протоколами. Эти протоколы используют заголовки фиксированного размера, которые нужно сохранить в буфере отдельно от остальных данных.

Есть одно обстоятельство, которое сбивает с толку. Оно заключается в том, что операция чтения из канала включает в себя запись (или помещение) байтов в буфер, а операция записи в канал — чтение (или сбор) байтов из буфера. Таким образом, когда я говорю, что метод `flip()` подготавливает буфер для чтения, я имею в виду, что он подготавливает буфер для канальной операции `write()`! Обратное утверждение верно для буферного метода `compact()`.

Кодирование и декодирование текста с помощью Charsets

Объект `java.nio.charset.Charset` представляет собой набор символов плюс кодировку для этого набора. Класс `Charset` и связанные с ним классы `CharsetEncoder` и `CharsetDecoder` определяют методы для кодирования строк символов в последовательности байтов и декодирования последовательностей байтов в строки символов. Так как эти классы являются частью нового API ввода/вывода, они используют классы `ByteBuffer` и `CharBuffer`:

```
// Самый простой случай. Применяем одну из операций Charset для конвертации.
Charset charset = Charset.forName("ISO-8859-1"); // Получить набор символов Latin-1
```

```
CharBuffer cb = CharBuffer.wrap("Hello World");  // Символы для кодирования
// Кодировать символы и сохранить байты в ByteBuffer
ByteBuffer bb = charset.encode(cb);
// Декодировать символы в CharBuffer и вывести их
System.out.println(charset.decode(bb));
```

В этом примере используется набор символов ISO-8859-1 (также известный как «Latin-1»). Этот 8-битный набор символов удобен для большинства восточно-европейских языков, включая английский. Программисты, работающие только с английским, могут задействовать 7-битный набор символов «US-ASCII». Класс `Charset` сам по себе не выполняет кодирование и декодирование, а ранее упомянутые операции просто создают классы `CharsetEncoder` и `CharsetDecoder`. Если вы планируете выполнять кодирование или декодирование несколько раз, то удобнее создать эти объекты самому:

```
Charset charset = Charset.forName("US-ASCII");   // Получить набор символов
CharsetEncoder encoder = charset.newEncoder();   // Создать из него кодировщик
CharBuffer cb = CharBuffer.wrap("Hello World!"); // Получить CharBuffer
WritableByteChannel destination;                 // Инициализирован в другом месте
destination.write(encoder.encode(cb));           // Кодировать символы и записать
```

Ранее упомянутый метод `CharsetEncoder.encode()` должен выделять новый буфер `ByteBuffer` каждый раз, когда он вызывается. Максимальную эффективность предоставляют низкоуровневые методы, которые можно вызывать для кодирования и декодирования данных в существующий буфер:

```
ReadableByteChannel source;                        // Инициализирован в другом месте
Charset charset = Charset.forName("ISO- 8859-1"); // Получить набор символов
CharsetDecoder decoder = charset.newDecoder();     // Создать из него декодировщик
ByteBuffer bb = ByteBuffer.allocateDirect(2048);  // Буфер для хранения байтов
CharBuffer cb = CharBuffer.allocate(2048);         // Буфер для хранения символов

while(source.read(bb) != -1) {              // Читать байты из канала, пока не достигнут конец
  bb.flip();         // Перевернуть буфер, чтобы приготовить его к декодированию
  decoder.decode(bb, cb, true);           // Декодировать байты в символы
  cb.flip();         // Перевернуть символьный буфер, чтобы подготовить его к выводу символов
  System.out.print(cb);                  // Вывести символы
  cb.clear();        // Очистить символьный буфер, чтобы приготовить его к декодированию
  bb.clear();        // Подготовить буфер байтов для следующей операции чтения из канала
}

source.close();      // Закончили работу с каналом, поэтому закроем его
System.out.flush();  // Убедимся, что все операции вывода завершены
```

Предыдущий код опирается на то, что кодировка «ISO-8895-1» является 8-битной, а соответствие символов и байтов – один к одному. В более сложных наборах символов, таких как «UTF-8» или «EUC-JP», используемых в японском языке, для некоторых или всех символов понадобится более одного байта. В таких случаях нет гарантии того, что все байты в буфере будут декодированы (конец буфера может содержать часть символа). Кроме того, поскольку один символ может быть закодирован несколькими байтами, то бывает трудно узнать, сколько байтов потребуется для кодирования строки. Следующий код показывает цикл, который можно применять для декодирования байтов более универсальным способом:

```
ReadableByteChannel source;                      // Инициализирован в другом месте
Charset charset = Charset.forName("UTF- 8");     // Кодировка Unicode
```

```
CharsetDecoder decoder = charset.newDecoder();   // Создать из нее декодировщик
ByteBuffer bb = ByteBuffer.allocateDirect(2048); // Буфер для хранения байтов
CharBuffer cb = CharBuffer.allocate(2048);       // Буфер для хранения символов

// Предписать декодировщику игнорировать ошибки, которые могут быть результатом декодирования
// неполных символов
decoder.onMalformedInput(CodingErrorAction.IGNORE);
decoder.onUnmappableCharacter(CodingErrorAction.IGNORE);

decoder.reset();                         // Установить декодировщик в исходное состояние
while(source.read(bb) != -1) {           // Читать байты из канала, пока не достигнут EOF
  bb.flip();          // Развернуть буфер, чтобы приготовить его к декодированию
  decoder.decode(bb, cb, false); // Декодировать байты в символы
  cb.flip();          // Развернуть символьный буфер, чтобы подготовить его к выводу символов
  System.out.print(cb);          // Вывести символы
  cb.clear();                    // Очистить символьный буфер
  bb.compact();                  // Отбросить декодированные символы
}
source.close();                  // Закончили работу с каналом, поэтому закроем его

// В этот момент буфер может все еще содержать байты для декодирования
bb.flip();                       // Подготовиться к декодированию
decoder.decode(bb, cb, true);    // Передать true, чтобы показать, что это последний вызов
decoder.flush(cb);               // Вывести оставшиеся символы
cb.flip();                       // Развернуть символьный буфер
System.out.print(cb);            // Вывести оставшиеся символы
```

Работа с файлами

Класс `FileChannel` **— это конкретная реализация класса** `Channel`**, которая выполняет файловые операции ввода/вывода и реализует интерфейсы** `ReadableByteChannel` **и** `WritableByteChannel`**. Однако его метод** `read()` **работает, если файл открыт только для чтения, а метод** `write()` **— если файл открыт для записи. Получить объект** `FileChannel` **можно с помощью пакета** `java.io`**. Создайте классы** `FileInputStream`**,** `FileOutputStream` **или** `RandomAccessFile`**, а затем вызовите метод** `getChannel()` **этого объекта (новшество в Java 1.4). Например, вы можете использовать два объекта** `FileChannel` **для копирования файла:**

```
String filename = "test";      // Имя файла для копирования
// Создать потоки для чтения исходного файла и записи копии
FileInputStream fin = new FileInputStream(filename);
FileOutputStream fout = new FileOutputStream(filename + ".copy");
// Использовать потоки для создания соответствующих объектов каналов
FileChannel in = fin.getChannel();
FileChannel out = fout.getChannel();
// Выделить низкоуровневый буфер размером 8 Кбайт для копирования
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
while(in.read(buffer) != -1 || buffer.position() > 0) {
  buffer.flip();     // Подготовиться к чтению из буфера и записи в файл
  out.write(buffer); // Записать часть или все содержимое буфера
  buffer.compact();  // Отбросить все записанные байты и приготовиться к чтению оставшихся
                     // байтов и записи в буфер.
in.close();          // Всегда закрываем каналы и потоки по окончании работы с ними
out.close();
fin.close();         // Обратите внимание, что закрытие FileChannel
fout.close();        // автоматически не закрывает соответствующий поток.
```

Класс FileChannel имеет специальные методы transferTo() и transferFrom(), которые позволяют легче (и на многих операционных системах эффективнее) передавать байты из канала FileChannel в другой канал или из другого канала в канал FileChannel. Эти методы позволяют нам упростить предыдущий код для копирования файла:

```
FileChannel in, out;              // Полагаем, что они инициализированы в предыдущем примере
long numbytes = in.size();        // Количество байтов в исходном файле
in.transferTo(0, numbytes, out); // Передать байты в выходной файл
```

В вышеприведенном фрагменте вместо метода transferTo можно использовать transferFrom. Следующий вариант ничуть не хуже (обратите внимание, что порядок записи аргументов у этих двух методов отличается):

```
long numbytes = in.size();
out.transferFrom(in, 0, numbytes);
```

Класс FileChannel также имеет дополнительные возможности, которых нет в других классах. Одна из наиболее важных – возможность «отобразить» файл или часть файла в памяти, то есть получить экземпляр класса MappedByteBuffer (подкласс класса ByteBuffer), который представляет собой содержимое файла и позволяет читать (а при необходимости записывать) содержимое файла, просто читая байты из буфера и записывая их в буфер. Отображение файла в памяти – дорогая операция, поэтому такая техника обычно эффективна только при работе с большими файлами, к которым необходим постоянный доступ. Отображение файла предоставляет еще один способ выполнять операции копирования:

```
long filesize = in.size();
ByteBuffer bb = in.map(FileChannel.MapMode.READ_ONLY, 0, filesize);
while(bb.hasRemaining()) out.write(bb);
```

Интерфейсы каналов, определенные в пакете java.nio.channels, включают в себя ByteChannel, но не CharChannel. API каналов – низкоуровневый. Он предоставляет только методы для чтения байтов. Все предыдущие примеры интерпретировали файлы как двоичные. Классы CharsetEncoder и CharsetDecoder, представленные ранее, можно применять для конвертации между байтами и символами, но при работе с текстовыми файлами классы Reader и Writer пакета java.io обычно легче использовать, чем CharBuffer. К счастью, класс Channels определяет удобные методы, которые помогают протянуть мост между новым и старым API. Здесь приведен код, который помогает «обернуть» объектами Reader и Writer каналы ввода/вывода, прочитать строки текста к кодировке «Latin-1» и записать их в выходной канал, изменив кодировку на «UTF-8»:

```
ReadableByteChannel in;  // Полагаем, что инициализированы
WritableByteChannel out; // в другом месте
// Создать объекты Reader и Writer, используя FileChannel и имя набора символов
BufferedReader reader = new BufferedReader(Channels.newReader(in, "ISO-8859-1"));
PrintWriter writer = new PrintWriter(Channels.newWriter(out, "UTF-8"));
String line;
while((line = reader.readLine()) != null) writer.println(line);
reader.close();
writer.close();
```

В отличие от классов FileInputStream и FileOutputStream, класс FileChannel позволяет организовать произвольный доступ к содержимому файла. Метод position(), вызываемый без аргументов, возвращает *указатель файла* (позицию следующего символа, который будет прочитан), а метод, вызываемый с одним аргументом, позволяет уста-

новить указатель в любое значение. Это дает вам возможность пропустить часть файла так же, как это делает java.io.RandomAccessFile. Вот пример:

```
// Полагаем, что у вас есть файл, содержащий записи, а последние 1024 байта являются индексом,
// который позволяет определить положение этих записей. Здесь приведен код, который читает
// индекс, находит позицию первой записи и считывает ее.
FileChannel in = new FileInputStream("test.data").getChannel(); // Канал
ByteBuffer index = ByteBuffer.allocate(1024);                   // Буфер для хранения индекса
long size = in.size();                                          // Размер файла
in.position(size - 1024);                                       // Позиция начала индекса
in.read(index);                                                 // Прочитать индекс
int record0Position = index.getInt(0);          // Получить первую позицию индекса
in.position(record0Position);                   // Позиционировать файл в эту точку
ByteBuffer record0 = ByteBuffer.allocate(128); // Получить буфер для хранения данных
in.read(record0);                               // И наконец, прочитать запись
```

И наконец, последняя особенность FileChannel, которая здесь рассматривается подробно – это возможность заблокировать файл или часть файла, исключая одновременный доступ (эксклюзивная блокировка) или одновременную запись (блокировка с возможностью совместного доступа). Обратите внимание, что одни операционные системы строго предписывают использовать все блокировки, а другие лишь рекомендуют программам сотрудничать и пытаться заблокировать разделяемую часть файла перед чтением или записью. Предположим, что в предыдущем примере случайного доступа к файлу нужно, чтобы никакая другая программа не изменила запись, пока мы ее читаем. Мы можем заблокировать часть файла, предоставив возможность совместного доступа:

```
FileLock lock = in.lock(record0Position, // Начало блока
                        region 128,      // Длина блока
                        true);           // Блокировка с возможностью доступа: запрещает
                                         // изменение, но разрешает чтение.
in.position(record0Position);            // Перейти к началу индекса
in.read(record0);                        // Прочитать индекс
lock.release();                          // Мы закончили, поэтому снимаем блокировку
```

Работа с сетью на стороне клиента

Наряду с возможностью доступа к файлам новый API ввода/вывода предоставляет возможность работы с сетью. Для общения по сети можно применять класс SocketChannel. Создайте SocketChannel, вызвав статический метод open(), а затем читайте и записывайте байты так, как в случае с любым другим каналом. Следующий код использует SocketChannel для отправки HTTP-запроса веб-серверу и сохранения ответа в файле (в том числе все HTTP-заголовки). Класс java.net.InetSocketAddress (подкласс java.net.SocketAddress) используется для того, чтобы сообщить SocketChannel место, к которому следует присоединяться. Эти классы также появились в Java 1.4 и были представлены как часть нового API ввода/вывода.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

// Создать SocketChannel, соединенный с сервером www.oreilly.com.
SocketChannel socket =
    SocketChannel.open(new InetSocketAddress("www.oreilly.com",80));
```

```
// Набор символов для кодирования HTTP-запроса
Charset charset = Charset.forName("ISO-8859-1");
// Отправить HTTP-запрос серверу. Сначала получить строку,
// обернуть ее в CharBuffer, перекодировать в ByteBuffer, а затем записать в сокет.
socket.write(charset.encode(CharBuffer.wrap("GET / HTTP/1.0\r\n\r\n")));
// Создать FileChannel для сохранения ответа сервера
FileOutputStream out = new FileOutputStream("oreilly.html");
FileChannel file = out.getChannel();
// Получить буфер для хранения байтов при передаче из сокета в файл
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
// В цикле перебрать все байты, прочитав их из сокета и записав в файл
while(socket.read(buffer) != -1 || buffer.position() > 0) {          // Мы закончили?
  buffer.flip();          // Подготовиться к чтению из буфера и записи в файл
  file.write(buffer);     // Записать часть байтов или все байты в файл
  buffer.compact();       // Отбросить записанные байты
}
socket.close();           // Закрыть канал сокета
file.close();             // Закрыть канал файла
out.close();              // Закрыть файл
```

Другой способ создать `SocketChannel` — использовать версию метода `open()` без аргументов, которая создает ни с чем не соединенный канал. Это позволяет вызвать метод `socket()` для получения соответствующего сокета, сконфигурировать его и присоединиться к нужному серверу. Например:

```
SocketChannel sc = SocketChannel.open();       // Открыть неприсоединенный сокет
channel Socket s = sc.socket();                // Получить соответствующий java.net.Socket
s.setSOTimeout(3000);                          // Установить тайм-аут в 3 секунды
// Теперь соединиться с выбранным сервером через соответствующий порт
sc.connect(new InetSocketAddress("www.davidflanagan.com", 80));

ByteBuffer buffer = ByteBuffer.allocate(8192); // Создать буфер
try { sc.read(buffer); }                       // Попытаться читать
catch(SocketTimeoutException e) {              // Перехватить тайм-аут
  System.out.println("Удаленный компьютер не отвечает.");
  sc.close();
}
```

В дополнение к классу `SocketChannel` пакет `java.nio.channels` определяет класс `DatagramChannel` для сетевых операций с дейтаграммами вместо сокетов. Этот класс здесь не продемонстрирован, но вы можете прочитать о нем в главе 14.

Одна из наиболее мощных особенностей нового API ввода/вывода состоит в том, что каналы `SocketChannel` и `DatagramChannel` могут применяться в неблокирующем режиме.

Работа с сетью на стороне сервера

Пакет `java.net` определяет класс `Socket` для общения клиента с сервером и класс `ServerSocket`, используемый сервером для прослушивания портов и установления соединения с клиентами. Пакет `java.nio.channels` аналогичен: он определяет класс `SocketChannel` для передачи данных и класс `ServerSocketChannel` для установления соединений. `ServerSocketChannel` — необычный канал, потому что он не реализует интерфейсы `ReadableByteChannel` и `WritableByteChannel`. Вместо методов `read()` и `write()` у него есть метод `accept()` для установления клиентского соединения и получения объекта `SocketChannel`, через который он общается с клиентом. Вот код для простого, однопотокового сервера, который прослушивает порт 8000 и возвращает текущее время любому клиенту, который к нему присоединяется:

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class DateServer {
  public static void main(String[] args) throws java.io.IOException {
    // Получить объект CharsetEncoder для кодирования текста,отправляемого клиенту
    CharsetEncoder encoder = Charset.forName("US- ASCII").newEncoder();

    // Создать новый ServerSocketChannel и связать его с портом 8000. Обратите внимание на то,
    // чта операция должна быть выполнена с использованием базового ServerSocket
    ServerSocketChannel server = ServerSocketChannel.open();
    server.socket().bind(new java.net.InetSocketAddress(8000));

    for(;;) {  // Сервер запущен «навечно»
      // Ожидать клиентское соединение
      SocketChannel client = server.accept();
      // Получить текущую дату и время в виде строки
      String response = new java.util.Date().toString() + "\r\n";
      // Обернуть, закодировать и отправить строку клиенту
      client.write(encoder.encode(CharBuffer.wrap(response)));
      // Отсоединиться от клиента
      client.close();
    }
  }
}
```

Неблокирующий ввод/вывод

Вышеупомянутый класс `DateServer` является простым сервером. Так как он не поддерживает соединение ни с одним клиентом, ему нет необходимости общаться более чем с одним клиентом одновременно. В любой момент времени задействован только один сокет `SocketChannel`. Более реалистичный сервер должен уметь общаться с несколькими клиентами одновременно. API `java.io` и `java.net` позволяют выполнять только блокирующие операции ввода/вывода, поэтому сервер, применяющий эти API, должен использовать отдельный поток для каждого клиента. В случае больших серверов с большим количеством клиентов этот принцип не совсем подходит. Для решения этой проблемы новый API ввода/вывода позволяет большинству каналов (но не каналу `FileChannel`) работать в неблокирующем режиме и обрабатывать все ожидающие соединения в одном потоке. Это реализуется с помощью объекта `Selector`, который следит за набором зарегистрированных каналов и может блокироваться, пока один или несколько каналов не готовы для ввода/вывода. Далее представлен более длинный пример, иллюстрирующий данную тему. Это работающий серверный класс, который управляет классом `ServerSocketChannel` и любым количеством клиентских соединений `SocketChannel` через объект `Selector`.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*; // Для Set и Iterator
public class NonBlockingServer {
  public static void main(String[] args) throws IOException {
```

```
// Получить кодировщики и декодировщики символов, которые понадобятся далее
Charset charset = Charset.forName("ISO-8859-1");
CharsetEncoder encoder = charset.newEncoder();
CharsetDecoder decoder = charset.newDecoder();

// Выделить буфер для связи с клиентами
ByteBuffer buffer = ByteBuffer.allocate(512);

// Все каналы в этом примере будут работать в неблокирующем режиме. Итак, создадим объект
// Selector, который будет заблокирован при слежении за всеми каналами и разблокируется,
// когда один или несколько каналов будут готовы к вводу/выводу.
Selector selector = Selector.open();

// Создать новый объект ServerSocketChannel и связать его с портом 8000. Обратите внимание
// на то, что операция должна быть выполнена с помощью базового ServerSocket
ServerSocketChannel server = ServerSocketChannel.open();
server.socket().bind(new java.net.InetSocketAddress(8000));
// Перевести ServerSocketChannel в неблокирующий режим
server.configureBlocking(false);
// Теперь зарегистрируем его с помощью Selector (обратите внимание, что метод register()
// вызывается для канала, а не для селектора). Объект SelectionKey представляет собой
// регистрацию этого канала и объекта Selector.
SelectionKey serverkey = server.register(selector, SelectionKey.OP_ACCEPT);

for(;;) {            // Главный цикл. Сервер всегда запущен.
  // Этот вызов блокирует выполнение, пока не активен ни один зарегистрированный канал.
  // Это ключевой метод при неблокирующем вводе/выводе.
  selector.select();

  // Получить объект java.util.Set, содержащий объект SelectionKey, для всех каналов,
  // которые готовы к вводу/выводу.
  Set keys = selector.selectedKeys();
  // Используем java.util.Iterator для перебора всех выбранных ключей
  for(Iterator i = keys.iterator(); i.hasNext(); ) {
    // Получить из набора следующий объект SelectionKey и удалить его. Он должен быть
    // удален явно, иначе он будет возвращен еще раз при следующем вызове метода select().
    SelectionKey key = (SelectionKey) i.next();
    i.remove();

    // Проверить, является ли SelectionKey объектом, полученным при регистрации
    // ServerSocketChannel.
    if (key == serverkey) {
      // Активность в ServerSocketChannel означает, что клиент пытается
      // соединиться с сервером.
      if (key.isAcceptable()) {
        // Принять клиентское соединение и получить объект SocketChannel
        // для связи с клиентом.
        SocketChannel client = server.accept();
        // Перевести клиентский канал в неблокирующий режим
        client.configureBlocking(false);
        // Теперь зарегистрировать его с объектом Selector, сообщая
        // ему, что вам хотелось бы знать, когда из канала можно будет читать данные.
        SelectionKey clientkey =
             client.register(selector, SelectionKey.OP_READ);
        // Связать с ключом состояние клиента. Это состояние будет учитываться
        // при общении с клиентом.
        clientkey.attach(new Integer(0));
      }
    }
```

```
        else {
          // Если ключ из набора не является объектом ServerSocketChannel, то он
          // представляет собой клиентское соединение. Получить канал.
          SocketChannel client = (SocketChannel) key.channel();

          // Если вы здесь, то в канале есть данные для чтения, но проверим это еще раз.
          if (!key.isReadable()) continue;

          // Теперь прочитаем байты от клиента, предполагая, что все байты помещаются в
          // одну операцию чтения.
          int bytesread = client.read(buffer);

          // Если метод read() возвращает -1, это означает конец потока, что, в свою очередь,
          // свидетельствует о том, что клиент был отсоединен и его нужно вычеркнуть
          // из списка зарегистрированных клиентов и закрыть канал.
          if (bytesread == -1) {
            key.cancel();
            client.close();
            continue;
          }

          // В противном случае декодировать байты в строку запроса
          buffer.flip();
          String request = decoder.decode(buffer).toString();
          buffer.clear();
          // Теперь ответить клиенту, основываясь на строке запроса
          if (request.trim().equals("quit")) {
            // Если это «quit»-запрос, отправить последнее
            // сообщение. Закрыть канал и разрегистрировать SelectionKey
            client.write(encoder.encode(CharBuffer. wrap("Bye.")));
            key.cancel();
            client.close();
          }
          else {
            // В противном случае отправить строку ответа, включающую номер этого запроса
            // и строку запроса в верхнем регистре. Обратите внимание на то, что вы  следите
            // за номером, связывая объект Integer с объектом SelectionKey и увеличивая
            // его каждый раз.

            // Получить номер от SelectionKey
            int num = ((Integer)key.attachment()).intValue();
            // Для строки ответа

            String response = num + ": " + request.toUpperCase();
            // Обернуть, закодировать и записать строку ответа
            client.write(encoder.encode(CharBuffer. wrap(response)));
             // Связать инкрементированный номер с ключом
            key.attach(new Integer(num+1));
          }
        }
      }
    }
  }
```

Неблокирующий ввод/вывод наиболее полезен при создании серверов. Кроме того, он полезен в клиентах, имеющих несколько ожидающих соединений в одно и то же время. Например, представьте себе броузер, одновременно загружающий текст и

изображения страницы. Другой интересный пример использования неблокирующего ввода/вывода – выполнение неблокирующих операций с сокетами. Идея состоит в следующем. Вы можете попросить объект SocketChannel установить соединение с удаленным сервером, а затем выполняете другие операции (например, создание интерфейса), пока операционная система устанавливает соединение по сети. Позднее вы вызываете метод select() для блокировки выполнения, пока соединение не будет установлено, если оно не установлено до сих пор. Код для неблокирующего соединения выглядит так:

```
// Создать новый неприсоединенный объект SocketChannel. Перевести его в неблокирующий режим,
// зарегистрировать новый Selector, а затем предписать ему выполнить соединение. Этот вызов
// вернет управление в основной поток вместо того, чтобы дожидаться установления соединения.
Selector selector = Selector.open();
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_CONNECT);
channel.connect(new InetSocketAddress(hostname, port));

// Теперь сделаем что-нибудь другое, пока соединение устанавливается.
// Например, создадим пользовательский интерфейс.

// Теперь блокируем управление до тех пор, пока SocketChannel не соединится.
// Так как в этом селекторе вы зарегистрировали только один канал, вам не нужно
// проверять состояние ключа; вы всегда знаете, какой канал готов.
while(selector.select() == 0) /* пустой цикл */;

// Этот вызов необходим для завершения неблокирующих соединений
channel.finishConnect();

// И наконец, закроем селектор, который удалит канал
selector.close();
```

XML

JAXP, Java API для обработки XML, был изначально определен как необязательное расширение для Java-платформы. Он доступен как самостоятельных продукт, однако в Java 1.4 средства JAXP вошли в состав ядра. API состоит из следующих пакетов (и подпакетов):

`javax.xml.parsers`

Этот пакет предоставляет интерфейсы высокого уровня для реализации программ синтаксического разбора для SAX и DOM; это «интерфейсный слой», который позволяет конечным пользователям или системным администраторам выбирать или даже заменять реализацию парсера по умолчанию.

`javax.xml.transform`

Этот пакет и его подпакеты определяют Java API для преобразования содержимого XML-документов и его представления с использованием стандарта XSLT. Кроме того, этот пакет предоставляет встраиваемый слой, который позволяет «встраивать» новые библиотеки обработки XSLT и применять их вместо реализаций по умолчанию.

`org.xml.sax`

Этот пакет и два его подпакета определяют фактические стандарты SAX API (Simple API для XML). SAX – событийно-ориентированный API для разбора XML: SAX вызывает методы объекта `ContentHandler` и вовлекает другие объекты-обработчики при разборе XML-документов. Структура и содержимое документов полностью описываются через вызовы методов. Это потоковый API, который не создает постоянного представления документа. Этим занимается реализация `ContentHandler`, которая сохраняет любое состояние или выполняет любые необходимые действия. Данный пакет включает в себя классы для SAX 2 API и устаревшие классы SAX 1.

`org.w3c.dom`

Этот пакет определяет интерфейсы, которые представляют XML-документ в виде дерева. Объектная модель документа (DOM) – это рекомендации (по сути, стандарт) консорциума W3C. DOM-обработчик читает XML-документ и конвертирует его в дерево узлов, представляющее полное содержимое документа. После создания документа в виде дерева программа может работать с ним так, как ей необходимо.

Примеры каждого из этих пакетов приведены в следующих подразделах.

Разбор XML с помощью SAX

Первым шагом при разборе документов с помощью SAX является получение парсера (SAX parser). Если у вас есть собственная реализация такого парсера, то вы можете легко создать класс. Обычно для создания классов удобнее использовать пакет `javax.xml.parsers` вне зависимости от того, какой парсер реализован в Java. Код выглядит так:

```
import javax.xml.parsers.*;

// Получить объект-фабрику для создания SAX-парсера
SAXParserFactory parserFactory = SAXParserFactory.newInstance();

// Сконфигурировать объект-фабрику, задав атрибуты парсера, который она создает
parserFactory.setValidating(true);
parserFactory.setNamespaceAware(true);

// Теперь создадим объект SAXParser
SAXParser parser = parserFactory.newSAXParser(); // Может генерировать исключение
```

Класс `SAXParser` – это простая обертка вокруг класса `org.xml.sax.XMLReader`. После его получения вы можете разбирать документ, вызывая один из разнообразных методов объекта `parse()`. Некоторые из этих методов применяют класс `HandlerBase` из устаревшего SAX 1, а другие методы используют класс `org.xml.sax.helpers.DefaultHandler` из текущего SAX 2. Класс `DefaultHandler` предоставляет пустую реализацию всех методов интерфейсов `ContentHandler`, `ErrorHandler`, `DTDHandler` и `EntityResolver`. Это методы, которые может вызвать SAX-парсер при разборе XML-документа. Создав подкласс `DefaultHandler` и определив необходимые методы, вы можете выполнять любые действия в ответ на вызовы методов, сгенерированные парсером. Следующий код иллюстрирует метод, использующий SAX для разбора XML-документа и определения количества XML-элементов, которые встречаются в документе, и количества символов простого текста (возможно, без «игнорируемых пробелов»), содержащегося в этих элементах:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXCount {
  public static void main(String[] args)
      throws SAXException,IOException, ParserConfigurationException
{
    // Создать парсер-фабрику, а затем сам парсер
    SAXParserFactory parserFactory = SAXParserFactory.newInstance();
    SAXParser parser = parserFactory.newSAXParser();
    // Это имя файла для разбора
    String filename = args[0];
    // Создать подкласс DefaultHandler для подсчета
    CountHandler handler = new CountHandler();
    // Запустить парсер. Он читает файл и вызывает методы-обработчики.
    parser.parse(new File(filename), handler);
    // После завершения работы выдать результаты
    System.out.println(filename + " содержит " + handler.numElements +
                                " элементов и " + handler.numChars + " других символов ");
  }

  // Этот внутренний класс расширяет класс DefaultHandler для подсчета элементов и текста
  // в XML-файле и сохраняет результаты в открытых полях. Существует множество
  // других методов DefaultHandler, которые можно заменить,но вам нужен только этот.
  public static class CountHandler extends DefaultHandler {
    public int numElements = 0, numChars = 0; // Здесь хранятся счетчики
    // Этот метод вызывается, когда парсер находит открытый тег любого XML-элемента.
    // Игнорируем аргументы, но считаем элементы.
    public void startElement(String uri, String localname, String qname,
                             Attributes attributes) {
        numElements++;
    }

    // Этот метод вызывается для любого элемента, содержащего обычный текст
    // Просто посчитаем количество символов в тексте

    public void characters(char[] text, int start, int length) {
        numChars += length;
    }
  }
}
```

Разбор XML с помощью DOM

DOM API довольно сильно отличается от SAX API. Хотя SAX подходит для сканирования XML-документа, он не достаточно удобен для программ, которым нужно менять документ. Вместо конвертации XML-документа в серию вызовов методов парсер DOM конвертирует документ в объект `org.w3c.dom.Document`, который является деревом объектов `org.w3c.dom.Node`. Полная конвертация XML-документа в дерево позволяет получить доступ к любой части документа, но может потребовать большого объема памяти.

В DOM API каждый узел в дереве документа реализует интерфейс `Node` и специфичный подынтерфейс (subinterface). В документах DOM наиболее часто применяются

узлы Element и Text. Следующий код использует JAXP для получения парсера DOM, который на языке JAXP называется DocumentBuilder. Затем он разбирает XML-файл и строит из него дерево документа. Далее программа исследует дерево Document, находит элементы <sect1> и выводит содержимое их тегов <title>.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class GetSectionTitles {
  public static void main(String[] args)
      throws IOException, ParserConfigurationException,
             org.xml.sax.SAXException
  {
    // Создать фабрику объектов для создания парсера DOM и сконфигурировать ее
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setIgnoringComments(true);    // Мы хотим игнорировать комментарии
    factory.setCoalescing(true);          // Конвертируем узлы CDATA в узлы Text
    factory.setNamespaceAware(false);     // Нет пространства имен: по умолчанию
    factory.setValidating(false);         // Не проверять DTD: так же по умолчанию

    // Теперь задействуем фабрику для создания парсера DOM,
    // также известного как DocumentBuilder
    DocumentBuilder parser = factory.newDocumentBuilder();

    // Разобрать файл и построить дерево Document для представления его содержимого
    Document document = parser.parse(new File(args[0]));

    // Найти в документе все элементы <sect1>
    NodeList sections = document.getElementsByTagName("sect1");
    // Перебрать все элементы <sect1> по одному
    int numSections = sections.getLength();
    for(int i = 0; i < numSections; i++) {
      Element section = (Element)sections.item(i);    // Элемент <sect1>
      // Первым дочерним элементом каждого объекта Element (<sect1>) должен быть элемент
      // <title>, но могут быть и пробелы
      // Сначала узел Text, переберем всех детей, пока не найдем первый дочерний элемент.
      Node title = section.getFirstChild();
      while(title != null && title.getNodeType()
            != Node.ELEMENT_NODE) title = title.getNextSibling();
      // Выведем текст, содержащийся в дочернем узле Text этого элемента
      if (title != null)
        System.out.println(title.getFirstChild().getNodeValue());
    }
  }
}
```

Преобразование XML-документов

Пакет javax.xml.transform определяет класс TransformerFactory для создания объекта Transformer. Объект Transformer может преобразовать документ из исходного представления Source в новое представление Result и применить XSLT-трансформацию к содержимому документа в процессе обработки. Три подпакета определяют конкретные реализации интерфейсов Source и Result, которые позволяют преобразовывать документ в то или иное представление (всего их три):

`javax.xml.transform.stream`

Представляет документ как поток XML-текста.

`javax.xml.transform.dom`

Представляет документ как DOM-дерево `Document`.

`javax.xml.transform.sax`

Представляет документ как последовательность вызовов методов SAX.

Следующий код показывает один из вариантов использования этих пакетов для преобразования представления документа из DOM-дерева `Document` в поток XML-текста. Интересная особенность этого кода заключается в том, что он не создает дерево `Document`, разбирая файл; вместо этого он выстраивает его самостоятельно.

```
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOMToStream {
  public static void main(String[] args)
    throws ParserConfigurationException,
           TransformerConfigurationException,
           TransformerException
  {
    // Создает DocumentBuilderFactory и DocumentBuilder
    DocumentBuilderFactory dbf =
      DocumentBuilderFactory.newInstance();
    DocumentBuilder db = dbf.newDocumentBuilder();
    // Вместо разбора XML-документа создадим пустой документ,
    // который можно наращивать самостоятельно.
    Document document = db.newDocument();

    // Теперь построим дерево документа, используя методы DOM
    Element book = document.createElement("book");          // Создадим новый элемент
    book.setAttribute("id", "javanut4");                    // Зададим его атрибут
    document.appendChild(book);                             // Добавим в документ
    for(int i = 1; i <= 3; i++) {
      // Добавим еще несколько элементов
      Element chapter = document.createElement("chapter");
      Element title = document.createElement("title");
      title.appendChild(document.createTextNode("Chapter " + i));
      chapter.appendChild(title);
      chapter.appendChild(document.createElement("para"));
      book.appendChild(chapter);
    }

    // Теперь создадим TransformerFactory и используем его для создания объекта Transformer
    // при преобразовании нашего DOM-документа в поток XML-текста.
    // Вызов newTransformer() без аргументов означает, что мы не задействуем XSLT
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // Создать объекты Source и Result для преобразования
    DOMSource source = new DOMSource(document);             // DOM-документа
    StreamResult result = new StreamResult(System.out);     // в XML-текст
```

```
    // Наконец, выполним преобразование
    transformer.transform(source, result);
  }
}
```

Наиболее интересно применение пакета `javax.xml.transform` при работе со стилями XSLT. XSLT – это сложная, но мощная грамматика XML, описывающая преобразование XML-документа в другую форму (например, XML, HTML или обычный текст). Обзор стилей XSLT выходит за рамки этой книги, но следующий код, содержащий только шесть ключевых строк, показывает, как можно применить стили, которые сами являются XML-документом, к другому XML-документу и записать результирующий документ в поток:

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Transform {
  public static void main(String[] args)
    throws TransformerConfigurationException,
           TransformerException
  {
    // Получить объекты Source и Result для ввода, стиля и вывода
    StreamSource input = new StreamSource(new File(args[0]));
    StreamSource stylesheet = new StreamSource(new File(args[1]));
    StreamResult output = new StreamResult(new File(args[2]));

    // Создать объект и выполнить преобразование
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer(stylesheet);
    transformer.transform(input, output);
  }
}
```

Процессы

Ранее в этой главе мы видели, как легко создать несколько потоков, которые будут выполняться внутри одного интерпретатора Java. Кроме того, в Java есть класс `java.lang.Process`, представляющий собой программу, работающую вне интерпретатора. Java-программа может общаться с внешним процессом с помощью потоков так же, как может общаться по сети с сервером, работающим на другом компьютере. Использование класса `Process` всегда зависит от платформы, а приложения очень редко поддаются переносу, но иногда это полезно:

```
// Максимально увеличим возможность переноса, находя в
// конфигурационном файле имя команды для выполнения.
java.util.Properties config;
String cmd = config.getProperty("sysloadcmd");
if (cmd != null) {
  // Выполнить команду; процесс p представляет собой выполняющуюся команду
  Process p = Runtime.getRuntime().exec(cmd);           // Выполнить команду
  InputStream pin = p.getInputStream();                 // Прочитать выходные байты
  InputStreamReader cin = new InputStreamReader(pin);   // Преобразовать их в символы
```

```
    BufferedReader in = new BufferedReader(cin);          // Прочитать строку символов
    String load = in.readLine();                          // Получить вывод команды
    in.close();                                           // Закрыть поток
  }
```

Безопасность

Пакет java.security определяет всего несколько классов, связанных с архитектурой контроля доступа Java, которая более подробно обсуждается в главе 5. Эти классы позволяют Java-программам выполнять ненадежный код в ограниченном окружении, где этот код не может нанести вреда. Несмотря на важность этих классов, они применяются редко. Более интересны классы, применяемые для аутентификации; примеры их использования представлены ниже.

Профили сообщений

Профиль сообщения – это значение, которое также известно как криптографическая контрольная сумма, или защищенный хеш. Оно вычисляется на основе последовательности байтов. Длина профиля обычно гораздо меньше длины данных, для которых он вычисляется, но любое изменение в исходных данных, каким бы маленьким оно ни было, повлечет за собой изменение профиля. При передаче данных (сообщения) вы можете передать вместе с ним профиль сообщения. Затем получатель сообщения может повторно вычислить профиль сообщения на основе полученных данных и, сравнив два профиля, определить, правильно ли было передано сообщение и нет ли потери данных. Ранее при обсуждении потоков мы уже видели способ вычисления профиля сообщения. Похожую технику можно задействовать при вычислении профиля сообщения для непотоковых двоичных данных:

```
import java.security.*;

// Получить объект для вычисления профиля сообщения с использованием «Алгоритма защищенного
// хеша»; этот метод может генерировать исключение NoSuchAlgorithmException.
MessageDigest md = MessageDigest.getInstance("SHA");

byte[] data, data1, data2, secret;    // Массивы данных, инициализированные в другом месте

// Создать профиль для одного массива данных
byte[] digest = md.digest(data);

// Создать профиль для нескольких цепочек данных
md.reset();                   // Необязательно: автоматически вызывается методом digest()
md.update(data1);             // Обработать первую цепочку данных
md.update(data2);             // Обработать вторую цепочку данных
digest = md.digest();         // Вычислить профиль

// Создать зашифрованный профиль, который можно проверить,если вы знаете секретные байты
md.update(data);              // Данные, которые должны быть переданы вместе с профилем
digest = md.digest(secret);   // Добавить секретные байты и вычислить профиль

// Проверить профиль можно так
byte[] receivedData, receivedDigest;           // Полученные данные и профиль
byte[] verifyDigest = md.digest(receivedData); // Вычислить профиль полученных данных
// Сравнить профили
boolean verified = java.util.Arrays.equals(receivedDigest, verifyDigest);
```

Цифровые подписи

Цифровая подпись объединяет алгоритм профиля сообщения и шифрование с открытым ключом. Отправитель сообщения, Алиса, может вычислить профиль сообщения и зашифровать его секретным ключом. Затем она отсылает сообщение и зашифрованный профиль получателю, Бобу. Боб знает открытый ключ Алисы (он открыт для всех), поэтому он может использовать его для расшифровки профиля и проверки целостности сообщения. Во время этой проверки Боб также узнает, что профиль был зашифрован секретным ключом Алисы, поскольку ему удалось расшифровать профиль, используя ее открытый ключ. Поскольку секретный ключ Алисы есть только у нее, то сообщение пришло именно от Алисы. Цифровая подпись получила такое название потому, что, подобно подписи на бумаге, она служит для аутентификации источника документа или сообщения. В отличие от подписи ручкой, цифровая подпись очень сложна, ее невозможно фальсифицировать или просто скопировать и вставить в другой документ.

Java позволяет легко создавать цифровые подписи. Для создания цифровой подписи вам нужен объект `java.security.PrivateKey`. Если хранилище ключей существует в вашей системе (см. документацию по *keytool* в главе 8), вы можете получить один из них с помощью следующего кода:

```
// Вот данные, которые нам потребуются
File homedir = new File(System.getProperty("user.home"));
File keyfile = new File(homedir, ".keystore");  // Или прочитать их из конфигурационного файла
String filepass = "KeyStore password"           // Пароль для всего файла
String signer = "david";                        // Прочитать из конфигурационного файла
String password = "Никто не сможет угадать!";   // Лучше попросить ввести пароль
PrivateKey key;                                 // Это ключ, который мы хотим получить

try {
  // Получить объект KeyStore и загрузить в него данные
  KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
  keystore.load(new BufferedInputStream(
    new FileInputStream(keyfile)), filepass.toCharArray());
  // Теперь попросим необходимый ключ
  key = (PrivateKey) keystore.getKey(signer, password.toCharArray());
}
catch (Exception e) { /* Здесь обработаем исключения */ }
```

Получив объект `PrivateKey`, вы создаете цифровую подпись с помощью объекта `java.security.Signature`:

```
PrivateKey key;                          // Инициализирован, как показано выше
byte[] data;                             // Данные, которые нужно подписать
Signature s =                            // Получить объект для создания и проверки подписей
  Signature.getInstance("SHA1withDSA"); // Может генерировать
                                         // исключение NoSuchAlgorithmException
s.initSign(key);      // Инициализируем его; может генерировать исключение InvalidKeyException
s.update(data);       // Данные для подписи; может генерировать исключение SignatureException
/* s.update(data2); */                // Вызовите несколько раз, указав все данные
byte[] signature = s.sign();          // Вычислите подпись
```

Объект `Signature` может проверить цифровую подпись:

```
byte[] data;                          // Подписанные данные; инициализированы в другом месте
byte[] signature;                     // Подпись для проверки; инициализирована в другом месте
String signername;                    // Создатель подписи; инициализирован в другом месте
```

```
KeyStore keystore;     // Место хранения сертификата; инициализируется так, как показано ранее

// Найти сертификат открытого ключа для подписавшей стороны
java.security.cert.Certificate cert = keystore.getCertificate(signername);
PublicKey publickey = cert.getPublicKey();                  // Получить из него открытый ключ
Signature s = Signature.getInstance("SHA1withDSA");         // Или использовать другой алгоритм
s.initVerify(publickey);                                    // Приготовиться к проверке
s.update(data);                                             // Указать подписанные данные
boolean verified = s.verify(signature);                     // Проверить подпись
```

Подписанные объекты

Класс `java.security.SignedObject` **является удобным средством для цифровой подписи объектов. Объект** `SignedObject` **может быть сериализован и передан получателю, который восстановит его и вызовет метод** `verify()` **для проверки подписи:**

```
Serializable o; // Объект, который нужно подписать; должен быть Serializable
PrivateKey k;   // Ключ для подписи; инициализирован в другом месте
Signature s = Signature.getInstance("SHA1withDSA");       // «Движок» подписи
SignedObject so = new SignedObject(o, k, s);              // Создать SignedObject

// Объект SignedObject инкапсулирует объект o; теперь он может
// быть сериализован и передан получателю.

// Вот как получатель может проверить объект SignedObject
SignedObject so;       // Восстановленный объект SignedObject
Object o;              // Исходный объект
PublicKey pk;          // Ключ для проверки
Signature s = Signature.getInstance("SHA1withDSA");       // «Движок» подписи

if (so.verify(pk,s))                                      // Если подпись верна,
  o = so.getObject();                                    // получить инкапсулированный объект.
```

Шифрование

Пакет `java.security` **включает в себя базовые классы для криптографии, но не содержит классов для реального шифрования и дешифрования. Этим занимается пакет** `javax.crypto`**. Данный пакет поддерживает шифрование с использованием симметричных ключей, в котором для шифрования и дешифрования применяется один и тот же ключ, известный отправителю данных и получателю.**

Секретные ключи

Интерфейс `SecretKey` **представляет собой ключ шифрования; первый шаг любой криптографической операции – получение соответствующего объекта** `SecretKey`**. К сожалению, пакет *keytool*, поставляемый с Java SDK, не может генерировать и хранить секретные ключи, поэтому программа должна сама побеспокоиться о выполнении этих задач. Вот код, который показывает различные способы работы с объектом** `SecretKey`**:**

```
import javax.crypto.*;
import javax.crypto.spec.*;

// Сгенерировать ключ шифрования с помощью объекта KeyGenerator
KeyGenerator desGen = KeyGenerator.getInstance("DES");                  // Алгоритм DES
SecretKey desKey = desGen.generateKey();                               // Сгенерировать ключ
```

```
KeyGenerator desEdeGen = KeyGenerator.getInstance("DESede");              // Тройной DES
SecretKey desEdeKey = desEdeGen.generateKey();                            // Сгенерировать ключ

// SecretKey - это непрозрачное представление ключа. Используйте SecretKeyFactory для
// конвертации ключа в прозрачное представление, которое можно сохранять в файле,
// передавать получателю и т. д.
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");
DESKeySpec desSpec = (DESKeySpec)
   desFactory.getKeySpec(desKey, javax.crypto.spec.DESKeySpec.class);
byte[] rawDesKey = desSpec.getKey();
// Сделать то же самое для ключа DESede
SecretKeyFactory desEdeFactory = SecretKeyFactory.getInstance("DESede");
DESedeKeySpec desEdeSpec = (DESedeKeySpec)
  desEdeFactory.getKeySpec(desEdeKey, javax.crypto.spec.DESedeKeySpec.class);
byte[] rawDesEdeKey = desEdeSpec.getKey();

// Конвертировать строку байтов ключа в объект SecretKey
DESedeKeySpec keyspec = new DESedeKeySpec(rawDesEdeKey);
SecretKey k = desEdeFactory.generateSecret(keyspec);

// Для создания ключей DES и DESede существует даже более простой путь
// SecretKeySpec реализует SecretKey; применяйте его для представления этих ключей
byte[] desKeyData = new byte[8];                                  // Прочитать 8 байт данных из файла
byte[] tripleDesKeyData = new byte[24];                           // Прочитать 24 байт данных из файла
SecretKey myDesKey = new SecretKeySpec(desKeyData, "DES");
SecretKey myTripleDesKey = new SecretKeySpec(tripleDesKeyData, "DESede");
```

Шифрование и дешифрование с помощью Cipher

Как только вы получили соответствующий объект `SecretKey`**, центральным классом для шифрования и дешифрования становится класс** `Cipher`**. Используйте его так:**

```
SecretKey key;     // Получить SecretKey, как показано ранее
byte[] plaintext;  // Данные для шифрования; инициализированы в другом месте

// Получить объект для выполнения шифрования или дешифрования
Cipher cipher = Cipher.getInstance("DESede"); // Шифрование с помощью тройного DES
// Инициализировать объект cipher для шифрования
cipher.init(Cipher.ENCRYPT_MODE, key);
// Теперь зашифровать данные
byte[] ciphertext = cipher.doFinal(plaintext);

// Если у нас есть несколько цепочек данных для шифрования, то мы можем это сделать так
cipher.update(message1);
cipher.update(message2);
byte[] ciphertext = cipher.doFinal();

// При расшифровании мы выполняем те же действия, но в обратном порядке
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] decryptedMessage = cipher.doFinal(ciphertext);

// Для расшифрования нескольких цепочек данных
byte[] decrypted1 = cipher.update(ciphertext1);
byte[] decrypted2 = cipher.update(ciphertext2);
byte[] decrypted3 = cipher.doFinal(ciphertext3);
```

Шифрование и дешифрование потоков

Класс `Cipher` **можно применять с потоком** `CipherInputStream` **или** `CipherOutputStream` **для шифрования или дешифрования потоковых данных при чтении или записи:**

```
byte[] data;                                    // Данные для шифрования
SecretKey key;                                  // Инициализировать, как показано ранее
Cipher c = Cipher.getInstance("DESede");        // Объект для выполнения шифрования
c.init(Cipher.ENCRYPT_MODE, key);               // Инициализировать его

// Создать поток для записи байтов в файл
FileOutputStream fos = new FileOutputStream("encrypted.data");

// Создать поток, который шифрует данные перед отправкой в этот поток
// См. также CipherInputStream для шифрования и расшифрования данных при чтении
CipherOutputStream cos = new CipherOutputStream(fos, c);

cos.write(data);                                // Зашифровать и записать данные в файл
cos.close();                                    // Не забывайте закрывать потоки
java.util.Arrays.fill(data, (byte)0);           // Удалить незашифрованные данные
```

Зашифрованные объекты

И наконец, класс `javax.crypto.SealedObject` **предоставляет самый простой способ шифрования. Этот класс сериализует специальный объект и шифрует результирующий поток байтов. Затем объект** `SealedObject` **может сам себя сериализовать и передать получателю. Получатель сможет принять соответствующий объект, если знает необходимый** `SecretKey`**:**

```
Serializable o;          // Объект для шифрования; должен быть Serializable
SecretKey key;           // Ключ для шифрования
Cipher c = Cipher.getInstance("Blowfish");      // Объект для выполнения шифрования
c.init(Cipher.ENCRYPT_MODE, key);               // Инициализировать его с помощью ключа
SealedObject so = new SealedObject(o, c);       // Создать объект SealedObject.

// Объект просто является оберткой для исходного зашифрованного объекта o; теперь он может
// быть сериализован и передан получателю. Вот как получатель расшифрует исходный объект
Object original = so.getObject(key);            // Должен применяться тот же SecretKey
```

Глава 5

Безопасность в Java

Java-программы могут динамически загружать классы из различных источников, в том числе из ненадежных, таких как веб-сайты, соединение с которыми происходит по незащищенной сети. Способность создавать мобильный код и работать с ним является одной из сильных сторон Java. В Java большое внимание уделяется архитектуре защиты, которая позволяет ненадежному коду безопасно выполняться на сервере.

Потребность в системе защиты чаще всего возникает при использовании апплетов – маленьких программ на Java, предназначенных для встраивания в веб-страницы.[1] Когда с помощью броузера, поддерживающего Java, пользователь посещает веб-страницу, содержащую апплет, броузер загружает файлы Java-классов, из которых состоит апплет, и выполняет их. При отсутствии системы защиты апплет мог бы причинить вред системе пользователя – удалить файлы, устаовить вирус или похитить конфиденциальную информацию. В более тонких случаях апплет может подменить адрес электронной почты, сгенерировать спам и организовать хакерские атаки на другие системы.

Основное средство защиты в Java – контроль доступа: ненадежной программе не предоставляется доступ к жизненно важным частям ядра Java API. Например, ненадежному апплету обычно не разрешается чтение, запись или удаление файлов в системе хоста или соединение через сеть с другим компьютером, за исключением веб-сервера, с которого загружен апплет. В этой главе описывается архитектура контроля доступа Java и некоторые другие аспекты системы безопасности.

Угрозы безопасности

Концепция безопасности заложена в основу Java еще на этапе разработки. Это дает Java большое преимущество перед многими существующими системами и платформами. Тем не менее ни одна система не может гарантировать безопасность на 100%, и Java здесь не исключение.

Архитектура системы безопасности Java была спроектирована экспертами в этой области, ее исследовали и тестировали многие специалисты по безопасности. Они пришли к соглашению, что сама по себе архитектура обладает устойчивостью и защищенностью и теоретически в ней нет брешей (по крайней мере, ни одна до сих пор не

---

[1] Документацию по апплетам можно найти в книге «Java Foundation Classes in a Nutshell» (O'Reilly), поэтому они не рассматриваются подробно в нашей книге. Апплеты служат здесь хорошими примерами.

найдена). Другое дело – ее реализация. Обнаружение ошибок в системе безопасности и их устранение в отдельных реализациях Java имеет долгую историю. Например, в апреле 1999 года была обнаружена ошибка в верификаторе классов Java 1.1. Вышли «заплатки» для Java 1.1.6 и 1.1.7, была исправлена ошибка в Java 1.1.8. Позже, в августе 1999, специалисты обнаружили серьезную ошибку в Microsoft Java Virtual Machine, которая используется в веб-броузерах Internet Explorer 4.0 и 5.0. Эта ошибка была особенно опасной, потому что она предоставляла апплетам неограниченный доступ к системе пользователя. Microsoft выпустила новую версию виртуальной машины. Не было зафиксировано ни одной атаки, использующей эту брешь.

По всей видимости, специалисты будут и дальше находить бреши системы безопасности в реализациях виртуальной машины и выпускать для них «заплатки». Тем не менее Java остается, наверное, самой защищенной платформой из всех существующих. Известно очень мало случаев, когда вредоносный Java-код воспользовался изъянами системы безопасности на практике. Платформа Java выглядит достаточно защищенной, особенно в сравнении с некоторыми альтернативными платформами, у которых слабая система безопасности и ненадежная защита от вирусов.

Защита виртуальной машины Java и верификация файлов классов

Система защиты Java на самом низком уровне использует особенности архитектуры виртуальной машины и исполняемого ею байт-кода. Виртуальная машина Java не допускает прямого доступа к ячейкам памяти базовой системы. Это предотвращает воздействие кода Java на операционную систему и аппаратное обеспечение. Внутренние ограничения виртуальной машины отражаются на самом языке Java, который не поддерживает указатели и адресную арифметику. Язык не допускает приведение целого типа к ссылке или наоборот. Нет абсолютно никакого способа получить адрес объекта в памяти. Без этих возможностей вредоносному коду просто не на что опереться.

Помимо того что виртуальная машина имеет безопасный набор инструкций, она также проводит *верификацию байт-кода*, когда загружает класс из ненадежного источника. Этот процесс гарантирует следующее: все байт-коды и операнды класса будут допустимыми; стек виртуальной машины не опустошится и не переполнится; локальные переменные не будут использоваться до инициализации; модификаторы полей, методов, управления доступом будут правильными и т. д. Верификация разработана для того, чтобы виртуальная машина не смогла выполнить программу, которая может аварийно завершиться или привести машину в неопределенное и непроверенное состояние. В этом состоянии программа становится уязвимой для атак злоумышленников. Верификация байт-кода – это защита от вредоносных программ и ненадежных компиляторов Java, которые могут сгенерировать недопустимый байт-код.

Аутентификация и криптография

В Java 1.1 и последующих версиях пакет `java.security` и его подпакеты предоставляют классы и интерфейсы для *аутентификации*. Как описано в главе 4, эти элементы архитектуры безопасности позволяют Java-коду создавать и проверять профили сообщений и цифровые подписи. Использование этих технологий гарантирует, что данные (например, файл класса Java) – подлинные, то есть действительно предоставлены заявленным поставщиком и не были случайно или умышленно изменены при пересылке.

Криптографическое дополнение к Java (Java Cryptography Extension, JCE) состоит из пакета `javax.crypto` и входящих в него подпакетов. В этих пакетах определены классы для шифрования данных. Во многих программах криптография используется для обеспечения защиты данных, но она не имеет прямого отношения к основной теме главы и здесь рассматриваться не будет.

Контроль доступа

Как было отмечено в начале главы, стержнем архитектуры безопасности Java является контроль доступа: ненадежной программе не предоставляются в распоряжение те части Java API, с помощью которых она может нанести вред. Модель контроля доступа в Java 1.2 была значительно усовершенствована по сравнению с Java 1.0; подробнее мы это обсудим в следующих разделах. Контроль доступа в Java 1.2 относительно стабилен; он не подвергся значительным изменениям в Java 1.3.

Java 1.0: «песочница»

В первой версии Java все программы этой платформы, локально установленные в системе, считаются надежными по определению. Все программы, загруженные по сети, не являются надежными и выполняются в ограниченной среде, в шутку названной «песочницей». Политика контроля доступа определяется текущим установленным объектом `java.lang.SecurityManager`. Перед выполнением операции, на которую наложены ограничения, например, чтения локального файла, системный код вызывает соответствующий метод (в данном случае `checkRead()`) объекта `SecurityManager`. Если выполняемый код является ненадежным, то генерируется исключение `SecurityException`, предотвращающее выполнение недопустимой операции.

Классом `SecurityManager` обычно пользуется броузер с поддержкой Java, который устанавливает объект этого класса, чтобы можно было безопасно запускать апплеты. Конечно, реализация системы безопасности зависит от конкретного броузера, но в общем случае на апплеты налагаются следующие ограничения:

- Апплет не может читать, записывать, переименовывать или удалять файлы. Он не может запрашивать длину или дату последнего изменения файла и даже проверять, существует ли такой файл. Соответственно, он не может создавать, удалять каталог или получать сведения о подкаталогах.
- Апплет не может создавать сетевое соединение или принимать запрос на соединение с другой системой, кроме той, с которой он был загружен. Он не может использовать привилегированные порты (до 1024 включительно).
- Апплету не разрешается производить действия системного уровня, такие как загрузка библиотеки, создание нового процесса или выход из интерпретатора Java. Апплет не может манипулировать потоками или группами потоков, кроме тех, которые создает сам. В Java 1.1 и последующих версиях апплеты не могут использовать Java Reflection API для получения информации о членах класса, которые не являются открытыми (public), за исключением тех классов, которые загрузила сама программа.
- Апплет не имеет доступа к некоторым графическим функциям и средствам GUI. Он не может создать задание для принтера или получить доступ к буферу обмена или очереди сообщений. Кроме того, все окна, созданные апплетом, помечены как небезопасные (визуальным индикатором). Это сделано для того, чтобы апплет не смог обмануть пользователя, приняв вид другого приложения.

- Апплету не разрешается чтение некоторых системных свойств, в первую очередь `user.home` и `user.dir`, которые определяют домашний каталог пользователя и текущий рабочий каталог.
- Апплет не может обойти эти ограничения, зарегистрировав новый объект `SecurityManager`.

Как работает «песочница»

Предположим, что апплет (или какой-нибудь другой ненадежный код, выполняющийся в безопасной среде) предпринимает попытку чтения из файла */etc/passwd*, передав его имя конструктору `FileInputStream()`. Программисты, которые писали класс `FileInputStream()`, знали, что класс предоставит доступ к системному ресурсу (файлу), поэтому при использовании класса необходим контроль доступа. Для этого был написан конструктор `FileInputStream()`, использующий класс `SecurityManager`.

При каждом вызове `FileInputStream()` проверяется, был ли установлен `SecurityManager`. Если да, то конструктор вызывает метод `checkRead()` объекта `SecurityManager`, передав в качестве аргумента имя файла (в нашем случае */etc/passwd*). Метод `checkRead()` не возвращает значения; он либо нормально завершается, либо генерирует исключение `SecurityException`. Если выполняется возврат, то конструктор `FileInputStream()` производит необходимую инициализацию и завершается. В противном случае исключение передается вызывающему методу. Когда это происходит, объект класса `FileInputStream` не создается, и апплет не получает доступа к файлу */etc/passwd*.

Java 1.1: классы с цифровой подписью

В Java 1.1 сохранилась модель «песочницы» Java 1.0, но добавился новый пакет `java.security` для работы с цифровыми подписями. За счет этого на классы Java можно ставить цифровую подпись, а также проверять ее. Таким образом, броузеры и другие системы, использующие Java, можно настроить так, что загруженная программа, имеющая цифровую подпись надежного источника, будет считаться надежной. Такой код воспринимается так же, как установленный локально, поэтому ему полностью предоставляются Java API. В эту версию Java включена программа *javakey*, которая управляет ключами и ставит цифровые подписи на JAR-файлы с кодом. Хотя в Java 1.1 добавилась возможность доверять любой программе с действительной цифровой подписью, в этой версии остается модель «песочницы»: надежная программа получает полные права, а ненадежная ограничивается в доступе.

Java 1.2: политики и права доступа

В архитектуре системы безопасности Java 1.2 появились новые возможности контроля доступа. В пакет `java.security` добавлены новые классы. Класс `Policy` – одно из самых важных нововведений. Он определяет политику безопасности Java. Объект `Policy` связывает объекты `CodeSource` с наборами объектов класса `Permission`. Объект `CodeSource` содержит информацию об источнике Java-программы: URL файла класса (может быть локальным файлом) и список источников, поставивших на файл свои электронные подписи. Объекты типа `Permission`, связанные с `CodeSource`, определяют права доступа, предоставленные коду из данного источника. Различные Java API содержат классы-потомки `Permission`, которые представляют разные виды прав доступа. Например, к ним относятся `java.lang.RuntimePermission`, `java.io.FilePermission` и `java.net.SocketPermission`.

В новой модели контроля доступа класс SecurityManager остается центральным; с помощью его методов выполняются запросы на контроль доступа. Но в реализации SecurityManager, установленной по умолчанию, большая часть этих запросов передается новому классу AccessController, который принимает решения о предоставлении доступа, основываясь на архитектуре прав доступа и политик.

Новая архитектура контроля доступа Java 1.2 имеет несколько важных особенностей:

- Программам из различных источников могут быть предоставлены разные наборы прав доступа. Другими словами, новая архитектура поддерживает настраиваемые уровни доверия. Даже локально установленная программа может получить статус ненадежной или частично ненадежной. В новой архитектуре абсолютно надежными считаются только системные классы и стандартные расширения.
- Для определения политики безопасности нет необходимости создавать собственный класс-потомок SecurityManager. Системный администратор может настроить политику путем внесения изменений в текстовый файл или с помощью новой программы *policytool*.
- В новой архитектуре набор методов для контроля доступа не ограничивается методами класса SecurityManager. Можно легко определить новые классы-потомки Permission для управления доступом к системным ресурсам, который можно открыть с помощью новых стандартных расширений.

Как работает система политик и прав доступа

Вернемся к примеру апплета, который предпринимает попытку создать объект FileInputStream для чтения файла */etc/passwd*. В Java 1.2 конструктор FileInputStream() работает точно так же, как в версиях 1.0 и 1.1: он проверяет, установлен ли SecurityManager, и если да, то вызывает метод checkRead(), передав имя файла в качестве параметра.

В Java 1.2 метод checkRead() работает иначе. Если программа не заменила менеджер безопасности своим собственным, то предустановленный менеджер безопасности создает объект FilePermission, представляющий запрошенный доступ. *Цель* (*target*) этого объекта имеет значение «/etc/passwd», а *действие* (*action*) – «read». Метод checkRead() передает объект FilePermission статическому методу checkPermission() класса java.security.AccessController.

В Java 1.2 именно AccessController и его метод checkPermission() выполняют работу по контролю доступа. Этот метод определяет объект CodeSource для каждого вызывающего его метода и использует текущую политику – объект Policy – для определения объектов Permission, связанных с этой политикой. Пользуясь данной информацией, AccessController может определить, давать ли разрешение на чтение файла */etc/passwd*.

Класс Permission представляет права доступа, данные объектом Policy, и права доступа, запрошенные каким-нибудь методом, например конструктором FileInputStream(). Когда происходит запрос прав доступа, Java обычно использует FilePermission (или другой потомок класса Permission) с конкретным значением цели, например «/etc/passwd» (один файл). Но, предоставляя разрешение, Policy обычно использует объект FilePermission, цель которого задается шаблоном для предоставления множества файлов. В нашем примере – «/etc/*». Одна из ключевых особенностей подклассов Permission (например FilePermission) – это метод implies(), который может определить, подразумевает ли разрешение на чтение «/etc/*» разрешение на чтение «/etc/passwd».

Безопасность для всех

Программисты, системные администраторы и конечные пользователи рассматривают систему безопасности с разных позиций и, следовательно, играют разные роли при работе с архитектурой безопасности Java 1.2.

Безопасность и системные программисты

Системные программисты создают новые программные интерфейсы (API), осуществляющие доступ к «чувствительным» системным ресурсам. Как правило, эти программисты работают с «родными» (native) методами, которые осуществляют незащищенный доступ к системе. Чтобы не допустить вызов этих методов ненадежным кодом на Java, нужно задействовать механизм управления доступом Java. Для этого системный программист должен вставлять вызовы методов класса `SecurityManager` в соответствующие фрагменты программного кода. Кроме того, можно использовать уже существующие потомки класса `Permission` для управления доступом к системным ресурсам, предоставляемым API, или создать специализированный подкласс класса `Permission`.

Системный программист несет большую ответственность за обеспечение безопасности системы: если в его программном коде не производится соответствующая проверка доступа, то нарушается целостность всей системы безопасности платформы Java. Подробно эта тема здесь не освещается в связи с ее сложностью. К счастью, задачи системного программирования, требующие написания «родных» методов, встречаются довольно редко. Большинство читателей этой книги могут применять готовые API.

Безопасность и прикладные программисты

Программисты, которые используют ключевые API Java и стандартные расширения, но не определяют новые расширения или «родные» методы, могут положиться на средства безопасности, созданные системными программистами – авторами этих API. Другими словами, большинство программистов могут использовать API Java в своих приложениях, не опасаясь нарушить работоспособность системы безопасности платформы Java.

В действительности разработчикам приложений редко приходится использовать механизм управления доступом. При написании программы, которая при выполнении будет считаться ненадежной, нужно помнить о типичных ограничениях, налагаемых политиками безопасности на такие программы. Следует иметь в виду, что некоторые методы (например, методы для чтения или записи файла) могут вызывать исключения типа `SecurityException`, но это не означает, что в приложении необходимо обрабатывать эти исключения. Зачастую `SecurityException` нужно игнорировать ; это вызовет завершение приложения.

Иногда требуется написать приложение (например, средство просмотра апплетов), которое будет загружать ненадежные классы и запускать их под контролем системы управления доступом. Для этого в Java 1.2 сначала нужно установить менеджер безопасности:

```
System.setSecurityManager(new SecurityManager());
```

Затем ненадежные классы загружаются с помощью объекта `java.net.URLClassLoader`. Объект `URLClassLoader` дает загружаемому классу набор прав доступа, определенный по умолчанию, но в некоторых случаях придется изменить права доступа, предостав-

ляемые загружаемой программе. Для этого нужно задействовать классы `Policy` и `PermissionCollection`.

Безопасность и системные администраторы

При работе с Java 1.2 и последующими версиями системные администраторы отвечают за определение политики безопасности на вверенных им системах. Политика, определенная по умолчанию, хранится в файле *lib/security/java.policy* каталога установки Java. Системный администратор может изменять текстовый файл вручную либо запустить программу *policytool* фирмы Sun, которая позволяет редактировать файл, используя графический интерфейс. Определять политики предпочтительнее с помощью *policytool*, поэтому синтаксис файла политик в данной книге не рассматривается.

Политика, определенная по умолчанию в файле *java.policy*, сильно напоминает политику в Java 1.0 и 1.1: системные классы и установленные расширения считаются полностью надежными, а все остальные программы считаются ненадежными и получают лишь небольшой набор простых прав доступа. Хотя такая политика подходит во многих случаях, в некоторых организациях программам, загружаемым из безопасной интрасети, требуется предоставить больше прав доступа.

Чтобы правильно и эффективно определить политику безопасности, системный администратор должен понимать назначение различных потомков класса `Permission` платформы Java, знать имена поддерживаемых целевых объектов и действий, а также последствия, которые может вызывать предоставление того или иного права доступа. Эти темы освещены в документе «Permissions in the Java 2 SDK», который входит в Java 1.2 и доступен (на время написания книги) по адресу *http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html*.

Система безопасности для конечных пользователей

Большинству конечных пользователей не приходится заниматься проблемами безопасности: им нужно только, чтобы Java-программы выполнялись безопасным образом и не требовали вмешательства пользователя. Однако некоторые продвинутые пользователи могут определять собственные политики безопасности. Для этого они могут запустить *policytool* и указать собственные файлы политик, дополняющие системную политику безопасности. Персональная политика безопасности по умолчанию хранится в файле *.java.policy* в домашнем каталоге пользователя. По умолчанию Java загружает эту политику и дополняет ею системную политику безопасности.

В Java 1.2 и последующих версиях пользователь может указать дополнительный файл политики, который будет использоваться при запуске интерпретатора Java. Для этого в командной строке нужно задать ключ `-D` и свойство `java.security.policy`. Например:

```
C:\> java -Djava.security.policy=policyfile UntrustedApp
```

Эта строка запускает класс `UntrustedApp`, дополнив системную и пользовательскую политики безопасности политикой, определенной в файле (или URL) *policyfile*. Чтобы новая политика не дополняла системные и пользовательские политики, а заменяла их собой, при указании свойства нужно поставить двойной знак равенства:

```
C:\> java -Djava.security.policy==policyfile UntrustedApp
```

Обратите внимание, что указание файла политик имеет смысл только в том случае, если установлен `SecurityManager`. Если пользователь не доверяет приложению, то ему, скорее всего, не нужно, чтобы данное приложение установило собственный менед-

жер безопасности. В таком случае он может определить системное свойство `java.security.manager`:

```
C:\> java -Djava.security.manager -Djava.security.policy=policyfile UntrustedApp
```

Неважно, какое значение имеет это свойство. Достаточно лишь задать его, а затем интерпретатор Java автоматически установит предопределенный объект `SecurityManager`, который будет следить за выполнением политик управления доступом, описанных в файлах системной, пользовательской политик и в `java.security.policy`.

Классы прав доступа

В табл. 5.1 перечислены различные потомки класса `Permission`, определенные в ядре платформы Java, и указаны предоставляемые ими права доступа. Подробности о каждом классе можно прочитать в справочном разделе. Детальное описание этих прав доступа, перечень имен целевых объектов и действий, список методов и требуемых для них прав доступа можно найти по адресу *http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html* (этот документ входит в стандартную документацию, которую можно получить вместе с JDK).

Таблица 5.1. Классы прав доступа Java

Класс права доступа	Описание
`java.security.AllPermission`	Это специальный класс прав доступа, включающий в себя все права доступа.
`javax.sound.sampled.AudioPermission`	Контролирует воспроизведение и запись звука.
`javax.security.auth.AuthPermission`	Контролирует доступ к аутентификационным методам пакета `javax.security.auth` и его подпакетов.
`java.awt.AWTPermission`	Управляет доступом к аутентификационным методам пакета `java.awt` и подпакетов.
`java.io.FilePermission`	Управляет доступом к файловой системе.
`java.net.NetPermission`	Управляет доступом к ресурсам, связанным с сетью – обработчикам потоков и системе аутентификации HTTP. См. также `java.net.SocketPermission`.
`java.util.PropertyPermission`	Управляет доступом к системным свойствам.
`java.lang.reflect.ReflectPermission`	Управляет доступом к классам и членам классов пакета `java.lang.reflect`, которые обычно должны быть недоступны.
`java.lang.RuntimePermission`	Контролирует большое количество методов и ресурсов. Многие из них находятся в пакетах `java.lang.System` и `java.lang.Runtime`.
`java.security.SecurityPermission`	Управляет доступом к различным методам, связанным с системой безопасности.
`java.io.SerializablePermission`	Управляет доступом к методам, связанным с сериализацией.
`java.net.SocketPermission`	Управляет доступом к сети.
`java.sql.SQLPermission`	Управляет возможностью указывать потоки протоколирования (logging stream) в `java.sql` JDBC API.

Глава 6

Компоненты JavaBeans

JavaBeans API обеспечивает основу для определения многократно используемых, модульных и встраиваемых компонентов ПО. Спецификация по JavaBeans определяет компонент Java так: «многократно используемый компонент ПО, которым можно визуально манипулировать в среде разработки». Следует отметить, что это достаточно расплывчатое определение; компонент JavaBeans может принимать различные формы. Наиболее широко компоненты Java применяются в элементах графического пользовательского интерфейса, например компоненты пакетов `java.awt` и `javax.swing`, описание которых представлено в книге «Java Foundation Classes in a Nutshell» (O'Reilly).[1] Хотя всеми компонентами JavaBeans можно манипулировать визуально, это не означает, что у каждого компонента Java есть свое собственное визуальное представление. Например, класс `javax.sql.RowSet`, описанный в книге «Java Enterprise in a Nutshell» (O'Reilly), является компонентом JavaBeans, который представляет данные, полученные по запросу к базе данных. Нет никаких ограничений, относящихся к простоте или сложности компонента JavaBeans. Самые простые компоненты Java – это, как правило, базовые компоненты графического интерфейса, такие как объект `java.awt.Button`. Даже сложные системы, например встраиваемые приложения для работы с таблицами, могут функционировать как индивидуальные компоненты.

Одна из целей создания модели JavaBeans – взаимодействие с похожими компонентными структурами. Так, например, Windows-программа при наличии соответствующего моста или объекта-обертки может использовать компонент JavaBeans так, словно он является компонентом COM или ActiveX. Детали такого взаимодействия в этой главе не рассматриваются.

Компонентная модель JavaBeans состоит из пакетов `java.beans` и `java.beans.beancontext`, а также ряда важных соглашений по именованию объектов и прикладным интерфейсам. Этим соглашениям должны соответствовать компоненты Java и инструментарий для работы с ними. JavaBeans – среда для работы с обычными компонентами, поэтому соглашения JavaBeans во многих отношениях даже более важны, чем сам API.

---

[1] Компоненты JavaBeans описываются именно в указанной книге, поскольку компонентная модель JavaBeans не является особенностью AWT- или Swing-программирования. И все же, вряд ли можно обсуждать компоненты Java, не упомянув компоненты AWT и Swing. Вероятнее всего, вы сможете извлечь для себя максимальную пользу из этого раздела, если знакомы с GUI-программированием на Java с использованием AWT и Swing.

Компоненты Java могут применяться на трех уровнях:

- Если вы пишете приложение, которое использует компоненты Java, разработанные другими программистами, или применяете инструмент *beanbox*[1] для создания приложения из этих компонентов, вам нужно ознакомиться с общими концепциями и терминологией JavaBeans. Кроме того, вам следует прочесть документацию по отдельным компонентам Java, которые вы используете в своем приложении, но вам не обязательно понимать JavaBeans API. Данный раздел начинается с обзора концепций JavaBeans, которые достаточны для программистов, использующих компоненты Java на этом уровне.
- Если вы создаете компоненты Java, вам следует понимать и следовать соглашениям по именованию компонентов JavaBeans и объединению их в пакеты. После введения в общую терминологию и концепции в данном разделе описаны основные соглашения по компонентам Java, которых должны придерживаться разработчики. Хотя компонент JavaBeans можно реализовать без JavaBeans API, большинство компонентов Java распространяются с различными вспомогательными классами, которые облегчают их использование в приложениях, задействующих эти компоненты. Вспомогательные классы в значительной степени опираются на JavaBeans API, благодаря чему они могут взаимодействовать с контейнерами компонентов (beanbox).
- Если вы разрабатываете GUI-редактор, конструктор приложений или другой инструмент, использующий компоненты, то вы применяете JavaBeans API для облегчения работы с компонентами Java в данном приложении. Вам следует хорошо понимать различные соглашения по программированию JavaBeans. Хотя в этом разделе описываются наиболее важные соглашения, вам также следует обращаться к первоисточнику, а именно к спецификации JavaBeans (см. *http://java.sun.com/beans*).

Основы компонентов Java

Компонентом Java может быть любой объект, который соответствует некоторым основным правилам; не существует класса Bean, от которого должны порождаться все компоненты Java. Многие компоненты Java являются AWT-компонентами, но также вполне возможно, а зачастую и полезно создавать «невидимые» компоненты Java, которые не видны на экране. Если у компонента Java нет представления на экране в законченном приложении, то это вовсе не значит, что с ним нельзя будет работать визуально в среде разработки.

Компонент Java характеризуется свойствами, событиями и методами, которые он экспортирует. Именно этими свойствами, событиями и методами оперирует разработчик приложений, применяя контейнер компонентов. *Свойство* – это часть внутреннего состояния компонента Java. Его можно установить программно и/или получить его значение – обычно при помощи стандартной пары методов доступа `get` и `set`.

С помощью генерации *событий* компонент Java общается с приложением, в которое он встроен, и с другими компонентами Java. JavaBeans API применяет ту же самую модель событий, которую используют AWT- и Swing-компоненты. Эта модель осно-

---

[1] *beanbox* – это название программы-примера для работы с компонентами Java, которая поставляется Sun в BDK (набор для разработки компонентов Java). Этот термин полезен; далее я буду применять его при описании любых инструментов для графического проектирования или разработки приложений, которые работают с компонентами Java.

вана на классе java.util.EventObject и интерфейсе java.util.EventListener; она детально описана в книге «Java Foundation Classes in a Nutshell» (O'Reilly). Вот, вкратце, схема работы модели событий:

- Компонент Java определяет событие, если он предоставляет методы add и remove для регистрации (добавления) и удаления объектов-слушателей данного события.
- Приложение, которое хочет получать уведомление о том, что произошло событие определенного типа, использует эти методы для регистрации объекта-слушателя событий соответствующего типа.
- Когда происходит событие, компонент Java уведомляет зарегистрированные приемники путем передачи событийного объекта, который описывает событие, методу, определенному интерфейсом слушателя событий.

Однонаправленное событие – это редкий вариант события, для которого может быть зарегистрирован единственный объект-приемник. Метод регистрации add для однонаправленного события вызывает исключение TooManyListenersException в том случае, если предпринимается попытка зарегистрировать более одного приемника.

Методы, экспортируемые компонентом Java, – это просто все методы с модификатором public, определенные компонентом Java, за исключением методов доступа к свойствам и методов, которые регистрируют и удаляют слушателей событий.

В дополнение к обычным свойствам, описанным выше, JavaBeans API поддерживает несколько специализированных подтипов. *Индексированное свойство* – это свойство, значением которого является массив, а также методы доступа, которые позволяют обращаться как к отдельным элементам массива, так и ко всему массиву в целом. *Связанное свойство* – это свойство, которое посылает событие PropertyChangeEvent любым заинтересованным объектам PropertyChangeListener, когда значение свойства изменяется. *Ограниченное свойство* – это свойство, любые изменения в котором могут быть заблокированы любым заинтересованным слушателем. Когда меняется значение ограниченного свойства компонента Java, он должен выслать PropertyChangeEvent списку заинтересованных объектов VetoableChangeListener. Если любой из этих объектов вызывает исключение PropertyVetoException, то значение свойства не меняется, а исключение PropertyVetoException возвращается методу, который устанавливает свойство.

Java позволяет динамически загружать классы, поэтому контейнеры компонентов могут загружать произвольные компоненты Java. Контейнер компонентов использует процесс под названием *интроспекция* (*introspection*) для того, чтобы определить свойства, события и методы, экспортируемые компонентом Java. Механизм интроспекции реализуется классом java.beans.Introspector; он опирается на механизм отражения java.lang.reflect и на ряд соглашений по именованию JavaBeans. Например, Introspector может определить список свойств, поддерживаемых компонентом Java, путем сканирования класса на предмет наличия методов, которые несут имена и сигнатуры, представляющие их как методы доступа к свойствам get и set.

Впрочем, механизм интроспекции опирается не только на возможности отражения, присутствующие в Java. Любой компонент Java может определять вспомогательный класс BeanInfo, который предоставляет дополнительную информацию о компоненте Java и его свойствах, событиях, методах. Introspector автоматически пытается найти и загрузить класс BeanInfo, принадлежащий компоненту Java.

Класс BeanInfo предоставляет дополнительную информацию о компоненте Java в форме объектов FeatureDescriptor, каждый из которых описывает одну функциональную особенность этого компонента. Каждый FeatureDescriptor предоставляет имя и краткое описание функциональной особенности, которую он документирует. Инструмент

beanbox может отображать имя и описание пользователю, что делает компонент Java самодокументируемым и более легким в использовании. Конкретные функциональные особенности компонентов Java – свойства, события и методы – описываются специфическими подклассами `FeatureDescriptor`, такими как `PropertyDescriptor`, `EventSetDescriptor` и `MethodDescriptor`.

Одна из главных задач контейнеров компонентов – предоставить пользователю возможность настройки компонента Java путем установки значений его свойств. Контейнер определяет *редакторы свойств* для часто используемых типов свойств, таких как числа, строки, шрифты и цвета. Если компонент Java имеет свойство более сложного типа, то он может определять класс `PropertyEditor`, который позволяет контейнеру компонентов предоставить редактор этого свойства.

Ко всему прочему, механизм отдельной настройки каждого свойства, который предоставляется большинством контейнеров компонентов, может оказаться недостаточным для сложного компонента Java. Такой компонент может определить класс `Customizer` для создания графического интерфейса, который позволит более удобно конфигурировать компонент Java. Особенно сложный компонент Java может определять модули настройки (customizers), которые служат в качестве «мастеров», предоставляющих пользователю возможность пошаговой настройки.

Контекст компонента Java – это логический (а часто и визуальный) контейнер для JavaBeans и, возможно, для других вложенных контекстов компонентов Java. На практике большая часть JavaBeans – это AWT- или Swing-компоненты или контейнеры. Контейнеры компонентов предусматривают это и позволяют составным компонентам Java размещаться внутри компонентов-контейнеров Java. Контекст компонента Java – это своего рода большегрузный контейнер, который формализует эту вложенность. Что более важно, контекст компонента Java может обеспечивать ряд сервисов (например, печать, отладку, связь с базой данных) для тех компонентов Java, которые он содержит. Можно написать компоненты Java, которые будут «осознавать» свой контекст и посылать запросы к контексту, чтобы воспользоваться доступными сервисами. Контексты компонентов Java реализуются с помощью `java.beans.beancontext` API. Эта возможность, появившаяся в Java 1.2, более подробно обсуждается далее.

Java 1.4 содержит JavaBeans Persistence API, который позволяет компоненту Java или дереву таких компонентов сохранять свое состояние в XML-файле, откуда впоследствии можно будет восстановить этот компонент или компоненты. Это выполняется с помощью классов `XMLEncoder` и `XMLDecoder` из пакета `java.beans`. Постоянство JavaBeans напоминает механизм сериализации пакета `java.io`, но он всецело основан на открытом API компонентов Java и, таким образом, является более надежным при одновременном использовании нескольких версий или реализаций этого API. Далее будут рассмотрены примеры нового JavaBeans Persistence API. Примеры API сериализации представлены в главе 4, а в главе 9 вы найдете детальное описание `XMLEncoder`, `XMLDecoder` и смежных классов.

Соглашения JavaBeans

Компонентная модель JavaBeans основывается на ряде правил и соглашений, которым должны следовать разработчики компонентов. Эти соглашения не являются частью самого JavaBeans API, но в общем и целом они более важны для разработчиков компонентов Java, чем сам API. Иногда соглашения называют *проектными шаб-*

лонами; они определяют такие понятия, как имена и сигнатуры методов доступа к свойствам, которые определяются компонентом Java.

Суть проектных шаблонов – возможность взаимодействия компонентов Java и контейнеров компонентов, которые с ними работают. Как мы уже видели, контейнеры компонентов могут использовать интроспекцию для определения списка свойств, событий и методов, поддерживаемых компонентом Java. Для того чтобы все это функционировало, разработчики компонентов Java должны использовать имена методов, которые понятны контейнерам компонентов. Модель JavaBeans облегчает процесс, устанавливая соглашения по именованию. Одно такое соглашение состоит, например, в том, чтобы методы доступа для получения и установки свойства начинались с `get` и `set`.

Не все эти шаблоны обязательны. Если компонент Java имеет методы доступа к свойству, которые не подчиняются соглашениями по именованию, то можно использовать объект `PropertyDescriptor` (представлен в классе `BeanInfo`) для обозначения таких методов. Хотя класс `BeanInfo` является альтернативой соглашению по именованию, которое основано на методе доступа к свойствам, такой метод должен придерживаться соглашений, задающих количество, тип его параметров и возвращаемое значение.

Компоненты Java

Сам компонент Java должен придерживаться следующих соглашений:

Имя класса

На имя класса компонента Java не налагается ограничений.

Родительский класс

Компонент Java может расширять любой другой класс. Компоненты Java часто являются AWT- или Swing-компонентами, но ограничений не существует.

Создание экземпляра

Компонент Java должен предоставлять конструктор без параметров или файл, содержащий сериализованный экземпляр. Контейнер компонентов может десериализовать этот экземпляр для использования в качестве прототипа при создании экземпляра компонента Java. Файл, который содержит компонент Java, должен иметь то же название, что и компонент Java, а также расширение *.ser*.

Имя компонента Java

Имя компонента Java – это имя класса, который его реализует, или имя файла, который содержит сериализированный экземпляр компонента Java без расширения *.ser* и символов разделения каталогов (/), которые будут переведены в точки (.).

Свойства

Компонент Java определяет свойство *p* типа *T* в том случае, если у него есть методы доступа, которые соответствуют этим шаблонам (если *T* – `boolean`, то разрешена особая форма метода get):

Метод, возвращающий значение (getter)

```
public T getP()
```

Метод, возвращающий значение boolean (boolean getter)

```
public boolean isP()
```

Метод, устанавливающий значение (setter)

```
public void setP(T)
```

Исключения

Методы доступа к свойству могут вызывать проверяемые и непроверяемые исключения любого типа.

Индексированные свойства

Индексированное свойство – это свойство типа «массив», которое предоставляет методы доступа, получающие и устанавливающие весь массив, равно как и методы, получающие и устанавливающие отдельные элементы массива. Компонент Java определяет индексированное свойство *p* типа *T*[], если он определяет следующие методы доступа:

Метод, возвращающий массив

```
public T[] getP()
```

Метод, возвращающий элемент

```
public T getP(int)
```

Метод, устанавливающий массив

```
public void setP(T[])
```

Метод, устанавливающий элемент

```
public void setP(int, T)
```

Исключения

Методы доступа к свойству могут вызывать проверяемые и непроверяемые исключения любого типа. В частности, они должны генерировать исключение `ArrayIndexOutOfBoundsException`, если запрашиваемый индекс выходит за пределы допустимого диапазона.

Связанные свойства

Связанное свойство – это свойство, которое генерирует событие `PropertyChangeEvent`, когда его значение меняется. Ниже приведены соглашения для связанного свойства:

Методы доступа

Методы доступа к связанному свойству отвечают тем же соглашениям, что и методы доступа к обычному свойству.

Интроспекция

При помощи одной только интроспекции контейнер компонентов не сможет отличить связанное свойство от несвязанного. Поэтому вам, возможно, следует реализовать класс `BeanInfo`, который возвращает объект `PropertyDescriptor` для этого свойства. Метод `isBound()` объекта `PropertyDescriptor` должен возвращать `true`.

Регистрация слушателя

Компонент Java, определяющий одно или несколько связанных свойств, должен определить пару методов для регистрации слушателей, которые получают уведомление при изменении значения какого-либо связанного свойства. Такие методы должны иметь следующие сигнатуры:

```
public void addPropertyChangeListener(PropertyChangeListener)
public void removePropertyChangeListener(PropertyChangeListener)
```

Регистрация слушателя для именованного свойства

Компонент Java может, но не обязан предоставлять дополнительные методы, позволяющие регистрировать слушателей событий, при которых изменяются значе-

ния отдельного связанного свойства. Этим методам передается имя свойства. Они имеют следующие сигнатуры:

```
public void addPropertyChangeListener(String, PropertyChangeListener)
public void removePropertyChangeListener(String, PropertyChangeListener)
```

Регистрация слушателя для отдельного свойства

Компонент Java может, но не обязан предоставлять дополнительные методы регистрации приемников событий, которые специфичны для конкретного свойства. Для свойства *p* эти методы имеют следующие сигнатуры:

```
public void addPListener(PropertyChangeListener)
public void removePListener(PropertyChangeListener)
```

Методы такого типа позволяют контейнерам компонентов отличать связанное свойство от несвязанного.

Уведомление

Когда изменяется значение связанного свойства, компонент Java должен обновить свое внутреннее состояние, чтобы отразить это изменение, а затем передать `PropertyChangeEvent` методу `propertyChange()` каждого объекта `PropertyChangeListener`, зарегистрированного для такого компонента Java или же для конкретного связанного свойства.

Поддержка

Класс `java.beans.PropertyChangeSupport` полезен при реализации связанных свойств.

Ограниченные свойства

Ограниченное свойство – это свойство, любые изменения в котором могут быть заблокированы зарегистрированными слушателями. Большая часть ограниченных свойств также является связанными свойствами. Ниже приведены соглашения для ограниченного свойства:

Метод, возвращающий значение

Возвращающий метод для ограниченного свойства – это то же самое, что и возвращающий метод для обычного свойства.

Метод, устанавливающий значение

Устанавливающий метод ограниченного свойства генерирует `PropertyVetoException`, если блокируется изменение свойства. Для свойства *p* типа *T* есть следующая сигнатура:

```
public void setP(T) throws PropertyVetoException
```

Регистрация слушателя

Компонент Java, который определяет одно или несколько ограниченных свойств, должен определить два метода для регистрации слушателей, которые получают уведомление, когда меняется значение свойства. Эти методы должны иметь следующие сигнатуры:

```
public void addVetoableChangeListener(VetoableChangeListener)
public void removeVetoableChangeListener(VetoableChangeListener)
```

Регистрация слушателя для именованного свойства

Компонент Java может, но не обязан предоставлять дополнительные методы, позволяющие регистрировать слушателей событий, при которых изменяется значе-

ние ограниченного свойства. Этим методам передается имя свойства. Они имеют следующие сигнатуры:

```
public void addVetoableChangeListener(String, VetoableChangeListener)
public void removeVetoableChangeListener(String, VetoableChangeListener)
```

Регистрация слушателя для отдельного свойства

Компонент Java может, но не обязан предоставлять дополнительные методы регистрации слушателей; такие методы специфичны для конкретного ограниченного свойства. Для свойства *p* эти методы имеют следующие сигнатуры:

```
public void addPListener(VetoableChangeListener)
public void removePListener(VetoableChangeListener)
```

Уведомление

Когда вызывается устанавливающий метод ограниченного свойства, компонент Java должен сгенерировать событие `PropertyChangeEvent`, которое описывает запрошенное изменение, и передать это событие методу `vetoableChange()` каждого объекта `VetoableChangeListener`, зарегистрированного для компонента Java или конкретного ограниченного свойства. Если какой-либо слушатель заблокирует это изменение, вызвав исключение `PropertyVetoException`, то компонент Java должен будет сгенерировать еще одно событие `PropertyChangeEvent` для того, чтобы вернуть свойство к его первоначальному значению. Затем компонент должен сам вызвать исключение `PropertyVetoException`. С другой стороны, если изменение свойства не будет заблокировано, то компонент Java должен обновить свое внутреннее состояние с тем, чтобы оно отражало проведенное изменение. Если ограниченное свойство также является связанным свойством, то компонент Java должен в этот момент уведомить объекты `PropertyChangeListener`.

Поддержка

Класс `java.beans.VetoableChangeSupport` полезен при реализации ограниченных свойств.

События

В дополнение к событиям `PropertyChangeEvent`, которые генерируются в момент изменения связанных и ограниченных свойств, компонент Java может генерировать и другие типы событий. Событие, названое *E*, должно соответствовать следующим соглашениям:

Класс события

Класс события должен напрямую или косвенно расширять `java.util.EventObject`. Кроме того, он должен называться *E*`Event`.

Интерфейс слушателя

Событие должно быть ассоциировано с интерфейсом слушателя событий, который расширяет `java.util.EventListener` и называется *E*`Listener`.

Методы слушателя

Интерфейс слушателя событий может определять любое количество методов, которые принимают единственный аргумент типа *E*`Event` и возвращают `void`.

Регистрация слушателя

Компонент Java должен определять пару методов для регистрации слушателей событий, которые хотят получать уведомления в случае, если происходит событие *E*. Такие методы должны иметь следующие сигнатуры:

```
public void addEListener(EListener)
public void removeEListener(EListener)
```

Однонаправленные события

Однонаправленное событие позволяет регистрировать только одного слушателя в некий момент времени. Если *E* является однонаправленным событием, то у метода регистрации слушателя должна быть следующая сигнатура:

```
public void addEListener(EListener) throws TooManyListenersException
```

Методы

Контейнер компонентов может предоставлять методы компонента Java разработчикам приложений. Единственное формальное соглашение состоит в том, что эти методы должны быть объявлены `public`. Тем не менее полезно ознакомиться со следующими положениями:

Имя метода

Метод может иметь любое имя, которое не противоречит соглашениям по именованию свойств и событий. Имя должно быть как можно более значимым.

Параметры

Метод может иметь любое количество параметров любого типа. Тем не менее контейнеры компонентов лучше всего работают с методами, не имеющими параметров, или с методами, которые имеют простые параметры примитивных типов.

Исключение методов

С помощью реализации `BeanInfo` компонент Java может явно указать список методов, которые он экспортирует.

Документация

Компонент Java может предоставлять дружественные и простые в восприятии локализованные имена и описания методов посредством объектов `MethodDescriptor`, которые возвращаются реализацией `BeanInfo`.

Вспомогательные классы

Компонент Java может предоставлять следующие вспомогательные классы:

`BeanInfo`

В целях предоставления дополнительной информации о компоненте *B* нужно реализовать интерфейс `BeanInfo` в классе с названием *B*`BeanInfo`.

Редактор свойств для конкретного типа

Чтобы предоставить контейнерам компонентов возможность работать со свойствами типа *T*, реализуйте интерфейс `PropertyEditor` в классе с названием *T*`Editor`. Класс должен иметь конструктор, не содержащий параметров.

Редактор свойств для конкретного свойства

Для настройки способа, с помощью которого контейнер компонентов позволяет вводить значения отдельно взятых свойств, определите класс, который реализует интерфейс `PropertyEditor` и имеет конструктор без параметров; зарегистрируйте этот класс, передав ему объект `PropertyDescriptor`, который возвращается классом `BeanInfo` для компонента Java.

Настройщики

Чтобы определить мастеры настройки для конфигурирования компонента *B*, задайте AWT- или Swing-компонент с конструктором без параметров, который выполняет настройку. Этот класс обычно называют *B*Customizer, но это не обязательно. Зарегистрируйте класс с помощью объекта BeanDescriptor, который возвращается классом BeanInfo для этого компонента Java.

Документация

Напишите документацию по умолчанию для компонента *B* в формате HTML 2.0 и сохраните ее в файле с именем *B.html*. Подготовьте локализованные переводы документации в файлах с тем же именем, но в каталогах, соответствующих конкретным регионам (locale).

Объединение компонентов в пакеты и их распространение

Компоненты Java распространяются в архивных файлах JAR, которые подчиняются следующим соглашениям:

Содержание

Класс или классы, которые реализируют компонент Java, должны быть включены в JAR-файл вместе со вспомогательными классами, такими как реализации BeanInfo и PropertyEditor. Если для компонента Java создается экземпляр из сериализованного ранее экземпляра, то такой экземпляр должен быть включен в архив JAR. Имя этого файла должно заканчиваться на *.ser*. JAR-файл может содержать HTML-документацию для компонента Java и любые файлы ресурсов, такие как файлы изображений, необходимые компоненту Java и его вспомогательным классам. Один JAR-файл может содержать несколько компонентов Java.

Атрибут Java-Bean

В манифесте (manifest) JAR-файла все файлы *.class* и *.ser*, которые определяют компонент Java, должны быть помечены следующим атрибутом:

```
Java-Bean: true
```

Атрибут Depends-On

В манифесте JAR-файла можно применять атрибут Depends-On для того, чтобы указать все остальные файлы в JAR-архиве, необходимые компоненту Java. Контейнер компонентов может использовать эту информацию во время генерации приложений или при повторном объединении компонентов Java. Каждый компонент Java может иметь несколько атрибутов Depends-On, каждый из которых может включать в себя несколько имен файлов, разделенных пробелами. В JAR-файле символ «/» всегда используется как разделитель каталогов.

Атрибут Design-Time-Only

В манифесте JAR-файла можно использовать атрибут Design-Time-Only для указания вспомогательных файлов, таких как реализации BeanInfo. Эти файлы задействуются инструментом beanbox, но не используются приложениями, применяющими компонент Java. Beanbox может использовать эту информацию при повторном объединении компонентов Java в пакеты.

Контексты и сервисы компонентов Java

Компонентная модель JavaBeans была представлена в Java 1.1. Java 1.2 расширяет эту модель введением протокола включения (containment) и служб, который описывается в пакете java.beans.beancontext. Контекст компонента Java – это коллекция компонентов java.util.Collection, которая реализует интерфейс BeanContext и обеспечивает контекст для компонентов Java, которые она содержит. Многие контексты компонентов Java определяют один или несколько сервисов. Например, компоненты Java могут получать и использовать сервис печати. Эти контексты компонентов Java реализуют интерфейс BeanContextServices. Все контексты компонентов Java являются реализациями BeanContextChild, поэтому контексты могут быть вложены друг в друга.

Многим компонентам Java никогда не пригодится знание контекстов, которые их содержат. Компонент Java, который действительно желает воспользоваться своим контекстом и теми сервисами, которые он предоставляет, реализует интерфейс BeanContextChild. Когда элемент контекста компонента Java добавляется в контекст компонента Java, метод setBeanContext() интерфейса BeanContextChild вызывается контекстом компонента. Реализация этого метода должна сохранить ссылку на контекст компонента для будущего использования. Метод setBeanContext() является связанным и ограниченным свойством, поэтому он должен уведомлять объекты VetoableChangeListener и PropertyChangeListener. По этой причине многие компоненты Java поручают эти обязанности объекту BeanContextChildSuport.

Если компонент (или контекст компонента) Java вложен внутрь контекста компонента, реализующего BeanContextServices, то компонент может использовать сервисы, которые предоставляются контекстом компонента. Сервис идентифицируется классом Java, который его определяет. Таким образом, сервис печати идентифицируется объектом Class класса java.awt.print.PrinterJob, а сервис буфера обмена представлен классом java.awt.datatransfer.Clipboard. Компонент Java может вызвать метод hasService() окружающего объекта BeanContextServices, чтобы определить, доступен ли указанный сервис. Если да, то с помощью getService() он может получить подходящий экземпляр сервисного класса. Если контекст компонента Java вложен в другой контекст, то он может передавать методы hasService() и getService() в окружающий контекст.

В дополнение к getService() и hasService() интерфейс BeanContext предоставляет несколько других методов, которые доступны компонентам Java. Методы getResource() и getResourceAsStream() заменяют методы с такими же именами, определенные в классах Class и ClassLoader. Метод isDesignTime() из интерфейса DesignMode позволяет компоненту Java установить, отображается ли он внутри beanbox либо запущен в приложении или апплете. Метод BeanContext более предпочтителен, чем статический метод Beans.isDesignTime(), потому что он не является глобальным, а привязан к контексту.

BeanContext и BeanContextServices являются крупными интерфейсами; при их реализации нужно придерживаться достаточно сложных спецификаций. Эти спецификации управляют их взаимодействием с компонентами Java, которые в них содержатся, и с контекстами, в которые они вложены. По этим причинам разработчики компонентов Java нечасто создают узкоспециализированные контексты. Напротив, они полагаются на контексты, предоставляемые создателем контейнера компонентов. Опытные разработчики компонентов Java, которым все-таки надо создать контексты компонентов Java, могут поручить многие методы классам BeanContextSupport и BeanContextServicesSupport, которые реализуют основные протоколы и структуру.

Глава 7

Соглашения по программированию и документированию в Java

В этом разделе объясняется ряд важных и полезных соглашений по программированию на Java. Если вы будете придерживаться этих соглашений, то ваш Java-код будет самодокументирован. Его будет легче читать, сопровождать и переносить.

Соглашения по именованию и применению прописных букв

Следующие общепринятые соглашения по именованию применимы к пакетам, классам, методам, полям и константам. Эти соглашения выполняются повсеместно. Поскольку они влияют на открытые API определяемых классов, их следует тщательно придерживаться:

Пакеты

Обеспечьте уникальность имен ваших пакетов путем добавления к ним префиксов в виде написанного наоборот имени вашего интернет-домена (например, `com.davidflanagan.utils`). Все имена пакетов, или по крайней мере их уникальные префиксы, должны быть написаны строчными буквами.

Классы

Имя класса следует писать с заглавной буквы и использовать в нем буквы разных регистров (например, `String`). Если имя класса состоит из нескольких слов, то каждое слово должно начинаться с заглавной буквы (например, `StringBuffer`). Если имя класса или одно из слов в имени класса является акронимом, то акроним может быть полностью написан заглавными буквами (например, `URL`, `HTMLParser`).

Так как классы разрабатывались для представления объектов, вам следует выбирать имена классов, которые являются существительными (например, `Thread`, `Teapot`, `FormatConverter`).

Интерфейсы

Имена интерфейсов соответствуют тем же соглашениями по применению заглавных букв, что и имена классов. Когда интерфейс используется для предоставления дополнительной информации о классах, которые его реализуют, то в качестве имени интерфейса выбирается прилагательное (например, `Runnable`, `Cloneable`, `Serializable`, `DataInput`). Когда интерфейс выступает как абстрактный родительский

класс, используйте имя, которое является существительным (например, `Document`, `FileNameMap`, `Collection`).

Методы

Имя метода всегда начинается со строчной буквы. Если имя состоит из нескольких слов, то каждое слово, следующее за первым, должно начинаться с заглавной буквы (например, `insert()`, `insertObject()`, `insertObjectAt()`). Имена методов обычно выбираются так, чтобы первое слово было глаголом. Имена методов могут быть такой длины, какая необходима для объяснения их предназначения, однако по возможности следует выбирать краткие имена.

Поля и константы

Имена полей, не являющихся константами, выбираются в соответствии с теми же соглашениями по применению заглавных букв, что и имена методов. Если поле представляет собой константу `static final`, она записывается заглавными буквами. Если имя константы состоит из нескольких слов, то слова должны быть разделены символом подчеркивания (например, `MAX_VALUE`). Имя поля должно выбираться так, чтобы наиболее ясно выразить суть поля или то значение, которое оно содержит.

Параметры

Имена параметров метода содержатся в документации к конкретному методу, поэтому следует выбирать такие имена, которые максимально четко определяют предназначение параметров. Старайтесь сводить имена параметров к одному слову и последовательно придерживаться такой практики. Например, если класс `WidgetProcessor` определяет несколько методов, которые принимают объект `Widget` в качестве первого параметра, называйте этот параметр `widget` или даже `w` в каждом методе.

Локальные переменные

Имена локальных переменных – это детали реализации, которые никогда не будут видны за пределами вашего класса. И все же выбор подходящих имен означает код, который легче читать, понимать и поддерживать. Переменным обычно присваиваются имена, соответствующие тем же соглашениям, что и в случае с методами и полями.

В дополнение к соглашениям по конкретным типам имен существуют соглашения по символам, которые допустимо использовать в именах. Java допускает символ «$» в любом идентификаторе, однако в соответствии с соглашениями он предназначен для использования в синтетических именах, которые были сгенерированы обработчиками исходного кода. Например, он применяется Java-компилятором для того, чтобы сделать внутренние классы работоспособными. Java также допускает в именах любые буквенно-цифровые символы из всего набора символов Unicode. Такой подход удобен для неанглоязычных программистов, однако использование символов Unicode ограничено локальными переменными, закрытыми методами и полями, а также другими именами, которые не является частью открытого API класса.

Соглашения по переносимости и правила чистого языка Java (Pure Java)

Девиз Sun выражает ключевое преимущество Java – «Напиши один раз и запускай где угодно». Java облегчает написание переносимых программ, но это не означает, что Java-программы автоматически будут работать на любой Java-платформе. Для

обеспечения переносимости вам следует придерживаться нескольких достаточно простых правил, которые можно обобщить следующим образом:

Методы, зависимые от платформы

Переносимый код Java может использовать любые методы ядра Java API, включая методы, зависимые от платформы (native). И все же переносимый код не должен определять свои собственные методы, зависимые от платформы. Такие методы следует переписывать под каждую новую платформу, поэтому они напрямую подрывают лозунг Java «Напиши один раз и запускай где угодно».

Метод `Runtime.exec()`

В переносимом коде редко допустим вызов метода `Runtime.exec()` для порождения процесса и выполнения внешней команды в родной системе. Причиной служит тот факт, что команда родной ОС, которую нужно выполнить, не обязательно существует и ведет себя одинаково на всех платформах. Использование `Runtime.exec()` допустимо лишь в том случае, если пользователю разрешено указать команду для выполнения: либо набрать команду во время работы, либо указать ее в конфигурационном файле или диалоговом окне настроек.

Метод `System.getenv()`

Использование `System.getenv()` зависит от платформы; здесь нет никаких исключений. По этой причине применять данный метод не рекомендуется.

Недокументированные классы

Переносимый Java-код должен использовать только те классы и интерфейсы, которые являются документированной частью Java-платформы. Большинство Java-реализаций поставляются с дополнительными недокументированными открытыми классами, которые являются частью реализации, но не указаны в спецификации Java-платформы. Эти классы можно применять в программе, но подобное решение не обеспечивает переносимости, поскольку нет гарантии того, что классы существуют во всех Java-реализациях на всех платформах.

Пакет `java.awt.peer`

Интерфейсы в пакете `java.awt.peer` – это часть Java-платформы, но они документированы только для реализации AWT. Приложения, которые напрямую используют эти интерфейсы, не являются переносимыми.

Функциональность, зависящая от конкретной реализации

Переносимый код не должен основываться на функциональности, зависящей от конкретной реализации. Например, компания Microsoft выпустила версию системы исполнения Java-приложений, которая включала ряд дополнительных методов, не являвшихся частью Java-платформы в том виде, в каком это определено Sun. Очевидно, что любая программа, которая применяет оригинальные расширения Microsoft, не является переносимой. Собственное расширение Microsoft для Java-платформы привело к судебному процессу между Microsoft и Sun. Дело закончилось тем, что Microsoft отказалась от поддержки Java.

Ошибки, зависящие от конкретной реализации

Переносимый код не зависит ни от функциональных особенностей отдельной реализации, ни от ошибок, специфичных для такой реализации. Если класс или метод ведет себя не так, как указано в спецификации, то переносимая программа не должна полагаться на такое поведение, поскольку оно может быть различным на разных платформах.

Поведение, зависящее от конкретной реализации

Иногда различные платформы и реализации могут представлять разные варианты поведения, причем все они могут соответствовать Java-спецификации. Переносимый код не зависит от особенностей той или иной платформы. Например, в спецификации Java не указано, будут ли потоки с равным приоритетом делить между собой центральный процессор или же поток, запущенный раньше других, будет отнимать для себя ресурсы у потоков с равным приоритетом. Если приложение предполагает то или иное поведение, то, возможно, оно не будет функционировать должным образом на всех платформах.

Стандартные расширения

Переносимый код может основываться на стандартных расширениях Java-платформы, но в таком случае он должен четко обозначать используемые расширения. Такой код должен корректно прекращать работу, выдавая соответствующее сообщение об ошибке, если он выполняется в системе, на которой такие расширения не установлены.

Полностью укомплектованные программы

Любая переносимая Java-программа должна быть полностью укомплектована и самодостаточна: она должна содержать все применяемые классы, за исключением базовых классов и классов стандартных расширений.

Определение системных классов

Переносимый Java-код никогда не определяет классы в каких-либо системных пакетах или пакетах стандартных расширений. В противном случае нарушаются защитные границы этих пакетов и становятся видимы детали реализации пакетов.

Жестко закодированные имена файлов

Переносимая программа не содержит жестко закодированных имен файлов или каталогов. Дело в том, что на различных платформах файловые системы организованы по-разному; в них применяются различные символы для разделения каталогов. Если вам необходимо поработать с файлом или каталогом, то пользователь должен указать имя файла или по крайней мере базовый каталог, в котором можно обнаружить этот файл. Это можно сделать во время выполнения программы и в конфигурационном файле. Кроме того, такие сведения можно передать в качестве аргумента командной строки. При сцеплении имени файла или каталога с другим именем каталога применяйте конструктор `File()` или константу `File.separator`.

Разделители строк

Различные системы используют разные символы или последовательности символов в качестве разделителей строк. Не следует применять жесткую кодировку вида «\n», «\r» или «\r\n». Напротив, применяйте метод `println()` класса `PrintStream` или `PrintWriter`, который автоматически завершает строку при помощи разделителя строки, подходящего для данной платформы. Кроме того, можно использовать значение системного свойства `line.separator`.

Смешанные модели событий

В промежутке между выпусками Java 1.0 и Java 1.1 модель событий AWT существенно изменилась. Зачастую есть возможность смешивать эти модели событий, однако итоговый код не будет переносимым.

Изложенные правила входят в программу аттестации переносимости «100% Pure Java» от Sun; дополнительную информацию об этой программе и требования «Pure Java» можно найти по адресу *http://java.sun.com/100percent/*.

Документация в комментариях

В большинстве случаев комментарии в Java-коде поясняют детали реализации этого кода. Спецификация языка Java определяет особый тип комментариев, известный как документирующий комментарий (doc comment). Такие комментарии служат для документирования API вашего кода. Документирующий комментарий (далее просто комментарий) – это обычный многострочный комментарий, который начинается с символов «/**» (вместо обычных /*) и заканчивается символами «*/». Такой комментарий помещается непосредственно перед определением класса, интерфейса, метода или поля и содержит документацию для этого класса, интерфейса, метода или поля. Документация может включать в себя простые теги форматирования HTML или другие специальные ключевые слова, которые предоставляют дополнительную информацию. Комментарии игнорируются компилятором, но при помощи программы *javadoc* их можно извлечь и автоматически преобразовать в HTML-документацию (подробная информация о *javadoc* представлена в главе 8). Вот пример класса, который содержит надлежащие комментарии:

```
/**
 * Этот постоянный класс представляет <i>комплексные числа</i>.
 *
 * @author David Flanagan
 * @version 1.0
 */
public class Complex {
  /**
   * Содержит действительную часть комплексного числа.
   * @see #y
   */
  protected double x;

  /**
   * Содержит мнимую часть комплексного числа.
   * @see #x
   */
  protected double y;

  /**
   * Создает новый объект Complex, который представляет комплексное число
   * x+yi.
   * @param x Действительная часть комплексного числа.
   * @param y Мнимая часть комплексного числа.
   */
  public Complex(double x, double y) {
    this.x = x; this.y = y;
  }

  /**
   * Добавляет два объекта Complex и производит
   * третий объект, который представляет их сумму.
   * @param c1 Объект Complex
   * @param c2 Еще один объект Complex
   * @return Новый объект Complex, который представляет сумму
   *         <code>c1</code> и
   *         <code>c2</code>.
   * @exception java.lang.NullPointerException
```

```
     *          Если любой из аргументов представляет собой <code>null</code>.
     */
    public static Complex add(Complex c1, Complex c2) {
      return new Complex(c1.x + c2.x, c1.y + c2.y);
    }
}
```

Структура документирующего комментария

Тело комментария должно начинаться с краткого описания (длиной в одно предложение) класса, интерфейса, метода или поля. Это предложение может отображаться само по себе, как обобщающая документация, поэтому оно должно выделяться из остального текста. За первоначальным предложением может следовать любое количество других предложений и параграфов, которые описывают класс, интерфейс, метод или поле.

После описательных разделов может размещаться любое количество других разделов, каждый из которых начинается специальным тегом комментария, например: `@author`, `@param` или `@returns`. Эти именованные разделы предоставляют конкретную информацию о классе, интерфейсе, методе или поле, которая стандартным образом отображается программой *javadoc*. Полный набор тегов комментариев представлен в следующем разделе.

Описательный материал может содержать простые теги разметки HTML, такие как `<I>` для визуального выделения, `<CODE>` для имен классов, методов и полей, `<PRE>` для примеров многострочного кода. Он может содержать теги `<P>` для разбиения описания на абзацы, а также `<UL>`, `<LI>` и подобные теги для отображения маркированных списков и других структур такого же типа. Тем не менее нужно помнить, что материал, который вы пишете, размещается внутри более крупного, более сложного HTML-документа. По этой причине комментарии не должны содержать структурных тегов параграфов HTML, таких как `<H2>` или `<HR>`, которые могут повлиять на структуру более крупного документа.

Избегайте использования тега `<A>` для включения гиперссылок или перекрестных ссылок в ваши комментарии. Вместо этого используйте особый тег для комментария `{@link}`, который, в отличие от других тегов, может размещаться в любом месте комментария. Как описано в следующем разделе, тег `{@link}` позволяет создавать гиперссылки на другие классы, интерфейсы, методы и поля. Для его применения не нужно знать соглашения по структурированию HTML и именам файлов, используемых *javadoc*.

Если вы хотите включить в комментарий изображение, поместите файл с изображением в подкаталог *doc-files* каталога с исходным кодом. Присвойте изображению то же имя, что и классу, сопроводив его суффиксом с целым числом. Например, второе изображение, которое появится в комментарии для класса с названием `Circle`, может быть присоединено при помощи такого HTML-тега:

```
<IMG src="doc-files/Circle-2.gif">
```

Строки документирующего комментария вставляются внутрь комментария Java, поэтому любые ведущие пробелы или звездочки (`*`) убираются с каждой строки комментария. Следовательно, вам не нужно беспокоиться о звездочках, появляющихся в сгенерированной документации, или об отступах комментариев, которые могут повлиять на отступы примеров с кодом, включенных в комментарий при помощи тега `<PRE>`.

Теги документирующих комментариев

Как уже упоминалось ранее, *javadoc* распознает ряд специальных тегов, каждый из которых начинается с символа `@`. Эти теги позволяют стандартным образом вносить специфичную информацию в ваши комментарии. Они позволяют *javadoc* выбирать подходящий формат вывода этой информации. Например, тег `@param` позволяет указать имя и значение одного параметра метода. *javadoc* может извлечь эту информацию и отобразить ее, используя HTML-список `<DL>` или HTML-таблицу `<TABLE>`.

Теги комментария, которые распознаются *javadoc*, перечислены далее; документирующий комментарий обычно содержит эти теги в том порядке, в каком они здесь перечислены:

`@author` *имя*

Добавляет элемент «Author:», который содержит заданное имя. Этот тег следует применять для определения каждого класса или интерфейса, но он не может использоваться для отдельных методов или полей. Если у класса несколько авторов, размещайте теги `@author` на смежных строках. Например:

```
@author David Flanagan
@author Paula Ferguson
```

Перечисляйте авторов в хронологическом порядке, начиная с первого. Если автор неизвестен, вы можете указать «unascribed» («не установлен»). Программа *javadoc* не выводит информацию об авторстве, пока не указан аргумент командной строки `-author`.

`@version` *текст*

Вставляет запись «Version:», которая содержит конкретный текст. Например:

```
@version 1.32, 08/26/99
```

Этот тег должен включаться в комментарий каждого класса или интерфейса, но не должен применяться для отдельных методов или полей. Его часто используют вместе с автоматизированными средствами нумерации версий, имеющимися в системах контроля версий SCCS, RCS или CVS. *javadoc* выводит информацию о версиях, если указан аргумент командной строки `-version`.

`@param` *имя-параметра описание*

Добавляет указанный параметр и его описание в раздел «Parameters:» текущего метода. Комментарий по какому-либо методу или конструктору должен содержать один тег `@param` для каждого параметра метода. Эти теги должны появляться в том же порядке, что и параметры, указанные методом. Тег нельзя применять в комментариях классов, интерфейсов или полей. Для лаконичности описания следует применять краткие фразы. Однако если параметр требует детальной документации, описание может состоять из нескольких строк и включать столько текста, сколько необходимо. Кроме того, для выравнивания описаний можно использовать пробелы. Например:

```
@param o      объект для вставки
@param index  позиция, в которой необходимо сделать вставку
```

`@return` *описание*

Вставляет раздел «Returns:», который содержит конкретное описание. Этот тег должен появляться в каждом комментарии к какому-либо методу, если только метод не возвращает `void` или не является конструктором. Тег не может появлять-

ся в комментариях к классам, интерфейсам или полям. Описание может быть сколь угодно длинным, но нужно стремиться к краткости. Например:

```
@return <code>true</code> если вставка прошла успешно или
        <code>false</ code> в списке уже содержится
        указанный объект.
```

`@exception` *полное-имя-класса описание*

Добавляет запись «Throws:», которая содержит конкретное имя исключения и описание. Комментарий к какому-либо методу или конструктору должен содержать тег `@exception` для каждого проверяемого исключения, которое появляется в его секции `throws`. Например:

```
@exception java.io.FileNotFoundException
           Если нельзя найти указанный файл.
```

Тег `@exception` может использоваться для документирования непроверяемых исключений (то есть подклассов `RuntimeException`), которые могут генерироваться методом. Если метод может генерировать несколько типов исключений, размещайте теги `@exception` в смежных строках и перечисляйте исключения в алфавитном порядке. Описание может быть коротким или длинным в зависимости от важности исключения. Этот тег комментария не может быть использован в комментариях к классам, интерфейсам или полям. Тег `@throws` является синонимом `@exception`.

`@throws` *полное-имя-класса описание*

Этот тег является синонимом `@exception`. Он был введен в Java 1.2.

`@see` *ссылка*

Добавляет запись «See Also:», которая содержит конкретную ссылку. Этот тег можно размещать в любом комментарии. *Ссылка* может быть трех видов. Если она начинается с символа кавычек, то предполагается, что это название книги или какого-либо другого печатного издания. Она отображается так, как в оригинале. Если *ссылка* начинается с символа «<» , то это произвольная гиперссылка HTML, которая использует тег `<A>`. Гиперссылка вставляется в исходящую документацию так, как она есть. Эта форма тега `@see` может вставлять ссылки на другие online-документы, такие как пособие программиста или руководство пользователя.

Если *ссылка* не является строкой, заключенной в кавычки, или гиперссылкой, то тег `@see` должен иметь следующую форму:

```
@see имя метка
```

В этом случае *javadoc* выводит текст, определяемый *меткой* (label), и кодирует его как гиперссылку на указанное *имя* (feature). Если *метку* пропускают (обычно так и бывает), то вместо нее *javadoc* использует указанное *имя*.

Имя может ссылаться на пакет, класс, интерфейс, метод, конструктор или поле, используя одну из следующих форм:

pkgname

Ссылка на именованный пакет. Например:

```
@see java.lang.reflect
```

pkgname.classname

Ссылка на класс или интерфейс, который указан вместе с полным именем пакета. Например:

```
@see java.util.List
```

classname

Ссылка на класс или интерфейс, который указан без имени пакета. Например:

```
@see List
```

javadoc разрешает эту ссылку путем поиска класса с таким именем в текущем пакете и списке импортированных классов.

classname#methodname

Ссылка на именованный метод или конструктор внутри конкретного класса. Например:

```
@see java.io.InputStream#reset
@see InputStream#close
```

Если класс указан без имени своего пакета, то он обрабатывается так же, как и в случае с *classname*. Этот синтаксис ведет к двусмысленности, если метод перегружен или класс определяет поле с тем же именем.

classname#methodname(paramtypes)

Ссылка на метод или конструктор с конкретным указанием на тип его параметров. Эта форма тега `@see` полезна в том случае, когда даются перекрестные ссылки на перегруженный метод. Например:

```
@see InputStream#read(byte[], int, int)
```

#methodname

Ссылка на неперегруженный метод, на конструктор в текущем классе или интерфейсе или на какой-либо из классов, родительских классов или родительских интерфейсов, содержащихся в текущем классе или интерфейсе. Используйте краткую форму для ссылки на другие методы в том же классе. Например:

```
@see #setBackgroundColor
```

#methodname(paramtypes)

Ссылка на метод или на конструктор в текущем классе или интерфейсе или на какой-либо из его родительских классов или подклассов. Эта форма работает с перегруженными методами, потому что в ней явно указываются типы параметров метода. Например:

```
@see #setPosition(int, int)
```

classname#fieldname

Ссылка на именованное поле внутри конкретного класса. Например:

```
@see java.io.BufferedInputStream#buf
```

Если класс указан без имени своего пакета, то он обрабатывается так же, как и в случае с *classname*.

#fieldname

Ссылка на поле в текущем классе или интерфейсе или на какой-либо из классов, родительских классов или родительских интерфейсов, содержащихся в текущем классе или интерфейсе. Например:

```
@see #x
```

`@deprecated` *объяснение*

В Java 1.1 данный тег указывает на то, что следующий класс, интерфейс, метод или поле устарели и применять их не следует. *javadoc* добавляет в документацию запись «Deprecated» и конкретный текст с *объяснением*. Этот текст должен сообщать о том, с какого времени класс или член больше не применяются. По возможности текст должен предлагать в качестве замены какой-либо класс или член и давать на него ссылку. Например:

```
@deprecated В версии 3.0, этот метод заменяется методом
            {@link #setColor}.
```

Хотя Java-компилятор игнорирует все комментарии, он принимает во внимание тег `@deprecated`, находящийся в комментариях к документу. Когда появляется данный тег, компилятор отмечает запрет в том файле класса, который он компилирует. Это позволяет выводить предупреждения для других классов, которые полагаются на устаревшую функциональность.

`@since` *версия*

Определяет, когда класс, интерфейс, метод или поле были добавлены к API. За этим тегом должен следовать номер или другая спецификация версии. Например:

```
@since JNUT 3.0
```

Каждый комментарий к классу или интерфейсу должен включать тег `@since`, а любые методы или поля, добавленные после первого выпуска класса или интерфейса, должны иметь теги `@since` в своих комментариях.

`@serial` *описание*

С технической точки зрения способ сериализации класса является частью открытого API. Если вы пишете класс, который будет сериализован, то вам следует задокументировать его сериализованный формат с помощью `@serial` и соответствующих тегов, которые перечислены далее. Тег `@serial` должен появляться в комментарии для любого поля, которое является частью сериализованного состояния класса `Serializable`. Применительно к классам, которые используют механизм сериализации по умолчанию, это означает все поля, которые не объявлены `transient`, включая поля, объявленные `private`. *Описание* должно представлять собой краткое описание поля и его назначение внутри сериализованного объекта.

В Java 1.4 тег `@serial` можно применять на уровне классов и пакетов, чтобы указать, надо ли генерировать «страницу сериализованной формы». Синтаксис будет таким:

```
@serial include
@serial exclude
```

`@serialField` *имя тип описание*

Класс `Serializable` может определять свой сериализованный формат путем объявления массива объектов `ObjectStreamField` в поле с названием `serialPersistentFields`. Для такого класса комментарий к `serialPersistentFields` должен включать тег `@serialField` для каждого элемента массива. Каждый тег указывает имя, тип и описание конкретного поля в сериализованном состоянии класса.

`@serialData` *описание*

Класс `Serializable` может определять метод `writeObject()` для записи данных, отличающихся от тех, которые записываются механизмом сериализации по умолчанию. Класс `Externalizable` определяет метод `writeExternal()`, который отвечает

за запись всего состояния объекта в поток сериализации. Тег `@serialData` нужно применять в комментариях к методам `writeObject()`и `writeExternal()`, а *описание* должно документировать формат сериализации, используемый методом.

`@beaninfo` *информация*

Этот нестандартный тег предоставляет информацию по компонентам JavaBeans и их методам. В настоящий момент он не используется *javadoc* (хотя этот вопрос рассматривается), но применяется в самой компании Sun. Соответствующий инструмент извлекает информацию из тегов `@beaninfo` для какого-либо класса и порождает соответствующий класс `java.beans.BeanInfo`. Этот тег появился в исходном коде компонент-классов Swing в Java 1.2. Вот как его применяют:

```
@beaninfo      bound: true
         description: фоновый цвет этого компонента JavaBeans.
```

В дополнение к предыдущим тегам инструмент *javadoc* поддерживает несколько *встроенных тегов* (*inline tags*), которые могут находиться в любом месте HTML-текста комментария. Эти теги размещаются непосредственно внутри HTML-текста, поэтому они требуют использования фигурных скобок для отделения текста с тегами от HTML-текста. Далее представлены поддерживаемые встроенные теги:

`{@link` *ссылка*`}`

В Java 1.2 и последующих версиях тег `@link` напоминает тег `@see`, за исключением того, что вместо размещения *ссылки* в специальном разделе «See Also:» он вставляет ссылку в месте своего появления. Тег `@link` может находиться в любом месте, где в комментарии разрешено использование HTML-текста. Другими словами, он может появиться в первоначальном описании класса, интерфейса, метода или поля и в описаниях, связанных с тегами `@param`, `@returns`, `@exception` и `@deprecated`. *Ссылка* для тега `@link` использует тот же синтаксис, что и в случае с тегом `@see`, который был рассмотрен ранее. Например:

```
@param regexp Регулярное выражение для поиска. Этот строковый
              аргумент должен следовать правилам синтаксиса,
              описанным для {@link RegExpParser}.
```

`{@linkplain` *ссылка*`}`

В Java 1.4 и последующих версиях тег `{@linkplain}` напоминает тег `{@link}`, за исключением того, что текст ссылки форматируется с использованием обычного шрифта, а не кодового шрифта, который задействован тегом `{@link}`. Это полезно в тех случаях, когда *ссылка* содержит как *имя* элемента (feature), на который делается ссылка, так и *метку* (label), которая задает дополнительный текст, отображаемый в ссылке. Описание *имени* и *метки*, составляющих аргумент *ссылки*, рассматривалось при обсуждении тега `@see` ранее в этом разделе.

`{@inheritDoc}`

Когда метод замещает другой метод в родительском классе или реализует метод в интерфейсе, вы можете пропустить комментарий; *javadoc* автоматически унаследует документацию от замещенного или реализованного метода. Тем не менее в Java 1.4 тег `{@inheritDoc}` позволяет наследовать только текст индивидуальных тегов. Если вы наследуете весь комментарий, то можно вставить унаследованный текст в ваш собственный текст. Для наследования индивидуальных тегов применяйте такой синтаксис:

```
@param index @{inheritDoc}
@return @{inheritDoc}
```

Для наследования всего комментария, включая собственный текст до и после комментария, используйте тег следующим образом:

```
Этот метод подменяет {@link java.langObject#toString}, который задокументирован следующим
образом:
<P>{@inheritDoc}
<P> Эта переопределенная версия метода возвращает строку формы…
```

`{@docRoot}`

Этот встроенный тег не имеет параметров и замещается ссылкой на корневой каталог сгенерированной документации. Он полезен в гиперссылках, которые ссылаются на внешний файл, например изображение или уведомление о наличии авторского права:

```
<img src="{@docroot}/images/logo.gif">Это <a href="{@docRoot}/ legal.html">защищенный
авторскими правами</a> материал.
```

`{@docRoot}` был введен в Java 1.3.

Документирующие комментарии для пакетов

Комментарии к классам, интерфейсам, методам, конструкторам и полям появляются в исходном коде Java непосредственно перед описаниями функциональных особенностей, которые они документируют.

javadoc может читать и отображать документацию, относящуюся ко всему пакету. Так как пакет определяется в каталоге, а не в отдельном файле исходного кода, то *javadoc* будет искать документацию пакета в файле с именем *package.html* в каталоге, который содержит исходный код классов пакета.

Файл *package.html* должен содержать простую HTML-документацию по пакету. Кроме того, он может содержать теги `@see`, `@link`, `@deprecated` и `@since`. Файл *package.html* не является файлом исходного кода Java, поэтому документация, которую он содержит, *не должна* представлять собой комментарий Java, то есть не должна обрамляться символами `/**` и `*/`. И наконец, любые теги `@see` и `@link`, которые появляются в *package.html*, должны использовать полностью определенные имена классов.

Помимо определения файла *package.html* для каждого пакета, вы можете предоставить высокоуровневую документацию для группы пакетов. Для этого нужно определить файл *overview.html* в дереве исходных текстов этих пакетов. Когда программа *javadoc* «передвигается» по дереву исходных текстов, она использует *overview.html* в качестве наиболее глубокого представления.

Глава 8

Средства разработки Java

В поставку Java фирмы Sun включено несколько инструментов для разработчиков. Безусловно, основные инструменты – это интерпретатор и компилятор Java, но есть и другие. В этой главе приводится описание инструментов, входящих в поставку Java 2 SDK (ранее известного как JDK). Здесь не рассмотрены средства работы с RMI и IDL, которые предназначены для корпоративного программирования. Их описание можно найти в книге «Java Enterprise in a Nutshell» (O'Reilly).

Инструменты, о которых пойдет речь, входят в пакет разработки от Sun; их следует рассматривать как особенности реализации, поскольку эти инструменты не описаны в спецификации Java. Если вы применяете другую среду разработки, обратитесь к документации по соответствующим инструментам.

В некоторых примерах, размещенных в этой главе, используются соглашения для имен файлов и путей, принятые в Unix. Если вашей платформой разработки является Windows, поменяйте символы косой черты (slash) в путях к файлам на символы обратной косой черты (backslash), а двоеточия – на точки с запятой.

appletviewer

JDK 1.0 и выше

Программа просмотра апплетов Java

Краткое описание

```
appletviewer [ ключи ] url | имя_файла...
```

Описание

Программа *appletviewer* читает или загружает HTML-документы, указанные в командной строке через имя файла или URL. Далее она загружает все апплеты, на которые ссылаются эти файлы, и запускает каждый апплет в отдельном окне. Если указанные документы не содержат апплетов, то *appletviewer* не производит никаких действий.

appletviewer распознает апплеты, указанные в теге `<APPLET>`, а в Java 1.2 и последующих версиях – в тегах `<OBJECT>` и `<EMBED>`.

Ключи командной строки

Программа *appletviewer* имеет следующие ключи командной строки:

`-debug`

Если указан этот ключ, то *appletviewer* запускается в отладчике Java (*jdb*). В этом режиме можно отлаживать апплеты, на которые ссылается документ.

`-encoding` *enc*

Этот ключ определяет кодировку, которую *appletviewer* будет использовать при чтении содержимого файла или URL. Он применяется для перекодировки значений параметров апплета в Unicode. Ключ доступен в Java 1.1 и последующих версиях.

`-J`*javaoption*

Этот ключ передает значение *javaoption* интерпретатору Java в качестве параметра командной строки. В *javaoption* не должно быть пробелов. Если передается параметр, состоящий из нескольких слов, то нужно поставить несколько ключей `-J` (допустимые ключи интерпретатора перечислены в разделе «java»). Ключ доступен в Java версии 1.1 и последующих версиях.

Также допускаются ключи `-classic`, `-native` и `-green`. Подробнее о них рассказано в разделе «java».

Команды

Каждое окно программы *appletviewer* имеет отдельное меню Applet, содержащее такие команды:

Restart

Команда останавливает и закрывает текущий апплет, заново инициализирует и запускает его.

Reload

Команда останавливает, закрывает и выгружает апплет, затем перезагружает его, повторно инициализирует и запускает.

Stop

Остановить апплет. Команда доступна в Java 1.1 и последующих версиях.

Save

Сериализовать (serialize) апплет и сохранить его в файле *Applet.ser* в домашнем каталоге пользователя. Выполнение апплета должно быть остановлено до того, как выбран этот пункт меню. Команда доступна в Java 1.1 и последующих версиях.

Start

Перезапустить остановленный апплет. Команда доступна в Java 1.1 и выше.

Clone

Создать новую копию апплета в новом окне.

Tag

Показать в диалоговом окне тег `<APPLET>`, из которого запущен текущий апплет, и все связанные с ним теги `<PARAM>`.

Info

Показать диалоговое окно с информацией об апплете. Программа получает информацию с помощью методов `getAppletInfo()` и `getParameterInfo()`, реализованных в апплете.

Edit

Эта команда не реализована. Пункт меню Edit недоступен.

Character Encoding

Показать текущую кодировку символов в строке состояния. Команда доступна в Java 1.1 и последующих версиях.

Print

Печать апплета. Команда доступна в Java 1.1 и последующих версиях.

Properties

Показать диалоговое окно, в котором пользователь может установить параметры программы *appletviewer*, включая настройки брандмауэра и кэширующих прокси-серверов (proxy servers).

Quit

Завершить работу *appletviewer*, закрыв все открытые окна.

Переменные окружения

`CLASSPATH`

В Java 1.0 и Java 1.1 *appletviewer* использует переменную среды `CLASSPATH` точно так же, как интерпретатор (см. подробности в разделе «java»). В Java 1.2 и последующих версиях *appletviewer* игнорирует эту переменную для более точной имитации веб-броузера.

Свойства

При запуске *appletviewer* читает определения свойств из файла *~/.hotjava/properties* (Unix) или *.hotjava\properties* относительно переменной среды `HOME` (Windows). Эти свойства хранятся в списке системных свойств и определяют текст различных сообщений об ошибках и о состоянии, которые отображаются программой *appletviewer*. Кроме того, в списке находятся настройки политик безопасности и использования прокси-серверов. Свойства безопасности и прокси-серверов описаны в следующих разделах. Рядовому пользователю программы *appletviewer* не нужно знать эти свойства.

Свойства безопасности

Следующие свойства определяют ограничения для ненадежных (untrusted) апплетов:

`acl.read`

Список файлов и каталогов, чтение которых разрешено ненадежным апплетам. Элементы списка должны быть разделены двоеточиями в среде Unix и точками с запятой в Windows. В Unix символ «~» заменяется на путь к домашнему каталогу текущего пользователя. Если в качестве элемента списка стоит знак «+», то он заменяется значением свойства `acl.read.default`. Это позволяет легко установить право на чтение, присвоив «+» свойству `acl.read`. По умолчанию ненадежным апплетам запрещено чтение любых файлов и каталогов.

`acl.read.default`

Список файлов и каталогов, чтение которых разрешено ненадежному апплету, если свойство `acl.read` содержит знак «+».

`acl.write`

Список файлов и каталогов, запись в которые разрешена ненадежному апплету. Элементы списка должны отделяться двоеточиями в Unix и точками с запятой в Windows. В Unix символ «~» заменяется на путь к домашнему каталогу текущего пользователя. Если в качестве элемента списка стоит знак «+», то он заменяется значением свойства `acl.write.default`. Это позволяет легко установить разрешение на запись, присвоив свойству `acl.write` значение «+». По умолчанию ненадежным апплетам запрещена запись в любые файлы и каталоги.

`acl.write.default`

Список файлов и каталогов, запись в которые разрешена ненадежному апплету, если свойство `acl.write` содержит знак «+».

`appletviewer.security.mode`

Определяет типы доступа к сети, разрешенные для ненадежных апплетов. Если свойство имеет значение «none», то апплетам запрещен доступ к сети. Значение по умолчанию – «host»; если оно задано, апплет может устанавливать соединение только с тем сервером, с которого он был загружен. Значение `unrestricted` разрешает апплетам устанавливать соединение с любым хостом без каких-либо ограничений.

`package.restrict.access.`*`префикс_пакета`*

Чтобы сделать недоступными для ненадежных апплетов классы из любого пакета, в имени которого *`префикс_пакета`* является первым компонентом, нужно присвоить этому свойству значение `true`. Например, чтобы запретить апплетам использование любых классов Sun (таких как компилятор Java и *appletviewer*), которые входят в поставку Java SDK, нужно установить следующее свойство:

```
package.restrict.access.sun=true
```

По умолчанию *appletviewer* устанавливает значение `true` для пакетов `sun.*` и `netscape.*`.

`package.restrict.definition.`*`префикс_пакета`*

Чтобы запретить ненадежным апплетам определять классы из любого пакета, в имени которого *`префикс_пакета`* является первым компонентом, нужно присвоить этому свойству значение `true`. Например, чтобы запретить апплетам определение любого класса из любого стандартного пакета, нужно установить следующее свойство:

```
package.restrict.definition.java=true
```

appletviewer присваивает этому свойству `true` по умолчанию для пакетов `java.*`, `sun.*` и `netscape.*`.

`свойство``.applet`

Если для свойства в этой форме установлено значение `true` (Java 1.1), то апплетам разрешено чтение *`свойства`* из списка системных свойств. По умолчанию апплетам разрешено чтение только 10 стандартных системных свойств (см. книгу «Java Foundation Classes in a Nutshell» O'Reilly). Например, чтобы разрешить апплетам чтение свойства `user.home`, нужно указать следующее:

```
user.home.applet=true
```

Свойства прокси-серверов

В *appletviewer* для настройки использования брандмауэра и кэширующих прокси-серверов используются следующие свойства:

`firewallHost`

Адрес брандмауэра, с которым устанавливается соединение, если `firewallSet` имеет значение `true`.

`firewallPort`

Номер порта брандмауэра, с которым устанавливается соединение, если `firewallSet` имеет значение `true`.

`firewallSet`

Определяет, должен ли апплет использовать брандмауэр. Может принимать значения `true` или `false`.

`proxyHost`

Хост кэширующего прокси-сервера, с которым устанавливается соединение, если `proxySet` имеет значение `true`.

`proxyPort`

Порт кэширующего прокси-сервера, с которым устанавливается соединение, если `proxySet` имеет значение `true`.

`proxySet`

Определяет, должен ли апплет использовать кэширующий прокси-сервер. Может принимать значение `true` или `false`.

См. также

java, javac, jdb

extcheck

Java 2 SDK 1.2 и выше

Утилита для определения конфликта версий JAR

Краткое описание

```
extcheck [-verbose] jar-файл
```

Описание

Программа *extcheck* проверяет, установлено ли расширение (extension), содержащееся в указанном *jar-файле*, или более поздняя версия этого расширения. Проверка осуществляется по атрибутам `Specification-Title` и `Specification-Version` файла манифеста (manifest) архива JAR и всех файлов JAR, содержащихся в системном каталоге расширений.

Программа предназначена для использования в сценариях автоматической установки. Если не указан ключ `-verbose`, результаты проверки не выводятся. В этом случае программа возвращает код завершения `0`, если указанное расширение может быть успешно установлено без конфликтов с уже установленными расширениями. Код завершения принимает ненулевое значение, если расширение с таким же именем уже установлено, а его номер версии больше или равен номеру версии указанного файла.

Ключи командной строки

`-verbose`

Составить список установленных расширений и вывести результаты проверки.

См. также

jar

Jar

JDK 1.1 и выше

Архиватор Java

Краткое описание

```
jar c|t|u|x[f][m][M][0][v] [jar-файл] [манифест] [-C каталог] [входные_файлы]
jar -i [jar-файл]
```

Описание

jar – это инструмент для создания архивов Java ARchive (JAR) и управления ими. Файл JAR – это ZIP-архив, содержащий файлы классов Java, различные файлы ресурсов, используемые этими классами и (необязательно) метаинформацию. Последняя включает в себя файл манифеста (manifest), в котором приводится содержание архива и дополнительная информация о каждом файле.

С помощью команды *jar* можно создавать архивы JAR, получать список файлов, содержащихся в архиве, и извлекать из него файлы. В Java 1.2 и последующих версиях с помощью этой команды можно добавлять файлы в существующий архив или обновлять файл манифеста архива. В Java 1.3 и последующих версиях программа *jar* также может добавлять в файл JAR элемент индекса.

Ключи

Синтаксис команды *jar* напоминает синтаксис команды *tar* (tape archive) в системе Unix. Большинство ключей *jar* не являются отдельными параметрами, а входят в блок связанных символов, передаваемых как один аргумент. Первая буква первого аргумента определяет, какое действие нужно выполнить. Остальные буквы необязательны. В зависимости от того, какие буквы указаны, применяются различные аргументы – имена файлов.

Командные ключи

Первая буква первого ключа определяет основное действие, которое должна выполнить программа *jar*. Возможны четыре значения:

`c`

Создать новый JAR-архив. Последним параметром командной строки должен быть список входных файлов и каталогов. В созданном архиве первым элементом является файл манифеста *META-INF/MANIFEST.MF*. В этом информационном файле, созданном автоматически, приведено содержание архива JAR. Здесь же находятся профили сообщений для каждого файла.

`t`

Получить список содержимого архива JAR.

`u`

Обновить содержимое архива JAR. Все файлы, перечисленные в командной строке, добавляются в архив. При использовании вместе с ключом `m` в файл комментария добавляется информация, указанная в командной строке. Ключ доступен в Java 1.2 и последующих версиях.

`x`

Извлечь содержимое архива. Файлы и каталоги, указанные в командной строке, извлекаются в текущий рабочий каталог. Если имена файлов и каталогов не указаны, извлекается все содержимое архива JAR.

Модифицирующие ключи

За каждым из четырех символов командных ключей может следовать дополнительный символ, дающий более подробную информацию о действии, которое нужно выполнить:

`f`

Указывает, что программа будет работать с файлом JAR, имя которого содержится в командной строке. Если этот ключ отсутствует, программа читает файл JAR

из стандартного входного потока или записывает файл архива в стандартный выходной поток. Если ключ `f` присутствует, командная строка должна содержать имя файла JAR, с которым будет выполнено действие.

`m`

Когда программа *jar* создает или обновляет файл архива, она автоматически создает или обновляет файл манифеста *META-INF/MANIFEST.MF*, содержащийся в архиве. По умолчанию манифест содержит только список файлов и каталогов архива JAR. Часто бывает необходимо, чтобы в манифесте файла JAR содержалась дополнительная информация; ключ `m` указывает на то, что в командной строке есть заготовка манифеста. Программа *jar* читает манифест и записывает его в создаваемый файл *META-INF/MANIFEST.MF*. Этот ключ следует применять только с командами `c` или `u` и нельзя применять с командами `t` или `x`.

`M`

Применяется с командами `c` и `u`. Если указан этот ключ, *jar* не создает файл манифеста.

`v`

Выводится подробная информация о процессе обработки.

`0`

Применяется с командами `c` и `u`. С этим ключом *jar* сохраняет файлы в архиве, не сжимая их. Внимание: в ключе используется цифра «нуль», а не буква O.

Файлы

Первый ключ программы *jar* состоит из первой буквы команды и стоящих за ней дополнительных символов. За первым ключом перечислены имена файлов:

jar

Если в первом ключе указана команда `f`, то за ним должно идти имя файла, над которым нужно выполнить действие.

манифест

Если в первом ключе указана команда `m`, то за ним должно идти имя файла, содержащего манифест. Если в первом ключе одновременно присутствуют буквы `f` и `m`, то имена файла JAR и файла манифеста должны идти в том же порядке, в каком идут буквы `f` и `m`. Другими словами, если `f` стоит перед `m`, то имя файла JAR должно стоять перед именем файла манифеста. Иначе, если `m` стоит перед `f`, то имя файла манифеста должно предшествовать имени файла JAR.

файлы

Перечисление файлов и каталогов, которые нужно включить в архив или извлечь из него.

Дополнительные ключи

Кроме вышеперечисленных, допускаются следующие ключи:

`-C` *каталог*

Применяется со списком файлов, над которыми необходимо выполнить действие. Определяет каталог для последующих файлов и каталогов. Последующие файлы и каталоги интерпретируются как элементы указанного *каталога* и включаются в архив JAR без префикса *каталог*. Можно использовать любое количество ключей `-C`; область действия каждого – до следующего ключа. Путь, указанный в ключе `-C`,

воспринимается относительно текущего рабочего каталога и не зависит от предыдущего ключа -C. Ключ -C доступен в Java 1.2 и последующих версиях.

-i *jar-файл*

Ключ -i применяется вместо команд c, t, u и x. Если указан этот ключ, создается индекс всех файлов JAR, перечисленных в *jar-файле*. Индекс помещается в файл *META-INF/INDEX.LIST*. Интерпретатор Java или программа просмотра апплетов могут использовать индекс для оптимизации алгоритма поиска классов и ресурсов и для предотвращения загрузки ненужных файлов JAR. Ключ доступен в Java 1.3 и последующих версиях.

Примеры

Набор ключей команды *jar* может показаться слишком запутанным, но пользоваться этой командой легко. Чтобы создать простой архив JAR, содержащий все файлы классов из текущего каталога и все файлы из подкаталога *images*, введите следующую команду:

```
% jar cf my.jar*.class images
```

Получить подробный список содержимого архива:

```
% jar tvf your.jar
```

Извлечь файл манифеста из архива JAR для просмотра или редактирования:

```
% jar xf the.jar META-INF/MANIFEST.MF
```

Обновить манифест файла JAR:

```
% jar ufm my.jar manifest.template
```

См. также

jarsigner

jarsigner

Java 2 SDK 1.2 и выше

Средство подписи и верификации файлов JAR

Краткое описание

```
jarsigner [ключи] jar-файл signer
jarsigner -verify jar-файл
```

Описание

jarsigner добавляет цифровую подпись в файл, указанный в параметре *jar-файл*. Если задан ключ -verify, *jarsigner* проверяет цифровую подпись (или подписи) файла JAR. *signer* – это псевдоним (не чувствителен к регистру) владельца цифровой подписи. Имя, указанное в параметре *signer*, используется для нахождения секретного ключа, генерирующего подпись.

Когда вы подписываете файл, вы тем самым ручаетесь за содержимое архива. Вы гарантируете, что файл JAR содержит только безвредный код, что он не нарушает законы об авторском праве и т. п. При проверке цифровой подписи вы можете установить личность подписавшегося и определить, изменялось ли содержимое файла, было ли оно повреждено с момента включения в него цифровой подписи. Не стоит путать понятия проверки цифровой подписи и доверия лицу, поставившему подпись.

Программы *jarsigner* и *keytool* заменяют *javakey* из Java 1.1.

Ключи

В *jarsigner* есть несколько ключей; многие из них определяют способ поиска секретного ключа для псевдонима *signer*. Большинство ключей не используются вместе с ключом -verify, который применяется для проверки файлов JAR:

-certs

Если этот ключ применяется вместе с -verify или -verbose, программа выводит информацию о сертификатах открытого ключа, связанных с подписанным файлом JAR.

-J*javaoption*

Передает указанный параметр *javaoption* непосредственно интерпретатору.

-keypass *пароль*

Пароль для открытого ключа владельца подписи *signer*. Если этот ключ не используется, *jarsigner* предлагает ввести пароль.

-keystore *url*

Хранилище ключей *keystore* – это файл, содержащий ключи и сертификаты. Данной командой определяется имя или URL файла ключей, в котором производится поиск сертификатов открытых и секретных ключей лица, поставившего подпись (*signer*). По умолчанию это файл *.keystore*, находящийся в домашнем каталоге пользователя, который определяется значением системного свойства user.home. Этот путь также указывает на местонахождение файла ключей, используемого программой *keytool*.

-sigfile *базовое_имя*

Определяет базовые имена файлов *.SF* и *.DSA*, добавленные в каталог *META-INF/* файла JAR. Если не указывать этот параметр, базовые имена файлов будут созданы на основе имени *signer*.

-signedjar *выходной_файл*

Определяет имя подписанного файла, который создается программой *jarsigner*. Если этот параметр не указан, программа заменяет *jar-файл*, указанный в командной строке.

-storepass *пароль*

Определяет пароль, подтверждающий целостность файла ключей, но не шифрующий секретный ключ. Если этот параметр опущен, *jarsigner* предлагает ввести пароль.

-storetype *тип*

Определяет тип файла ключей, указанного в параметре -keystore. По умолчанию используется системный тип; в большинстве систем это Java Keystore, или «JKS». Если у вас установлено расширение Java Cryptography Extension, вы можете вместо «JKS» использовать «JCEKS».

-verbose

Показывать дополнительную информацию о процессе подписи и верификации.

-verify

С этим ключом *jarsigner* проверяет подпись указанного файла JAR.

См. также

jar, keytool, javakey

java

JDK 1.0 и выше

Интерпретатор Java

Краткое описание

```
java [ ключи_интерпретатора ] имя_класса [ параметры_программы]
java [ ключи_интерпретатора ] -jar jar-файл [ параметры_программы ]
```

Описание

`java` – это интерпретатор байт-кода Java; он запускает программы, написанные на языке Java. *имя_класса* – имя класса программы, которую нужно запустить. Оно должно быть полным (содержать имя пакета класса), но без расширения *.class*. Например:

```
% java david.games.Checkers
% java Test
```

В классе должен быть определен метод `main()`, имеющий следующую сигнатуру:

```
public static void main(String[] args)
```

Этот метод служит точкой входа в программу: интерпретатор начинает выполнение именно с него.

В Java 1.2 и последующих версиях программу можно упаковать в исполняемый файл JAR. Чтобы запустить такую программу, используйте ключ `-jar` для указания файла JAR. Манифест исполняемого файла JAR должен содержать атрибут `Main-Class`, определяющий, в каком из файлов классов внутри архива находится метод `main()`, с которого интерпретатор начинает выполнение программы.

Любые параметры командной строки, предшествующие имени файла класса или файла JAR, – это ключи интерпретатора Java. Все параметры, которые идут после имени класса или архива JAR, представляют собой параметры самой программы, они игнорируются интерпретатором Java и передаются в виде массива строк методу `main()`.

Интерпретатор Java работает до завершения метода `main()`. Все потоки программы (кроме потоков, помеченных как daemon) тоже завершаются.

Версии интерпретатора

Программа *java* – это базовая версия интерпретатора. Кроме нее есть несколько других версий интерпретатора Java. Каждая из них похожа на *java*, но обладает специальными возможностями. Существуют следующие разновидности интерпретатора:

java

Это базовая версия интерпретатора Java; обычно нужно использовать именно ее. В Java 1.2 набор поддерживаемых ключей существенно изменился по сравнению с Java 1.1; изменилось и действие ключей. В остальных выпусках были проведены незначительные изменения.

oldjava

Эта версия интерпретатора включена в пакет Java 1.2 и Java 1.3 для совместимости с Java 1.1. Она загружает классы по схеме, применяемой в Java 1.1. Программ, использующих эту версию интерпретатора, очень мало, поэтому она не вошла в состав Java 1.4.

javaw

Эта версия интерпретатора включена только в платформы для Windows. Используйте *javaw*, если нужно, чтобы при запуске программы Java (например, из сце-

нария) не появлялось окно консоли. В Java 1.2 и Java 1.3 включена программа *oldjavaw*, совмещающая в себе возможности *oldjava* и *javaw*.

java_g

В Java 1.0 и Java 1.1 программа *java_g* – это отладочная версия интерпретатора Java. В ней доступно несколько специальных ключей командной строки, но эта версия применяется редко. В платформах для Windows есть программа *javaw_g*. Инструмент *java_g* не входит в Java 1.2 и последующие версии.

Серверная и клиентская ВМ

Виртуальная машина «HotSpot» фирмы Sun имеет две разновидности: первая оптимизирована для использования с клиентскими приложениями, имеющими малое время жизни, а вторая предназначена для серверных программ, имеющих большое время жизни. В Java 1.4 с помощью ключа `-server` можно запустить серверную виртуальную машину. Клиентскую машину можно выбрать с помощью ключа `-client`, являющегося ключом по умолчанию.

Классическая ВМ

В Java 1.3 можно задать ключ `-classic`, чтобы выбрать «классическую ВМ» (по существу, это то же самое, что и виртуальная машина Java 1.2).

Динамический компилятор

В Java 1.2 и Java 1.3 при указании ключа `-classic` интерпретатор Java использует динамический компилятор (just-in-time compiler, JIT), если он доступен на вашей платформе. JIT преобразует байт-код Java в машинные инструкции на этапе исполнения. Это значительно увеличивает скорость работы большинства Java-программ. Если вы не хотите использовать JIT, вы можете отключить его, присвоив переменной среды `JAVA_COMPILER` значение «NONE» или установив для системного свойства `java.compiler` это же значение с помощью ключа `-D`:

```
% setenv JAVA_COMPILER NONE                  // Синтаксис Unix-оболочки csh
% java -Djava.compiler=NONE MyProgram
```

Если требуется использовать другую реализацию компилятора JIT, присвойте переменной среды или системному свойству имя требуемого компилятора. Эта переменная не используется в Java 1.4 с виртуальной машиной HotSpot, применяющей более эффективную технологию JIT.

Потоковые системы

В ОС Solaris и других Unix-платформах можно выбрать тип потоков для интерпретатора Java 1.2 или «классической ВМ» Java 1.3. Чтобы использовать тип потоков, применяемый данной ОС, в командной строке укажите ключ `-native`. Для использования виртуальных, или «зеленых», потоков (по умолчанию) укажите `-green`. В Java 1.3 клиентская машина по умолчанию применяет родные потоки ОС. При указании ключей `-green` или `-native` в Java 1.3 автоматически добавляется ключ `-classic`. В Java 1.4 эти ключи больше не поддерживаются за ненадобностью.

Ключи командной строки

`-classic`

В Java 1.3 этот ключ запускает «классическую ВМ» вместо высокопроизводительной клиентской машины, используемой по умолчанию.

`-classpath` *путь*

Определяет каталоги, файлы JAR или ZIP, которые просматривает программа *java* при попытке загрузить класс. В Java 1.0 и Java 1.1, а также при использова-

нии *oldjava*, этот ключ определяет расположение системных классов, классов расширений и классов приложений. В Java 1.2 и последующих версиях этот ключ определяет только расположение классов приложений.

`-client`

Оптимизировать инкрементную компиляцию BM HotSpot для типичных клиентских приложений. Доступен в Java 1.4 и последующих версиях. См. также ключ `-server`.

`-cp`

Синоним `-classpath`. Доступен в Java 1.2 и последующих версиях.

`-cs, -checksource`

С этими ключами *java* проверяет время изменения указанного файла класса и соответствующего файла с исходным кодом. Если файл класса не найден или устарел, исходный код автоматически перекомпилируется. Ключ доступен только в Java 1.0 и Java 1.1, а в Java 1.2 и последующих версиях он не поддерживается.

`-D`*имя_свойства=значение*

Присвоить *значение* указанному системному *свойству*. Java-программа может осуществлять поиск этого значения по имени свойства. Можно указать любое количество ключей `-D`, например:

```
% java -Dawt.button.color=gray -Dmy.class.pointsize=14 my.class
```

`-d32`

Запуск в 32-разрядном режиме. Ключ допустим в Java 1.4 и последующих версиях, но в настоящее время реализован только для ОС Solaris.

`-d64`

Запуск в 64-разрядном режиме. Ключ допустим в Java 1.4 и последующих версиях, но в настоящее время реализован только для ОС Solaris.

`-da[:`*where*`]`

Отменить утверждения (assertions). См. `-disableassertions`. Доступен в Java 1.4 и последующих версиях.

`-debug`

Запустить программу *java* таким образом, чтобы отладчик *jdb* мог подключиться к сессии интерпретатора. В Java 1.2 и последующих версиях этот ключ заменен ключом `-Xdebug`.

`-disableassertions[:`*where*`]`

Отменить утверждения (assertions). Ключ впервые появился в Java 1.4. Допускается сокращенный вариант `-da`. Если указан только сам ключ (по умолчанию), то отменяются все утверждения (кроме тех, которые находятся в системных классах). Чтобы отменить утверждения только из одного класса, нужно сразу за ключом поставить двоеточие и указать полное имя класса. Чтобы отменить утверждения во всем пакете (и всех входящих в него пакетах), после ключа нужно поставить двоеточие, имя пакета и три точки. См. также `-enableassertions` и `-disablesystemassertions`.

`-disablesystemassertions`

Отменить утверждения во всех системных классах (по умолчанию). Ключ впервые появился в Java 1.4. Допускается сокращенная запись `-dsa`. Этот ключ не принимает никаких параметров.

`-dsa`

Сокращенная запись ключа `-disablesystemassertions`. Ключ доступен в Java 1.4 и последующих версиях.

`-ea[:where]`

Использовать утверждения. Это сокращенная запись ключа `-enableassertions`. Доступен в Java 1.4 и последующих версиях.

`-enableassertions[:where]`

Использовать утверждения. Ключ впервые появился в Java 1.4. Допускается сокращенная форма `-ea`. Если ключ указан без параметров, будут использоваться все утверждения (кроме тех, которые находятся в системных классах). Чтобы разрешить утверждения только из одного класса, поставьте после ключа двоеточие и полное имя класса. Чтобы использовать утверждения из всего пакета (и всех входящих в него пакетов), после ключа нужно поставить двоеточие, имя пакета и три точки. См. также `-disableassertions` и `-enablesystemassertions`.

`-enablesystemassertions`

Использовать утверждения из всех системных классов. Допускается сокращенный вариант `-esa`. Ключ доступен в Java 1.4 и последующих версиях.

`-esa`

Сокращенный вариант ключа `-enablesystemassertions`. Доступен в Java 1.4 и последующих версиях.

`-green`

В операционных системах, поддерживающих несколько типов потоков (например, Linux и Solaris), этим ключом выбирается виртуальный, или «зеленый», тип потоков. Данный ключ применяется по умолчанию в Java 1.2. В Java 1.3 при выборе этого ключа автоматически выбирается ключ `-classic`. См. также `-native`. Ключ доступен только в Java 1.2 и Java 1.3.

`-help, -?`

Выводится справочная информация, и программа завершается. См. также `-X`.

`-jar` *jar-файл*

Запустить исполняемый *jar-файл*. Манифест этого файла должен содержать атрибут `Main-Class`. Этот атрибут указывает, в каком классе находится `main()` — метод, с которого начинается выполнение программы. Ключ доступен в Java 1.2 и последующих версиях.

`-native`

В операционных системах, поддерживающих разные типы потоков (например, Solaris и Linux), этим ключом выбирается тип потоков, используемый данной ОС. По умолчанию применяется другой, «зеленый» тип потоков. В некоторых случаях, например при запуске программы на многопроцессорном компьютере, лучше использовать основной тип потоков. В Java 1.3 виртуальная машина HotSpot применяет основные потоки. Ключ доступен только в Java 1.2 и 1.3.

`-showversion`

Этот ключ выполняет те же действия, что и `-version`. Отличие состоит в том, что с этим ключом интерпретатор продолжает работу после отображения информации о версии. Доступен в Java 1.3 и последующих версиях.

`-verbose`, `-verbose:class`

Выводить сообщение при каждой загрузке класса. В Java 1.2 и последующих версиях ключ `-verbose:class` можно использовать как синоним.

`-verbose:gc`

Выводить сообщение при каждой сборке мусора. Доступен в Java 1.2 и последующих версиях. В версиях, предшествующих 1.2, используйте ключ `-verbosegc`.

`-verbose:jni`

Выводить сообщение при каждом вызове методов Java, зависимых от платформы (native). Ключ доступен в Java 1.2 и последующих версиях.

`-version`

Программа *java* выводит версию интерпретатора и завершает работу.

`-X`

Выводится справочная информация для нестандартных ключей командной строки интерпретатора (тех, которые начинаются с `-X`), и программа завершает работу. См. также `-help`. Ключ доступен в Java 1.2 и последующих версиях.

`-Xbatch`

Предписать виртуальной машине HotSpot выполнить динамическую компиляцию (just-in-time compilation) с высоким приоритетом, независимо от времени, необходимого для компиляции. Без этого ключа виртуальная машина выполняет низкоприоритетную компиляцию методов во время их интерпретации с высоким приоритетом. Доступен в Java 1.3 и последующих версиях.

`-Xbootclasspath:`*путь*

Определяет путь поиска системных классов, состоящий из имен каталогов, файлов ZIP и JAR. Этот ключ используется очень редко. Доступен в Java 1.2 и последующих версиях.

`-Xbootclasspath/a:`*путь*

Добавляет указанный *путь* в конец списка путей к системным классам. Доступен в Java 1.3 и последующих версиях.

`-Xbootclasspath/p:`*путь*

Добавляет указанный *путь* в начало списка путей к системным классам. Доступен в Java 1.3 и последующих версиях.

`-Xcheck:jni`

Выполняется дополнительная проверка при каждом использовании функций из Java Native Interface. Ключ доступен в Java 1.2 и последующих версиях.

`-Xdebug`

Запустить интерпретатор так, чтобы с ним мог общаться отладчик. Ключ доступен в Java 1.2 и последующих версиях. В версиях, предшествующих 1.2, используйте ключ `-debug`.

`-Xfuture`

Выполнить строгую проверку формата всех загруженных файлов классов. Без этого ключа *java* производит такую же проверку, как в Java 1.1. Ключ доступен в Java 1.2 и последующих версиях.

`-Xincgc`

Использовать инкрементную сборку мусора. В этом режиме сборщик мусора постоянно работает в фоновом режиме, а выполняемая программа очень редко при-

останавливается для сборки мусора. Тем не менее использование этого ключа снижает общую производительность на 10%. Ключ доступен в Java 1.3 и последующих версиях.

`-Xint`

С этим ключом ВМ HotSpot работает только в режиме интерпретатора, без выполнения динамической компиляции. Доступен в Java 1.3 и последующих версиях.

`-Xloggc:`*`имя_файла`*

Вести журнал регистрации событий, связанных со сборкой мусора, в указанном файле.

`-Xmixed`

С этим ключом ВМ HotSpot выполняет динамическую компиляцию наиболее часто используемых («горячих») методов, а остальные методы выполняет в режиме интерпретатора. Такой режим работы установлен по умолчанию. Сравните ключи `-Xbatch` и `-Xint`. Ключ доступен в Java 1.3 и последующих версиях.

`-Xms` *`initmem`*`[k|m]`

Определяет, сколько памяти будет выделено под «кучу» при старте интерпретатора. По умолчанию объем памяти (*`initmem`*) указывается в байтах. Можно указать размер в килобайтах или мегабайтах, поставив `k` или `m` соответственно.

По умолчанию размер «кучи» равен 1 Мбайт. Производительность больших программ или программ, интенсивно использующих память (таких как компилятор Java), можно увеличить, выделив больше памяти при запуске интерпретатора. Минимальный начальный размер «кучи» – 1000 байт. Ключ доступен в Java 1.2 и последующих версиях. В версиях, предшествующих 1.2, применяйте ключ `-ms`.

`-Xmx` *`maxmem`*`[k|m]`

Определяет максимальный размер «кучи», которую интерпретатор использует для динамически создаваемых объектов и массивов. По умолчанию размер указывается в байтах, но его можно указать в килобайтах или мегабайтах, поставив `k` или `m` соответственно. По умолчанию максимальный размер «кучи» равен 16 Мбайт; и он не может быть меньше 1000 байт. Ключ доступен в Java 1.2 и последующих версиях. В версиях, предшествующих 1.2, применяйте ключ `-mx`.

`-Xnoclassgc`

Не собирать мусор классов. Ключ доступен в Java 1.2 и последующих версиях. В Java 1.1 применяйте ключ `-noclassgc`.

`-Xprof`

Выводит в стандартный выходной поток информацию о выполнении кода. Ключ доступен в Java 1.3 и последующих версиях. При использовании ключа `-classic`, а также в Java 1.2 применяйте `-Xrunhprof`. В версиях Java, предшествующих 1.2, указывайте ключ `-prof`.

`-Xrs`

Использовать меньше сигналов операционной системы. При этом на некоторых системах может улучшиться производительность. Ключ доступен в Java 1.2 и последующих версиях.

`-Xrunhprof:`*`suboptions`*

Включить профилирование процессора, «кучи», монитора. *`suboptions`* – это список элементов вида `имя=значение`, разделенных запятыми. Для получения списка поддерживаемых параметров и значений задавайте ключ `-Xrunhprof:help`. Ключ под-

держивается в Java 1.2 и последующих версиях. В версиях, предшествующих 1.2, профилирование поддерживается частично. Оно включается с помощью ключа `-prof`. В Java 1.3 данный ключ работает только в классическом режиме (ключ `-classic`) и не поддерживается новой ВМ HotSpot. См. `-Xprof`.

`-Xss`*`size`*`[k|m]`

Определяет размер стека потока в байтах, килобайтах или мегабайтах. Ключ доступен в Java 1.3 и последующих версиях.

Загрузка классов

Интерпретатору Java известно, где нужно искать системные классы, входящие в состав платформы Java. В Java 1.2 и последующих версиях ему также известно, где искать файлы классов расширений, установленных в каталог системных расширений. Но интерпретатору требуется указать, где искать несистемные классы запускаемого приложения.

Файлы классов хранятся в каталогах с именами, соответствующими именам их пакетов. Например, класс `com.davidflanagan.utils.Util` хранится в файле *com/davidflanagan/utils/Util.class*. По умолчанию в качестве корневого каталога интерпретатор использует текущий рабочий каталог и ищет в нем и в его подкаталогах все классы.

Интерпретатор может осуществлять поиск классов внутри архивов ZIP и JAR. Чтобы сообщить интерпретатору, где нужно искать классы, укажите в параметре *classpath* список каталогов и архивов ZIP и JAR. При поиске класса интерпретатор просматривает все пути в том порядке, в котором они указаны.

Самый простой способ определить путь к классам – создать переменную среды `CLASSPATH`, аналогичную переменной `PATH` в командной оболочке Unix или в командном интерпретаторе Windows. В Unix введите такую команду:

```
% setenv CLASSPATH .:~/myclasses:/usr/lib/javautils.jar:/usr/lib/javaapps
```

В Windows используйте следующую команду:

```
C:\> set CLASSPATH=.;c:\myclasses;c:\javatools\classes.zip;d:\javaapps
```

Обратите внимание на то, что в Unix и Windows при указании пути используются различные символы-разделители.

Путь к классу можно определить при помощи ключей командной строки `-classpath` или `-cp` интерпретатора Java. Путь, указанный в одном из этих ключей, замещает любой путь, определенный в переменной среды `CLASSPATH`. В Java 1.2 и последующих версиях ключ `-classpath` определяет только путь поиска классов приложения и пользовательских классов. В версиях Java, предшествующих 1.2, или при использовании интерпретатора *oldjava* этот ключ определяет путь поиска для всех классов, включая системные классы и классы расширений.

См.также

javac, jdb

javac

JDK 1.0 и выше

Компилятор Java

Краткое описание

```
javac [ ключи ] файлы
oldjavac [ ключи ] файлы
```

Описание

Программа *javac* – это компилятор Java. Он компилирует исходный код Java, находящийся в файлах *.java*, в байт-код (файлы *.class*). Сам компилятор написан на Java. В Java 1.3 компилятор полностью переписан, а его производительность существенно повысилась. Хотя новый *javac* в основном совместим с предыдущими версиями, старый компилятор также включен в поставку под именем *oldjavac* (только в Java 1.3).

Программе *javac* можно передать любое количество исходных файлов Java, имена которых должны иметь расширение *.java*. Программа *javac* создает отдельный файл с расширением *.class* для каждого класса, определенного в исходных файлах. В каждом исходном файле может содержаться любое количество классов, но только один из классов верхнего уровня может иметь модификатор `public`. Название исходного файла (без расширения *.java*) должно соответствовать имени класса `public`, содержащегося в этом файле.

Если в Java 1.2 и последующих версиях имя файла, указанное в командной строке, начинается с @, то этот файл интерпретируется не как файл с исходным кодом, а как список ключей компилятора и исходных файлов. Таким образом, если записать имена исходных файлов из одного проекта в файл с именем *project.list*, можно скомпилировать сразу все файлы одной командой:

```
% javac @project.list
```

Чтобы компилятор *javac* смог скомпилировать исходный файл, для него должны быть доступны объявления всех классов из исходного файла. Компилятор ищет объявления как в исходных файлах, так и в файлах классов. При этом он автоматически компилирует исходные файлы, не имеющие соответствующих файлов классов, а также файлы, которые были изменены с момента последней компиляции.

Ключи командной строки

`-bootclasspath` *путь*

Определяет *путь*, который используется компилятором для поиска системных классов. Этот ключ удобно применять, когда *javac* используется как кросс-компилятор для компиляции классов, написанных под разные версии Java API. Например, компилятор Java 1.3 можно применять для компиляции классов в среде Java 1.2. Этот ключ не может определять системные классы, которые используются для запуска самого компилятора; можно задать только системные классы, которые доступны для чтения компилятору. См. также `-extdirs` и `-target`. Ключ доступен в Java 1.2 и последующих версиях.

`-classpath` *путь*

Определяет *путь*, который используется компилятором для поиска классов, на которые ссылается исходный код. Этот ключ замещает любой путь из переменной среды `CLASSPATH`. *путь* – это упорядоченный список каталогов, архивов ZIP и JAR, разделенных двоеточиями в Unix и точками с запятой в Windows. Если не указан ключ `-sourcepath`, этот ключ также устанавливает путь поиска исходных файлов.

В версиях Java, предшествующих 1.2, этот ключ определяет путь к системным классам и классам расширений, а также к классам приложений и пользовательским классам, поэтому он требует аккуратного использования. В Java 1.2 и последующих версиях этот ключ устанавливает путь поиска только для классов приложений. См. раздел «Загрузка классов» в описании команды *java*.

`-d` *каталог*

Устанавливает каталог, в котором (или в подкаталогах которого) нужно сохранить файлы классов. По умолчанию *javac* сохраняет полученные файлы *.class* в том же каталоге, где находятся файлы *.java* с объявлениями этих классов. Если задан ключ `-d`, *каталог* рассматривается как корневой в иерархии классов, а файлы *.class* помещаются в него или в его подкаталоги в зависимости от названия пакета класса. Таким образом, команда

```
% javac -d /java/classes Checkers.java
```

помещает файл *Checkers.class* в каталог */java/classes*, если в файле *Checkers.java* нет оператора `package`. Если же в исходном файле обозначена принадлежность к пакету:

```
package com.davidflanagan.games;
```

то файл *.class* сохраняется в каталоге */java/classes/com/davidflanagan/games*. Когда применяется ключ `-d`, компилятор автоматически создает по соответствующему пути все каталоги, в которые будут помещены файлы классов.

`-depend`

Включить рекурсивный поиск устаревших файлов классов, которые необходимо перекомпилировать. С этим ключом производится полная компиляция, которая может сильно замедлить процесс. В Java 1.2 и последующих версиях этот ключ переименован в `-Xdepend`.

`-deprecation`

С этим ключом при каждом нерекомендуемом (deprecated) использовании API программа *javac* выводит предупреждение. По умолчанию *javac* вырабатывает одно предупреждение о неверном использовании API на каждый исходный файл. Ключ доступен в Java 1.1 и последующих версиях.

`-encoding` *кодировка*

Если кодировка символов исходного файла отличается от системной, то она должна быть указана в этом ключе.

`-extdirs` *путь*

Определяет список каталогов, в которых будет производиться поиск JAR-файлов расширений. Этот ключ используется вместе с `-bootclasspath` при кросс-компилировании для различных версий среды выполнения Java. Доступен в Java 1.2 и последующих версиях.

`-g`

Добавить в выходной файл класса номера строк, исходный код и информацию о локальных переменных, необходимые отладчикам. По умолчанию *javac* добавляет только номера строк.

`-g:none`

Не включать отладочную информацию в выходные файлы. Доступен в Java 1.2 и последующих версиях.

`-g:`*список_ключевых_слов*

Определяет вид отладочной информации. Типы отладочной информации указываются в виде *списка_ключевых_слов* и разделяются запятыми. Допустимые ключевые слова: `source` (ннформация об исходном коде), `lines` (номера строк), `vars` (информация о локальных переменных). Ключ доступен в Java 1.2 и последующих версиях.

`-help`

Вывести список ключей.

`-J`*`javaoption`*

Передать аргумент *`javaoption`* непосредственно интерпретатору Java. Например: `-J-Xmx32m`. В параметре *`javaoption`* не должно быть пробелов; если нужно передать интерпретатору несколько аргументов, применяйте несколько ключей `-J`. Ключ доступен в Java 1.1 и последующих версиях.

`-nowarn`

Не показывать предупреждений. При использовании этого ключа сообщения об ошибках все равно выводятся.

`-nowrite`

Не создавать файлы классов. Исходный код анализируется компилятором, но запись выходных файлов не производится. Применяйте этот ключ, чтобы проверить, как будет компилироваться файл, не производя при этом компиляции. Ключ доступен только в Java 1.0 и Java 1.1. В Java 1.2 и последующих версиях он отсутствует.

`-O`

Разрешить оптимизацию файла класса для увеличения скорости его исполнения. При использовании этого ключа файлы классов имеют больший размер и дольше компилируются. Кроме того, их сложно отлаживать. Во всех версиях Java, предшествующих 1.2, этот ключ не совместим с `-g`; при выборе `-O` автоматически выключается `-g` и включается `-depend`.

`-source` *`номер_релиза`*

Определяет версию Java, в которой написан код. Для компиляции кода, в котором присутствует оператор `assert`, применяйте ключ `-source 1.4`. Этот ключ также устанавливает опцию `-target`. Ключ доступен в Java 1.4 и последующих версиях.

`-sourcepath` *`путь`*

Определяет список каталогов, архивов ZIP и JAR, в которых производится поиск исходных файлов. Каждый найденный файл компилируется, если для него нет соответствующего файла класса или исходный файл новее файла класса. По умолчанию путь поиска исходных файлов совпадает с путем поиска файлов класса. Ключ доступен в Java 1.2 и последующих версиях.

`-target` *`версия`*

Определяет формат файла класса, который будет использоваться при создании выходных файлов. *`версия`* по умолчанию – 1.1, а созданные файлы классов могут быть прочитаны и выполнены виртуальной машиной Java 1.0 и последующими версиями ВМ. Если указать версию 1.2, *javac* увеличивает номер версии файла класса, и такой файл не может исполняться интерпретаторами Java 1.0 и Java 1.1. Форматы файлов классов разных версий практически не отличаются; указание версии в файле – это удобный способ не допустить запуск класса, использующего новые возможности Java 1.2, в старых версиях интерпретатора.

`-verbose`

Показывать сообщение о каждом действии. В частности, при этом выводятся имена всех компилируемых исходных файлов, в том числе и тех, которые не были указаны в командной строке.

`-X`

Вывести справочную информацию о нестандартных ключах (они начинаются с `-X`). Этот ключ доступен в Java 1.2, Java 1.4 и последующих версиях.

`-Xdepend`

Включить рекурсивный поиск файлов, которые необходимо перекомпилировать. С этим ключом производится полная компиляция, которая может сильно замедлить процесс. Ключ доступен только в Java 1.2; в Java 1.3 его нет.

`-Xstdout`

При использовании этого ключа сообщения о предупреждениях и об ошибках выводятся в стандартный выходной поток, а не в поток сообщений об ошибках. Ключ доступен только в Java 1.2.

`-Xstdout` *имя_файла*

Записывать сообщения о предупреждениях и об ошибках в указанный файл, а не выводить их в окно консоли. Ключ доступен в Java 1.4 и последующих версиях.

`-Xswitchcheck`

Выводить предупреждения о секциях `case` операторов `switch`, не имеющих завершающего оператора `break`.

`-Xverbosepath`

Выводить информацию о расположении найденных файлов классов и исходных файлов. Ключ доступен только в Java 1.2.

Переменные окружения

`CLASSPATH`

Определяет упорядоченный список (с разделителем «:» в Unix или «;» в Windows) каталогов, файлов ZIP и JAR, в которых будет производиться поиск классов пользователя и исходных файлов. Если указан ключ `-classpath`, то он замещает эту переменную.

См. также

java, jdb

javadoc JDK 1.0 и выше

Программа для создания документации Java

Краткое описание

```
javadoc [ ключи ] @list пакет... исходные_файлы...
```

Описание

Программа *javadoc* генерирует документацию API в формате HTML (по умолчанию) для указанных пакетов и классов. В командной строке может быть указано любое количество имен пакетов и исходных файлов Java. Для удобства при работе с большим количеством ключей командной строки или имен пакетов и классов можно поместить их в произвольный файл и указать в командной строке символ «@», а сразу за ним – имя этого файла.

Программа *javadoc* использует компилятор *javac* при работе с указанными исходными файлами и файлами из указанных пакетов. На основе полученной от компилятора информации она создает подробную документацию API. При этом используются

все комментарии из исходных файлов. О том, как писать комментарии в Java-коде, рассказано в главе 7.

Кроме имени входного файла, необходимо указать полный путь к нему. Программу *javadoc* можно применять для создания документации ко всему пакету. Для того чтобы программа *javadoc* могла правильно найти исходный код пакета, путь к которому не входит в список путей к классам, нужно указать ключ `-sourcepath`.

По умолчанию программа *javadoc* создает документы в формате HTML, но можно определить класс doclet, генерирующий документацию в нужном формате. Вы можете написать собственный класс doclet с помощью doclet API из пакета `com.sun.javadoc`. Этот пакет описан в стандартной документации к Java 1.2 и последующим версиям.

В Java 1.2 программа *javadoc* полностью обновлена. Здесь приведена документация к этой программе из Java 1.2 и последующих версий, а отличия от предыдущих версий не рассмотрены.

Ключи командной строки

Команда *javadoc* допускает множество различных ключей. Часть из них представляет собой стандартные ключи, всегда распознаваемые программой. Другие ключи определены в классе doclet, создающем документацию. Ниже приведены ключи, допустимые при использовании стандартного doclet в формате HTML.

`-1.1`

Использовать стиль выходного файла и структуру каталогов, принятые в *javadoc* из Java 1.1. Этот ключ доступен только в Java 1.2 и Java 1.3 и удален в Java 1.4.

`-author`

Включить в выходной документ информацию об авторстве, определенную параметром `@author`. Ключ доступен только в doclet, установленном по умолчанию.

`-bootclasspath`

Определяет расположение дополнительных системных классов. Этот ключ можно использовать при кросс-компилировании. Подробное описание представлено в разделе «javac».

`-bottom` *текст*

Поместить *текст* в конец каждого созданного документа. *текст* может содержать теги HTML. См. также `-footer`. Ключ доступен только в doclet, установленном по умолчанию.

`-breakiterator`

Использовать алгоритм `java.text.BreakIterator` для определения конца сводного предложения (summary sentence) в комментариях. Ключ доступен только в doclet, установленном по умолчанию.

`-charset` *кодировка*

Установить набор символов выходного файла. Кодировка выходного файла зависит от кодировки комментариев из исходного файла. Значение параметра *кодировка* используется в теге `<META>` выходного файла HTML. Ключ доступен только в doclet, установленном по умолчанию.

`-classpath` *путь*

Определяет путь, используемый программой *javadoc* при поиске файлов классов. Если указан ключ `-sourcepath`, путь также применяется при поиске файлов с исходным кодом. Поскольку программа *javadoc* использует компилятор *javac*, ей требуется путь к файлам всех классов, на которые ссылается документируемый

пакет. Ключ `-sourcepath` и переменная среды `CLASSPATH` описаны в разделах «java» и «javac».

`-d` *каталог*

Установить каталог, в который будут помещены выходные документы HTML. Если этот ключ не указан, используется текущий каталог. Ключ доступен только в doclet, установленном по умолчанию.

`-docencoding` *кодировка*

Определяет кодировку выходных документов HTML. Имя кодировки может не соответствовать имени набора символов, указанного в ключе `-charset`. Ключ доступен только в doclet, установленном по умолчанию.

`-docfilessubdirs`

С этим ключом программа проводит рекурсивное копирование всех подкаталогов каталога *doc-files*, а не просто копирование всех файлов, содержащихся в каталоге *doc-files*. Ключ доступен только в doclet, установленном по умолчанию.

`-doclet` *имя_класса*

Определяет имя класса doclet, который будет использоваться при создании документов. Если этот ключ не указан, программа *javadoc* генерирует документы с помощью doclet в формате HTML, установленного по умолчанию.

`-docletpath` *путь*

Этот ключ устанавливает путь, по которому находится класс, указанный в ключе `-doclet`, если этот путь не указан в списке путей к классам.

`-doctitle` *текст*

Ключ определяет заголовок обзорного файла документации. Этот файл обычно видят читатели во время просмотра документации. Заголовок может содержать теги HTML. Ключ доступен только в doclet, установленном по умолчанию.

`-encoding` *название_кодировки*

Указать кодировку символов в комментариях из входных файлов. Она может отличаться от кодировки выходных файлов, установленной ключом `-docencoding`. По умолчанию используется кодировка, являющаяся основной в данной ОС.

`-exclude` *пакеты*

Исключить *пакеты* из набора пакетов, определенного ключом `-subpackages`. Параметр *пакеты* представляет собой список имен пакетов, разделенных двоеточиями. Ключ доступен только в doclet, установленном по умолчанию.

`-excludedocfilessubdir` *каталоги*

Эта опция исключает указанные подкаталоги каталога *doc-files*, если задан ключ `-docfilessubdirs`. Например, его можно применять для исключения каталогов управления версиями. Параметр *каталоги* представляет собой список имен каталогов, разделенных двоеточиями и являющихся подкаталогами каталога *doc-files*. Ключ доступен только в doclet, установленном по умолчанию.

`-extdirs` *список_каталогов*

Установить список каталогов стандартных расширений. Этот ключ бывает необходим только при кросс-компилировании вместе с ключом `-bootclasspath`. Подробная информация представлена в разделе «javac».

`-footer` *текст*

Этот ключ определяет текст, отображаемый в конце каждого файла, справа от навигационной панели. Параметр *текст* может содержать теги HTML. См. также `-bottom` и `-header`. Ключ доступен только в doclet, установленном по умолчанию.

`-group` *заголовок список_пакетов*

С этим ключом программа *javadoc* создает обзорную страницу верхнего уровня, в которой перечислены все пакеты, описанные в создаваемой документации. По умолчанию создается отдельная таблица, составленная из имен этих пакетов в алфавитном порядке. С помощью этого ключа можно разбить их на группы связанных пакетов. *заголовок* определяет имя группы пакетов, например «базовые пакеты». Параметр *список_пакетов* — это список имен пакетов, разделенных двоеточием. Имена пакетов могут заканчиваться групповым символом «*». Командная строка программы *javadoc* может содержать любое количество ключей `-group`, например:

```
javadoc -group "AWT Packages" java.awt*
     -group "Swing Packages" javax.accessibility:javax.swing*
```

`-header` *текст*

Определить заголовки, отображаемые в верхней части каждого файла, справа от верхней навигационной панели. *текст* может содержать теги HTML. См. также `-footer`, `-doctitle` и `-windowtitle`. Ключ доступен только в doclet, установленном по умолчанию.

`-help`

Вывести справочную информацию о программе *javadoc*.

`-helpfile` *файл*

Определить файл HTML, содержащий справку по использованию документации. Программа *javadoc* помещает ссылки на него во все созданные файлы. Если этот ключ не указан, создается файл справки по умолчанию. Ключ доступен только в doclet, установленном по умолчанию.

`-J`*javaoption*

Передать *javaoption* непосредственно интерпретатору Java. При обработке большого количества пакетов для увеличения объема выделяемой для программы *javadoc* памяти может потребоваться использование этого ключа. Например:

```
% javadoc -J-Xmx64m
```

В связи с тем что ключи `-J` передаются непосредственно интерпретатору Java до старта программы *javadoc*, их нельзя помещать во внешний файл, указанный в командной сторке в виде `@list`.

`-link` *url*

Этот ключ определяет абсолютную или относительную ссылку-URL на каталог верхнего уровня другого документа, созданного с помощью *javadoc*. Указанная ссылка является базовой для всех ссылок текущего документа на пакеты, классы, методы и поля, которые не описаны в этом документе. Например, при создании документации к вашим собственным пакетам вы можете с помощью этого ключа связать ее с документацией базовых Java API. Ключ доступен только в doclet, установленном по умолчанию.

Каталог, указанный в параметре *url*, должен содержать файл с именем *package-list*, который должен быть доступен для чтения во время выполнения *javadoc*.

Этот файл был создан командой *javadoc*. Он содержит список пакетов, описанных в документах, на которые ссылается *url*.

Можно использовать сразу несколько ключей -link, но в ранних выпусках Java 1.2 это не будет работать. Если ключ -link не указан, гиперссылки на классы, не описанные в текущей документации, не создаются.

-linkoffline *url packagelist*

Производит те же действия, что и ключ -link, за исключением того, что *package-list* указывается в командной строке. Ключ -linkoffline нужно использовать, если каталог, на который ссылается *url*, не содержит файл *package-list* или если этот файл недоступен во время исполнения программы *javadoc*. Ключ доступен только в doclet, установленном по умолчанию.

-linksource

При использовании этого ключа создается HTML-версия каждого обрабатываемого исходного файла, а во все страницы добавляются ссылки на эту версию. Ключ доступен только в doclet, установленном по умолчанию.

-locale *language_country_variant*

Определяет региональные параметры (locale), используемые в документации. *language_country_variant* используется при поиске файла ресурсов, содержащего перевод сообщений и текста для выходных файлов.

-nocomment

С этим ключом все комментарии игнорируются, а создаваемая документация содержит в себе только API без сопровождающего текста. Ключ доступен только в doclet, установленном по умолчанию.

-nodeprecated

Пропускать некорректные (deprecated) места кода. При указании этого ключа автоматически используется -nodeprecatedlist. Ключ доступен только в doclet, установленном по умолчанию.

-nodeprecatedlist

При указании этого ключа программа *javadoc* не создает файл *deprecated-list.html* и не включает ссылку на него в навигационную панель. Ключ доступен только в doclet, установленном по умолчанию.

-nohelp

При использовании этого ключа *javadoc* не создает файл справки и не включает ссылку на него в навигационную панель. Ключ доступен только в doclet, установленном по умолчанию.

-noindex

Не создавать файлов с индексом. Ключ доступен только в doclet, установленном по умолчанию.

-nonavbar

Не создавать навигационные панели в начале и в конце файлов. При этом текст, указанный в ключах -header и -footer, также будет игнорироваться. Данный ключ полезен при создании документации, предназначенной для печати. Ключ доступен только в doclet, установленном по умолчанию.

-noqualifier *пакеты* | all

Если указан этот ключ, программа *javadoc* пропускает имена пакетов в названиях классов, входящих в текущий обрабатываемый пакет. Кроме того, пропускаются

имена всех пакетов, указанных в параметре. Если в качестве параметра задано ключевое слово `all`, то исключаются имена всех пакетов. *пакеты* – это список имен пакетов, разделенный запятыми; в списке может присутствовать групповой символ `*`, указывающий на подпакеты. Например, `-noqualifier java.io:java.nio.*` исключает имена пакетов из названий всех классов пакетов java.io, java.nio и его подпакетов. Ключ доступен только в doclet, установленном по умолчанию.

`-nosince`

Игнорировать все теги `@since` в комментариях. Ключ доступен только в doclet, установленном по умолчанию.

`-notree`

С этим ключом *javadoc* не создает диаграмму иерархии классов *tree.html* и не помещает ссылку на нее в навигационную панель. Ключ доступен только в doclet, установленном по умолчанию.

`-overview` *имя_файла*

С этим ключом *javadoc* читает обзорный комментарий из указанного файла и использует его при создании обзорной страницы. Этот файл не содержит исходного Java-кода, поэтому комментарий в нем можно не выделять символами `/**` и `*/`.

`-package`

При создании выходных файлов помимо членов классов, объявленных как `public` или `protected`, учитываются члены классов, видимые только внутри пакета.

`-private`

При создании выходных файлов учитываются все члены классов, в том числе видимые только внутри пакета или объявленные как `private`.

`-protected`

Учитывать только члены классов, объявленные как `public` и `protected`. Этот ключ используется по умолчанию.

`-public`

Учитывать только члены классов, объявленные как `public`. Все остальные члены классов игнорируются.

`-quiet`

Запретить вывод любой информации, кроме предупреждений и сообщений об ошибках.

`-serialwarn`

Выдавать предупреждение, если комментарии к сериализуемому классу не содержат формат сериализации в тегах `@serial`. Ключ доступен только в doclet, установленном по умолчанию.

`-source 1.4`

Этот ключ используется при обработке кода на Java 1.4, в котором применяется оператор `assert` нового типа.

`-sourcepath` *путь*

Устанавливает путь поиска исходных файлов (как правило, это один каталог верхнего уровня). *javadoc* использует этот путь при поиске файлов с исходным кодом указанного пакета.

`-splitindex`

Создать несколько индексных файлов, каждый из которых соответствует одной букве алфавита. Этот ключ следует применять при создании документации к большому количеству кода, иначе программа *javadoc* создаст только один индексный файл, неудобный в использовании из-за большого размера. Ключ доступен только в doclet, установленном по умолчанию.

`-stylesheetfile` *файл*

Определяет файл таблицы стилей CSS для создаваемых HTML-документов. *javadoc* вставляет в эти документы ссылки на данный файл. Ключ доступен только в doclet, установленном по умолчанию.

`-subpackages` *пакеты*

Обрабатывать указанные пакеты вместе со всеми подпакетами. Параметр *пакеты* представляет собой список имен пакетов или префиксов имен пакетов, разделенных запятыми. Часто удобнее указать этот ключ, нежели полностью перечислять имена всех необходимых пакетов. Например:

```
-subpackages java:javax
```

См. также `-exclude`. Ключ доступен только в doclet, установленном по умолчанию.

`-tag` *имя_тега:where:текст_заголовка*

При использовании этого ключа программа *javadoc* обрабатывает тег, указанный в параметре *имя_тега*, следующим образом: вставляет *текст_заголовка* рядом с любым текстом, идущим за тегом. Это дает возможность использовать в комментариях простые пользовательские теги (с тем же синтаксисом, как у тегов `@return` и `@author`). Параметр *where* – это строка из символов, определяющих тип комментариев, в которых разрешен пользовательский тег. Здесь допускаются следующие символы: `a` (тег разрешен везде), `p` (в пакетах), `t` (в типах – классах и интерфейсах), `c` (в конструкторах), `m` (в методах) и `f` (в полях).

Второе применение ключа `-tag` – определение порядка, в котором обрабатываются теги и появляются выходные данные. После ключа `-tag` можно указать имена стандартных тегов, чтобы задать нужный порядок. В данный список можно включать пользовательские теги и классы вида «taglet». Ключ доступен только в doclet, установленном по умолчанию.

`-taglet` *имя_класса*

Определяет имя класса типа «taglet», который будет обрабатывать пользовательский тег. Техника написания таких классов здесь не рассматривается. Теги, указанные в ключе `-taglet`, могут стоять между тегами, указанными в ключе `-tag`, определяя порядок, в котором они будут обрабатываться. Ключ доступен только в doclet, установленном по умолчанию.

`-tagletpath` *путь*

В этом ключе указывается через запятую список файлов JAR или каталогов, в которых будет производиться поиск классов типа «taglet». Ключ доступен только в doclet, установленном по умолчанию.

`-use`

Создает для каждого класса и пакета дополнительный файл, в который помещаются ссылки на каждый случай использования этого класса или пакета.

`-verbose`

Отображать дополнительные информационные сообщения при обработке исходных файлов.

`-version`

Этот ключ используется, чтобы включить данные из тегов `@version` в создаваемые файлы (а не для вывода версии программы *javadoc* на экран). Ключ доступен только в doclet, установленном по умолчанию.

`-windowtitle` *текст*

Поместить *текст* в тег `<TITLE>` каждого создаваемого файла. Обычно текст из этого тега помещается в строку заголовка, а также в список закладок и в журнал веб-броузера. Этот *текст* не должен содержать HTML-тегов. См. также `-doctitle` и `-header`. Ключ доступен только в doclet, установленном по умолчанию.

Переменные окружения

`CLASSPATH`

Эта переменная определяет путь, который *javadoc* использует по умолчанию при поиске файлов классов и исходных файлов. Ключи `-classpath` или `-sourcepath` замещают значение этой переменной. Подробнее она рассмотрена в разделах «java» и «javac».

См. также

java, *javac*

javah

JDK 1.0 и выше

Генератор заглушек на языке C для методов, зависимых от платформы

Краткое описание

```
javah [ ключи ] имена_классов
```

Описание

Программа *javah* создает заголовочные и исходные файлы на C (файлы *.h* и *.c*), которые используются при реализации родных (native) методов Java на языке C. Интерфейс таких методов в Java 1.1 претерпел изменения. В Java 1.1 и предыдущих версиях программа *javah* создает файлы методов старого типа. В Java 1.1 программа *javah* с ключом `-jni` создает файлы нового типа. В Java 1.2 и последующих версиях этот ключ применяется по умолчанию.

Представленная информация относится только к самой программе *javah*. Полное описание способов реализации собственных методов Java на языке C выходит за рамки этой книги.

Ключи

`-bootclasspath`

Определяет путь поиска системных классов. См. раздел «javac». Ключ доступен в Java 1.2 и последующих версиях.

`-classpath` *путь*

Определяет путь поиска классов, указанных в командной строке. Этот ключ замещает значение переменной окружения `CLASSPATH`. В версиях Java, предшествующих 1.2, в этом ключе можно указать расположение системных классов и расши-

рений. В Java 1.2 и последующих версиях можно указать только путь к классам приложений (см. -bootclasspath). Путь поиска классов подробно рассмотрен в разделе «java».

-d *каталог*

Определяет каталог, в который будут записаны файлы, созданные программой *javah*. По умолчанию она сохраняет файлы в текущем каталоге. Этот ключ нельзя использовать одновременно с -o.

-force

Всегда записывать выходные файлы, даже если они не содержат ничего полезного.

-help

С этим ключом программа *javah* выводит краткую справку и завершается.

-jni

Создавать заголовочные файлы, совместимые с новым интерфейсом Java Native Interface (JNI), вместо файлов старого типа, использовавшегося в JDK 1.0. В Java 1.2 и последующих версиях этот ключ используется по умолчанию. См. также -old. Ключ доступен в Java 1.1 и последующих версиях.

-o *выходной_файл*

Собрать все выходные данные в один *выходной_файл*, а не создавать отдельные файлы для каждого класса.

-old

Создавать выходные файлы для методов старого типа, зависимых от платформы (Java 1.0). В версиях Java, предшествовавших 1.2, такой тип файлов использовался по умолчанию. См. также -jni. Ключ доступен в Java 1.2 и последующих версиях.

-stubs

Создавать для классов вместо заголовочных файлов файлы-заглушки *.c* (stub files). Этот ключ предназначен для использования только с интерфейсом методов Java 1.0, зависимых от платформы. См. также -old.

-td *каталог*

Определяет каталог, в который программа *javah* будет помещать временные файлы. В ОС Unix по умолчанию используется каталог */tmp*.

-trace

Включить в файлы-заглушки команды для трассировки. В Java 1.2 и последующих версиях этот ключ убран за ненадобностью. Вместо этого можно применять интерпретатор Java с ключом -verbose:jni.

-v, -verbose

Включить режим отображения информационных сообщений. В этом режиме *javah* выводит сообщения о каждом действии. Ключ -v доступен в Java 1.2 и последующих версиях. Допускается полная форма: -verbose.

-version

Вывести номер версии программы *javah*.

Переменные окружения

CLASSPATH

Определяет путь поиска классов, используемый по умолчанию. Переменная подробно описана в разделе «java».

См. также

java, javac

javap

JDK 1.0 и выше

Дизассемблер классов Java

Краткое описание

```
javap [ ключи ] имена_классов
```

Описание

Программа *javap* читает файлы классов, указанные в параметре *классы* командной строки, и отображает текст API этого класса в понятной пользователю форме. Эта программа может также отображать методы в виде байт-кода ВМ.

Ключи

`-b`

Ключ предназначен для обратной совместимости с *javap* из Java 1.1. Используется в программах, зависящих от точного формата выходных данных дизассемблера *javap*. Ключ доступен в Java 1.2 и последующих версиях.

`-bootclasspath` *путь*

Устанавливает путь поиска системных классов. Этот ключ, используемый довольно редко, подробно рассмотрен в разделе «javac». Доступен в Java 1.2 и последующих версиях.

`-c`

Показывать код (то есть байт-код ВМ) всех методов, независимо от их уровня видимости, для каждого указанного класса.

`-classpath` *путь*

Определяет путь поиска классов, указанных в командной строке. Этот ключ замещает переменную окружения `CLASSPATH`. В версиях Java, предшествующих 1.2, аргумент *путь* определяет путь ко всем системным классам, классам расширений и приложений. В Java 1.2 и последующих версиях он определяет путь только к классам приложений. См. также `-bootclasspath` и `-extdirs`. Подробнее путь к классам рассмотрен в разделах «java» и «javac».

`-extdirs` *каталоги*

Этот ключ определяет один или несколько каталогов, в которых будет производиться поиск классов расширений. Применяется довольно редко. Подробно рассмотрен в разделе «javac». Доступен во всех версиях Java, начиная с 1.2.

`-l`

Показывать таблицы номеров строк и локальных переменных, если соответствующая информация содержится в файлах классов. Как правило, этот ключ используется одновременно с `-c`. По умолчанию компилятор *javac* не включает информацию о локальных переменных в созданный файл класса. См. ключ `-g` команды *javac*.

`-help`

Программа *javac* выводит справочное сообщение и завершает свою работу.

`-J`*javaoption*

Параметр *javaoption* передается непосредственно интерпретатору Java.

`-package`

С этим ключом отображаются методы, видимые только внутри пакета, и методы, объявленные как `protected` и `public`; методы со спецификатором доступа `private` не выводятся. Этот ключ используется по умолчанию.

`-private`

Отображаются все члены класса, в том числе объявленные как `private`.

`-protected`

Отображаются только члены класса, объявленные как `protected` и `public`.

`-public`

Отображаются только члены класса со спецификатором доступа `-public`.

`-s`

При задании этого ключа объявления членов класса отображаются в формате, используемом для описания сигнатур типов и методов внутри виртуальной машины, а не в виде более привычного исходного кода на Java.

`-verbose`

Включить отображение дополнительной информации. В этом режиме выводится дополнительная информация о каждом члене указанных классов в виде комментариев Java.

`-verify`

С этим ключом программа *javap* выполняет частичную верификацию указанных классов и выводит результаты на экран. Эта опция доступна только в Java 1.0 и Java 1.1 и удалена в последующих версиях из-за неудовлетворительного качества верификации.

`-version`

Показать номер версии *javap*.

Переменные окружения

`CLASSPATH`

Определяет путь поиска классов приложений. Ключ `-classpath` замещает значение этой переменной. Путь к классам подробно рассмотрен в разделе «java».

См. также

java, javac

jdb

JDK 1.0 и выше

Отладчик Java

Краткое описание

```
jdb [ ключи ] класс [ ключи_программы ]
jdb ключи_соединения
```

Описание

Программа *jdb* – это отладчик классов Java. Она работает в текстовом режиме, управляется из командной строки. Команды *jdb* напоминают команды отладчиков *dbx* и *gdb* (ОС Unix) для программ на C и C++.

Программа *jdb* написана на Java, поэтому она сама исполняется интерпретатором Java. Если *jdb* запускается с указанием имени класса, она запускает вторую копию интерпретатора *java* со всеми параметрами, указанными в ее командной строке. Новая копия интерпретатора запускается со специальными ключами, разрешающими сообщение с *jdb*. Эта копия интерпретатора запускает указанный файл класса, а затем останавливается до начала выполнения байт-кода в ожидании отладочных команд.

С помощью *jdb* можно также отлаживать программы, уже исполняемые другой копией интерпретатора Java. Для этого нужно использовать специальные опции интерпретатора и отладчика. В Java 1.3 архитектура отладки претерпела серьезные изменения. В частности, появилась возможность подключения *jdb* к уже выполняющемуся интерпретатору.

Синтаксис выражений jdb

Такие команды отладчика, как `print`, `dump` и `suspend` позволяют обращаться к классам, объектам, методам, полям и потокам отлаживаемой программы. Можно обращаться к классам по имени, указывать имя пакета не обязательно. К статическим членам класса можно также обращаться по имени. К объектам можно обращаться по идентификатору (восьмизначное шестнадцатеричное целое число). Если отлаживаемый класс содержит информацию о локальных переменных, то при обращении к объектам можно использовать их имена. Для обращения к полям объекта и к элементам массива можно применять привычный синтаксис языка Java. Вы можете использовать этот синтаксис и для написания достаточно сложных выражений. В Java 1.3 разрешается применять этот синтаксис даже для вызова методов.

Для многих команд *jdb* необходимо указать поток. Каждому потоку присваивается целочисленный идентификатор. Для обращения к потоку используется синтаксис `t@`*n*, где *n* – идентификатор потока.

Ключи

При вызове *jdb* с указанием имени файла класса можно использовать любые ключи интерпретатора Java. Кроме того, *jdb* поддерживает следующие ключи:

`-attach [`*хост*`:]`*порт*

Подключиться к клиентской ВМ, выполняющейся на указанном хосте (по умолчанию на локальном хосте) и ожидающей соединения на указанном порту. Ключ доступен в Java 1.3 и последующих версиях.

Чтобы отладчик *jdb* мог соединиться с ВМ во время выполнения, виртуальная машина должна быть запущена командой следующего вида:

```
% java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

Архитектурой *jdb* в Java 1.3 предусмотрен большой набор опций для управления соединением отладчика с интерпретатором. Подробное описание этих опций в нашей книге не приводится.

`-help`

Выдать информацию о доступных ключах.

`-host` *имя_хоста*

В Java 1.2 и предыдущих версиях этот ключ используется для соединения с уже выполняющимся интерпретатором. Параметр ключа определяет имя хоста, на котором выполняется сессия интерпретатора (по умолчанию это локальный хост). Данный ключ необходимо применять с `-password`. В Java 1.3 этот ключ заменен ключом `-attach`.

`-launch`

Запустить указанное приложение при старте *jdb*. Это позволяет избежать явного использования команды `run`. Ключ доступен в Java 1.3 и последующих версиях.

`-password` *пароль*

В Java 1.2 и последующих версиях этот ключ определяет пароль к виртуальной машине Java на указанном хосте. При использовании вместе с `-hostname` данный ключ позволяет *jdb* соединиться с работающим интерпретатором, который должен быть запущен с ключом `-debug` или `-Xdebug`, включающим отображение пароля. В Java 1.3 этот ключ заменен на `-attach`.

`-sourcepath` *путь*

Установить путь поиска исходных файлов, которые будут использоваться при отладке соответствующих файлов классов. Если этот ключ не указан, *jdb* выбирает путь поиска, определенный по умолчанию. Ключ доступен в Java 1.3 и последующих версиях.

`-tclassic`

Использовать «классическую» ВМ вместо клиентской (HotSpot), применяемой по умолчанию в Java 1.3. Ключ доступен в Java 1.3 и последующих версиях.

`-version`

С этим ключом *jdb* показывает номер версии и завершает свою работу.

Команды

Программа *jdb* понимает следующие команды:

`?` **или** `help`

Вывести список всех команд с кратким описанием каждой.

`!!`

Заменяется текстом последней введенной команды. За данной командой может идти дополнительный текст, который присоединяется к предыдущей команде.

`catch [` *исключение* `]`

Ставить точку останова в местах, где возникает указанное исключение. Если *исключение* не указано, выводится список перехваченных на данный момент исключений. Для отмены точек останова по этим исключениям применяйте команду `ignore`.

`classes`

Вывести список всех загруженных классов.

`clear`

Вывести список всех точек останова.

`clear` *класс.метод*`[(`*тип_параметра...*`)]`

Снять все точки останова, установленные в указанном методе указанного класса.

`clear [` *класс:строка* `]`

Снять точку останова со строки указанного класса.

`cont`

Продолжить выполнение. Эта команда необходима, если текущий поток остановлен в точке останова.

`down [ n ]`

Спуститься вниз на *n* фреймов стека вызовов текущего потока. Если параметр *n* не указан, производится спуск на один фрейм.

`dump id...`

Вывести значение всех полей указанного объекта (объектов). Если указать имя класса, команда `dump` выводит значения всех статических методов и переменных этого класса, а также родительский класс и список реализованных интерфейсов. В качестве параметра можно указать имя или восьмизначный шестнадцатеричный идентификатор объектов или классов. Потоки можно указывать так: `t@`*номер_потока*.

`exit` или `quit`

Завершить работу *jdb*.

`gc`

Запустить сборщик мусора для утилизации неиспользуемых объектов.

`ignore` *исключение*

Не останавливаться на указанном исключении. Эта команда отменяет команду `catch`. Указанные исключения будет игнорировать только отладчик, а не интерпретатор Java.

`list [` *номер_строки* `]`

Вывести строку исходного кода с указанным номером, а также несколько предшествующих и последующих строк; если номер строки не указан, применяется номер строки из текущего фрейма стека текущего потока. Выводятся строки исходного файла текущего фрейма стека, принадлежащего текущему потоку. Чтобы указать *jdb* местоположение исходных файлов, используйте команду `use`.

`list` *метод*

Вывести исходный код указанного метода.

`load` *имя_класса*

Загрузить указанный класс в отладчик.

`locals`

Вывести список локальных переменных из текущего фрейма стека. Для того чтобы информация о локальных переменных была доступна, код Java нужно скомпилировать с ключом `-g`.

`memory`

Показать информацию об использовании памяти отлаживаемой программой.

`methods` *класс*

Вывести список всех методов указанного класса. Для вывода списка переменных экземпляра объекта или статических переменных класса используйте команду `dump`.

`print id...`

Вывести на экран значение указанного элемента. Элемент может быть классом, объектом, полем или локальной переменной; можно указать его имя или восьмизначный шестнадцатеричный идентификатор. Для обращения к потокам нужно применять синтаксис `t@`*номер_потока*. Команда `print` выводит на экран значение объекта, используя его метод `toString()`.

`next`

Выполнить текущую строку исходного кода, включая вызовы методов. См. также `step`.

`resume [ id_потока... ]`

Возобновить выполнение указанного потока или потоков. Если поток не указан, возобновляется выполнение всех приостановленных потоков. См. также `suspend`.

`run [ класс ] [ аргументы ]`

Запустить метод `main()` указанного класса, передав ему *аргументы*. Если не указаны *класс* или *аргументы*, они берутся из командной строки *jdb*.

`step`

Выполнить текущую строку текущего потока и остановиться. Если в строке вызывается метод, то выполняется шаг внутрь этого метода с последующим остановом. См. также `next`.

`stepi`

Выполнить одну инструкцию виртуальной машины Java.

`step up`

Выполнять до возвращения из текущего метода и остановиться в месте вызова.

`stop`

Вывести список всех точек останова.

`stop at класс:строка`

Установить точку останова на указанной строке указанного класса. По достижении этой строки программа останавливается. Для удаления точки останова применяйте `clear`.

`stop in класс.метод[(param-type...)]`

Установить точку останова в начале указанного метода указанного класса. Программа останавливается при входе в этот метод. Для удаления точки останова используйте `clear`.

`suspend [ id_потока... ]`

Приостановить выполнение указанного потока(ов). Если потоки не указаны, команда останавливает все выполняющиеся потоки. Чтобы возобновить выполнение, используйте команду `resume`.

`thread id_потока`

Сделать указанный поток текущим. Этот поток неявно применяется другими командами *jdb*. Можно указывать имя потока или его номер.

`threadgroup имя`

Установить текущую группу потоков.

`threadgroups`

Вывести список всех групп потоков из отладочной сессии интерпретатора.

`threads [ группа_потоков ]`

Вывести список всех потоков из указанной группы. Если группа не указана, выводится список всех потоков текущей группы, определенной командой `threadgroup`.

`up [ n ]`

Подняться на *n* фреймов стека вызовов текущего потока (на один фрейм, если не указано число *n*).

`use [ путь_к_исходному_файлу ]`

Установить путь поиска исходных файлов отлаживаемых классов. Если путь не указан, выводится текущий путь поиска.

`where [id_потока] [all]`

Показать распечатку стека указанного потока. Если поток не указан, на экран выводится распечатка стека текущего потока. Если указанно `all`, выводится распечатка стека для всех потоков.

`wherei [id_потока x]`

Показать распечатку стека указанного или текущего потока, включая детальную информацию о счетчике команд.

Переменные окружения

`CLASSPATH`

Определяет упорядоченный список каталогов, файлов ZIP и JAR (разделенных двоеточиями в Unix или точками с запятой в Windows), в которых *jdb* будет искать определения классов. Если эта переменная определена, *jdb* автоматически помещает в конец списка путь к системным классам. В противном случае путем поиска по умолчанию считается текущий каталог и каталог системных классов. Эта переменная среды замещается ключом `-classpath`.

См. также

java

keytool

Java 2 SDK 1.2 и выше

Программа управления ключами и сертификатами

Краткое описание

```
keytool команда ключи
```

Описание

Программа *keytool* управляет хранилищем открытых и секретных ключей и сертификатов открытых ключей (*keystore*). В ней определено множество команд для создания ключей, импорта, экспорта и отображения данных из хранилища ключей. В хранилище для ключей и сертификатов определены псевдонимы, которые не чувствительны к регистру. *keytool* обращается к ключу или сертификату по псевдониму.

В качестве первого параметра командной строки должна идти основная команда. Следующие параметры уточняют основную команду. Нужно указывать только одну команду. Если ей нужны параметры, для которых не определены значения по умолчанию, программа *keytool* предлагает ввести эти значения.

Команды

`-certreq`

Создать запрос на подпись сертификата с указанным псевдонимом в формате PKCS#10. Запрос записывается в указанный файл или в стандартный выходной поток, а затем отправляется в центр сертификации (certificate authority), который проводит аутентификацию запрашивающей стороны и отсылает обратно подписанный сертификат, подтверждающий подлинность открытого ключа. Далее подписанный сертификат можно импортировать в хранилище ключей командой

-import. С данной командой используются ключи -alias, -file, -keypass, -keystore, -sigalg, -storepass, -storetype и -v.

-delete

Удалить псевдоним из указанного хранилища ключей. С этой командой применяются следующие ключи: -alias, -file, -keystore, -rfc, -storepass, -storetype и -v.

-export

Записать сертификат, связанный с указанным псевдонимом, в файл или стандартный выходной поток. С этой командой используются следующие опции: -alias, -file, -keystore, -rfc, -storepass, -storetype и -v.

-genkey

Создать открытый и секретный ключ и подписанный сертификат X.509 для открытого ключа. Часто эта команда используется вместе с -certreq, поскольку подписанные сертификаты сами по себе почти не применяются. С этой командой используются следующие ключи: -alias, -dname, -keyalg, -keypass, -keysize, -keystore, -sigalg, -storepass, -storetype, -v и -validity.

-help

Вывести список всех команд *keytool* и их ключей. Эта команда выполняется без ключей.

-identitydb

Записать ключи и сертификаты из базы данных ключей программы *javakey* (Java 1.1) в хранилище ключей. База данных считывается из файла, если он указан, или из стандартного входного потока в противном случае. Ключи и сертификаты записываются в указанный файл хранилища ключей. Если такой файл не существует, он создается автоматически. С этой командой применяются следующие ключи: -file, -keystore, -storepass, -storetype и -v.

-import

При указании этой команды программа читает сертификат или цепочку сертификатов PKCS#7 из указанного файла или стандартного входного файла и сохраняет сертификаты в хранилище ключей как надежные сертификаты с указанными псевдонимами. С этой командой применяются следующие ключи: -alias, -file, -keypass, -keystore, -noprompt, -storepass, -storetype, -trustcacerts и -v.

-keyclone

Создать копию указанного элемента хранилища ключей и сохранить ее под другим псевдонимом. С этой командой применяются ключи -alias, -dest, -keypass, -keystore, -new, -storepass, -storetype и -v.

-keypasswd

Поменять пароль для шифрования секретного ключа, связанного с указанным псевдонимом. С этой командой применяются следующие ключи: -alias, -keypass, -new, -storetype и -v.

-list

Вывести в стандартный выходной поток отпечаток (fingerprint) сертификата с указанным псевдонимом. Если указан ключ -v, то выводится информация о сертификате. С ключом -rfc выводится содержимое сертификата в машинном представлении. С этой командой применяются такие ключи: -alias, -keystore, -rfc, -storepass, -storetype и -v.

-printcert

Вывести содержимое сертификата из указанного файла или из стандартного входного потока. В отличие от большинства команд, эта команда не использует хранилище ключей. С ней применяются ключи -file и -v.

-selfcert

При задании этой команды программа создает подписанный сертификат для открытого ключа с указанным псевдонимом. Данный сертификат заменяет все сертификаты или цепочки сертификатов, связанные с этим псевдонимом. С командой применяются следующие ключи: -alias, -dname, -keypass, -keystore, -sigalg, -storepass, -storetype, -v и -validity.

-storepasswd

Поменять пароль, обеспечивающий целостность всего хранилища ключей. Минимальная длина пароля – 6 символов. С этой командой применяются следующие ключи: -keystore, -new, -storepass, -storetype и -v.

Ключи

Команды программы *keytool* могут выполняться с различными ключами, перечень которых приведен ниже. Многие ключи имеют значения по умолчанию. Если такие значения не определены и не указаны в командной строке, программа *keytool* предлагает пользователю ввести соответствующие значения.

-alias *имя*

Указывает псевдоним из хранилища ключей, над которым будут выполняться последующие действия. Значение по умолчанию – «mykey».

-dest *новый_псевдоним*

Определяет имя нового псевдонима (или псевдонима, в который будет сделана запись) для команды -keyclone. Если ключ не указан, *keytool* предлагает пользователю ввести значение.

-dname *X.500-distinguished-name*

Указать отличительное имя (distinguished name) сертификата X.500, создаваемого командой -selfcert или -genkey. Отличительное имя – это подробное имя, которое служит глобальным и уникальным идентификатором. Например:

```
CN=David Flanagan, OU=Editorial, O=OReilly, L=Cambridge, S=Massachusetts, C=US
```

Если имя не указано в командной строке, а ключ применяется с командой -genkey, то *keytool* предлагает пользователю ввести имя. По умолчанию команда -selfcert использует отличительное имя текущего сертификата.

-file *файл*

Определяет входной или выходной файл для многих команд *keytool*. Если этот ключ не указан, *keytool* использует для чтения стандартный входной поток, а для записи – стандартный выходной поток.

-keyalg *имя_алгоритма*

Используется вместе с командой -genkey. Определяет тип криптографических ключей, которые создаст программа. В стандартной поставке Java фирмы Sun поддерживается только алгоритм «DSA» (значение по умолчанию).

-keypass *пароль*

Определяет пароль для шифрования секретного ключа, находящегося в хранилище ключей. Если этот ключ не указан, *keytool* сначала пытается применить па-

роль, указанный ключом -storepass. Если он не подходит, пользователю предлагается ввести пароль.

-keysize *размер*

Используется командой -genkey и определяет размер ключа (в битах), который будет сгенерирован. По умолчанию создается 1024-битный ключ.

-keystore *имя_файла*

Определяет путь к файлу хранилища ключей. По умолчанию применяется файл *.keystore* из домашнего каталога пользователя.

-new *новый_пароль_или_псевдоним*

Используется с командой -keyclone для указания нового имени псевдонима, а также с командами -keypasswd и -storepasswd для указания нового пароля. Если этот ключ не указан, *keytool* предлагает пользователю ввести соответствующее значение.

-noprompt

Используется вместе с командой -import для отмены диалога с пользователем в случае, если программе не удалось установить цепочку доверия (chain of trust) для импортируемого сертификата. Если этот ключ не указан, пользователю предлагается ввести необходимые данные.

-rfc

Используется вместе с командами -list и -export и устанавливает выходной формат сертификата (printable encoding format), задаваемый RFC-1421. Без этого ключа команда -export выводит сертификат в двоичном формате, а -list выводит только отпечаток (fingerprint). В команде -list нельзя одновременно использовать ключи -rfc и -v.

-sigalg *имя_алгоритма*

Определяет имя алгоритма цифровой подписи сертификата. Значение по умолчанию зависит от типа открытого ключа. Для ключей типа DSA алгоритмом по умолчанию является «SHA1withDSA», а для ключей RSA – «MD5withRSA».

-storepass *пароль*

Определяет пароль к файлу хранилища ключей. Он также используется как пароль по умолчанию к секретным ключам, для которых не указан ключ -keypass. Если ключ -storepass не указан, программа *keytool* предлагает пользователю ввести пароль. Минимальная длина пароля – 6 символов.

-storetype *тип*

Определяет тип хранилища ключей. Если этот ключ не указан, значение по умолчанию берется из файла свойств системной безопасности. Часто по умолчанию применяется тип «JKS» фирмы Sun.

-trustcacerts

Используется вместе с командой -import. Указывает, что сертификаты из хранилища ключей *jre/lib/security/cacerts*, подписанные центром сертификации, могут считаться надежными (trusted). Если ключ не указан, программа *keytool* игнорирует этот файл.

-v

Включает режим вывода дополнительных сообщений, если он доступен. В этом режиме многие команды выводят дополнительную информацию.

`-validity` *срок_действия*

Используется с командами `-genkey` и `-selfcert`, определяет срок действия (в днях) создаваемого сертификата. Значение по умолчанию – 90 дней.

См. также

jarsigner, *javakey*, *policytool*

native2ascii

JDK 1.1 и выше

Программа преобразования кода Java в ASCII

Краткое описание

```
native2ascii [ ключи ] [ входной_файл [ выходной_файл ]]
```

Описание

Программа *javac* может обрабатывать файлы только в 8-битной кодировке Latin-1, а любые другие символы в них должны быть представлены в виде `\uxxxx` (формат Unicode). *native2ascii* – это несложная программа, преобразующая исходные Java-файлы, использующие локальную кодировку, в формат «Latin-1-plus-ASCII-encoded-Unicode», требуемый *javac*.

Указывать параметры *входной_файл* и *выходной_файл* необязательно. Если они не указаны, используются стандартные потоки, поэтому программу *native2ascii* удобно применять с каналами (pipes).

Ключи командной строки

`-encoding` *кодировка*

Определяет кодировку исходных файлов. Если этот ключ не указан, используется значение системного свойства `file.encoding`.

`-reverse`

Предписывает программе *native2ascii* провести конвертацию в обратном направлении – символы `\uxxxx` представить в локальной кодировке.

См. также

`java.io.InputStreamReader`, `java.io.OutputStreamWriter`

policytool

Java 2 SDK 1.2 и выше

Менеджер файлов политик

Краткое описание

```
policytool
```

Описание

Программа *policytool* применяет пользовательский интерфейс Swing, дающий возможность легко редактировать конфигурационные файлы политик. Архитектура системы безопасности Java основана на файлах политик, в которых определен набор разрешений для программ из различных источников. По умолчанию система безопасности Java определяется файлом системных политик *jre/lib/security/java.policy* и файлом пользовательских политик *.java.policy* из домашнего каталога пользователя. Системные администраторы и пользователи могут изменять эти файлы в тексто-

вом редакторе, но синтаксис данных файлов довольно сложный, поэтому удобнее использовать программу *policytool*.

Выбор файла политик для редактирования

При старте программа *policytool* по умолчанию открывает файл *.java.policy*, находящийся в домашнем каталоге пользователя. Чтобы создать новый файл, открыть существующий файл или сохранить изменения, применяйте соответственно команды New, Open и Save из меню File.

Редактирование файла политик

В основном окне программы *policytool* элементы файла политик отображаются в виде списка. Каждый элемент этого списка определяет источник программ и права, которые даются программам из этого источника. Окно содержит кнопки, с помощью которых можно добавить новый элемент, редактировать или удалить существующий элемент файла политик.

Когда в ядро платформы Java 1.4 был введен JAAS API, в программу *policytool* были внесены изменения, позволяющие определять `Principal`, которому предоставляются права.

С каждым файлом политик связано хранилище ключей, из которого извлекаются сертификаты, необходимые при проверке цифровых подписей Java-программ. Как правило, используется хранилище ключей, определенное по умолчанию, но при необходимости можно связать с файлом политик другое хранилище. Для этого служит команда Change Keystore меню Edit главного окна программы.

Добавление и изменение статей политик

В окне редактора политик отображается источник программ данной статьи и список связанных с ним прав. Кроме того, в окне содержатся кнопки для добавления нового права, а также удаления или изменения существующего.

Первым шагом при создании новой политики является определение источника программ. Он определяется URL и перечнем цифровых подписей, которые должны содержаться в этих программах. Введите URL источника и (необязательно) перечислите через запятую псевдонимы сертификатов из хранилища ключей, связанного с редактируемым файлом политик.

После того как источник кода определен, укажите права для программ из этого источника. При добавлении или редактировании прав применяйте соответственно кнопки Add Permission и Edit Permission. После нажатия этих кнопок появится окно редактора прав.

Определение прав

Чтобы установить право, в окне редактора прав сначала выберите тип права из выпадающего списка Permission. Затем выберите соответствующий объект из списка Target Name. Этот список автоматически изменяется в зависимости от выбранного типа права. Для некоторых типов прав, например `FilePermission`, список объектов не составляется, поэтому значение нужно вводить вручную. Например, если объектом является каталог */tmp*, введите «/tmp», а если все файлы из этого каталога и его подкаталоги, то введите «/tmp/*» или «/tmp/-». Поддерживаемые типы объектов описаны в документации к соответствующему классу `Permission`.

В зависимости от типа права может потребоваться выбрать тип действия из меню Actions. Когда указано право, объект и действие, нажмите OK.

См. также

jarsigner, keytool

serialver JDK 1.1 и выше

Генератор версии класса

Краткое описание

```
serialver [-show] имя_класса...
```

Описание

Программа *serialver* показывает номер версии класса (или классов). Номер версии используется при сериализации: он меняется каждый раз, когда изменяется формат сериализации класса.

Если в классе объявлена константа `longserialVersionUID`, то выводится ее значение. Иначе программа вычисляет уникальный номер версии, применяя алгоритм Secure Hash Algorithm (SHA) к API этого класса. Основное назначение программы – вычислить уникальный номер версии класса, который затем можно поместить в класс в виде константы. На выходе программа создает Java-код, который можно вставить в объявление класса.

Ключи командной строки

`-show`

Если указан этот ключ, программа *serialver* отображает окно, содержащее поле для ввода имени класса (можно ввести только одно имя) и отображающее серийный идентификатор UID. При использовании ключа `-show` в командной строке нельзя указывать имя класса.

Переменные окружения

`CLASSPATH`

Поскольку программа *serialver* написана на Java, она использует эту переменную так же, как интерпретатор *java*. Поиск указанных классов производится с учетом этой переменной.

См. также

`java.io.ObjectStreamClass`

Часть II

Справочник по API

Часть II – основа книги. Здесь содержится справочная информация по важнейшим API платформы Java. Следующий раздел «Как использовать этот справочник» поможет извлечь максимальную пользу из данного материала.

Как использовать этот справочник

В данном справочнике значительный объем информации умещается всего лишь на 650 страницах. В этом вводном разделе объясняется, как наиболее эффективно воспользоваться этой информацией. Здесь будет показано, как устроен справочник и как следует читать его отдельные статьи.

Как найти статью в справочнике

Справочник разделен на главы, каждая из которых документирует один пакет Java-платформы или группы смежных пакетов. Пакеты перечислены в алфавитном порядке, поэтому вам не придется запоминать, какая именно глава документирует конкретный пакет: вы можете просто осуществлять поиск по алфавиту, как в словаре. Документация по каждому пакету начинается со статьи справочника по самому пакету. Такая статья включает краткий обзор пакета и перечисление классов и интерфейсов, включенных в данный пакет. В списке содержимого пакета классы и интерфейсы группируются по категориям (например, интерфейсы, классы и исключения). Внутри каждой категории они группируются на основе иерархии классов, а отступы обозначают уровень иерархии. И наконец, классы и интерфейсы, расположенные на одном и том же уровне иерархии, перечисляются в алфавитном порядке.

За обзором каждого пакета следуют отдельные справочные статьи по классам и интерфейсам, которые определяются в данном пакете. Все статьи в этом справочном материале организованы в алфавитном порядке в соответствии с именами класса *и* пакета, поэтому смежные классы располагаются рядом друг с другом. Это означает следующее: чтобы посмотреть справочную статью по конкретному классу, необходимо выяснить имя пакета, содержащего этот класс. Как правило, имя пакета очевидно из его контекста, и вы легко найдете интересующую вас справочную статью. Имя обсуждаемого пакета или класса вынесено в колонтитул, что также поможет при поиске необходимого пакета или класса.

Иногда необходимо посмотреть класс, пакет которого вам еще не известен. В этом случае обратитесь к главе 25. Данный перечень позволит найти класс по имени и выяснить, частью какого пакета он является.

Как читать статью в справочнике

Справочные статьи по классам и интерфейсам содержат большое количество информации. Далее следуют разделы, в которых описана структура справочной статьи, а также вид информации, ее местонахождение и назначение. Во время чтения последующих описаний есть смысл полистать сам справочник и ознакомиться с примерами функциональных особенностей, описываемых в конкретной статье.

Имя класса, имя пакета, доступность и флаги

Каждая справочная статья начинается с заголовка, состоящего из четырех частей. В нем указано имя класса, имя пакета и доступность класса. Кроме того, в заголовке могут быть указаны дополнительные флаги, которые описывают класс. Имя класса напечатано жирным шрифтом в левой верхней части заголовка. Имя пакета напечатано более мелким шрифтом в нижней левой части (под именем класса).

Верхняя правая часть заголовка определяет доступность класса; здесь указывается наиболее ранний выпуск, в котором содержался класс. Если класс был введен в Java 1.1, то в этой части заголовка будет запись «Java 1.1». Часть заголовка, в которой указывается доступность, также применяется для обозначения устаревшего класса (deprecated). Если класс был признан устаревшим, то здесь указывается, в каком выпуске это произошло. Например, в этой части может быть написано следующее: «Java 1.1; устарел в Java 1.2».

В нижнем правом углу заголовка может содержаться список флагов, которые описывают класс. Далее приведены возможные флаги и их значения:

проверяемое

Данный класс является проверяемым исключением. Это означает, что он расширяет `java.lang.Exception`, а не `java.lang.RuntimeException`. Другими словами, исключение должно быть объявлено в выражении `throws` любого метода, который может его сгенерировать.

клонируемый

Класс или родительский класс реализует `java.lang.Cloneable`.

коллекция

Класс или родительский класс реализует `java.util.Collection` или `java.util.Map`.

сравнимый

Класс или родительский класс реализует `java.lang.Comparable`.

ошибка

Класс расширяет `java.lang.Error`.

событие

Класс расширяет `java.util.EventObject`.

адаптер события

Класс или родительский класс реализует `java.util.EventListener`, а имя класса заканчивается на «Adapter».

слушатель события

Класс или родительский класс реализует `java.util.EventListener`.

запускаемый

Класс или родительский класс реализует `java.lang.Runnable`.

сериализуемый

Класс или родительский класс реализует `java.io.Serializable`.

непроверяемое

Данный класс является непроверяемым исключением; это означает, что он расширяет `java.lang.RuntimeException`. Поэтому такое исключение не нуждается в объявлении внутри выражения `throws` того метода, который может его сгенерировать.

Описание

За названием каждой справочной статьи следует краткое описание наиболее важных функциональных особенностей класса или интерфейса. Это описание может быть от двух предложений до нескольких абзацев.

Иерархия

Если класс или интерфейс имеет нетривиальную иерархию классов, то за описанием следует рисунок, который иллюстрирует иерархию и помогает понять класс в контексте этой иерархии. На диаграмме все классы изображены в виде прямоугольников (за исключением абстрактных классов, которые представлены в скошенных прямоугольниках, или параллелограммах). Интерфейсы изображены в виде закругленных прямоугольников. Текущий класс, являющийся предметом обсуждения, заключен в прямоугольник с более жирной линией. Прямоугольники соединены линиями: жирные линии обозначают связи на уровне «расширяет», а тонкие линии обозначают отношения на уровне «реализует». Иерархию, основанную на отношении «родительский класс/подкласс», следует читать слева направо в самом верхнем ряду прямоугольников в диаграмме (может быть и один ряд). Интерфейсы обычно расположены под классами, которые их реализуют, хотя в простых случаях интерфейс иногда располагается на той же строке, что и класс. Такое размещение приводит к более компактной диаграмме. Имейте в виду, что диаграмма иерархии показывает только родительские классы класса. Если у класса есть подклассы, то они перечисляются в конце справочной статьи по конкретному классу.

Синопсис

Наиболее важная часть каждой справочной статьи – это синопсис класса, который следует за заголовком и описанием. Синопсис класса в значительной степени похож на исходный код класса, за исключением того, что пропускаются тела методов и добавляются дополнительные аннотации. Если вы знаете синтаксис Java, то сможете прочесть синопсис класса.

Первая строка синопсиса содержит информацию о самом классе. Она начинается со списка модификаторов класса, таких как `public`, `abstract` и `final`. За этими модификаторами следует ключевое слово `class` или `interface`, а затем имя класса. За именем класса может следовать выражение `extends`, которое определяет родительский класс, и выражение `implements`, которое определяет любые интерфейсы, реализуемые классом.

За строкой с определением класса следует список полей и методов, представленных классом. Зная синтаксис Java, вы сможете легко понять эти строки. Для каждого члена (member) представлены модификаторы, тип и имя члена. В случае методов синопсис также включает в себя тип и имя каждого параметра метода и (необязательно) выражение `throws`, в котором перечислены исключения, генерируемые методом. Имена членов напечатаны жирным шрифтом, поэтому можно легко просмотреть

список членов и найти необходимый. Имена параметров метода приведены курсивом, означающим, что их не следует применять буквально.

Доступность члена и флаги

Каждое представление члена – это единая строка, которая определяет API для данного члена. В таком представлении применяется синтаксис Java, поэтому оно понятно любому Java-программисту. Есть еще дополнительная информация, ассоциированная с синопсисом каждого члена, однако она требует пояснений.

Вспомните, что каждый элемент справочника начинается с маленького раздела, включающего версию, в которой класс был впервые определен. В случае когда член введен в класс после первоначальной версии класса, версия, в которой член был введен, указывается мелким шрифтом слева от синопсиса члена. Например, если класс был впервые введен в Java 1.1, но имеет новый метод, добавленный в Java 1.2, то заголовок содержит строку «Java 1.1», а запись для нового члена предваряется числом «1.2». Кроме того, если член был признан устаревшим, этот факт обозначается с помощью знака «решетка» (#) слева от синопсиса члена.

Пространство справа от синопсиса члена задействовано для показа разнообразных флагов, предоставляющих дополнительную информацию о члене. Некоторые флаги указывают на дополнительные детали описания, которые не проявляются непосредственно в API члена. Другие флаги содержат информацию, специфичную для реализации. Эта информация может оказаться полезной для понимания класса и отладки вашего кода, однако знайте, что она может изменяться от реализации к реализации. Флаги, специфичные для реализации, базируются на реализации Java для Linux, выполненной Sun.

Следующие флаги могут стоять справа от синопсиса члена:

зависит от платформы (native)

Флаг, специфичный для реализации. Показывает, что метод реализован платформозависимым кодом (native). Хотя `native` является ключевым словом Java и может присутствовать в сигнатурах методов, это часть реализации метода, а не его спецификации. Поэтому данная информация представляется флагами члена, а не является частью его листинга. Этот флаг полезен в качестве подсказки об ожидаемой производительности метода.

синхронизирован (synchronized)

Флаг, специфичный для реализации. Показывает, что реализация метода объявлена как `synchronized`. Это означает, что метод фиксирует объект или класс перед выполнением. Подобно ключевому слову `native`, ключевое слово `synchronized` является частью реализации метода, а не частью его спецификации, поэтому оно присутствует как флаг, а не указывается в синопсисе метода. Данный флаг является удобной подсказкой о том, что метод, возможно, реализован как потокобезопасный (thread-safe).

Информация о том, является ли метод потокобезопасным или нет, – это часть спецификации метода, поэтому данная информация должна присутствовать в документации метода (хотя зачастую это не так). Однако существует большое количество различных способов сделать метод потокобезопасным, а объявление метода с ключевым словом `synchronized` – только одна из возможных реализаций. Другими словами, метод, который не имеет флага `synchronized`, все же может быть потокобезопасным.

Замещает: (Overrrides)

Этот флаг показывает, что метод класса замещает метод в одном из его родительских классов. Следом за флагом идет имя родительского класса, чей метод замещается. Это детали спецификации, а не реализации. Как мы увидим в следующем разделе, замещающие методы обычно сгруппированы в отдельный раздел синопсиса класса. Флаг `Замещает:` применяется только в случае, когда замещающий метод не представлен таким образом.

Реализует: (Implements)

Этот флаг показывает, что метод реализует метод интерфейса. За флагом следует имя реализуемого интерфейса. Это детали спецификации, а не реализации. Как мы увидим в следующем разделе, методы, реализующие интерфейс, обычно группируются в специальный раздел синопсиса класса. Флаг `Реализует:` применяется только для методов, не сгруппированных таким образом.

пустой

Этот флаг показывает, что реализация метода имеет пустое тело. Это подсказка программисту: возможно, метод необходимо заменить в подклассе.

константа

Флаг, специфичный для реализации. Показывает, что метод имеет ограниченную реализацию. Только методы с возвращаемым типом `void` могут быть действительно пустыми. Любой метод, объявленный как возвращающий значение, обязан иметь оператор `return`. Флаг «константа» показывает, что реализация метода пуста, исключая оператор `return`, возвращающий постоянное значение. Такой метод может иметь тело `return null;` или `return false;`. Как и флаг «пустой», данный флаг может показывать, что метод нуждается в замене.

по умолчанию:

Этот флаг применяется с методами доступа к свойствам, которые читают значение свойства (то есть с методами, чьи имена начинаются с «get» и которые не принимают аргументов). За флагом следует значение свойства по умолчанию. Строго говоря, значения свойства по умолчанию являются деталями спецификации. Однако на практике значения по умолчанию не всегда документированы. Необходимо быть начеку, поскольку значения по умолчанию могут изменяться от реализации к реализации.

Этот флаг имеют не все методы доступа к свойствам. Значение по умолчанию определяется динамической загрузкой рассматриваемого класса. С помощью конструктора без аргументов создается экземпляр класса, а последующий вызов метода определяет его возвращаемое значение. Данный метод можно применять только для классов, в которых доступна динамическая загрузка и создание экземпляра и которые имеют конструкторы без аргументов, поэтому значения, заданные по умолчанию, отображаются только для таких классов. Кроме того, при создании экземпляра класса с применением другого конструктора значения, заданные по умолчанию, могут различаться.

=

В полях с модификаторами `static final` за этим флагом следует постоянное значение поля. Представлены только константы примитивных типов, константы типа `String` и константы, имеющие значение `null`. Некоторые постоянные значения относятся к деталям спецификации, тогда как другие являются деталями реализации. Однако назначение символических констант – предоставить возможность писать код, не ссылаясь непосредственно на значение константы. Этот флаг слу-

жит подспорьем для понимания класса, но не полагайтесь на значения этих констант в ваших программах.

Функциональная группировка членов

В синопсисе класса члены не перечислены в строгом алфавитном порядке. Вместо этого они разбиты на функциональные группы и представлены в алфавитном порядке внутри каждой группы. Конструкторы, методы, поля и внутренние классы перечислены отдельно. Методы экземпляра хранятся отдельно от статических методов класса. Константы отделены от переменных полей. Открытые члены перечислены отдельно от защищенных. Группировка членов по категориям разбивает класс на более мелкие и понятные части. Таким образом, класс становится более простым для понимания. Кроме того, становится проще найти необходимый член.

Функциональные группы в синопсисе класса отделены одна от другой комментариями Java, такими как «// Открытые конструкторы», «// Внутренние классы» и «// Методы, реализующие DataInput». Существуют следующие функциональные категории (в порядке появления в синопсисе класса):

Конструкторы

Показывает конструкторы класса. Открытые и защищенные конструкторы показаны в отдельных подгруппах. Если класс вообще не определяет конструкторы, то компилятор Java добавляет конструктор по умолчанию, не имеющий аргументов. Если класс определяет только закрытые конструкторы, то его экземпляр не может быть создан, а специальная запись «Конструктор отсутствует» указывает на этот факт. Сначала перечисляются конструкторы, поскольку при работе с большинством классов прежде всего создаются их экземпляры путем вызова конструктора.

Константы

Показывает все константы, определяемые классом (то есть поля, объявленные как `static` и `final`). Открытые и защищенные константы показаны в отдельных подгруппах. Константы перечисляются в начале синопсиса класса, поскольку их значения часто применяются как допустимые значения для параметров методов и возвращаемых значений.

Внутренние классы

Группирует все внутренние классы и интерфейсы, определяемые классом или интерфейсом. Для каждого внутреннего класса присутствует однострочный синопсис. Кроме того, каждый внутренний класс имеет собственную запись в справочнике, включающую полный синопсис этого внутреннего класса. Подобно константам, внутренние классы перечисляются в начале синопсиса класса, поскольку они часто применяются другими членами класса.

Статические методы

Перечисляет статические методы класса, разбитые на подгруппы открытых статических методов и защищенных статических методов.

Методы регистрации слушателей событий

Перечисляет открытые методы экземпляра, которые регистрируют и удаляют объекты слушателей событий в классе. Имена этих методов начинаются со слов «add» и «remove» и заканчиваются на «Listener». Таким методам всегда передается объект `java.util.EventListener`. Методы обычно определяются парами, поэтому пары описываются вместе. Методы представлены в алфавитном порядке по имени события, а не по имени метода.

Методы доступа к свойствам

Перечисляет открытые методы экземпляра, устанавливающие или запрашивающие значение свойства или атрибута класса. Имена этих методов начинаются со слов «set», «get» и «is», а их сигнатуры соответствуют шаблонам, установленным в спецификации JavaBeans. Несмотря на то что соглашения об именовании и шаблоны сигнатур методов определены только для JavaBeans, классы и интерфейсы всей платформы Java определяют методы доступа к свойствам в соответствии с этими соглашениями и шаблонами. Взгляд на класс с точки зрения его свойств является мощным средством для понимания этого класса, поэтому методы свойств сгруппированы вместе. Методы доступа к свойствам перечисляются в алфавитном порядке по имени свойства, а не по имени метода. Это означает, что для того или иного свойства методы «set», «get» и «is» показаны вместе.

Открытые методы экземпляра

Содержит все открытые методы экземпляра, не сгруппированные где-либо еще.

Реализующие методы

Группирует методы, реализующие один и тот же интерфейс. На каждый интерфейс, реализуемый классом, приходится одна подгруппа. Методы, определенные в одном интерфейсе, почти всегда связаны друг с другом, поэтому это удобный способ функциональной группировки методов.

Если метод интерфейса является методом регистрации события или методом доступа к свойству, то он приводится в двух местах: в этой группе и в группе событий или свойств. Такая ситуация возникает не часто, но повторное перечисление оправдывается важностью и удобством всех способов функциональной группировки. Когда метод интерфейса приводится в группе событий или свойств, он сопровождается флагом «Реализует:», указывающим имя интерфейса, частью которого является этот метод.

Замещающие методы

Группирует методы, замещающие методы родительского класса, разбивая их на подгруппы по имени родительского класса. Это удобный способ группировки, поскольку он помогает понять, как класс изменяет поведение его родительских классов, заданное по умолчанию. Зачастую методы, замещающие один и тот же родительский класс, функционально связаны друг с другом.

Иногда метод, замещающий метод родительского класса, является также методом доступа к свойствам или (реже) методом регистрации события. В таких случаях метод группируется с методами свойств или событий и приводится с флагом, показывающим, какой метод родительского класса он замещает. Однако метод не перечисляется вместе с другими замещающими методами. Обратите внимание на это отличие от интерфейсных методов, которые могут перечисляться в обеих группах благодаря их тесной функциональной связи.

Защищенные методы экземпляра

Содержит все защищенные методы экземпляра, не сгруппированные где-либо еще.

Поля

Перечисляет все переменные поля класса, разбивая их на подгруппы для открытых и защищенных статических полей и открытых и защищенных полей экземпляра. Множество классов не определяют никаких открытых полей. Программисты, создающие объектно-ориентированные приложения, предпочитают не применять эти поля напрямую, а использовать методы доступа к свойствам, когда такие методы доступны.

Устаревшие члены

Устаревшие методы и поля группируются в самом низу синопсиса класса. Их применение не одобряется.

Перекрестные ссылки

За секцией синопсиса в справочнике могут следовать разделы перекрестных ссылок. Они обозначают другие связанные классы и методы, которые могут представлять интерес. Вот эти разделы:

Подклассы

Здесь перечислены подклассы класса, если они существуют.

Реализации

Здесь перечислены классы, реализующие интерфейс.

Передается методам

Здесь перечислены все методы и конструкторы, которым в качестве аргумента передается объект этого типа. Это удобно, когда при наличии объекта данного типа нужно определить, что с ним можно сделать.

Возвращается методами

Здесь перечислены все методы (но не конструкторы), возвращающие объект данного типа. Это удобно, когда вы хотите работать с объектом данного типа, но не знаете, как его получить.

Генерируется методами

Для классов проверяемых исключений здесь перечислены все методы и конструкторы, генерирующие исключения данного типа. Этот материал поможет определить, когда данное исключение или ошибка могут быть сгенерированы. Однако этот раздел основывается на типах исключений, перечисленных в выражениях `throws` методов и конструкторов. Подклассы `RuntimeException` и `Error` могут не перечисляться в выражениях `throws`, поэтому нет возможности создать полный набор перекрестных ссылок для методов, генерирующих непроверяемые исключения этих типов.

Экземпляры

Здесь перечислены все поля и константы этого типа. Данный раздел поможет определить, как получить объект данного типа.

Замечание по поводу имен классов

На всем протяжении справочника мы ссылаемся на классы либо только по имени класса, либо только по имени класса и имени пакета. Если всегда указывать имена пакетов, то синопсисы классов будут очень длинными и трудными для восприятия. С другой стороны, если бы имена пакетов никогда не применялись, то порой было бы трудно понять, на какой класс мы ссылаемся. Правила, регламентирующие указание имени пакета, достаточно сложны. Однако в первом приближении они могут быть обобщены так:

- Если имя класса само по себе неоднозначно, то указывается имя пакета.
- Если класс является частью пакета `java.lang` или применяется очень широко, например `java.io.Serializable`, то имя пакета опускается.
- Если рассматриваемый класс является частью текущего пакета и имеет элемент справочника в текущем разделе, то имя пакета опускается.

Глава 9

java.beans и java.beans.beancontext

В этой главе описаны пакеты java.beans и java.beans.beancontext. В java.beans определены базовые классы и интерфейсы, лежащие в основе компонентов JavaBeans, а в java.beans.beancontext — контейнеры JavaBeans.

Пакет java.beans

Java 1.1

Пакет java.beans содержит классы и интерфейсы для поддержки компонентов JavaBeans. Большинство из них используются программами управления компонентами (bean), а не самими компонентами. Эти классы и интерфейсы также используются или реализуются вспомогательными классами, с помощью которых разработчики компонентов предоставляют дополнительную информацию для инструментов управления компонентами.

Класс Beans определяет несколько широко используемых статических методов. Особенно важен метод instantiate(). Класс Introspector используется для получения информации о компоненте и его экспортируемых свойствах, событиях и методах. Для этого он использует класс FeatureDescriptor вместе с его классами-потомками. Пакет java.beans также определяет класс PropertyChangeEvent и интерфейс PropertyChangeListener, которые часто используются в библиотеках AWT и Swing для рассылки сообщений при изменении связанного свойства компонента GUI.

В Java 1.4 добавлено несколько новых классов для поддержки механизма сохранения и восстановления (persistence mechanism) JavaBeans. См. также XMLEncoder.

Полностью введение в модель компонентов JavaBeans представлено в главе 6.

Интерфейсы

```
public interface AppletInitializer;
public interface BeanInfo;
public interface Customizer;
public interface DesignMode;
public interface ExceptionListener;
public interface PropertyEditor;
public interface Visibility;
```

События

```
public class PropertyChangeEvent extends java.util.EventObject;
```

Слушатели событий

```
public interface PropertyChangeListener extends java.util.EventListener;
public interface VetoableChangeListener extends java.util.EventListener;
```

Другие классы

```
public class Beans;
public class Encoder;
    └ public class XMLEncoder extends Encoder;
public class EventHandler Implements java.lang.reflect.InvocationHandler;
public class FeatureDescriptor;
    └ public class BeanDescriptor extends FeatureDescriptor;
    └ public class EventSetDescriptor extends FeatureDescriptor;
    └ public class MethodDescriptor extends FeatureDescriptor;
    └ public class ParameterDescriptor extends FeatureDescriptor;
    └ public class PropertyDescriptor extends FeatureDescriptor;
        └ public class IndexedPropertyDescriptor extends PropertyDescriptor;
public class Introspector;
public abstract class PersistenceDelegate;
    └ public class DefaultPersistenceDelegate extends PersistenceDelegate;
public class PropertyChangeListenerProxy extends java.util.EventListenerProxy
                        implements PropertyChangeListener;
public class PropertyChangeSupport Implements Serializable;
public class PropertyEditorManager;
public class PropertyEditorSupport Implements PropertyEditor;
public class SimpleBeanInfo Implements BeanInfo;
public class Statement;
    └ public class Expression extends Statement;
public class VetoableChangeListenerProxy extends java.util.EventListenerProxy
                        implements VetoableChangeListener;
public class VetoableChangeSupport Implements Serializable;
public class XMLDecoder;
```

Исключения

```
public class IntrospectionException extends Exception;
public class PropertyVetoException extends Exception;
```

AppletInitializer Java 1.2

java.beans

Этот интерфейс определяет общие методы для инициализации нового объекта `Applet`. `AppletInitializer` можно передать методу `Beans.instantiate()`, который позволяет при создании компонента-апплета правильно инициализировать. Метод `initialize()` связывает объект типа `Applet` с соответствующими объектами `AppletContext` и `AppletStub`. Для этого нужно поместить апплет в соответствующий контейнер, а затем вызвать метод `init()`. Метод `activate()` активирует апплет, вызывая его метод `start()`. Этот интерфейс обычно применяется разработчиками контекста компонентов. Разработчикам приложений могут потребоваться объекты `AppletInitializer`, но им следует избегать прямых вызовов или реализации методов этого интерфейса.

```
public interface AppletInitializer {
  // Открытые методы экземпляра
  public abstract void activate(java.applet.Applet newApplet);
  public abstract void initialize (java.applet.Applet newAppletBean,
     java.beans.beancontext.BeanContext bCtxt);
}
```

Передается методам: `Beans.instantiate()`

BeanDescriptor Java 1.1

java.beans

Объект `BeanDescriptor` — это потомок `FeatureDescriptor`, описывающий компонент JavaBeans. Класс `BeanInfo` может создавать и инициализировать объект `BeanDescriptor`, описывающий компонент. Обычно `BeanDescriptor` используется только средами разработки и подобными программами. Чтобы создать объект `BeanDescriptor`, укажите класс компонента и (необязательно) класс объекта `Customizer` этого компонента. Можно также использовать методы класса `FeatureDescriptor` для получения дополнительной информации о компоненте.


```
public class BeanDescriptor extends FeatureDescriptor {
// Открытые конструкторы
   public BeanDescriptor(Class beanClass);
   public BeanDescriptor(Class beanClass, Class customizerClass);
// Открытые методы экземпляра
   public Class getBeanClass();
   public Class getCustomizerClass();
}
```

Возвращается методами:

`BeanInfo.getBeanDescriptor(), SimpleBeanInfo.getBeanDescriptor()`

BeanInfo Java 1.1

java.beans

В интерфейсе `BeanInfo` определены методы, с помощью которых можно осуществлять экспорт информации о компоненте JavaBeans. Основную информацию о компоненте можно получить с помощью класса `Introspector`. Чтобы сделать компонент более удобным для использования, можно реализовать этот интерфейс и предоставить дополнительную информацию (иконки и строки с описанием каждого свойства, события и метода). Обратите внимание, что при разработке компонента нужно реализовать методы этого интерфейса в отдельном классе. Как правило, эти методы применяются только средами разработки и подобными программами. Методы `getBeanDescriptor()`, `getEventSetDescriptors()`, `getPropertyDescriptors()` и `getMethodDescriptors()` должны возвращать либо соответствующий объект-описатель компонента, либо `null`, если не предоставлены объекты-дескрипторы для компонента, наборов событий, свойств или методов соответственно. Методы `getDefaultEventIndex()` и `getDefaultPropertyIndex()` возвращают значения, определяющие событие или свойство по умолчанию, которое может представлять наибольший интерес для программиста, использующего компо-

нент. Эти методы должны возвращать -1, если значение по умолчанию не определено. Метод getIcon() должен возвращать изображение для компонента, которое будет отображено в палитре компонентов или меню. Этому методу в качестве аргумента должна передаваться одна из четырех констант, определенных в классе; данные константы определяют тип и размер иконки. Если требуемая иконка не доступна, getIcon() должен возвращать null.

Класс BeanInfo может возвращать -1, если требуемая информация не доступна. В таком случае базовую информацию, полученную путем интроспекции (introspection) компонента, может предоставить класс Introspector. Простейшую реализацию этого интерфейса, пригодную для наследования, можно найти в классе SimpleBeanInfo.

```
public interface BeanInfo {
// Открытые константы
   public static final int ICON_COLOR_16x16;                    // =1
   public static final int ICON_COLOR_32x32;                    // =2
   public static final int ICON_MONO_16x16;                     // =3
   public static final int ICON_MONO_32x32;                     // =4
// Методы доступа к свойствам (по имени свойства)
   public abstract BeanInfo[] getAdditionalBeanInfo();
   public abstract BeanDescriptor getBeanDescriptor();
   public abstract int getDefaultEventIndex();
   public abstract int getDefaultPropertyIndex();
   public abstract EventSetDescriptor[] getEventSetDescriptors();
   public abstract MethodDescriptor[] getMethodDescriptors();
   public abstract PropertyDescriptor[] getPropertyDescriptors();
// Открытые методы экземпляра
   public abstract java.awt.Image getIcon(int iconKind);
}
```

Реализация:
SimpleBeanInfo, java.beans.beancontext.BeanContextServiceProviderBeanInfo

Возвращается методами: BeanInfo.getAdditionalBeanInfo(), Introspector.getBeanInfo(), SimpleBeanInfo.getAdditionalBeanInfo(), java.beans.beancontext.BeanContextServiceProviderBeanInfo.getServicesBeanInfo()

Beans Java 1.1

java.beans

Класс Beans не предназначен для создания экземпляров, его статические методы предоставляют функциональность компонента JavaBeans. Метод instantiate() создает экземпляр компонента. Указанное имя компонента должно быть либо именем сериализованного файла компонента, либо именем файла класса компонента; его интерпретация зависит от того, какой объект ClassLoader указан.

Методами setDesignTime() и isDesignTime() соответственно устанавливается и запрашивается флаг, определяющий, используется ли компонент средой разработки. Аналогично setGuiAvailable() и isGuiAvailable() устанавливают или запрашивают флаг, определяющий, доступен ли GUI в среде, в которой работает виртуальная машина Java. Обратите внимание: ненадежный апплет не может вызывать методы setDesignTime() и setGuiAvailable().

Метод isInstanceOf() заменяет оператор Java instanceof при работе с компонентами. В настоящее время этот метод действует так же, как instanceof, но в будущем он может применяться к наборам объектов Java, имеющим различное представление о

компоненте. Аналогично метод getInstanceOf() заменяет оператор приведения типа. Этот метод приводит компонент к типу родительского класса или интерфейса; в настоящее время он работает как оператор приведения, но в дальнейшем он будет совместим с многоклассовыми компонентами.

```
public class Beans {
// Открытые конструкторы
   public Beans();
// Открытые методы класса
   public static Object getInstanceOf(Object bean, Class targetType);
   public static Object instantiate(ClassLoader cls, String beanName)
        throws java.io.IOException, ClassNotFoundException;
1.2 public static Object instantiate(ClassLoader cls, String beanName,
                                     java.beans.beancontext.BeanContext beanContext)
        throws java.io.IOException, ClassNotFoundException;
1.2 public static Object instantiate(ClassLoader cls, String beanName,
                                     java.beans.beancontext.BeanContext beanContext,
                                     AppletInitializer initializer)
        throws java.io.IOException, ClassNotFoundException;
   public static boolean isDesignTime();
   public static boolean isGuiAvailable();
   public static boolean isInstanceOf(Object bean, Class targetType);
   public static void setDesignTime(boolean isDesignTime) throws SecurityException;
   public static void setGuiAvailable(boolean isGuiAvailable) throws SecurityException;
}
```

Customizer

Java 1.1

java.beans

Интерфейс Customizer содержит методы, которые должны быть реализованы во всех классах, настраивающих компонент JavaBeans. Такие классы должны реализовать этот интерфейс, а также наследовать класс java.awt.Component и иметь конструктор без аргументов, чтобы среда разработки могла создать экземпляр этого класса.

Классы типа Customizer обычно применяются в сложных компонентах и дают возможность пользователю легко настраивать компонент. Они являются альтернативой простому списку свойств и значений. Если для компонента определен класс Customizer, то этот класс должен быть связан с компонентом с помощью объекта BeanDescriptor, возвращаемого классом BeanInfo этого компонента. Обратите внимание, что класс Customizer создается на этапе разработки компонента, а его объект создается и применяется только инструментальными программами.

После того как создан экземпляр класса Customizer, вызывается (один раз) метод setObject() для указания настраиваемого объекта. Можно вызвать методы addPropertyChangeListener() и removePropertyChangeListener() соответственно для регистрации и отмены регистрации объектов типа PropertyChangeListener. Объект Customizer должен посылать событие PropertyChangeEvent всем зарегистрированным слушателям (listener) каждый раз, когда он меняет свойства своего компонента.

```
public interface Customizer {
// Методы регистрации событий (по имени события)
   public abstract void addPropertyChangeListener(PropertyChangeListener listener);
   public abstract void removePropertyChangeListener(PropertyChangeListener listener);
// Открытые методы экземпляра
   public abstract void setObject(Object bean);}
```

DefaultPersistenceDelegate Java 1.4

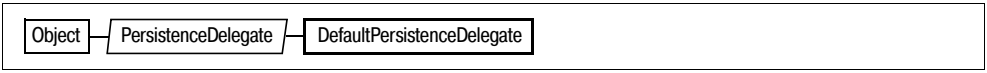
java.beans

Этот класс является делегатом механизма сохранения постоянства и восстановления (persistence delegate) и используется для кодирования экземпляров классов, которые не имеют собственных делегатов, определенных в BeanInfo, или зарегистрированных пользовательских делегатов. Для создания объекта класса DefaultPersistenceDelegate механизм постоянства JavaBeans будет вызывать конструктор без параметров. Полученный в результате делегат механизма восстановления предполагает, что в классе есть конструктор без параметров и что его состояние может быть полностью описано с помощью методов доступа к свойствам (property accessor methods). Эти методы должны соответствовать стандартному соглашению об именах get/set, принятому в JavaBeans. При работе с грамотно написанными классами JavaBeans приложению не нужно создавать экземпляры класса DefaultPersistenceDelegate, потому что это выполняется автоматически механизмом сохранения и восстановления компонента.

Экземпляры этого класса могут служить делегатами механизма восстановления для тех компонентов, у которых нет конструктора без параметров; при этом конструктору передается значение свойств «только для чтения». В этом случае можно создать экземпляр класса DefaultPersistenceDelegate, передав конструктору массив строк с именами его аргументов «только для чтения». Конечно же, в классе компонента должны быть определены методы для чтения этих свойств (property getter methods). Эти методы должны соответствовать соглашению об именах, принятому в JavaBeans, или должны быть определены в классе BeanInfo. В качестве примера рассмотрим класс java.awt.Color. (Документацию по нему можно найти в книге «Java Foundation Classes in a Nutshell» [O'Reilly].) Если механизм сохранения состояния и восстановления JavaBeans еще не определил делегат для этого класса, это можно сделать вручную следующим образом:

```
PersistenceDelegate delegate =
        new DefaultPersistenceDelegate(new String[]{"red", "green","blue"});,
```

Такая схема работает, поскольку в классе Color определен конструктор с тремя параметрами, соответствующими возвращаемым значениям методов чтения свойств getRed(), getGreen() и getBlue().




```
public class DefaultPersistenceDelegate extends PersistenceDelegate {
// Открытые конструкторы
   public DefaultPersistenceDelegate();
   public DefaultPersistenceDelegate(String[] constructorPropertyNames);
// Защищенные методы, замещающие  PersistenceDelegate
   protected void initialize(Class type, Object oldInstance, Object newInstance, Encoder out);
   protected Expression instantiate(Object oldInstance, Encoder out);
   protected boolean mutatesTo(Object oldInstance, Object newInstance);
}
```

DesignMode Java 1.2

java.beans

В этом интерфейсе определено единственное булево свойство designTime, которое определяет, действует ли компонент в среде разработки или в самостоятельном приложении или апплете. Как правило, этот интерфейс реализуется в контексте контейнера или компонента, так что потомки компонента могут обращаться к этому свойству.

```
public interface DesignMode {
// Открытые константы
   public static final String PROPERTYNAME;  // ="designTime"
// Открытые методы экземпляра
   public abstract boolean isDesignTime();
   public abstract void setDesignTime(boolean designTime);
}
```

Реализации: java.beans.beancontext.BeanContext

Encoder Java 1.4

java.beans

Этот класс лежит в основе механизма сохранения и восстановления состояния JavaBeans; задача объекта Encoder – кодировать и возвращать объекты Expression и Statement, созданные объектом PersistenceDelegate и описывающие состояние компонента. Хотя Encoder и не является абстрактным классом, он никогда не применяется непосредственно, поскольку сам по себе не производит полезного кодирования. Подробности представлены в разделе «XMLEncoder».

```
public class Encoder {
// Открытые конструкторы
   public Encoder();
// Открытые методы экземпляра
   public Object get(Object oldInstance);
   public ExceptionListener getExceptionListener();
   public PersistenceDelegate getPersistenceDelegate(Class type);
   public Object remove(Object oldInstance);
   public void setExceptionListener(ExceptionListener exceptionListener);
   public void setPersistenceDelegate(Class type,
        PersistenceDelegate persistenceDelegate);
   public void writeExpression(Expression oldExp);
   public void writeStatement(java.beans.Statement oldStm);
// Защищенные методы экземпляра
   protected void writeObject(Object o);
}
```

Подклассы: XMLEncoder

Передается методам: DefaultPersistenceDelegate.{initialize(), instantiate()}, PersistenceDelegate.{initialize(), instantiate(), writeObject()}

EventHandler

Java 1.4

java.beans

Этот класс использует Proxy и другие классы из пакета java.lang.reflect для создания объектов, которые могут выступать в роли простых слушателей событий. Такие слушатели могут легко сериализоваться механизмом сохранения и восстановления состояния JavaBeans. При этом применяется способ, недоступный для слушателей, реализованных как анонимные классы.

Как правило, приложения не работают непосредственно с объектами EventHandler и конструктором EventHandler(). Вместо этого они используют статические методы create(), чтобы получить объект, в котором реализован требуемый интерфейс слушателя событий. Слушатель, созданный методом EventHandler.create(), при активации осуществляет один вызов определенного метода. Необходимо указать имя, объект и (необязательно) параметр этого метода. В качестве параметра может быть передан объект события, активизировавшего обработчик, или свойство этого объекта. Метод create() принимает следующие параметры:

listenerInterface

Объект типа Class, представляющий слушателя EventListener, который будет возвращен. Например: PropertyChangeListener.class.

target

Объект, метод которого будет вызван полученным обработчиком событий.

action

Имя вызываемого метода объекта *target*. В качестве этого параметра может быть имя свойства объекта *target*; в этом случае вызывается метод для записи этого свойства. Поэтому вместо того, чтобы указывать имя метода «setX()» в качестве параметра *action*, можно указать имя свойства «x».

eventPropertyName

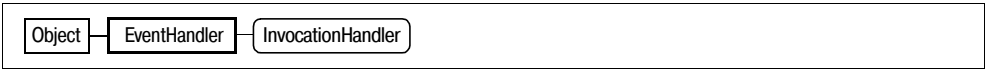
Этот необязательный аргумент определяет имя свойства объекта EventObject (или любого другого объекта, переданного методу слушателя), значение которого будет передано методу *action* объекта *target*. Если используется create() с тремя параметрами или в качестве eventPropertyName передается null, то в качестве параметра метода *action* будет передан EventHandler либо не будет передано никаких параметров.

Если параметр *eventPropertyName* имеет значение «x», то EventHandle сначала ищет в объекте, переданном методу слушателя (как правило, это EventObject), метод чтения getX(), не принимающий аргументов. Если такого метода нет, проводится поиск метода чтения «isX», не принимающего аргументов. Если такой метод также не найден, проводится поиск метода «x», не принимающего аргументов.

В качестве аргумента *eventPropertyName* можно указать несколько свойств различного уровня, разделенных точками. Например, значение аргумента *eventPropertyName* «propertyName.toLowerCase» будет преобразовано в вызовы методов getPropertyName().toLowerCase() объекта события. Возвращенное значение будет передано методу *action* в качестве параметра.

listenerMethodName

Этот аргумент определяет имя метода в *listenerInterface*, который должен быть реализован в возвращаемом объекте. Если указан null или используется метод create() с тремя или четырьмя параметрами, то все методы возвращаемого объекта будут вызывать метод *action* объекта *target*.




```
public class EventHandler Implements java.lang.reflect.InvocationHandler {
// Открытые конструкторы
   public EventHandler(Object target, String action, String eventPropertyName,
                       String listenerMethodName);
// Открытые методы класса
   public static Object create(Class listenerInterface, Object target, String action);
   public static Object create(Class listenerInterface, Object target,
        String action, String eventPropertyName);
   public static Object create(Class listenerInterface, Object target, String action,
                               String eventPropertyName, String listenerMethodName);
// Открытые методы экземпляра
   public String getAction();
   public String getEventPropertyName();
   public String getListenerMethodName();
   public Object getTarget();
// Методы, реализующие InvocationHandler
   public Object invoke(Object proxy, java.lang.reflect.Method method, Object[] arguments);
}
```

EventSetDescriptor

Java 1.1

java.beans

Класс EventSetDescriptor – это потомок FeatureDescriptor. Он описывает определенный набор событий, поддерживаемых компонентом JavaBeans. Набор событий соответствует методу или методам, поддерживаемым определенным интерфейсом EventListener. Класс BeanInfo может создавать объекты типа EventSetDescriptor для описания наборов событий, поддерживаемых компонентом. Как правило, методы объектов EventSetDescriptor используются только средами разработки и аналогичными инструментами, чтобы получить описательную информацию о наборе событий.

Чтобы создать объект типа EventSetDescriptor, нужно указать класс компонента, который поддерживает набор событий, базовое имя набора событий, класс интерфейса EventListener, соответствующий набору событий, и методы этого интерфейса, которые вызываются, когда происходит какое-нибудь событие из этого набора. Можно также определить методы класса компонента, которые добавляют и удаляют объекты EventListener. В различных конструкторах можно указывать методы по имени, передавая их как объекты java.lang.reflect.Method или MethodDescriptor. В Java 1.4 и последующих версиях можно также указать метод «get listeners», который будет использоваться для запроса списка зарегистрированных слушателей набора событий.

Когда создан объект EventSetDescriptor, нужно с помощью метода setUnicast() указать, представляет ли он однонаправленное (unicast) событие, а с помощью метода setInDefaultEventSet() указать, должен ли набор событий рассматриваться как набор событий по умолчанию для сред разработки. Методы родительского класса FeatureDescriptor позволяют получить дополнительную информацию о свойстве, которое будет указано.

```
public class EventSetDescriptor extends FeatureDescriptor {
// Открытые конструкторы
   public EventSetDescriptor(Class sourceClass, String eventSetName, Class listenerType,
                             String listenerMethodName) throws IntrospectionException;
  public EventSetDescriptor(String eventSetName, Class listenerType, java.lang.reflect.Method[]
                             listenerMethods, java.lang.reflect.Method addListenerMethod,
                             java.lang.reflect.Method removeListenerMethod) throws
                             IntrospectionException;
   public EventSetDescriptor(String eventSetName, Class listenerType,
                             MethodDescriptor[] listenerMethodDescriptors,
                             java.lang.reflect.Method addListenerMethod,
                             java.lang.reflect.Method removeListenerMethod) throws
                             IntrospectionException;
1.4 public EventSetDescriptor(String eventSetName, Class listenerType, java.lang.reflect.Method[]
                             listenerMethods, java.lang.reflect.Method addListenerMethod,
                             java.lang.reflect.Method removeListenerMethod,
                             java.lang.reflect.Method getListenerMethod)
                             throws IntrospectionException;
   public EventSetDescriptor(Class sourceClass, String eventSetName, Class listenerType,
                             String[] listenerMethodNames, String addListenerMethodName,
                             String removeListenerMethodName) throws IntrospectionException;
1.4 public EventSetDescriptor(Class sourceClass, String eventSetName, Class listenerType,
                             String[] listenerMethodNames String addListenerMethodName,
                             String removeListenerMethodName, String getListenerMethodName)
                             throws IntrospectionException;
// Методы доступа к свойствам (по имени свойства)
   public java.lang.reflect.Method getAddListenerMethod();
1.4 public java.lang.reflect.Method getGetListenerMethod();
   public boolean isInDefaultEventSet();
   public void setInDefaultEventSet(boolean inDefaultEventSet);
   public MethodDescriptor[] getListenerMethodDescriptors();
   public java.lang.reflect.Method[] getListenerMethods();
   public Class getListenerType();
   public java.lang.reflect.Method getRemoveListenerMethod();
   public boolean isUnicast();
   public void setUnicast(boolean unicast);
}
```

Возвращается методами:

`BeanInfo.getEventSetDescriptors()`, `SimpleBeanInfo.getEventSetDescriptors()`

ExceptionListener

Java 1.4

java.beans

Этот интерфейс реализуется объектами, которым необходимо получать уведомления об исключениях, возникающих во время процесса кодирования или декодирования, проводимого классом `Encoder` или `XMLDecoder`. Когда возникает исключение, вызывается метод `exceptionThrown()`.

```
public interface ExceptionListener {
// Открытые методы экземпляра
   public abstract void exceptionThrown(Exception e);
}
```

Передается методам:
`Encoder.setExceptionListener()`, `XMLDecoder.{setExceptionListener()`, `XMLDecoder()}`

Возвращается методами:
`Encoder.getExceptionListener()`, `XMLDecoder.getExceptionListener()`

Expression Java 1.4

java.beans

Этот класс является потомком класса `Statement` и заключает в себе вызов метода, который возвращает значение. Для вызова `Expression` нужно использовать не унаследованный метод `execute()`, а метод `getValue()`, который вызывает вышеописанный метод и возвращает значение, возвращенное этим методом. `getValue()` сохраняет это значение, поэтому базовый метод запускается только при первом вызове `getValue()`. Можно указать некоторое значение, передав его конструктору с четырьмя аргументами или методу `setValue()`. Это выполняется для повышения эффективности при кодировании метода, у которого известно возвращаемое значение. О том, как определить объект, имя метода и массив аргументов, рассказано в разделе «Statement».

Для класса `Expression` определены два специальных имени метода. Чтобы вызвать конструктор, в качестве объекта `target` укажите нужный класс, а в качестве имени метода используйте «`new`». Чтобы обратиться к элементу массива, нужно указать массив в качестве объекта `target`, а в качестве имени метода использовать «`get`» и передать методу единственный аргумент – индекс элемента.


```
public class Expression extends java.beans.Statement {
// Открытые конструкторы
   public Expression(Object target, String methodName, Object[] arguments);
   public Expression(Object value, Object target, String methodName, Object[] arguments);
// Открытые методы экземпляра
   public Object getValue() throws Exception;
   public void setValue(Object value);
// Открытые методы, замещающие Statement
   public String toString();
}
```

Передается методам: `Encoder.writeExpression()`, `XMLEncoder.writeExpression()`

Возвращается методами:
`DefaultPersistenceDelegate.instantiate()`, `PersistenceDelegate.instantiate()`

FeatureDescriptor Java 1.1

java.beans

`FeatureDescriptor` – это базовый класс для `MethodDescriptor`, `PropertyDescriptor` и для других классов, используемых механизмом интроспекции (introspection) JavaBeans. Он предоставляет базовую информацию об определенном элементе компонента (например, методе, свойстве или событии). Как правило, методы, начинающиеся с `get` или `is`, применяются средами разработки и другими инструментальными програм-

мами для запроса элементов компонента, а с помощью методов set авторы компонента могут определить информацию о нем.

setName() определяет регионо-независимое внутреннее (программное) имя элемента; setDisplayName() определяет локализованное имя, удобное для восприятия; setShortDescription() определяет короткую локализованную строку (длиной около 40 символов), описывающую данный элемент. Локализованному имени и краткому описанию по умолчанию присваивается значение внутреннего имени. С помощью методов setExpert() и setHidden() можно разрешить использовать данный элемент только экспертам и средам разработки и скрыть его от пользователей среды. Наконец, метод setValue() может связать произвольно указанное значение с данным элементом.

```
public class FeatureDescriptor {
// Открытые конструкторы
   public FeatureDescriptor();
// Методы доступа к свойствам (по имени свойства)
   public String getDisplayName();                          // по умолчанию:null
   public void setDisplayName(String displayName);
   public boolean isExpert();                               // по умолчанию:false
   public void setExpert(boolean expert);
   public boolean isHidden();                               // по умолчанию:false
   public void setHidden(boolean hidden);
   public String getName();                                 // по умолчанию:null
   public void setName(String name);
1.2 public boolean isPreferred();                           // по умолчанию:false
1.2 public void setPreferred(boolean preferred);
   public String getShortDescription();                     // по умолчанию:null
   public void setShortDescription(String text);
// Открытые методы экземпляра
   public java.util.Enumeration attributeNames();
   public Object getValue(String attributeName);
   public void setValue(String attributeName, Object value);
}
```

Подклассы: BeanDescriptor, EventSetDescriptor, MethodDescriptor, ParameterDescriptor, PropertyDescriptor

IndexedPropertyDescriptor Java 1.1

java.beans

Объект IndexedPropertyDescriptor — это потомок PropertyDescriptor, описывающий свойство компонента, являющееся массивом или ведущее себя как массив. Класс BeanInfo может создавать и инициализировать объекты IndexedPropertyDescriptor, описывающие индексированные свойства, которые поддерживаются компонентом. Как правило, только среды разработки и аналогичные инструментальные программы используют объекты-дескрипторы для получения информации об индексированных свойствах.

Чтобы создать IndexedPropertyDescriptor, нужно указать имя индексированного свойства и объект Class для компонента. Если стандартные соглашения для имен методов доступа к свойствам не соблюдены, такие методы можно указать либо по имени, либо как объект класса java.lang.reflect.Method. Как только создан объект IndexedPropertyDescriptor, можно с помощью методов классов PropertyDescriptor и FeatureDescriptor предоставить дополнительную информацию об индексированном свойстве.

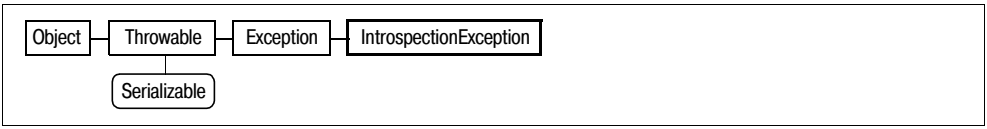

```
public class IndexedPropertyDescriptor extends PropertyDescriptor {
// Открытые конструкторы
   public IndexedPropertyDescriptor(String propertyName, Class beanClass)
        throws IntrospectionException;
   public IndexedPropertyDescriptor(String propertyName, java.lang.reflect.Method getter,
                                    java.lang.reflect.Method setter, java.lang.reflect.Method
                                    indexedGetter, java.lang.reflect.Method indexedSetter)
                                    throws IntrospectionException;
   public IndexedPropertyDescriptor(String propertyName, Class beanClass, String getterName,
                                    String setterName String indexedGetterName, String
                                    indexedSetterName) throws IntrospectionException;
// Открытые методы экземпляра
   public Class getIndexedPropertyType();
   public java.lang.reflect.Method getIndexedReadMethod();
   public java.lang.reflect.Method getIndexedWriteMethod();
1.2 public void setIndexedReadMethod(java.lang.reflect.Method getter)
        throws IntrospectionException;
1.2 public void setIndexedWriteMethod(java.lang.reflect.Method setter)
        throws IntrospectionException;
// Открытые методы, замещающие  PropertyDescriptor
1.4 public boolean equals(Object obj);
}
```

IntrospectionException

Java 1.1

java.beans ***сериализуемое, проверяемое***

Исключение IntrospectionException оповещает о том, что интроспекция компонента JavaBeans не может завершиться. Обычно это свидетельствует о наличии ошибки в определении компонента или связанного с ним класса BeanInfo.




```
public class IntrospectionException extends Exception {
// Открытые конструкторы
   public IntrospectionException(String mess);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

Introspector

Java 1.1

java.beans

Экземпляры класса Introspector никогда не создаются. Статические методы getBeanInfo() этого класса позволяют получить информацию о компоненте JavaBeans и используются, как правило, только средами разработки и аналогичными программами. Метод getBeanInfo() сначала ищет класс BeanInfo для указанного класса компо-

нента. Если класс называется x, метод ищет класс BeanInfo с именем xBeanInfo в текущем пакете, а затем во всех пакетах по пути поиска BeanInfo.

Если не найден класс BeanInfo или найденный класс не предоставляет полной информации о свойствах, событиях и методах компонента, метод getBeanInfo() проводит интроспекцию класса компонента, используя пакет java.lang.reflect, и получает недостающую информацию. Если класс BeanInfo предоставляет подробное описание компонента, то getBeanInfo() считает эту информацию окончательной. Если информация получена путем интроспекции, то проверяется по очереди BeanInfo из каждого родительского класса компонента. При вызове getBeanInfo() можно также указать второй аргумент типа Class, определяющий класс, начиная с которого (и двигаясь вверх) getBeanInfo() не проводит интроспекцию.

```
public class Introspector {
// Конструктор отсутствует
// Открытые константы
1.2 public static final int IGNORE_ALL_BEANINFO;                    // =3
1.2 public static final int IGNORE_IMMEDIATE_BEANINFO;              // =2
1.2 public static final int USE_ALL_BEANINFO;                       // =1
// Открытые методы класса
   public static String decapitalize(String name);
1.2 public static void flushCaches();
1.2 public static void flushFromCaches(Class clz);
   public static BeanInfo getBeanInfo(Class beanClass) throws IntrospectionException;
1.2 public static BeanInfo getBeanInfo(Class beanClass, int flags)
        throws IntrospectionException;
   public static BeanInfo getBeanInfo(Class beanClass, Class stopClass)
        throws IntrospectionException;
   public static String[] getBeanInfoSearchPath();                  // синхронизирован
   public static void setBeanInfoSearchPath(String[] path);         // синхронизирован
}
```

MethodDescriptor

Java 1.1

java.beans

Объект MethodDescriptor — это потомок FeatureDescriptor, описывающий один из методов, поддерживаемых компонентом JavaBeans. Класс BeanInfo может создавать объекты типа MethodDescriptor, описывающие методы, экспортируемые компонентом. Хотя класс BeanInfo создает объекты MethodDescriptor, их, как правило, используют только среды разработки и подобные программы для получения информации о компоненте.

При создании MethodDescriptor укажите объект java.lang.reflect.Method для требуемого метода; можно также указать массив объектов ParameterDescriptor, описывающих параметры метода. Когда объект MethodDescriptor создан, с помощью методов класса FeatureDescriptor можно предоставить дополнительную информацию о каждом методе.


```
public class MethodDescriptor extends FeatureDescriptor {
// Открытые конструкторы
   public MethodDescriptor(java.lang.reflect.Method method);
   public MethodDescriptor(java.lang.reflect.Method method,
        ParameterDescriptor[] parameterDescriptors);
// Открытые методы экземпляра
```

```
   public java.lang.reflect.Method getMethod();
   public ParameterDescriptor[] getParameterDescriptors();
}
```

Передается методам: EventSetDescriptor.EventSetDescriptor()

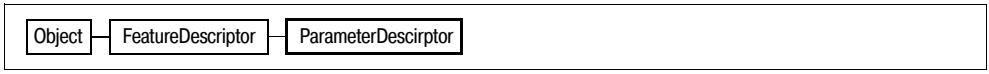
Возвращается методами: BeanInfo.getMethodDescriptors(), EventSetDescriptor.getListenerMethodDescriptors(), SimpleBeanInfo.getMethodDescriptors()

ParameterDescriptor Java 1.1

java.beans

Класс ParameterDescriptor является потомком класса FeatureDescriptor, описывающим аргумент или параметр метода компонента JavaBeans. Класс BeanInfo может создавать объекты ParameterDescriptor, которые описывают параметры методов, экспортируемых компонентом. Хотя класс BeanInfo создает и инициализирует объекты ParameterDescriptor, они, как привило, используются средами разработки и подобными программами для получения информации о параметрах методов, которые поддерживает компонент.

Класс ParameterDescriptor является простейшим подклассом FeatureDescriptor и не определяет новых методов. Поэтому, чтобы предоставить информацию о параметрах, используйте методы класса FeatureDescriptor.




```
public class ParameterDescriptor extends FeatureDescriptor {
// Открытые конструкторы
   public ParameterDescriptor();
}
```

Передается методам: MethodDescriptor.MethodDescriptor()

Возвращается методами: MethodDescriptor.getParameterDescriptors()

PersistenceDelegate Java 1.4

java.beans

Класс PersistenceDelegate играет главную роль в механизме сохранения и восстановления состояния JavaBeans. Он определяет текущее состояние компонента и описывает его на языке открытого API этого компонента в виде последовательных вызовов конструкторов и методов (представленных объектами Statement и Expression), которые могут в дальнейшем использоваться для восстановления сохраненного состояния компонента.

writeObject() является ключевым методом: Encoder передает объект методу writeObject() соответствующего делегата PersistenceDelegate. Последовательность вызовов конструкторов и методов, необходимая для восстановления, определяется объектом PersistenceDelegate; он также должен сообщить об этих методах кодировщику (Encoder), передав объекты Statement и Expression методам writeStatement() и writeExpression() кодировщика. Затем Encoder может закодировать объекты Statement и Expression в определенный выходной формат, например в виде документа XML. Обратите внимание на то, что writeObject() – это единственный открытый метод класса PersistenceDe-

legate. Он реализован на основе трех защищенных (protected) методов, которые могут замещаться в классах-потомках, если нужно изменить поведение механизма сохранения и восстановления состояния.

В механизме сохранения и восстановления состояния JavaBeans предусмотрены делегаты для компонентов AWT и Swing GUI и для всех типов (таких как цвета и шрифты), используемых в этих компонентах для свойств. Класс DefaultPersistenceDelegate работает с любым компонентом, который может выражать свое конечное состояние через аргументы конструкторов и методы доступа к свойствам. Если приходится использовать механизм сохранения и восстановления состояния JavaBeans для сериализации объектов, не соблюдающих эти соглашения JavaBeans, то может потребоваться создание класса, который наследует PersistenceDelegate и замещает какие-либо из трех защищенных методов. Руководство по написанию собственных представителей постоянства не входит в нашу книгу, но попробуем вкратце изложить эту тему. Задача метода instantiate() – возвращать объект Expression, описывающий вызов конструктора или метода-фабрики, который создает объект с таким же состоянием, что и указанный. Метод mutatesTo() должен определить возможность изменить один объект таким образом, чтобы он принял такое же состояние, как и другой объект. Если этот метод возвратил true, то PersistenceDelegate может вызывать метод initialize(), который и должен записать в объекты Expression и Statement вызовы методов, необходимые для инициализации.

Если нужно использовать экземпляр класса DefaultPersistenceDelegate или вашего собственного класса PersistenceDelegate, предназначенного для сохранения состояния компонента и связанного с ним объекта, то необходимо сначала связать делегат с соответствующим классом, вызвав метод setPersistenceDelegate() объекта XMLEncoder. (Обратите внимание: этот метод не определен в XMLEncoder, а унаследован от класса Encoder.) Также можно определить объект PersistenceDelegate для компонента JavaBean с помощью класса BeanInfo, возвращающего объект BeanDescriptor. Значение, возвращаемое при вызове getValue("persistenceDelegate") из объекта BeanDescriptor, является именем представителя постоянства.

```
public abstract class PersistenceDelegate {
// Открытые конструкторы
   public PersistenceDelegate();
// Открытые методы экземпляра
   public void writeObject(Object oldInstance, Encoder out);
// Защищенные методы экземпляра
   protected void initialize(Class type, Object oldInstance, Object newInstance, Encoder out);
   protected abstract Expression instantiate(Object oldInstance, Encoder out);
   protected boolean mutatesTo(Object oldInstance, Object newInstance);
}
```

Подклассы: DefaultPersistenceDelegate

Передается методам: Encoder.setPersistenceDelegate()

Возвращается методами: Encoder.getPersistenceDelegate()

PropertyChangeEvent Java 1.1

java.beans ***сериализуемый, событие***

Класс PropertyChangeEvent является потомком java.util.EventObject. Событие этого типа посылается его слушателям – объектам PropertyChangeListener – каждый раз, когда компонент JavaBeans меняет связанное (bound) свойство или когда объекты Pro-

pertyEditor и Customizer изменяют значение свойства. Событие PropertyChangeEvent также посылается зарегистрированным объектам VetoableChangeListener, когда компонент предпринимает попытку изменить значение свойства с ограничениями (constrained property).

При создании PropertyChangeEvent обычно указывается компонент, создавший событие, а также внутреннее (регионо-независимое) имя свойства, которое было изменено, и его старое и новое значения. Если значения нельзя определить, нужно передать null. Если событие связано с изменением значений нескольких свойств, то вместо имени свойства ставится null. Поскольку объекты PropertyChangeEvent создаются компонентами JavaBeans, то, как правило, получателями этих событий являются среды разработки и инструментальные программы.


```
public class PropertyChangeEvent extends java.util.EventObject {
// Открытые конструкторы
   public PropertyChangeEvent(Object source, String propertyName,
         Object oldValue, Object newValue);
// Открытые методы экземпляра
   public Object getNewValue();
   public Object getOldValue();
   public Object getPropagationId();
   public String getPropertyName();
   public void setPropagationId(Object propagationId);
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: PropertyVetoException.getPropertyChangeEvent()

PropertyChangeListener Java 1.1

java.beans ***слушатель события***

Этот интерфейс является расширением интерфейса java.util.EventListener. Он определяет метод для реализации в классе, который должен быть оповещен при изменении свойств. Событие PropertyChangeEvent посылается всем зарегистрированным объектам PropertyChangeListener, когда компонент меняет одно из связанных свойств или когда объекты PropertyEditor и Customizer меняют значение некоторого свойства.


```
public interface PropertyChangeListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void propertyChange(PropertyChangeEvent evt);
}
```

Реализации: Классов слишком много, чтобы их перечислить.

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

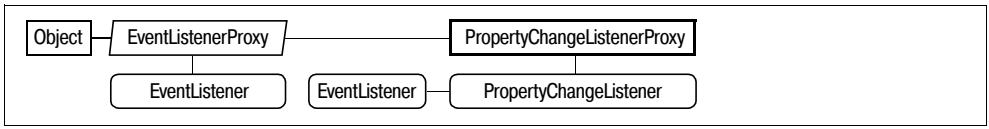
Экземпляры:

javax.swing.plaf.basic.BasicColorChooserUI.propertyChangeListener,
javax.swing.plaf.basic.BasicComboBoxUI.propertyChangeListener,
javax.swing.plaf.basic.BasicComboPopup.propertyChangeListener,
javax.swing.plaf.basic.BasicInternalFrameTitlePane.propertyChangeListener,
javax.swing.plaf.basic.BasicInternalFrameUI.propertyChangeListener,
javax.swing.plaf.basic.BasicListUI.propertyChangeListener,
javax.swing.plaf.basic.BasicMenuUI.propertyChangeListener,
javax.swing.plaf.basic.BasicOptionPaneUI.propertyChangeListener,
javax.swing.plaf.basic.BasicScrollBarUI.propertyChangeListener,
javax.swing.plaf.basic.BasicScrollPaneUI.spPropertyChangeListener,
javax.swing.plaf.basic.BasicSliderUI.propertyChangeListener,
javax.swing.plaf.basic.BasicSplitPaneUI.propertyChangeListener,
javax.swing.plaf.basic.BasicTabbedPaneUI.propertyChangeListener,
javax.swing.plaf.basic.BasicToolBarUI.propertyListener,
javax.swing.plaf.metal.MetalToolBarUI.rolloverListener

PropertyChangeListenerProxy Java 1.4

java.beans

Этот класс реализует интерфейс PropertyChangeListener и служит «оберткой» (wrapper) для другого объекта PropertyChangeListener. Если в компоненте реализован метод addPropertyChangeListener() с двумя аргументами для возможности регистрации слушателей на изменение заданного свойства, то он может возвращать экземпляры этого класса через метод getPropertyChangeListeners() (если он реализован). Чтобы определить имя свойства, на которое зарегистрирован слушатель, используйте getPropertyName(), а для получения базового объекта PropertyChangeListener применяйте унаследованный метод getListener().




```
public class PropertyChangeListenerProxy extends java.util.EventListenerProxy
       Implements PropertyChangeListener {

// Открытые конструкторы
   public PropertyChangeListenerProxy(String propertyName, PropertyChangeListener listener);
// Открытые методы экземпляра
   public String getPropertyName();
// Методы, реализующие PropertyChangeListener
   public void propertyChange(PropertyChangeEvent evt);
}
```

PropertyChangeSupport Java 1.1

java.beans ***сериализуемый***

Класс PropertyChangeSupport поддерживает список зарегистрированных объектов PropertyChangeListener и предоставляет метод firePropertyChange(), который посылает объект события PropertyChangeEvent всем зарегистрированным слушателям. Для кор-

ректного выполнения синхронизации потоков рекомендуется всем JavaBeans, поддерживающим связанные свойства, либо расширять этот класс, либо, как чаще всего и бывает, создавать экземпляр этого класса, которому компонент может перенаправить вызовы методов addPropertyChangeListener() и removePropertyChangeListener(). В Java 1.4 компоненты, в которых определены методы getPropertyChangeListeners(), могут также передавать эти методы объекту PropertyChangeSupport.


```
public class PropertyChangeSupport Implements Serializable {
// Открытые конструкторы
   public PropertyChangeSupport(Object sourceBean);
// Методы регистрации событий (по имени события)
   public void addPropertyChangeListener(PropertyChangeListener listener);      // синхронизирован
   public void removePropertyChangeListener(PropertyChangeListener listener);   // синхронизирован
// Открытые методы экземпляра
1.2 public void addPropertyChangeListener(String propertyName, PropertyChangeListener listener);
                                                                                  // синхронизирован
1.2 public void firePropertyChange(PropertyChangeEvent evt);
1.2 public void firePropertyChange(String propertyName, int oldValue, int newValue);
1.2 public void firePropertyChange(String propertyName, boolean oldValue, boolean newValue);
   public void firePropertyChange(String propertyName, Object oldValue, Object newValue);
1.4 public PropertyChangeListener[] getPropertyChangeListeners();              // синхронизирован
1.4 public PropertyChangeListener[] getPropertyChangeListeners(String propertyName);
                                                                                  // синхронизирован
1.2 public boolean hasListeners(String propertyName);                          // синхронизирован
1.2 public void removePropertyChangeListener(String propertyName, PropertyChangeListener
                                                                  listener);   // синхронизирован
}
```

Подклассы: javax.swing.event.SwingPropertyChangeSupport

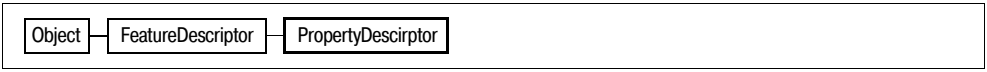
Экземпляры: java.awt.Toolkit.desktopPropsSupport, java.beans.beancontext.BeanContextChildSupport.pcSupport

PropertyDescriptor Java 1.1

java.beans

Объект PropertyDescriptor наследует FeatureDescriptor и описывает определенное свойство компонента JavaBeans. Класс BeanInfo компонента может создавать и инициализировать объекты PropertyDescriptor, описывающие свойства, которые поддерживает этот компонент. Как правило, методы и свойства этого класса используют только среды разработки и аналогичные программы для получения описаний свойств.

При создании объекта PropertyDescriptor нужно указать имя свойства и объект типа Class для компонента. Если не соблюдены стандартные соглашения для имен методов доступа, можно также указать такой метод для этого свойства. Когда создан объект PropertyDescriptor, то с помощью методов setBound() и setConstrained() можно указать, связано ли свойство и имеются ли у него ограничения. С помощью метода setPropertyEditorClass() можно определить редактор для этого свойства. Это может быть удобно, если свойство является перечислимым типом с фиксированным множеством допустимых значений. Методы родительского класса FeatureDescriptor позволяют получить дополнительную информацию об указанном свойстве.




```
public class PropertyDescriptor extends FeatureDescriptor {
// Открытые конструкторы
   public PropertyDescriptor(String propertyName, Class beanClass) throws IntrospectionException;
   public PropertyDescriptor(String propertyName, java.lang.reflect.Method getter,
                             java.lang.reflect.Method setter) throws IntrospectionException;
   public PropertyDescriptor(String propertyName, Class beanClass, String getterName, String
                             setterName) throws IntrospectionException;
// Методы доступа к свойствам (по имени свойства)
   public boolean isBound();
   public void setBound(boolean bound);
   public boolean isConstrained();
   public void setConstrained(boolean constrained);
   public Class getPropertyEditorClass();
   public void setPropertyEditorClass(Class propertyEditorClass);
   public Class getPropertyType();
   public java.lang.reflect.Method getReadMethod();
1.2 public void setReadMethod(java.lang.reflect.Method getter) throws IntrospectionException;
   public java.lang.reflect.Method getWriteMethod();
1.2 public void setWriteMethod(java.lang.reflect.Method setter) throws IntrospectionException;
// Открытые методы, замещающие  Object
1.4 public boolean equals(Object obj);
}
```

Подклассы: IndexedPropertyDescriptor

Возвращается методами:

BeanInfo.getPropertyDescriptors(), SimpleBeanInfo.getPropertyDescriptors()

PropertyEditor

Java 1.1

java.beans

Интерфейс PropertyEditor определяет методы, которые должны быть реализованы в редакторе свойств JavaBeans, предназначенном для использования в средах разработки и в аналогичных программах. PropertyEditor является сложным интерфейсом, в нем определены методы для поддержки различных способов отображения и изменения значений свойств.

Для свойства с именем x авторы компонентов обычно реализуют редактор с именем xEditor. Этот интерфейс реализуется разработчиком компонента, а используется только средами разработки (или классом Customizer данного компонента). Кроме реализации интерфейса PropertyEditor, редактор свойств должен иметь конструктор без аргументов, позволяющий среде разработки легко создавать этот объект. Еще он должен регистрировать и отменять регистрацию объектов PropertyChangeListener и посылать событие PropertyChangeEvent всем зарегистрированным слушателям при изменении значения редактируемого свойства. Класс PropertyEditorSupport является простейшей реализацией интерфейса PropertyEditor. Он предназначен для создания подклассов и поддержки списка слушателей PropertyChangeListener.

```
public interface PropertyEditor {
// Методы регистрации событий (по имени события)
   public abstract void addPropertyChangeListener(PropertyChangeListener listener);
   public abstract void removePropertyChangeListener(PropertyChangeListener listener);
```

```
// Методы доступа к свойствам (по имени свойства)
   public abstract String getAsText();
   public abstract void setAsText(String text) throws IllegalArgumentException;
   public abstract java.awt.Component getCustomEditor();
   public abstract String getJavaInitializationString();
   public abstract boolean isPaintable();
   public abstract String[] getTags();
   public abstract Object getValue();
   public abstract void setValue(Object value);
// Открытые методы экземпляра
   public abstract void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box);
   public abstract boolean supportsCustomEditor();
}
```

Реализации: PropertyEditorSupport

Возвращается методами: PropertyEditorManager.findEditor()

PropertyEditorManager Java 1.1

java.beans

Класс PropertyEditorManager не предусматривает создание экземпляров. Статические методы, определенные в нем, выполняют поиск и регистрируют классы PropertyEditor для указанного свойства. Чтобы связать в компоненте определенный класс PropertyEditor с заданным свойством, можно определить его в описателе PropertyDescriptor этого свойства. Если этого не сделано, то для поиска и регистрации редактора применяется PropertyEditorManager. Компонент или среда разработки для регистрации редактора PropertyEditor свойств определенного типа вызывает метод Editor(). Чтобы получить объект PropertyEditor для определенного типа свойств, среды разработки и классы Customizer данного компонента могут вызывать метод findEditor(). Если для данного типа не зарегистрирован ни один редактор, объект PropertyEditorManager предпринимает попытку назначить его самостоятельно. Для типа x производится поиск класса xEditor сначала в том же пакете, где определен x, а затем во всех пакетах, перечисленных в пути поиска редакторов свойств.

```
public class PropertyEditorManager {
// Открытые конструкторы
   public PropertyEditorManager();
// Открытые методы класса
   public static PropertyEditor findEditor(Class targetType);       // синхронизирован
   public static String[] getEditorSearchPath();                    // синхронизирован
   public static void registerEditor(Class targetType, Class editorClass);
   public static void setEditorSearchPath(String[] path);           // синхронизирован
}
```

PropertyEditorSupport Java 1.1

java.beans

Класс PropertyEditorSupport является простейшей реализацией интерфейса PropertyEditor. Он предоставляет «пустую» реализацию большинства методов, поэтому можно определять потомки этого класса, в которых замещаются не все, а только необходимые методы. Кроме того, в PropertyEditorSupport определены рабочие реализации методов addPropertyChangeListener() и removePropertyChangeListener(), а также метода firePropertyChange(), посылающего событие PropertyChangeEvent всем зарегистрирован-

ным слушателям. Классы PropertyEditor могут создавать объект PropertyEditorSupport только для поддержки списка слушателей. В таком случае при создании объекта PropertyEditorSupport нужно указывать объект-источник, который будет использоваться в событиях PropertyChangeEvent.

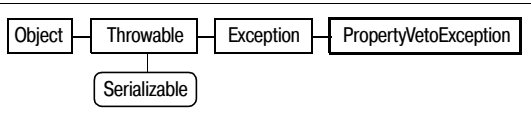

```
public class PropertyEditorSupport Implements PropertyEditor {
// Защищенные конструкторы
   protected PropertyEditorSupport();
   protected PropertyEditorSupport(Object source);
// Методы регистрации событий (по имени события)
   public void addPropertyChangeListener(PropertyChangeListener listener);
                                          // Реализует:PropertyEditor; синхронизирован
   public void removePropertyChangeListener(PropertyChangeListener listener);
                                          // Реализует:PropertyEditor; синхронизирован
// Открытые методы экземпляра
   public void firePropertyChange();
// Методы, реализующие PropertyEditor
   public void addPropertyChangeListener(PropertyChangeListener listener);   // синхронизирован
   public String getAsText();
   public java.awt.Component getCustomEditor();                              // константа
   public String getJavaInitializationString();
   public String[] getTags();                                                // константа
   public Object getValue();
   public boolean isPaintable();                                             // константа
   public void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box);    // пустой
   public void removePropertyChangeListener(PropertyChangeListener listener);// синхронизирован
   public void setAsText(String text) throws IllegalArgumentException;
   public void setValue(Object value);
   public boolean supportsCustomEditor();                                    // константа
}
```

PropertyVetoException

Java 1.1

java.beans ***сериализуемое, проверяемое***

Исключение PropertyVetoException оповещает о том, что объект VetoableChangeListener, получивший событие PropertyChangeEvent при изменении свойства с ограничениями, запретил предложенное изменение. При получении этого исключения данному свойству должно быть присвоено предыдущее значение, а всем объектам класса VetoableChangeListener, которые уже получили уведомление об изменении свойства, необходимо повторно сообщить о возвращении свойству исходного значения. Класс VetoableChangeSupport автоматически проводит повторное уведомление и снова генерирует исключение PropertyVetoException, чтобы сообщить в место вызова, что в изменении отказано.




```
public class PropertyVetoException extends Exception {
// Открытые конструкторы
```

```
   public PropertyVetoException(String mess, PropertyChangeEvent evt);
// Открытые методы экземпляра
   public PropertyChangeEvent getPropertyChangeEvent();
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

SimpleBeanInfo Java 1.1

java.beans

Класс SimpleBeanInfo является простейшей реализацией интерфейса BeanInfo. Все методы этого класса возвращают null или -1 в случае, если информация о компоненте отсутствует. Чтобы использовать этот класс, нужно подменить один или несколько методов, возвращающих тот тип информации, который вы хотите предоставить. Кроме того, в SimpleBeanInfo есть метод loadImage(), принимающий имя ресурса в качестве аргумента и возвращающий объект типа Image. Этот метод удобен для определения метода getIcon().


```
public class SimpleBeanInfo Implements BeanInfo {
// Открытые конструкторы
   public SimpleBeanInfo();
// Открытые методы экземпляра
   public java.awt.Image loadImage(String resourceName);
// Методы, реализующие BeanInfo
   public BeanInfo[] getAdditionalBeanInfo();              // константа; по умолчанию:null
   public BeanDescriptor getBeanDescriptor();              // константа; по умолчанию:null
   public int getDefaultEventIndex();                      // константа; по умолчанию:-1
   public int getDefaultPropertyIndex();                   // константа; по умолчанию:-1
   public EventSetDescriptor[] getEventSetDescriptors();   // константа; по умолчанию:null
   public java.awt.Image getIcon(int iconKind);            // константа
   public MethodDescriptor[] getMethodDescriptors();       // константа; по умолчанию:null
   public PropertyDescriptor[] getPropertyDescriptors();   // константа; по умолчанию:null
}
```

Statement Java 1.4

java.beans

Это простой класс, описывающий вызов метода без возвращаемого значения. В рамках механизма сохранения и восстановления состояния JavaBeans объекты типа Statement создаются классом PersistenceDelegate и транслируются в некоторый текстовый формат кодировщиком, например классом XMLEncoder.

Чтобы создать объект Statement, необходимо указать целевой объект, в котором будет вызываться указанный метод, имя метода и массив его аргументов. Если все аргументы метода представляют собой значения простых типов, используйте соответствующий объект-обертку. Например, для значения типа int используйте Integer. Чтобы вызвать статический метод, в качестве целевого объекта укажите соответствующий Class. Если целевой объект является массивом, можно присвоить значение элементу массива, указав «set» в качестве имени метода и передав объект Integer, определяющий индекс массива, и объект Object, определяющий присваиваемое значение. Ме-

тод execute() для вызова указанного метода использует пакет java.lang.reflect. Потомок класса Expression описывает вызов метода, возвращающего значение.

```
public class Statement {
// Открытые конструкторы
   public Statement(Object target, String methodName, Object[] arguments);
// Открытые методы экземпляра
   public void execute() throws Exception;
   public Object[] getArguments();
   public String getMethodName();
   public Object getTarget();
// Открытые методы, замещающие  Object
   public String toString();
}
```

Подклассы: Expression

Передается методам: Encoder.writeStatement(), XMLEncoder.writeStatement()

VetoableChangeListener Java 1.1

java.beans ***слушатель события***

Этот интерфейс дополняет java.util.EventListener. VetoableChangeListener определяет методы, которые нужно реализовать в классе, чтобы получать сообщения об изменении компонентом свойства с ограничениями. Когда это происходит, событие PropertyChangeEvent передается методу vetoableChange(). Если необходимо запретить изменение, этот метод должен сгенерировать исключение PropertyVetoException.


```
public interface VetoableChangeListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
}
```

Реализации: VetoableChangeListenerProxy, java.beans.beancontext.BeanContextSupport

Передается методам: java.awt.KeyboardFocusManager.{addVetoableChangeListener(), removeVetoableChangeListener()}, VetoableChangeListenerProxy.VetoableChangeListenerProxy(), VetoableChangeSupport.{addVetoableChangeListener(), removeVetoableChangeListener()}, java.beans.beancontext.BeanContextChild.{addVetoableChangeListener(), removeVetoableChangeListener()}, java.beans.beancontext.BeanContextChildSupport.{addVetoableChangeListener(), removeVetoableChangeListener()}, javax.swing.JComponent.{addVetoableChangeListener(), removeVetoableChangeListener()}

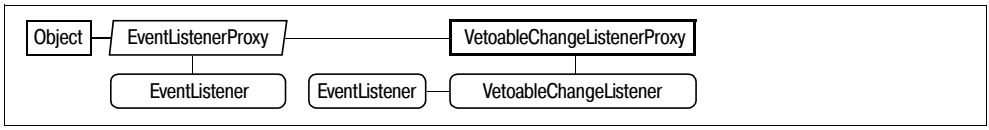
Возвращается методами: java.awt.KeyboardFocusManager.getVetoableChangeListeners(), VetoableChangeSupport.getVetoableChangeListeners(), java.beans.beancontext.BeanContextSupport.getChildVetoableChangeListener(), javax.swing.JComponent.getVetoableChangeListeners()

VetoableChangeListenerProxy Java 1.4

java.beans

Этот класс реализует интерфейс VetoableChangeListener и служит оберткой для другого объекта VetoableChangeListener. Если в компоненте определен метод addVetoable-

ChangeListener() с двумя аргументами для регистрации слушателя свойств с указанными именами, то метод getVetoableChangeListeners(), если он реализован, возвращает объекты этого класса. Чтобы получить имя свойства, связанного со слушателем, используйте метод getPropertyName(). С помощью унаследованного метода getListener() можно получить внутренний объект VetoableChangeListener.

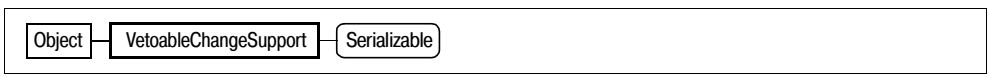



```
public class VetoableChangeListenerProxy extends java.util.EventListenerProxy
       Implements VetoableChangeListener {
// Открытые конструкторы
   public VetoableChangeListenerProxy(String propertyName, VetoableChangeListener listener);
// Открытые методы экземпляра
   public String getPropertyName();
// Методы, реализующие VetoableChangeListener
   public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
}
```

VetoableChangeSupport Java 1.1

java.beans ***сериализуемый***

Вспомогательный класс VetoableChangeSupport хранит список зарегистрированных объектов типа VetoableChangeListener и предоставляет метод fireVetoableChange(), посылающий событие PropertyChangeEvent всем зарегистрированным слушателям. Если какой-нибудь зарегистрированный слушатель наложил запрет на изменение свойства, то fireVetoableChange() посылает еще один объект PropertyChangeEvent. Этот объект уведомляет слушателей, получивших предыдущее сообщение, о том, что свойству возвращено исходное значение. Из-за сложности обработки изменений с возможностью запрета, а также из-за того, что поддержание списка слушателей связано с синхронизацией потоков, рекомендуется во всех компонентах, поддерживающих события с ограничениями, создавать объект VetoableChangeSupport, которому можно перенаправить вызовы методов addVetoableChangeListener() и removeVetoableChangeListener(). В Java 1.4 в компонентах также можно определять метод getVetoableChangeListeners() и передать его объекту VetoableChangeSupport.




```
public class VetoableChangeSupport Implements Serializable {
// Открытые конструкторы
   public VetoableChangeSupport(Object sourceBean);
// Методы регистрации событий (по имени события)
   public void addVetoableChangeListener(VetoableChangeListener listener);     // синхронизирован
   public void removeVetoableChangeListener(VetoableChangeListener listener); // синхронизирован
// Открытые методы экземпляра
1.2 public void addVetoableChangeListener(String propertyName, VetoableChangeListener listener);
                                                                               // синхронизирован
1.2 public void fireVetoableChange(PropertyChangeEvent evt) throws PropertyVetoException;
1.2 public void fireVetoableChange(String propertyName, int oldValue, int newValue)
```

```
        throws PropertyVetoException;
   public void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
        throws PropertyVetoException;
1.2 public void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue)
        throws PropertyVetoException;
1.4 public VetoableChangeListener[] getVetoableChangeListeners();              // синхронизирован
1.4 public VetoableChangeListener[] getVetoableChangeListeners(String propertyName);
                                                                               // синхронизирован
1.2 public boolean hasListeners(String propertyName);                          // синхронизирован
1.2 public void removeVetoableChangeListener(String propertyName,
                                             VetoableChangeListener listener); // синхронизирован
}
```

Экземпляры: java.beans.beancontext.BeanContextChildSupport.vcSupport

Visibility Java 1.1

java.beans

Этот интерфейс предназначен для реализации в универсальных компонентах, которые могут работать как с графическим интерфейсом пользователя (GUI), так и без него. Методы этого интерфейса позволяют указать, нужен ли компоненту GUI. При этом среда выполнения может сообщить компоненту, доступен ли графический интерфейс. Если компоненту необходим GUI, то метод needsGui() должен возвращать true. Если компонент выполняется без использования графического интерфейса, то avoidingGui() должен возвращать true. Если GUI недоступен, то компоненту можно сообщить об этом с помощью метода dontUseGui(), а если доступен – с помощью okToUseGui().

```
public interface Visibility {
// Открытые методы экземпляра
   public abstract boolean avoidingGui();
   public abstract void dontUseGui();
   public abstract boolean needsGui();
   public abstract void okToUseGui();
}
```

Реализации: java.beans.beancontext.BeanContext

Возвращается методами:

java.beans.beancontext.BeanContextSupport.getChildVisibility()

XMLDecoder Java 1.4

java.beans

Этот класс восстанавливает компонент JavaBeans, который был сохранен в формате XML объектом XMLEncoder. Объект XMLDecoder можно создать, указав поток java.io.InputStream, из которого будут читаться сериализованные компоненты в формате XML. Затем нужно вызвать метод readObject() необходимое число раз, чтобы прочитать из файла закодированные компоненты. Когда компонентов для чтения больше нет, этот метод вызывает исключение ArrayIndexOutOfBoundsException. Когда XMLDecoder выполнил все необходимые операции, нужно вызвать метод close(), чтобы закрыть входной поток. Поскольку XMLEncoder кодирует состояние компонента в виде последовательных вызовов открытых методов, процесс декодирования сравнительно эффективен, а декодирующему приложению нет необходимости загружать и использовать классы PersistenceDelegate, с помощью которых был закодирован компонент.

Процессу декодирования обеспечено устойчивое выполнение, а после любых исключений всегда предпринимаются попытки восстановления процесса. Если нужно получать уведомления об исключениях, возникающих во время процесса декодирования, передайте конструктору `XMLDecoder()` объект `ExceptionListener`. (`XMLDecoder` начинает процесс декодирования сразу же после вызова конструктора, поэтому передавать `ExceptionListener` методу `setExceptionListener()` бесполезно: слишком поздно.)

Если в декодируемый поток на этапе кодирования был включен вызов метода `setOwner()`, то нужно передать объект-владелец конструктору `XMLDecoder()`, чтобы могли быть декодированы методы, использующие этот объект. (В этом классе есть метод `setOwner()`, но прямо вызывать его бесполезно по тем же причинам, что и `setExceptionListener()`.)

```
public class XMLDecoder {
// Открытые конструкторы
   public XMLDecoder(java.io.InputStream in);
   public XMLDecoder(java.io.InputStream in, Object owner);
   public XMLDecoder(java.io.InputStream in, Object owner, ExceptionListener exceptionListener);
// Открытые методы экземпляра
   public void close();
   public ExceptionListener getExceptionListener();
   public Object getOwner();
   public Object readObject();
   public void setExceptionListener(ExceptionListener exceptionListener);
   public void setOwner(Object owner);
}
```

XMLEncoder

Java 1.4

java.beans

Этот класс создает описание компонента JavaBeans или дерева компонентов в формате XML и записывает его в поток. Этот поток нужно указать при создании объекта `XMLEncoder`. Затем нужно вызвать метод `writeObject()` необходимое число раз для сериализации компонента JavaBeans с использованием формата XML. Обратите внимание, что этот метод сериализует не только указанный объект, но и рекурсивно все объекты, ссылки на которые встречаются в открытом интерфейсе этого объекта. После окончания работы кодировщика `XMLEncoder` необходимо вызвать метод `close()`, который сбрасывает внутренние буферы, создает закрывающий тег XML, сбрасывает и закрывает выходной поток. При необходимости методом `flush()` можно сбросить буферы, не закрывая кодировщик. Для восстановления компонентов, закодированных с помощью этого класса, используйте `XMLDecoder`. Сравните этот механизм с `java.io.ObjectOutputStream`, который производит сериализацию, используя двоичный формат.

Процессу кодирования компонента обеспечено устойчивое выполнение, а при каждом внутреннем исключении в нем предпринимаются попытки восстановления. Если нужно зарегистрировать слушатель `ExceptionListener` для получения сообщений об исключениях во время процесса кодирования, используйте унаследованный метод `setExceptionListener()`.

В процессе кодирования компонента используется объект `PersistenceDelegate`. Класс `DefaultPersistenceDelegate` используется для всех компонентов, которые удовлетворяют соглашениям JavaBeans, имеют конструктор без аргументов и инкапсулируют свое состояние в свойствах с открытыми методами доступа `get` и `set`. В реализации Java также представлены защищенные (private) классы типа `PersistenceDelegate` для всех классов `java.awt.Component` из пакетов AWT и Swing (они описаны в справочнике

«Java Foundation Classes in a Nutshell»), а также для классов, которые используются в этих компонентах в качестве свойств (например, java.awt.Font и java.awt.Color). Если в кодируемом компоненте не соблюдены соглашения JavaBeans по конструкторам и свойствам, то для него нужно создать свой объект PersistenceDelegate и связать его с объектом Class для этого компонента, вызвав унаследованный метод setPersistenceDelegate(). Подробное описание представлено в разделе «PersistenceDelegate».

Кроме описания состояния компонента, этот класс можно использовать для кодирования вызова произвольного метода с помощью методов writeExpression() и writeStatement(). Это особенно удобно при использовании метода setOwner(). С его помощью можно определить объект, который не будет сериализован вместе с компонентом. Затем нужно закодировать вызовы методов объекта-владельца (owner) с помощью writeStatement(). Когда архив XML декодируется классом XMLDecoder, нужно вызвать аналогичный метод setOwner() данного класса, чтобы указать объект-владелец для этих методов. Если этот объект представляет собой серверный процесс приложения, то его можно связать с сериализованным GUI, зарегистрировав слушателей событий или передав ему определенные компоненты.


```
public class XMLEncoder extends Encoder {
// Открытые конструкторы
   public XMLEncoder(java.io.OutputStream out);
// Открытые методы экземпляра
   public void close();
   public void flush();
   public Object getOwner();
   public void setOwner(Object owner);
// Открытые методы, замещающие  Encoder
   public void writeExpression(Expression oldExp);
   public void writeObject(Object o);
   public void writeStatement(java.beans.Statement oldStm);
}
```

Пакет java.beans.beancontext

Java 1.2

Пакет java.beans.beancontext расширяет модель компонентов JavaBeans, добавляя в нее понятие иерархии вложений (containment). Он также поддерживает контейнеры компонентов, которые предоставляют содержащимся в них компонентам контекст выполнения и могут предоставлять набор сервисов. Как правило, этот пакет используется опытными разработчиками компонентов и программ для управления компонентами. Разработчикам приложений, которые лишь используют компоненты, этот пакет не требуется.

BeanContext — главный интерфейс пакета. Он является контейнером для компонентов и определяет несколько методов, которые возвращают информацию о контексте для компонентов. BeanContextServices расширяет интерфейс BeanContext: в нем определены методы, позволяющие компонентам из контейнера запрашивать доступные сервисы. В компоненте, который нужно уведомлять о его принадлежности контейнеру, должна быть реализация интерфейса BeanContextChild. Сам контекстный объект

(BeanContext) является дочерним контекстом (BeanContextChild), то есть контексты могут быть вложены в другие контексты.

Информация о компонентах и контекстах представлена в главе 6.

Интерфейсы

```
public interface BeanContextChild;
public interface BeanContextChildComponentProxy;
public interface BeanContextContainerProxy;
public interface BeanContextProxy;
public interface BeanContextServiceProvider;
public interface BeanContextServiceProviderBeanInfo extends java.beans.BeanInfo;
```

Коллекции

```
public interface BeanContext extends BeanContextChild, java.util.Collection,
       java.beans.DesignMode, java.beans.Visibility;
public class BeanContextSupport extends BeanContextChildSupport
                                Implements BeanContext, java.beans.PropertyChangeListener,
                                Serializable, java.beans.VetoableChangeListener;
   └ public class BeanContextServicesSupport extends BeanContextSupport
                                             Implements BeanContextServices;
```

События

```
public abstract class BeanContextEvent extends java.util.EventObject;
public class BeanContextMembershipEvent extends BeanContextEvent;
public class BeanContextServiceAvailableEvent extends BeanContextEvent;
public class BeanContextServiceRevokedEvent extends BeanContextEvent;
```

Слушатели событий

```
public interface BeanContextMembershipListener extends java.util.EventListener;
public interface BeanContextServiceRevokedListener extends java.util.EventListener;
public interface BeanContextServices extends BeanContext, BeanContextServicesListener;
public interface BeanContextServicesListener extends BeanContextServiceRevokedListener;
```

Другие классы

```
public class BeanContextChildSupport
             Implements BeanContextChild, BeanContextServicesListener, Serializable;
```

Защищенные внутренние классы

```
protected class BeanContextServicesSupport.BCSSProxyServiceProvider
          Implements BeanContextServiceProvider, BeanContextServiceRevokedListener;
protected static class BeanContextServicesSupport.BCSSServiceProvider Implements Serializable;
protected class BeanContextSupport.BCSChild Implements Serializable;
     └ protected class BeanContextServicesSupport.BCSSChild extends BeanContextSupport.BCSChild;
protected static final class BeanContextSupport.BCSIterator Implements java.util.Iterator;
```

BeanContext Java 1.2

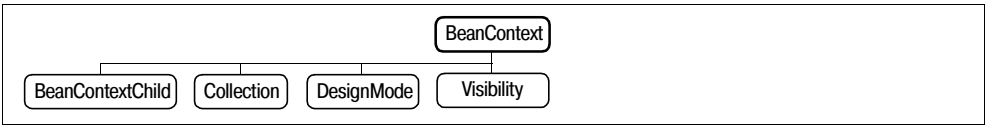
java.beans.beancontext *коллекция*

В этом интерфейсе определены методы, которые должны быть реализованы в любом классе, желающем выступить в качестве логического контейнера для компонентов JavaBeans. Каждый контекстный объект (BeanContext) является также дочерним контекстом (BeanContextChild), поэтому он может быть вложен в контекстный объект более высокого уровня. BeanContext расширяется интерфейсом BeanContextServices. Лю-

бой контекст компонентов, который будет предоставлять им сервисы, должен реализовать более специализированный интерфейс BeanContextServices.

BeanContext расширяет интерфейс java.util.Collection. Доступ к дочерним объектам осуществляется при помощи методов java.util.Collection. Кроме того, в самом интерфейсе BeanContext определено несколько важных методов. Метод instantiateChild() создает новый экземпляр компонента точно так же, как стандартный метод Beans.instantiate(), а затем делает этот компонент дочерним элементом контекстного объекта. Вызывается данный метод подобно методу Beans.instantiate() с тремя аргументами. Методы getResource() и getResourceAsStream() являются аналогами одноименных методов из классов java.lang.Class и java.lang.ClassLoader. В одних реализациях контекста компонентов может быть предусмотрено особое поведение этих методов, а в других просто используются методы объектов Class или ClassLoader, относящихся к компоненту. И наконец, еще два метода предназначены для регистрации и отмены регистрации слушателей событий, которых BeanContext оповещает о добавлении и удалении дочерних элементов контекста.

Задача реализации BeanContext является более специализированной, чем разработка компонента JavaBeans. Разработчику компонентов практически никогда не требуется самому реализовать этот интерфейс. Если все же приходится это делать, удобно использовать BeanContextSupport – либо расширив его, либо задав его экземпляр в качестве «заместителя» (proxy).




```
public interface BeanContext extends BeanContextChild, java.util.Collection,
                                java.beans.DesignMode, java.beans.Visibility {
// Открытые константы
   public static final Object globalHierarchyLock;
// Методы регистрации событий (по имени события)
   public abstract void
        addBeanContextMembershipListener(BeanContextMembershipListener bcml);
   public abstract void
        removeBeanContextMembershipListener(BeanContextMembershipListener bcml);
// Открытые методы экземпляра
   public abstract java.net.URL getResource(String name, BeanContextChild bcc)
        throws IllegalArgumentException;
   public abstract java.io.InputStream getResourceAsStream(String name, BeanContextChild bcc)
        throws IllegalArgumentException;
   public abstract Object instantiateChild(String beanName) throws java.io.IOException,
      ClassNotFoundException;
}
```

Реализации: BeanContextServices, BeanContextSupport

Передается методам: java.beans.AppletInitializer.initialize(), java.beans.Beans.instantiate(), BeanContextChild.setBeanContext(), BeanContextChildSupport.{setBeanContext(), validatePendingSetBeanContext()}, BeanContextEvent.{BeanContextEvent(), setPropagatedFrom()}, BeanContextMembershipEvent.BeanContextMembershipEvent(), BeanContextSupport.BeanContextSupport()

Возвращается методами: BeanContextChild.getBeanContext(), BeanContextChildSupport.getBeanContext(), BeanContextEvent.{getBeanContext(), getPropagatedFrom()}, BeanContextSupport.getBeanContextPeer()

Экземпляры: BeanContextChildSupport.beanContext, BeanContextEvent.propagatedFrom

BeanContextChild Java 1.2

java.beans.beancontext

Этот интерфейс реализуется в компонентах JavaBeans, которые предназначены для использования внутри контекста и должны быть уведомлены о вхождении в контекст. В BeanContextChild определено единственное свойство – beanContext, которое определяет, в каком контексте содержится компонент. Это свойство является связанным и имеет ограничения: при вызове setBeanContext() должны посылаться события PropertyChangeEvent, а вызов этого метода может создать исключение PropertyVetoException, если какой-нибудь из объектов VetoableChangeListener запретит изменения. Метод setBeanContext() не предназначен для использования компонентами или приложениями. Когда компонент создается или восстанавливается из сериализованной формы, окружающий контекст вызывает этот метод, чтобы представить себя этому компоненту. Последний должен сохранить ссылку на свой контекст BeanContext во временном (transient) поле, чтобы контекст не был сериализован вместе с компонентом.

Реализовать с нуля интерфейс BeanContextChild довольно сложно, поскольку нужно корректно обработать протокол VetoableChangeListener и соблюсти в реализации множество соглашений, например о хранении ссылки на BeanContext во временном поле. Поэтому большинство разработчиков компонентов не реализуют этот интерфейс непосредственно, а вместо этого используют класс BeanContextSupport в качестве родительского класса либо используют его экземпляр для обработки вызовов методов.

```
public interface BeanContextChild {
// Открытые методы экземпляра
   public abstract void addPropertyChangeListener(String name,
        java.beans.PropertyChangeListener pcl);
   public abstract void addVetoableChangeListener(String name,
        java.beans.VetoableChangeListener vcl);
   public abstract BeanContext getBeanContext();
   public abstract void removePropertyChangeListener(String name,
        java.beans.PropertyChangeListener pcl);
   public abstract void removeVetoableChangeListener(String name,
        java.beans.VetoableChangeListener vcl);
   public abstract void setBeanContext(BeanContext bc)
        throws java.beans.PropertyVetoException;
}
```

Реализации: BeanContext, BeanContextChildSupport

Передается методам: BeanContext.{getResource(), getResourceAsStream()}, BeanContextChildSupport.BeanContextChildSupport(), BeanContextServices.{getService(), releaseService()}, BeanContextServicesSupport.{getService(), releaseService()}, BeanContextSupport.{getResource(), getResourceAsStream()}

Возвращается методами: BeanContextChildSupport.getBeanContextChildPeer(), BeanContextProxy.getBeanContextProxy(), BeanContextSupport.getChildBeanContextChild()

Экземпляры: BeanContextChildSupport.beanContextChildPeer

BeanContextChildComponentProxy Java 1.2

java.beans.beancontext

Если класс, реализующий интерфейс BeanContextChild, не является потомком класса Component, но связан с объектом Component, который содержит в себе визуальное представление компонента, то в этом классе может быть реализован интерфейс BeanContextChildComponentProxy, позволяющий осуществить доступ к этому объекту Component.

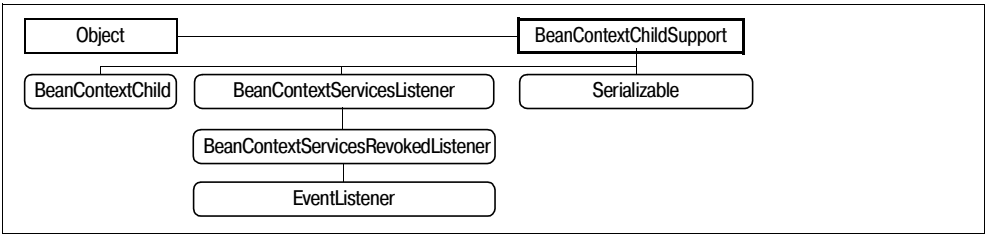
```
public interface BeanContextChildComponentProxy {
// Открытые методы экземпляра
   public abstract java.awt.Component getComponent();
}
```

BeanContextChildSupport Java 1.2

java.beans.beancontext ***сериализуемый***

Этот класс позволяет реализовать интерфейс BeanContextChild таким образом, чтобы он в точности соответствовал спецификации контекста компонентов. В классе-потомке должны быть реализованы методы initializeBeanContextResources() и releaseBeanContextResources() для получения и освобождения любых ресурсов, требуемых контекстом, таких как объекты сервисов, которые можно получить из контекста. Эти методы вызываются, когда контекст представляет себя компоненту с помощью setBeanContext(). Любой ресурс, полученный с помощью этих методов, должен быть помещен во временные поля, чтобы он не был сериализован вместе с компонентом. Если в компоненте нужно проверять правильность вызова setBeanContext() до его успешного завершения, то можно реализовать метод validatePendingSetBeanContext(). Если он возвращает false, значит соответствующий метод setBeanContext() вызовет исключение PropertyVetoException.

Многие компоненты JavaBeans являются компонентами AWT или Swing и не могут наследоваться одновременно от классов Component и BeanContextChildSupport. Поэтому большинство разработчиков компонентов считают, что нужно использовать внутренний экземпляр BeanContextChildSupport. Сделать это можно несколькими способами. Во-первых, можно реализовать в компоненте интерфейс BeanContextProxy и возвращать из метода getBeanContextProxy() объект BeanContextChildSupport. Во-вторых, можно самостоятельно реализовать интерфейс BeanContextChild, предоставив «пустые» методы, которые содержат только вызов соответствующих методов из BeanContextChildSupport. При этом нужно передать экземпляр вашего компонента конструктору BeanContextChildSupport(). Использование этого класса будет прозрачным для событий, которые будут передаваться непосредственно вашему компоненту. Кроме того, можно непосредственно создать экземпляр класса BeanContextChildSupport. Однако для реализации таких методов, как initializeBeanContextResources(), часто приходится создавать собственный подкласс (возможно, как внутренний класс).



```
public class BeanContextChildSupport Implements BeanContextChild,
                                  BeanContextServicesListener, Serializable {
// Открытые конструкторы
   public BeanContextChildSupport();
   public BeanContextChildSupport(BeanContextChild bcc);
// Открытые методы экземпляра
   public void firePropertyChange(String name, Object oldValue, Object newValue);
   public void fireVetoableChange(String name, Object oldValue, Object newValue)
        throws java.beans.PropertyVetoException;
   public BeanContextChild getBeanContextChildPeer(); // по умолчанию:BeanContextChildSupport
   public boolean isDelegated();                                  // по умолчанию:false
   public boolean validatePendingSetBeanContext(BeanContext newValue); // константа
// Методы, реализующие BeanContextChild
   public void addPropertyChangeListener(String name, java.beans.PropertyChangeListener pcl);
   public void addVetoableChangeListener(String name, java.beans.VetoableChangeListener vcl);
   public BeanContext getBeanContext();              // синхронизирован; по умолчанию:null
   public void removePropertyChangeListener(String name, java.beans.PropertyChangeListener pcl);
   public void removeVetoableChangeListener(String name, java.beans.VetoableChangeListener vcl);
   public void setBeanContext(BeanContext bc) throws java.beans.PropertyVetoException;
                                                                        // синхронизирован
// Методы, реализующие BeanContextServiceRevokedListener
   public void serviceRevoked(BeanContextServiceRevokedEvent bcsre);     // пустой
// Методы, реализующие BeanContextServicesListener
   public void serviceAvailable(BeanContextServiceAvailableEvent bcsae); // пустой
// Защищенные методы экземпляра
   protected void initializeBeanContextResources();                     // пустой
   protected void releaseBeanContextResources();                        // пустой
// Public Instance Fields
   public BeanContextChild beanContextChildPeer;
// Защищенные поля экземпляра
   protected transient BeanContext beanContext;
   protected java.beans.PropertyChangeSupport pcSupport;
   protected transient boolean rejectedSetBCOnce;
   protected java.beans.VetoableChangeSupport vcSupport;
}
```

Подклассы: BeanContextSupport

BeanContextContainerProxy

Java 1.2

java.beans.beancontext

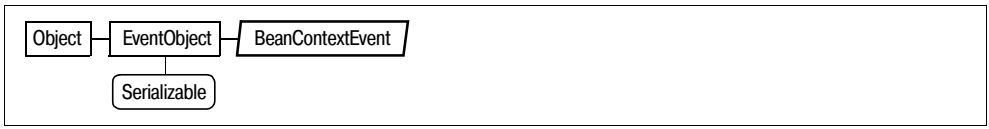
Этот интерфейс реализуется контекстом BeanContext**, с которым связан класс** java.awt.Container**. Метод** getContainer() **позволяет получить ссылку на контейнер, связанный с контекстным объектом. На практике контекстные объекты часто бывают связаны с контейнерами. К сожалению,** BeanContext **реализует интерфейс** java.util.Collection**, имена методов которого конфликтуют с именами методов класса** java.awt.Container**, поэтому ни один подкласс** Container **не может реализовать интерфейс** BeanContext**. См. также интерфейс** BeanContextProxy**, в котором направление взаимодействия «заместителей» (proxy) изменено на противоположное.**

```
public interface BeanContextContainerProxy {
// Открытые методы экземпляра
   public abstract java.awt.Container getContainer();
}
```

BeanContextEvent Java 1.2

java.beans.beancontext ***сериализуемый, событие***

Этот абстрактный класс является родительским классом всех событий, связанных с контекстными объектами. Метод `getBeanContext()` возвращает источник события. Если `isPropagated()` возвращает `true`, событие прошло через иерархически связанные контексты, а метод `getPropagatedFrom()` возвращает последний контекстный объект, через который прошло это событие.



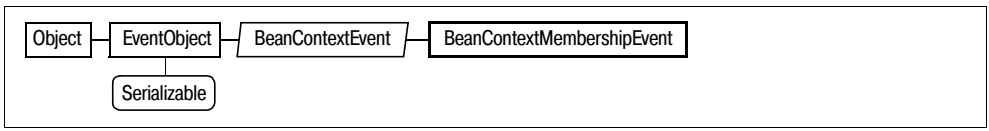

```
public abstract class BeanContextEvent extends java.util.EventObject {
// Защищенные конструкторы
   protected BeanContextEvent(BeanContext bc);
// Открытые методы экземпляра
   public BeanContext getBeanContext();
   public BeanContext getPropagatedFrom();                    // синхронизирован
   public boolean isPropagated();                             // синхронизирован
   public void setPropagatedFrom(BeanContext bc);             // синхронизирован
// Защищенные поля экземпляра
   protected BeanContext propagatedFrom;
}
```

Подклассы: `BeanContextMembershipEvent`, `BeanContextServiceAvailableEvent`, `BeanContextServiceRevokedEvent`

BeanContextMembershipEvent Java 1.2

java.beans.beancontext ***сериализуемый, событие***

Событие этого типа генерируется при добавлении компонентов в объект `BeanContext` и при их удалении. Объект события содержит список дочерних элементов, которые были добавлены или удалены, и позволяет несколькими способами получить доступ к этому списку. Метод `size()` возвращает количество дочерних объектов, которые были добавлены или удалены. Метод `contains()` проверяет, был ли среди них указанный объект. `toArray()` возвращает список этих объектов в виде массива, а метод `iterator()` – в форме `java.util.Iterator`.




```
public class BeanContextMembershipEvent extends BeanContextEvent {
// Открытые конструкторы
   public BeanContextMembershipEvent(BeanContext bc, Object[] changes);
   public BeanContextMembershipEvent(BeanContext bc, java.util.Collection changes);
// Открытые методы экземпляра
   public boolean contains(Object child);
   public java.util.Iterator iterator();
```

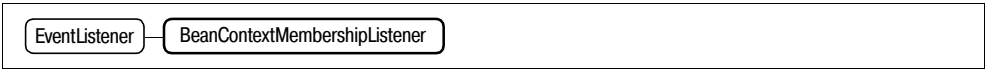
```
   public int size();
   public Object[] toArray();
// Защищенные поля экземпляра
   protected java.util.Collection children;
}
```

Передается методам: BeanContextMembershipListener.{childrenAdded(), childrenRemoved()}, BeanContextSupport.{fireChildrenAdded(), fireChildrenRemoved()}

BeanContextMembershipListener Java 1.2

java.beans.beancontext ***слушатель события***

Этот интерфейс должен быть реализован во всех объектах, которые хотят получать уведомления о добавлении дочерних элементов к объекту BeanContext и их удалении.




```
public interface BeanContextMembershipListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void childrenAdded(BeanContextMembershipEvent bcme);
   public abstract void childrenRemoved(BeanContextMembershipEvent bcme);
}
```

Передается методам: BeanContext.{addBeanContextMembershipListener(), removeBeanContextMembershipListener()}, BeanContextSupport.{addBeanContextMembershipListener(), removeBeanContextMembershipListener()}

Возвращается методами: BeanContextSupport.getChildBeanContextMembershipListener()

BeanContextProxy Java 1.2

java.beans.beancontext

Этот интерфейс реализуется в компонентах JavaBeans (часто, но не всегда, объектами AWT Component или Container), которые сами не являются объектами BeanContext или BeanContextChild, но с ними связан объект того или другого типа. Метод getBeanContextProxy() возвращает связанный объект. Типом возвращаемого значения этого метода является BeanContextChild. Но в зависимости от контекста, в котором вызывается этот метод, тип возвращаемого объекта в действительности может быть BeanContext или BeanContextServices. Это следует проверять с помощью оператора instanceof до приведения объекта к другим типам.

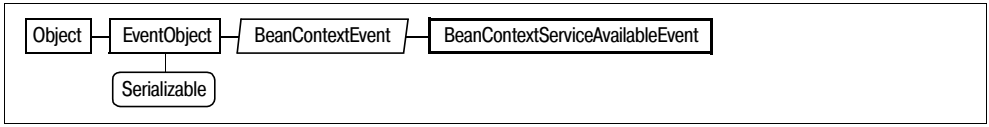
```
public interface BeanContextProxy {
// Открытые методы экземпляра
   public abstract BeanContextChild getBeanContextProxy();
}
```

BeanContextServiceAvailableEvent Java 1.2

java.beans.beancontext ***сериализуемый, событие***

Событие этого типа генерируется для уведомления заинтересованных объектов BeanContextServicesListener о том, что в объекте BeanContextServices появился новый класс

сервиса. Метод getServiceClass() возвращает класс этого сервиса, а getCurrentServiceSelectors() – набор дополнительных аргументов, которые параметризуют этот сервис.




```
public class BeanContextServiceAvailableEvent extends BeanContextEvent {
// Открытые конструкторы
   public BeanContextServiceAvailableEvent(BeanContextServices bcs, Class sc);
// Открытые методы экземпляра
   public java.util.Iterator getCurrentServiceSelectors();
   public Class getServiceClass();
   public BeanContextServices getSourceAsBeanContextServices();
// Защищенные поля экземпляра
   protected Class serviceClass;
}
```

Передается методам: BeanContextChildSupport.serviceAvailable(), BeanContextServicesListener.serviceAvailable(), BeanContextServicesSupport.{fireServiceAdded(), serviceAvailable()}

BeanContextServiceProvider Java 1.2

java.beans.beancontext

Этот интерфейс определяет методы, которые должны быть реализованы в классе-фабрике, предоставляющем компонентам объекты сервисов. Чтобы предоставить сервис, BeanContextServiceProvider нужно передать методу addService() объекта BeanContextServices. Таким образом, в объекте BeanContextServices класс сервиса (например, java.awt.print.PrinterJob) связывается с объектом BeanContextServiceProvider, который может возвращать экземпляр класса, предоставляющего сервис.

Когда BeanContextChild запрашивает сервис определенного класса из своего контейнера BeanContextServices, BeanContextServices находит соответствующий объект BeanContextServiceProvider и направляет запрос своему методу getService(). Когда компонент освобождает сервис, вызывается метод releaseService(). В качестве дополнительного параметра, или селектора сервиса, запрос для getService() может содержать в себе произвольный объект. Поставщики сервисов (service providers), которые используют этот селектор и поддерживают конечное множество разрешенных значений селектора, должны реализовать метод getCurrentServiceSelectors(), позволяющий получить список всех допустимых значений селектора.

Как правило, разработчикам компонентов не требуется реализовывать или использовать этот интерфейс. С точки зрения компонента, являющегося потомком контекстного объекта, сервисные объекты предоставляются объектами BeanContextServices. Однако разработчики, создающие классы BeanContextServices, должны реализовать соответствующий интерфейс BeanContextServiceProvider, который будет предоставлять сервисы.

```
public interface BeanContextServiceProvider {
// Открытые методы экземпляра
   public abstract java.util.Iterator getCurrentServiceSelectors(BeanContextServices bcs,
                                                    Class serviceClass);
   public abstract Object getService(BeanContextServices bcs, Object requestor,
```

```
                                    Class serviceClass, Object serviceSelector);
   public abstract void releaseService(BeanContextServices bcs, Object requestor, Object service);
}
```

Реализации: BeanContextServicesSupport.BCSSProxyServiceProvider

Передается методам: BeanContextServices.{addService(), revokeService()}, BeanContextServicesSupport.{addService(), createBCSSServiceProvider(), revokeService()}

Возвращается методами: BeanContextServicesSupport.BCSSServiceProvider.getServiceProvider()

Экземпляры: BeanContextServicesSupport.BCSSServiceProvider.serviceProvider

BeanContextServiceProviderBeanInfo Java 1.2

java.beans.beancontext

Объект BeanContextServiceProvider, который будет предоставлять программе для разработки GUI информацию о своем сервисе или сервисах, должен реализовать интерфейс BeanContextServiceProviderBeanInfo, являющийся потомком BeanInfo. В соответствии со стандартными соглашениями об именах JavaBeans, имя реализующего класса должно состоять из имени поставщика сервисов и суффикса «BeanInfo». Это позволяет инструментальным программам при необходимости искать и динамически загружать класс, содержащий информацию о компоненте.

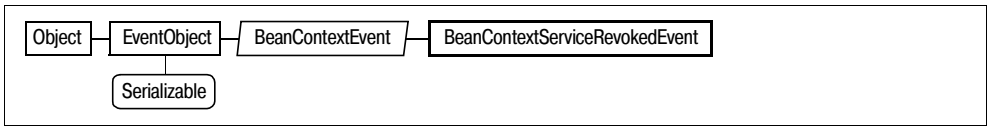
Метод getServicesBeanInfo() должен возвращать массив объектов BeanInfo для каждого класса сервисов, предоставляемых BeanContextServiceProvider. Эти объекты BeanInfo позволяют визуально настраивать объект сервиса в инструментальной программе. Это полезно, если в объектах сервисов не соблюдены стандартные соглашения об именах JavaBeans.


```
public interface BeanContextServiceProviderBeanInfo extends java.beans.BeanInfo {
// Открытые методы экземпляра
   public abstract java.beans.BeanInfo[] getServicesBeanInfo();
}
```

BeanContextServiceRevokedEvent Java 1.2

java.beans.beancontext ***сериализуемый, событие***

Этот класс предоставляет информацию об отмене сервиса, инициированной объектом BeanContextServices. Метод getServiceClass() определяет класс освобождаемого сервиса. С помощью метода isCurrentServiceInvalidNow() можно определить недействительность текущего объекта сервиса. Если этот метод возвращает true, то компонент, получивший это событие, должен немедленно прекратить использование сервиса. Если возвращается false, компонент может продолжать применять объект сервиса, но дальнейшие запросы сервиса этого класса получат отказ.



```
public class BeanContextServiceRevokedEvent extends BeanContextEvent {
// Открытые конструкторы
   public BeanContextServiceRevokedEvent(BeanContextServices bcs, Class sc, boolean invalidate);
// Открытые методы экземпляра
   public Class getServiceClass();
   public BeanContextServices getSourceAsBeanContextServices();
   public boolean isCurrentServiceInvalidNow();
   public boolean isServiceClass(Class service);
// Защищенные поля экземпляра
   protected Class serviceClass;
}
```

Передается методам: BeanContextChildSupport.serviceRevoked(), BeanContextServiceRevokedListener.serviceRevoked(), BeanContextServicesSupport.{fireServiceRevoked(), serviceRevoked()}, BeanContextServicesSupport.BCSSProxyServiceProvider.serviceRevoked()

BeanContextServiceRevokedListener Java 1.2

java.beans.beancontext ***слушатель события***

Этот интерфейс определяет метод, который вызывается, когда объект сервиса, возвращаемый объектом BeanContextServices, принудительно отменяется. В отличие от других типов слушателей событий, регистрация и отмена регистрации объекта BeanContextServiceRevokedListener методами add и remove не проводится. Вместо этого объект, реализующий этот интерфейс, должен передаваться при каждом вызове методу getService() объекта BeanContextServices. Если возвращенный сервис отменялся предоставляющим его объектом BeanContextServiceProvider до того, как его освободил компонент, вызывается метод serviceRevoked() этого интерфейса.

Когда сервис отменяется, все дальнейшие запросы этого сервиса получат отказ. Кроме того, текущие объекты сервиса стали недействительными и больше не должны использоваться. Метод serviceRevoked() должен вызывать метод isCurrentServiceInvalidNow() объекта событий, чтобы определить, так ли это. Если так, он должен немедленно прекратить использование объекта сервиса.


```
public interface BeanContextServiceRevokedListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void serviceRevoked(BeanContextServiceRevokedEvent bcsre);
}
```

Реализации:
BeanContextServicesListener, BeanContextServicesSupport.BCSSProxyServiceProvider

Передается методам:
BeanContextServices.getService(), BeanContextServicesSupport.getService()

BeanContextServices Java 1.2

java.beans.beancontext ***коллекция, слушатель события***

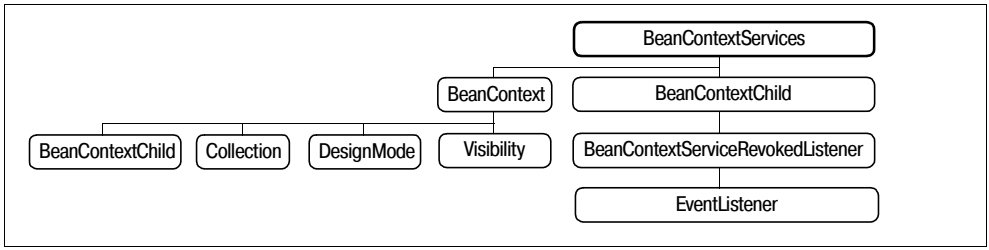
Этот интерфейс определяет дополнительные методы, которые должен реализовать класс контекста, если он будет предоставлять сервисы содержащимся в нем компонентам. Компонент вызывает метод hasService(), чтобы определить, доступен ли сер-

вис определенного типа в его контексте. Для запроса объекта сервиса указанного класса компонент вызывает метод getService(), а когда ему сервис больше не нужен – метод releaseService(). Компонент, которому нужно найти полный список доступных сервисов, может затем вызвать getCurrentServiceClasses(). Некоторые сервисы допускают (или требуют) объект селектора сервисов в качестве параметра метода getService(), чтобы получить дополнительную информацию о сервисе. Если в сервисе определен фиксированный набор допустимых селекторов, то компонент может с помощью метода getCurrentServiceSelectors() просмотреть весь набор селекторных объектов. Компонент, который будет получать уведомления, если становится доступным новый сервис или отменяется уже существующий, должен зарегистрировать объект BeanContextServicesListener с помощью метода addBeanContextServicesListener().

Если объект BeanContextServices не может предоставить запрошенный сервис, но вложен в другой объект BeanContext, то он должен проверить, может ли предоставить этот сервис какой-нибудь из родительских контекстных объектов. Интерфейс BeanContextServices расширяет BeanContextServicesListener. Это означает, что любой объект BeanContextServices может быть слушателем событий, исходящих от сервисов, которые доступны в его контейнере.

Компоненты вызывают упомянутые выше методы, чтобы получить сервисы. В интерфейсе BeanContextServices определены различные методы, которые используются поставщиками сервисов для доставки сервисов. Метод addService() определяет объект BeanContextServiceProvider для сервисов с указанным классом. Метод revokeService() удаляет связь между классом сервиса и его поставщиком и определяет, что сервис больше не доступен. Когда сервис отменяется, объект BeanContextServices должен сообщить об этом всем компонентам, до сих пор получающим этот сервис. Это выполняется путем уведомления объектов BeanContextServiceRevokedListener, переданных методу getService().

Возможно, разработчикам контекстов компонентов будет удобнее наследовать класс BeanContextServicesSupport в собственных классах или использовать его экземпляры, а не реализовывать с нуля интерфейс BeanContextServices.




```
public interface BeanContextServices extends BeanContext, BeanContextServicesListener {
// Методы регистрации событий (по имени события)
   public abstract void addBeanContextServicesListener(BeanContextServicesListener bcsl);
   public abstract void removeBeanContextServicesListener(BeanContextServicesListener bcsl);
// Открытые методы экземпляра
   public abstract boolean addService(Class serviceClass,
        BeanContextServiceProvider serviceProvider);
   public abstract java.util.Iterator getCurrentServiceClasses();
   public abstract java.util.Iterator getCurrentServiceSelectors(Class serviceClass);
  public abstract Object getService(BeanContextChild child, Object requestor, Class serviceClass,
                              Object serviceSelector, BeanContextServiceRevokedListener bcsrl)
      throws java.util.TooManyListenersException;
```

```
    public abstract boolean hasService(Class serviceClass);
    public abstract void releaseService(BeanContextChild child, Object requestor, Object service);
    public abstract void revokeService(Class serviceClass, BeanContextServiceProvider
                        serviceProvider, boolean revokeCurrentServicesNow);
}
```

Реализации: BeanContextServicesSupport

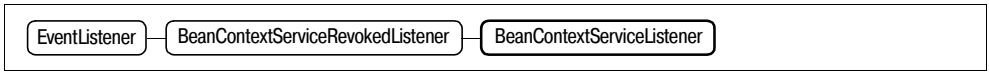
Передается методам: BeanContextServiceAvailableEvent.BeanContextServiceAvailableEvent(), BeanContextServiceProvider.{getCurrentServiceSelectors(), getService(), releaseService()}, BeanContextServiceRevokedEvent.BeanContextServiceRevokedEvent(), BeanContextServicesSupport.BeanContextServicesSupport(), BeanContextServicesSupport.BCSSProxyServiceProvider.{getCurrentServiceSelectors(), getService(), releaseService()}

Возвращается методами: BeanContextServiceAvailableEvent.getSourceAsBeanContextServices(), BeanContextServiceRevokedEvent.getSourceAsBeanContextServices(), BeanContextServicesSupport.getBeanContextServicesPeer()

BeanContextServicesListener Java 1.2

java.beans.beancontext ***слушатель события***

Этот интерфейс расширяет интерфейс BeanContextServiceRevokedListener, добавляя к его методу serviceRevoked() метод serviceAvailable(). Слушатели этого типа могут регистрироваться в объектах BeanContextServices. BeanContextServicesListener получает уведомление, когда становится доступным новый класс сервисов или отменяется существующий класс.




```
public interface BeanContextServicesListener extends BeanContextServiceRevokedListener {
// Открытые методы экземпляра

public abstract void serviceAvailable(BeanContextServiceAvailableEvent bcsae);
}
```

Реализации: BeanContextChildSupport, BeanContextServices

Передается методам: BeanContextServices.{addBeanContextServicesListener(), removeBeanContextServicesListener()}, BeanContextServicesSupport.{addBeanContextServicesListener(), removeBeanContextServicesListener()}

Возвращается методами:

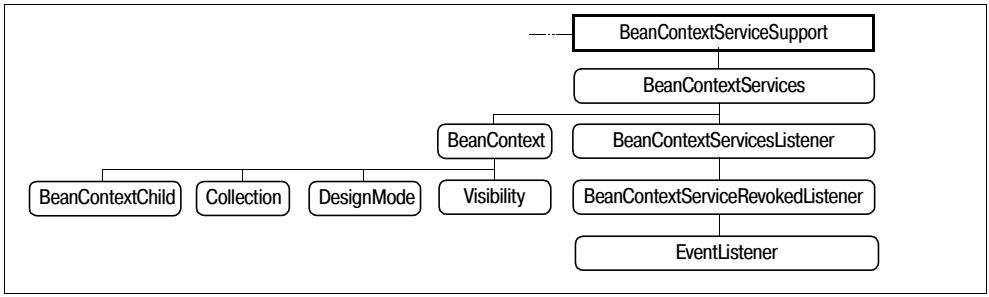
BeanContextServicesSupport.getChildBeanContextServicesListener()

BeanContextServicesSupport Java 1.2

java.beans.beancontext ***сериализуемый, коллекция***

Этот класс является полезной реализацией интерфейса BeanContextServices. В нем соблюдены спецификации и соглашения по контексту компонентов. Многие разработчики контекстов считают, что удобнее пользоваться этим классом, наследуя его в собственных классах или создавая его экземпляры, нежели с нуля реализовывать интерфейс BeanContextServices. Наиболее распространенный способ заключается в ре-

ализации интерфейса `BeanContextProxy` **так, чтобы его метод** `getBeanContextProxy()` **возвращал экземпляр класса** `BeanContextServicesSupport`**.**




```
public class BeanContextServicesSupport extends BeanContextSupport
       Implements BeanContextServices {
// Открытые конструкторы
   public BeanContextServicesSupport();
   public BeanContextServicesSupport(BeanContextServices peer);
   public BeanContextServicesSupport(BeanContextServices peer, java.util.Locale lcle);
   public BeanContextServicesSupport(BeanContextServices peer, java.util.Locale lcle,
                         boolean dtime);
   public BeanContextServicesSupport(BeanContextServices peer, java.util.Locale lcle, boolean
                         dTime, boolean visible);
// Внутренние классы
   protected class BCSSChild extends BeanContextSupport.BCSChild;
   protected class BCSSProxyServiceProvider Implements BeanContextServiceProvider,
      BeanContextServiceRevokedListener;
   protected static class BCSSServiceProvider Implements Serializable;
// Защищенные методы класса
   protected static final BeanContextServicesListener
        getChildBeanContextServicesListener(Object child);
// Методы регистрации событий (по имени события)
   public void addBeanContextServicesListener(BeanContextServicesListener bcsl);
                                                        // Реализует:BeanContextServices
   public void removeBeanContextServicesListener(BeanContextServicesListener bcsl);
                                                        // Реализует:BeanContextServices
// Открытые методы экземпляра
   public BeanContextServices getBeanContextServicesPeer();
                                                  // по умолчанию:BeanContextServicesSupport
// Методы, реализующие BeanContextServiceRevokedListener
   public void serviceRevoked(BeanContextServiceRevokedEvent bcssre);
// Методы, реализующие BeanContextServices
   public void addBeanContextServicesListener(BeanContextServicesListener bcsl);
   public boolean addService(Class serviceClass, BeanContextServiceProvider bcsp);
   public java.util.Iterator getCurrentServiceClasses();
                                            // по умолчанию:BeanContextSupport.BCSIterator
   public java.util.Iterator getCurrentServiceSelectors(Class serviceClass);
   public Object getService(BeanContextChild child, Object requestor, Class serviceClass,
      Object serviceSelector, BeanContextServiceRevokedListener bcsrl) throws
java.util.TooManyListenersException;
   public boolean hasService(Class serviceClass);                        // синхронизирован
   public void releaseService(BeanContextChild child, Object requestor, Object service);
   public void removeBeanContextServicesListener(BeanContextServicesListener bcsl);
   public void revokeService(Class serviceClass, BeanContextServiceProvider bcsp,
```

```
        boolean revokeCurrentServicesNow);
// Методы, реализующие BeanContextServicesListener
    public void serviceAvailable(BeanContextServiceAvailableEvent bcssae);
// Открытые методы, замещающие  BeanContextSupport
    public void initialize();
// Защищенные методы, замещающие  BeanContextSupport
    protected void bcsPreDeserializationHook(java.io.ObjectInputStream ois)
        throws java.io.IOException,ClassNotFoundException;            // синхронизирован
    protected void bcsPreSerializationHook(java.io.ObjectOutputStream oos)
        throws java.io.IOException;                                   // синхронизирован
    protected void childJustRemovedHook(Object child, BeanContextSupport.BCSChild bcsc);
    protected BeanContextSupport.BCSChild createBCSChild(Object targetChild, Object peer);
// Защищенные методы, замещающие  BeanContextChildSupport
    protected void initializeBeanContextResources();                  // синхронизирован
    protected void releaseBeanContextResources();                     // синхронизирован
// Защищенные методы экземпляра
    protected boolean addService(Class serviceClass, BeanContextServiceProvider bcsp,
                                 boolean fireEvent);
    protected BeanContextServicesSupport.BCSSServiceProvider createBCSSServiceProvider(Class sc,
                                                        BeanContextServiceProvider bcsp);
    protected final void fireServiceAdded(BeanContextServiceAvailableEvent bcssae);
    protected final void fireServiceAdded(Class serviceClass);
    protected final void fireServiceRevoked(BeanContextServiceRevokedEvent bcsre);
    protected final void fireServiceRevoked(Class serviceClass, boolean revokeNow);
// Защищенные поля экземпляра
    protected transient java.util.ArrayList bcsListeners;
    protected transient BeanContextServicesSupport.BCSSProxyServiceProvider proxy;
    protected transient int serializable;
    protected transient java.util.HashMap services;
}
```

BeanContextServicesSupport.BCSSChild Java 1.2

java.beans.beancontext ***сериализуемый***

Этот класс предназначен для внутреннего использования классом BeanContextServicesSupport, чтобы связывать дополнительную информацию с каждым дочерним компонентом контекстного объекта. В этом классе нет ни открытых, ни защищенных методов и полей, но его можно видоизменить через наследование.

```
protected class BeanContextServicesSupport.BCSSChild extends BeanContextSupport.BCSChild {
// Конструктор отсутствует
}
```

BeanContextServicesSupport.BCSSProxyServiceProvider Java 1.2

java.beans.beancontext

Этот внутренний класс используется в BeanContextServicesSupport для того, чтобы корректно поддерживать делегирование службам, предоставляемым контекстом, в котором находится компонент. Он реализует интерфейс BeanContextServiceProvider в терминах методов объекта BeanContextServices, в котором содержится соответствующий компонент.

```
protected class BeanContextServicesSupport.BCSSProxyServiceProvider
      Implements BeanContextServiceProvider, BeanContextServiceRevokedListener {
```

```
// Конструктор отсутствует
// Методы, реализующие BeanContextServiceProvider
   public java.util.Iterator getCurrentServiceSelectors(BeanContextServices bcs,
                                                        Class serviceClass);
   public Object getService(BeanContextServices bcs, Object requestor, Class serviceClass,
                                                        Object serviceSelector);
   public void releaseService(BeanContextServices bcs, Object requestor, Object service);
// Методы, реализующие BeanContextServiceRevokedListener
   public void serviceRevoked(BeanContextServiceRevokedEvent bcsre);
}
```

Экземпляры: BeanContextServicesSupport.proxy

BeanContextServicesSupport.BCSSServiceProvider Java 1.2

java.beans.beancontext ***сериализуемый***

Этот класс является простейшей «оберткой» для объекта BeanContextServiceProvider. Подклассы, которым необходимо связать дополнительную информацию с каждым поставщиком сервисов, могут наследовать этот класс и замещать метод createBCSSServiceProvider() класса BeanContextServicesSupport.

```
protected static class BeanContextServicesSupport.BCSSServiceProvider Implements Serializable {
// Конструктор отсутствует
// Защищенные методы экземпляра
   protected BeanContextServiceProvider getServiceProvider();
// Защищенные поля экземпляра
   protected BeanContextServiceProvider serviceProvider;
}
```

Возвращается методами: BeanContextServicesSupport.createBCSSServiceProvider()

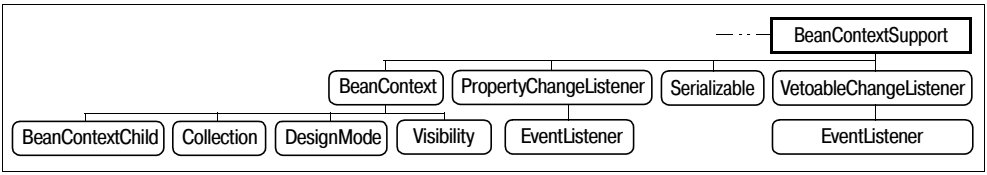
BeanContextSupport Java 1.2

java.beans.beancontext ***сериализуемый, коллекция***

Этот класс представляет собой простую и легко настраиваемую реализацию интерфейса BeanContext. Большинство разработчиков реализаций контекстных объектов считают, что удобнее наследовать BeanContextSupport в собственном классе или создавать объект этого класса, нежели с нуля реализовывать интерфейс BeanContext.

Контексты компонентов часто являются контейнерами для компонентов AWT или Swing и не могут реализовать BeanContext из-за конфликта имен методов. Поэтому контекстные объекты реализуют интерфейс BeanContextProxy таким образом, что метод getBeanContextProxy() возвращает объект BeanContext.

Но иногда необходимо изменить поведение контекстного объекта. BeanContextSupport разработан таким образом, что его можно легко приспособить для своих целей путем создания подклассов. При этом особенно полезны защищенные методы, такие как childJustAddedHook() и validatePendingAdd().



```
public class BeanContextSupport extends BeanContextChildSupport Implements BeanContext,
      java.beans.PropertyChangeListener,  Serializable, java.beans.VetoableChangeListener {
// Открытые конструкторы
   public BeanContextSupport();
   public BeanContextSupport(BeanContext peer);
   public BeanContextSupport(BeanContext peer, java.util.Locale lcle);
   public BeanContextSupport(BeanContext peer, java.util.Locale lcle, boolean dtime);
   public BeanContextSupport(BeanContext peer, java.util.Locale lcle, boolean dTime,
        boolean visible);
// Внутренние классы
   protected class BCSChild Implements Serializable;
   protected static final class BCSIterator Implements java.util.Iterator;
// Защищенные методы класса
   protected static final boolean classEquals(Class first, Class second);
   protected static final BeanContextChild getChildBeanContextChild(Object child);
   protected static final BeanContextMembershipListener getChildBeanContextMembershipListener(
                                                          Object child);
   protected static final java.beans.PropertyChangeListener
        getChildPropertyChangeListener(Object child);
   protected static final Serializable getChildSerializable(Object child);
   protected static final java.beans.VetoableChangeListener
        getChildVetoableChangeListener(Object child);
   protected static final java.beans.Visibility getChildVisibility(Object child);
// Методы регистрации событий (по имени события)
   public void addBeanContextMembershipListener(BeanContextMembershipListener bcml);
                                                          // Реализует:BeanContext
   public void removeBeanContextMembershipListener(BeanContextMembershipListener bcml);
                                                          // Реализует:BeanContext
// Методы доступа к свойствам (по имени свойства)
   public BeanContext getBeanContextPeer();             // по умолчанию:BeanContextSupport
   public boolean isDesignTime();// Реализует:DesignMode; синхронизирован; по умолчанию:false
   public void setDesignTime(boolean dTime);         // Реализует:DesignMode; синхронизирован
   public boolean isEmpty();                        // Реализует:Collection по умолчанию:true
   public java.util.Locale getLocale();                                  // синхронизирован
   public void setLocale(java.util.Locale newLocale) throws java.beans.PropertyVetoException;
                                                                         // синхронизирован
   public boolean isSerializing();                                  // по умолчанию:false
// Открытые методы экземпляра
   public boolean containsKey(Object o);
   public final void readChildren(java.io.ObjectInputStream ois) throws java.io.IOException,
                                        ClassNotFoundException;
   public final void writeChildren(java.io.ObjectOutputStream oos) throws java.io.IOException;
// Методы, реализующие BeanContext
   public void addBeanContextMembershipListener(BeanContextMembershipListener bcml);
   public java.net.URL getResource(String name, BeanContextChild bcc);
   public java.io.InputStream getResourceAsStream(String name, BeanContextChild bcc);
   public Object instantiateChild(String beanName) throws java.io.IOException,
      ClassNotFoundException;
   public void removeBeanContextMembershipListener(BeanContextMembershipListener bcml);
// Методы, реализующие Collection
   public boolean add(Object targetChild);
   public boolean addAll(java.util.Collection c);
   public void clear();
   public boolean contains(Object o);
   public boolean containsAll(java.util.Collection c);
   public boolean isEmpty();                                        // по умолчанию:true
   public java.util.Iterator iterator();
```

```
   public boolean remove(Object targetChild);
   public boolean removeAll(java.util.Collection c);
   public boolean retainAll(java.util.Collection c);
   public int size();
   public Object[] toArray();
   public Object[] toArray(Object[] arry);
// Методы, реализующие DesignMode
   public boolean isDesignTime();                    // синхронизирован; по умолчанию:false
   public void setDesignTime(boolean dTime);                              // синхронизирован
// Методы, реализующие PropertyChangeListener
   public void propertyChange(java.beans.PropertyChangeEvent pce);
// Методы, реализующие VetoableChangeListener
   public void vetoableChange(java.beans.PropertyChangeEvent pce)
        throws java.beans.PropertyVetoException;
// Методы, реализующие Visibility
   public boolean avoidingGui();
   public void dontUseGui();                                              // синхронизирован
   public boolean needsGui();                                             // синхронизирован
   public void okToUseGui();                                              // синхронизирован
// Защищенные методы экземпляра
   protected java.util.Iterator bcsChildren();
   protected void bcsPreDeserializationHook(java.io.ObjectInputStream ois)
        throws java.io.IOException, ClassNotFoundException;    // пустой
   protected void bcsPreSerializationHook(java.io.ObjectOutputStream oos) throws
java.io.IOException;  // пустой
   protected void childDeserializedHook(Object child, BeanContextSupport.BCSChild bcsc);
   protected void childJustAddedHook(Object child, BeanContextSupport.BCSChild bcsc);   // пустой
   protected void childJustRemovedHook(Object child, BeanContextSupport.BCSChild bcsc); // пустой
   protected final Object[] copyChildren();
   protected BeanContextSupport.BCSChild createBCSChild(Object targetChild, Object peer);
   protected final void deserialize(java.io.ObjectInputStream ois, java.util.Collection coll)
      throws java.io.IOException, ClassNotFoundException;
   protected final void fireChildrenAdded(BeanContextMembershipEvent bcme);
   protected final void fireChildrenRemoved(BeanContextMembershipEvent bcme);
   protected void initialize();                                           // синхронизирован
   protected boolean remove(Object targetChild, boolean callChildSetBC);
   protected final void serialize(java.io.ObjectOutputStream oos, java.util.Collection coll)
throws java.io.IOException;
   protected boolean validatePendingAdd(Object targetChild);          // константа
   protected boolean validatePendingRemove(Object targetChild);       // константа
// Защищенные поля экземпляра
   protected transient java.util.ArrayList bcmListeners;
   protected transient java.util.HashMap children;
   protected boolean designTime;
   protected java.util.Locale locale;
   protected boolean okToUseGui;
}
```

Подклассы: BeanContextServicesSupport

BeanContextSupport.BCSChild

Java 1.2

java.beans.beancontext ***сериализуемый***

Этот класс предназначен для внутреннего использования объектами BeanContextSupport. Он отслеживает дополнительную информацию об их дочерних объектах. В частности, он отслеживает объекты BeanContextChild, связанные с дочерними объектами, в которых реализован интерфейс BeanContextProxy. В этом классе нет открытых и защи-

щенных методов и полей. Потомки BeanContextSupport, которым нужно связать дополнительную информацию с каждым дочерним объектом, могут наследовать этот класс и замещать метод createBCSChild() так, чтобы он создавал экземпляр нового подкласса.

```
protected class BeanContextSupport.BCSChild Implements Serializable {
// Конструктор отсутствует
}
```

Подклассы: BeanContextServicesSupport.BCSSChild

Передается методам: BeanContextServicesSupport.childJustRemovedHook(), BeanContextSupport.{childDeserializedHook(), childJustAddedHook(), childJustRemovedHook()}

Возвращается методами: BeanContextServicesSupport.createBCSChild(), BeanContextSupport.createBCSChild()

BeanContextSupport.BCSIterator

Java 1.2

java.beans.beancontext

В этом классе реализован интерфейс java.util.Iterator. Его экземпляр возвращается методом iterator(), реализованном в классе BeanContextSupport. Метод remove() имеет в этом классе «пустую» реализацию и не выполняет никаких действий по удалению дочерних объектов из контекста.

```
protected static final class BeanContextSupport.BCSIterator Implements java.util.Iterator {
// Конструктор отсутствует
// Методы, реализующие Iterator
   public boolean hasNext();
   public Object next();
   public void remove();              // пустой
}
```

Глава 10

java.io

Пакет java.io

java 1.0

Пакет java.io является достаточно большим, однако основная часть его классов подпадает под иерархию с четкой структурой.

Большая часть пакета состоит из байтовых потоков (подклассов InputStream или OutputStream) и потоков символов (подклассов Reader или Writer). Каждый из этих подтипов потока имеет конкретное предназначение, поэтому несмотря на свой размер java.io является понятным и удобным пакетом. В Java 1.4 пакет java.io дополнен новым API ввода/вывода, который определяется в пакете java.nio и его подпакетах. Пакет java.nio является абсолютно новым, хотя в какой-то мере он совместим с классами из пакета java.io. Пакет java.nio был разработан для обеспечения высокоскоростного ввода/вывода, в особенности для использования на серверах. Он имеет низкоуровневый программный интерфейс приложения по сравнению с данным пакетом. Средства ввода/вывода java.io по-прежнему достаточно адекватны для большей части ввода/вывода, которая требуется обычным клиентским приложениям.

Перед тем как приступить к рассмотрению классов потоков, которые составляют основную часть этого пакета, рассмотрим те важные классы, которые непосредственно не работают с потоками. File представляет файл или имя каталога независимо от системы. Он предоставляет методы для чтения содержимого каталогов, доступа к атрибутам файлов, переименования и удаления файлов. FilenameFilter – это интерфейс, который определяет метод, принимающий или отклоняющий конкретные имена файлов. Он используется File для того, чтобы указать те типы файлов, которые следует включать в перечни файлов каталога. RandomAccessFile позволяет вам читать или записывать что-либо в произвольные участки файла. Тем не менее зачастую предпочтительнее использовать последовательный доступ к файлу. Кроме того, вам следует использовать один из классов потоков.

InputStream и OutputStream являются абстрактными классами, которые определяют методы для чтения и записи байтов. Их подклассы позволяют считывать и записывать байты в разнообразные приемники и источники. FileInputStream и FileOutputStream считывают данные из файлов и записывают их в файлы. ByteArrayInputStream и ByteArrayOutputStream считывают данные из массива байтов в памяти и записывают их в массив байтов в памяти. PipedInputStream считывает байты с PipedOutputStream, а PipedOutputStream записывает байты в PipedInputStream. Эти классы работают

вместе для того, чтобы реализовать *канал* (pipe), связывающий разные потоки выполнения (threads). FilterInputStream и FilterOutputStream являются особенными; они фильтруют входящие и исходящие байты. Когда вы создаете FilterInputStream, то вы указываете InputStream, подлежащий фильтрации.

Когда вы вызываете метод read(), принадлежащий FilterInputStream, он вызывает метод read() своего InputStream, обрабатывает байты, которые он считывает, и возвращает отфильтрованные байты. Когда вы аналогичным образом создаете FilterOutputStream, то вы указываете OutputStream, подлежащий фильтрации. Вызов метода write(), принадлежащего FilterOutputStream, приводит к тому, что он каким-либо образом обрабатывает ваши байты, а затем передает эти отфильтрованные байты методу write() своего OutputStream.

FilterInputStream и FilterOutputStream сами не выполняют какой-либо фильтрации; это делают их подклассы.

BufferedInputStream и BufferedOutputStream являются отфильтрованными потоками, которые обеспечивают буферизацию ввода и вывода и могут увеличить эффективность операций ввода/вывода. DataInputStream считывает необработанные байты из потока и интерпретирует их в различные двоичные форматы. У него есть различные методы для чтения примитивных типов данных Java в их стандартных двоичных форматах. DataOutputStream позволяет записывать примитивные типы данных Java в двоичном формате.

Описанные потоки байтов дополнены аналогичным набором потоков ввода/вывода символов. Reader является родительским классом всех потоков ввода символов. Writer является родительским классом всех потоков вывода символов. Большая часть потоков Reader и Writer имеют явные аналоги типа байт-потока. Поток BufferedReader широко используется; он обеспечивает буферизацию, улучшающую эффективность, и предоставляет метод readLine(), позволяющий читать строку текста за раз. PrintWriter является другим распространенным потоком; его методы выводят текстовое представление любого примитивного Java-типа или любого объекта (через метод объекта toString()). Классы ObjectInputStream и ObjectOutputStream – особенные. Эти классы байт-потоков применяются для сериализации и десериализации внутреннего состояния объектов с целью хранения или межпроцессного взаимодействия.

Интерфейсы

```
public interface DataInput;
public interface DataOutput;
public interface Externalizable extends Serializable;
public interface FileFilter;
public interface FilenameFilter;
public interface ObjectInput extends DataInput;
public interface ObjectInputValidation;
public interface ObjectOutput extends DataOutput;
public interface ObjectStreamConstants;
public interface Serializable;
```

Классы

```
public class File implements Comparable, Serializable;
public final class FileDescriptor;
public final class FilePermission extends java.security.Permission implements Serializable;
public abstract class InputStream;
   └ public class ByteArrayInputStream extends InputStream;
   └ public class FileInputStream extends InputStream;
```

```
   └ public class FilterInputStream extends InputStream;
       └ public class BufferedInputStream extends FilterInputStream;
       └ public class DataInputStream extends FilterInputStream implements DataInput;
       └ public class LineNumberInputStream extends FilterInputStream;
       └ public class PushbackInputStream extends FilterInputStream;
   └ public class ObjectInputStream extends InputStream
     implements ObjectInput, ObjectStreamConstants;
    └ public class PipedInputStream extends InputStream;
   └ public class SequenceInputStream extends InputStream;
   └ public class StringBufferInputStream extends InputStream;
public abstract static class ObjectInputStream.GetField;
public abstract static class ObjectOutputStream.PutField;
public class ObjectStreamClass implements Serializable;
public class ObjectStreamField implements Comparable;
public abstract class OutputStream;
    └ public class ByteArrayOutputStream extends OutputStream;
   └ public class FileOutputStream extends OutputStream;
   └ public class FilterOutputStream extends OutputStream;
       └ public class BufferedOutputStream extends FilterOutputStream;
      └ public class DataOutputStream extends FilterOutputStream implements DataOutput;
      └ public class PrintStream extends FilterOutputStream;
   └ public class ObjectOutputStream extends OutputStream
                                       implements ObjectOutput, ObjectStreamConstants;
   └ public class PipedOutputStream extends OutputStream;
public class RandomAccessFile implements DataInput, DataOutput;
public abstract class Reader;
   └ public class BufferedReader extends Reader;
       └ public class LineNumberReader extends BufferedReader;
   └ public class CharArrayReader extends Reader;
    └ public abstract class FilterReader extends Reader;
       └ public class PushbackReader extends FilterReader;
   └ public class InputStreamReader extends Reader;
       └ public class FileReader extends InputStreamReader;
   └ public class PipedReader extends Reader;
    └ public class StringReader extends Reader;
public final class SerializablePermission extends java.security.BasicPermission;
public class StreamTokenizer;
public abstract class Writer;
   └ public class BufferedWriter extends Writer;
   └ public class CharArrayWriter extends Writer;
   └ public abstract class FilterWriter extends Writer;
   └ public class OutputStreamWriter extends Writer;
      └ public class FileWriter extends OutputStreamWriter;
   └ public class PipedWriter extends Writer;
   └ public class PrintWriter extends Writer;
   └ public class StringWriter extends Writer;
```

Исключения

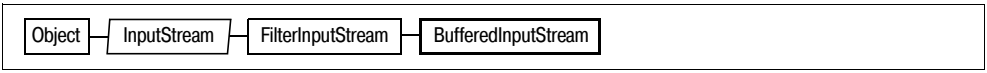
```
public class IOException extends Exception;
   └ public class CharConversionException extends IOException;
   └ public class EOFException extends IOException;
   └ public class FileNotFoundException extends IOException;
   └ public class InterruptedIOException extends IOException;
   └ public abstract class ObjectStreamException extends IOException;
     └ public class InvalidClassException extends ObjectStreamException;
     └ public class InvalidObjectException extends ObjectStreamException;
```

```
    └ public class NotActiveException extends ObjectStreamException;
    └ public class NotSerializableException extends ObjectStreamException;
    └ public class OptionalDataException extends ObjectStreamException;
    └ public class StreamCorruptedException extends ObjectStreamException;
    └ public class WriteAbortedException extends ObjectStreamException;
public class SyncFailedException extends IOException;
public class UnsupportedEncodingException extends IOException;
public class UTFDataFormatException extends IOException;
```

BufferedInputStream Java 1.0

java.io

Этот класс является потомком класса `FilterInputStream`, который обеспечивает буферизацию входных данных; эффективность повышается благодаря считыванию большого количества данных и их хранению во внутреннем буфере. Когда запрашиваются данные, они извлекаются из буфера. Таким образом, большинство запросов на считывание данных на самом деле не должны обращаться за данными к диску, сети или другому медленному источнику. Создайте `BufferedInputStream`, указав в запросе к конструктору на конкретный `InputStream`, подлежащий буферизации. См. также `BufferedReader`.



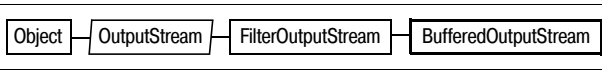

```
public class BufferedInputStream extends FilterInputStream {
// Открытые конструкторы
   public BufferedInputStream(java.io.InputStream in);
   public BufferedInputStream(java.io.InputStream in, int size);
// Открытые методы, замещающие методы класса FilterInputStream
   public int available() throws IOException;                          // синхронизирован
1.2 public void close() throws IOException;
   public void mark(int readlimit);                                    // синхронизирован
   public boolean markSupported();                                     // константа
   public int read() throws IOException;                               // синхронизирован
   public int read(byte[] b, int off, int len) throws IOException;     // синхронизирован
   public void reset() throws IOException;                             // синхронизирован
   public long skip(long n) throws IOException;                        // синхронизирован
// Защищенные поля экземпляра
   protected byte[] buf;
   protected int count;
   protected int marklimit;
   protected int markpos;
   protected int pos;
}
```

BufferedOutputStream Java 1.0

java.io

Этот класс является потомком класса `FilterOutputStream`, который обеспечивает буферизацию выходных данных; эффективность вывода увеличивается за счет хранения значений, подлежащих записи, в буфере. Фактическая запись этих значений происходит только в том случае, если переполняется буфер или вызывается метод `flush()`.

Создайте BufferedOutputStream, указав в запросе к конструктору на конкретный OutputStream, подлежащий буферизации. См. также BufferedWriter.



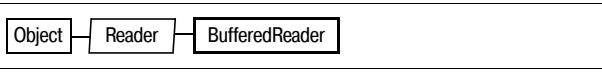

```
public class BufferedOutputStream extends FilterOutputStream {
// Открытые конструкторы
   public BufferedOutputStream(java.io.OutputStream out);
   public BufferedOutputStream(java.io.OutputStream out, int size);
// Открытые методы, замещающие методы класса FilterOutputStream
   public void flush() throws IOException;                                  // синхронизирован
   public void write(int b) throws IOException;                             // синхронизирован
   public void write(byte[] b, int off, int len) throws IOException;        // синхронизирован
// Защищенные поля экземпляра
   protected byte[] buf;
   protected int count;
}
```

BufferedReader

Java 1.1

java.io

Этот класс буферизует поток входных символов и таким образом улучшает эффективность их ввода. Вы создаете BufferedReader путем указания на какой-либо другой поток ввода символов, подлежащих буферизации. На этом этапе вы также можете задать размер буфера, хотя размер по умолчанию обычно вполне подходит. Как правило, такая буферизация применяется с FileReader или InputStreamReader. BufferedReader определяет стандартный набор методов Reader и предоставляет метод readLine(), который считывает строку текста (за исключением символа-разделителя строк) и возвращает ее как String. BufferedReader — это аналог BufferedInputStream, представленный как поток символов. Он также является заменой устаревшему методу readLine(), который представлен в DataInputStream и не способен корректно переводить байты в символы.




```
public class BufferedReader extends Reader {
// Открытые конструкторы
   public BufferedReader(Reader in);
   public BufferedReader(Reader in, int sz);
// Открытые методы экземпляра
   public String readLine() throws IOException;
// Открытые методы, замещающие методы класса Reader
   public void close() throws IOException;
   public void mark(int readAheadLimit) throws IOException;
   public boolean markSupported();                                          // константа
   public int read() throws IOException;
   public int read(char[] cbuf, int off, int len) throws IOException;
   public boolean ready() throws IOException;
   public void reset() throws IOException;
   public long skip(long n) throws IOException;
}
```

Подклассы: LineNumberReader

BufferedWriter Java 1.1

java.io

Этот класс буферизует поток вывода символов, улучшая эффективность вывода путем объединения небольших запросов на запись в один более крупный запрос. Вы создаете BufferedWriter путем указания на какой-либо другой поток вывода символов, которому он посылает свой буферизованный и объединенный вывод. (На этом этапе вы можете задать размер буфера, хотя размер по умолчанию обычно вполне приемлем.) Как правило, такая буферизация применяется с FileWriter или OutputStreamWriter. BufferedWriter определяет стандартные методы write(), flush() и close(), определяемые всеми потоками вывода, но он добавляет к потоку метод newLine(), который выводит разделитель строк, зависящий от конкретной платформы (обычно символ новой строки, символ возврата каретки или оба эти символа). BufferedWriter – это аналог BufferedOutputStream, представленный как поток символов.


```
public class BufferedWriter extends Writer {
// Открытые конструкторы
   public BufferedWriter(Writer out);
   public BufferedWriter(Writer out, int sz);
// Открытые методы экземпляра
   public void newLine() throws IOException;
// Открытые методы, замещающие методы класса Writer
   public void close() throws IOException;
   public void flush() throws IOException;
   public void write(int c) throws IOException;
   public void write(char[] cbuf, int off, int len) throws IOException;
   public void write(String s, int off, int len) throws IOException;
}
```

ByteArrayInputStream Java 1.0

java.io

Этот класс является подклассом InputStream, в котором входные данные берутся из указанного массива значений байтов. Это полезно в том случае, если нужно прочесть данные, находящиеся в памяти, как будто они поступают из файла, канала или сокета. Обратите внимание, что указанный массив байтов не копируется, когда создается ByteArrayInputStream. См. также CharArrayReader.


```
public class ByteArrayInputStream extends java.io.InputStream {
// Открытые конструкторы
   public ByteArrayInputStream(byte[] buf);
   public ByteArrayInputStream(byte[] buf, int offset, int length);
// Открытые методы, замещающие методы класса InputStream
   public int available();                                    // синхронизирован
1.2 public void close() throws IOException;                   // пустой
1.1 public void mark(int readAheadLimit);
```

```
1.1 public boolean markSupported();                                    // константа
   public int read();                                                  // синхронизирован
   public int read(byte[] b, int off, int len);                        // синхронизирован
   public void reset();                                                // синхронизирован
   public long skip(long n);                                           // синхронизирован
// Защищенные поля экземпляра
   protected byte[] buf;
   protected int count;
1.1 protected int mark;
   protected int pos;
}
```

ByteArrayOutputStream

java 1.0

java.io

Этот класс является подклассом `OutputStream`, в котором выходные данные хранятся во внутреннем массиве байтов. Внутренний массив увеличивается по мере необходимости и может быть извлечен при помощи методов `toByteArray()` или `toString()`. Метод `reset()` удаляет все данные, которые в настоящий момент хранятся во внутреннем массиве, и начинает записывать данные с самого начала. См. также `CharArrayWriter`.


```
public class ByteArrayOutputStream extends java.io.OutputStream {
// Открытые конструкторы
   public ByteArrayOutputStream();
   public ByteArrayOutputStream(int size);
// Открытые методы экземпляра
   public void reset();                                                // синхронизирован
   public int size();
   public byte[] toByteArray();                                        // синхронизирован
1.1 public String toString(String enc) throws UnsupportedEncodingException;
   public void writeTo(java.io.OutputStream out) throws IOException;   // синхронизирован
// Открытые методы, замещающие методы класса OutputStream
1.2 public void close() throws IOException;                            // пустой
   public void write(int b);                                           // синхронизирован
   public void write(byte[] b, int off, int len);                      // синхронизирован
// ПОткрытые методы, замещающие методы класса Object
   public String toString();
// Защищенные поля экземпляра
   protected byte[] buf;
   protected int count;
// Устаревшие открытые методы
# public String toString(int hibyte);
}
```

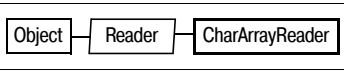
CharArrayReader

Java 1.1

java.io

Этот класс является потоком ввода символов, который использует массив символов в качестве источника тех символов, которые он возвращает. Вы создаете `CharArrayReader` путем указания массива символов (или части массива), из которого он должен

осуществлять чтение. CharArrayReader определяет обычные методы Reader и поддерживает методы mark() и reset(). Обратите внимание на то, что массив символов, передаваемый конструктору CharArrayReader(), не копируется. Это означает, что те изменения, которые вы внесете в элементы массива после создания входящего потока, повлияют на значения, считываемые из массива. CharArrayReader — это аналог ByteArrayInputStream, представленный как массив символов; он подобен StringReader.



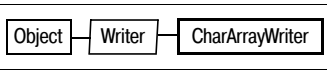

```
public class CharArrayReader extends Reader {
// Открытые конструкторы
   public CharArrayReader(char[] buf);
   public CharArrayReader(char[] buf, int offset, int length);
// Открытые методы, замещающие методы класса Reader
   public void close();
   public void mark(int readAheadLimit) throws IOException;
   public boolean markSupported();  // константа
   public int read() throws IOException;
   public int read(char[] b, int off, int len) throws IOException;
   public boolean ready() throws IOException;
   public void reset() throws IOException;
   public long skip(long n) throws IOException;
// Защищенные поля экземпляра
   protected char[] buf;
   protected int count;
   protected int markedPos;
   protected int pos;
}
```

CharArrayWriter

Java 1.1

java.io

Этот класс является потоком вывода символов, который использует внутренний массив символов в качестве приемника возвращаемых символов. Когда вы создаете CharArrayWriter, то вы можете задать начальный размер массива символов, однако не указывайте массив символов; этот массив контролируется самим CharArrayWriter и увеличивается по мере необходимости, чтобы вместить все символы. Методы toString() и toCharArray() возвращают копию всех символов, записанных в поток, в виде строки и массива символов соответственно. CharArrayWriter определяет стандартные методы write(), flush() и close(), которые определяются всеми подклассами Writer. Он также определяет ряд других полезных методов. size() возвращает количество символов, которые были записаны в поток. reset() приводит поток к его изначальному состоянию (с пустым массивом символов); это более эффективно, чем создание нового CharArrayWriter. И наконец, writeTo() записывает содержимое внутреннего массива символов в какой-либо другой поток символов. CharArrayWriter — это аналог ByteArrayOutputStream, представленный как поток символов; он подобен StringWriter.



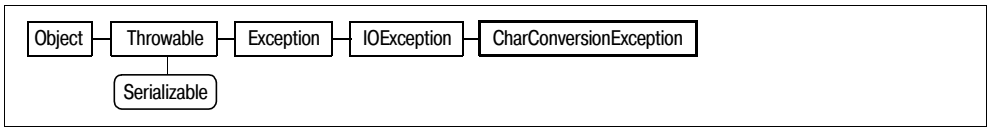
```
public class CharArrayWriter extends Writer {
// Открытые конструкторы
   public CharArrayWriter();
   public CharArrayWriter(int initialSize);
// Открытые методы экземпляра
   public void reset();
   public int size();
   public char[] toCharArray();
   public void writeTo(Writer out) throws IOException;
// Открытые методы, замещающие методы класса Writer
   public void close();  // пустой
   public void flush();  // пустой
   public void write(int c);
   public void write(char[] c, int off, int len);
   public void write(String str, int off, int len);
// Открытые методы, замещающие методы класса Object
   public String toString();
// Защищенные поля экземпляра
   protected char[] buf;
   protected int count;
}
```

CharConversionException

Java 1.1

java.io ***сериализуемое, проверяемое***

CharConversionException обозначает ошибку, которая произошла во время перевода байтов в символы или наоборот.




```
public class CharConversionException extends IOException {
// Открытые конструкторы
   public CharConversionException();
   public CharConversionException(String s);
}
```

DataInput

Java 1.0

java.io

Этот интерфейс определяет методы, необходимые потокам, которые могут считывать примитивные типы данных Java в машинно-независимом двоичном формате. Его реализуют классы DataInputStream и RandomAccessFile. Для получения более подробной информации о методах обратитесь к описанию DataInputStream.

```
public interface DataInput {
// Открытые методы экземпляра
   public abstract boolean readBoolean() throws IOException;
   public abstract byte readByte() throws IOException;
   public abstract char readChar() throws IOException;
   public abstract double readDouble() throws IOException;
```

```
    public abstract float readFloat() throws IOException;
    public abstract void readFully(byte[] b) throws IOException;
    public abstract void readFully(byte[] b, int off, int len) throws IOException;
    public abstract int readInt() throws IOException;
    public abstract String readLine() throws IOException;
    public abstract long readLong() throws IOException;
    public abstract short readShort() throws IOException;
    public abstract int readUnsignedByte() throws IOException;
    public abstract int readUnsignedShort() throws IOException;
    public abstract String readUTF() throws IOException;
    public abstract int skipBytes(int n) throws IOException;
}
```

Реализации: `java.io.DataInputStream`, `ObjectInput`, `RandomAccessFile`, `javax.imageio.stream.ImageInputStream`

Передается методам: `java.io.DataInputStream.readUTF()`, `java.rmi.server.UID.read()`

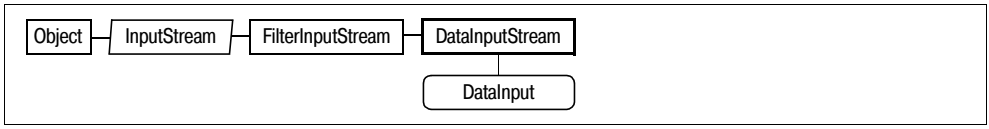
DataInputStream

Java 1.0

java.io

Этот класс является видом `FilterInputStream`. Он позволяет читать бинарные представления примитивных типов данных Java вне зависимости от платформы. Создайте `DataInputStream`, указав при вызове конструктора на конкретный `InputStream`, подлежащий фильтрации. `DataInputStream` считывает только примитивные Java-типы; используйте `ObjectInputStream` для считывания значений объектов.

Большинство методов считывают и возвращают из потока одиночную переменную примитивного Java-типа (в двоичном формате). Методы `readUnsignedByte()` и `readUnsignedShort()` считывают беззнаковые значения и возвращают их как значения `int`, поскольку беззнаковые типы `byte` и `short` не поддерживаются в Java. `read()` считывает данные в массив байтов, блокируя выполнение программы до тех пор, пока не будут доступны какие-нибудь данные. В отличие от него, `readFully()` считывает данные в массив байтов, но блокирует выполнение до получения всех данных, которые были запрошены. `skipBytes()` осуществляет блокировку до тех пор, пока не будет считано и пропущено указанное количество байтов. `readLine()` считывает символы из потока до тех пор, пока он не обнаружит символ новой строки, возврат каретки или пару, состоящую из символа новой строки и возврата каретки. Возвращенная строка не завершается символами новой строки или возврата каретки. Этот метод признан устаревшим Java 1.1; см. `BufferedReader` на предмет альтернативного решения. `readUTF()` считывает строку текста в Unicode, которая закодирована с помощью видоизмененной версии преобразовательного формата UTF-8. UTF-8 – это кодировка, совместимая с Американским стандартным кодом для обмена информацией (ASCII). Она часто применяется для передачи и хранения текста в Unicode. Этот класс использует модифицированную кодировку UTF-8, которая не содержит вложенных нуль-символов.




```
public class DataInputStream extends FilterInputStream implements DataInput {
// Открытые конструкторы
```

```
   public DataInputStream(java.io.InputStream in);
// Открытые методы класса
   public static final String readUTF(DataInput in) throws IOException;
// Методы, реализующие DataInput
   public final boolean readBoolean() throws IOException;
   public final byte readByte() throws IOException;
   public final char readChar() throws IOException;
   public final double readDouble() throws IOException;
   public final float readFloat() throws IOException;
   public final void readFully(byte[] b) throws IOException;
   public final void readFully(byte[] b, int off, int len) throws IOException;
   public final int readInt() throws IOException;
   public final long readLong() throws IOException;
   public final short readShort() throws IOException;
   public final int readUnsignedByte() throws IOException;
   public final int readUnsignedShort() throws IOException;
   public final String readUTF() throws IOException;
   public final int skipBytes(int n) throws IOException;
// Открытые методы, замещающие методы класса FilterInputStream
   public final int read(byte[] b) throws IOException;
   public final int read(byte[] b, int off, int len) throws IOException;
// Устаревшие открытые методы
# public final String readLine() throws IOException;                    // Реализует:DataInput
}
```

Передается методам: javax.swing.text.html.parser.DTD.read()

DataOutput Java 1.0

java.io

Этот интерфейс определяет методы, требующиеся для потоков, которые могут записывать примитивные типы данных Java в машинно-независимом двоичном формате. Он реализуется классами DataOutputStream и RandomAccessFile. Более детальная информация о методах представлена в описании DataOutputStream.

```
public interface DataOutput {
// Открытые методы экземпляра
   public abstract void write(byte[] b) throws IOException;
   public abstract void write(int b) throws IOException;
   public abstract void write(byte[] b, int off, int len) throws IOException;
   public abstract void writeBoolean(boolean v) throws IOException;
   public abstract void writeByte(int v) throws IOException;
   public abstract void writeBytes(String s) throws IOException;
   public abstract void writeChar(int v) throws IOException;
   public abstract void writeChars(String s) throws IOException;
   public abstract void writeDouble(double v) throws IOException;
   public abstract void writeFloat(float v) throws IOException;
   public abstract void writeInt(int v) throws IOException;
   public abstract void writeLong(long v) throws IOException;
   public abstract void writeShort(int v) throws IOException;
   public abstract void writeUTF(String str) throws IOException;
}
```

Реализации: java.io.DataOutputStream, ObjectOutput, RandomAccessFile, javax.imageio.stream.ImageOutputStream

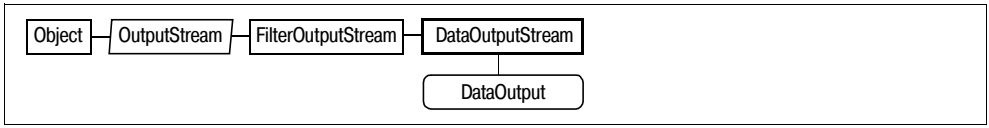
Передается методам: java.rmi.server.UID.write()

DataOutputStream Java 1.0

java.io

Этот класс является подклассом FilterOutputStream. Он позволяет записывать примитивные типы данных Java в машинно-независимом двоичном формате. Создайте DataOutputStream, указав в запросе к конструктору на конкретный OutputStream, подлежащий фильтрации. DataOutputStream имеет методы, которые выводят только примитивные типы; для вывода значений объектов применяйте ObjectOutputStream.

Многие из методов этого класса записывают одиночный примитивный Java-тип в выходной поток (в двоичном формате). write() записывает один байт, массив или подмассив байтов. flush() выводит любые буферизованные данные. size() возвращает количество байтов, которые были записаны на текущий момент. writeUTF() выводит Java-строку в символах Unicode, используя видоизмененную версию преобразовательного формата UTF-8. UTF-8 – это кодировка, совместимая с Американским стандартным кодом для обмена информацией (ASCII). Она часто применяется для передачи и хранения текста в Unicode. За исключением метода writeUTF(), этот класс применяется для двоичного вывода данных. Текстовый вывод должен производиться при помощи PrintWriter (или PrintStream в Java 1.0).



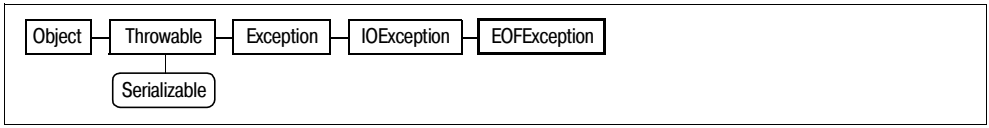

```
public class DataOutputStream extends FilterOutputStream implements DataOutput {
// Открытые конструкторы
   public DataOutputStream(java.io.OutputStream out);
// Открытые методы экземпляра
   public final int size();
// Методы, реализующие DataOutput
   public void write(int b) throws IOException;                              // синхронизирован
   public void write(byte[] b, int off, int len) throws IOException;         // синхронизирован
   public final void writeBoolean(boolean v) throws IOException;
   public final void writeByte(int v) throws IOException;
   public final void writeBytes(String s) throws IOException;
   public final void writeChar(int v) throws IOException;
   public final void writeChars(String s) throws IOException;
   public final void writeDouble(double v) throws IOException;
   public final void writeFloat(float v) throws IOException;
   public final void writeInt(int v) throws IOException;
   public final void writeLong(long v) throws IOException;
   public final void writeShort(int v) throws IOException;
   public final void writeUTF(String str) throws IOException;
// Открытые методы, замещающие методы класса FilterOutputStream
   public void flush() throws IOException;
// Защищенные поля экземпляра
   protected int written;
}
```

EOFException

Java 1.0

java.io *сериализуемое, проверяемое*

Исключение `EOFException` – это `IOException`, которое обозначает конец файла.




```
public class EOFException extends IOException {
// Открытые конструкторы
   public EOFException();
   public EOFException(String s);
}
```

Externalizable

Java 1.1

java.io *сериализуемый*

Этот интерфейс определяет методы, которые должны быть реализованы объектом, желающим иметь полный контроль над процессом своей сериализации. Методы `writeExternal()` и `readExternal()` должны быть реализованы для записи и считывания данных объекта в каком-либо произвольном формате с использованием методов интерфейсов `DataOutput` и `DataInput`. Объекты `Externalizable` должны сериализовать свои собственные поля; кроме того, они отвечают за сериализацию полей родительских классов. Большая часть объектов не нуждается в определении специализированного формата вывода и может использовать интерфейс `Serializable` вместо `Externalizable` для сериализации.


```
public interface Externalizable extends Serializable {
// Открытые методы экземпляра
   public abstract void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
   public abstract void writeExternal(ObjectOutput out) throws IOException;
}
```

Реализации: `java.awt.datatransfer.DataFlavor`, `java.rmi.server.RemoteRef`

File

Java 1.0

java.io *сериализуемый, сравнимый*

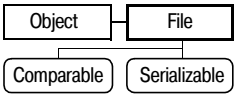
Этот класс поддерживает определение имен файлов и каталогов, которое не зависит от платформы. Кроме того, он предоставляет методы для перечисления файлов в каталоге; для проверки существования, читаемости, записываемости, типа, размера и времени изменения файлов и каталогов; для создания новых каталогов; для переименования файлов и каталогов; для создания и удаления временных файлов и файлов блокировки. Константы, определяемые этим классом, являются символами разделения пути и каталога, которые зависят от платформы. Они доступны как `String` и `char`.

getName() возвращает имя File без имен каталогов. getPath() возвращает полное имя файла, включая имя каталога. getParent() и getParentFile() возвращают каталог, который содержит File; единственное различие между этими методами заключается в том, что один из них возвращает String, а другой – File. isAbsolute() проверяет, задан ли File абсолютным именем. Если нет, то getAbsolutePath() возвращает абсолютное имя файла, созданное путем добавления относительного имени файла к текущему каталогу. getAbsoluteFile() возвращает эквивалентный абсолютный объект File. getCanonicalPath() и getCanonicalFile() похожи: они возвращают абсолютное имя файла или объект File, переведенный в его каноническую форму, зависимую от системы. Такая возможность полезна при сравнении двух объектов File, когда нужно понять, ссылаются ли они на один и тот же файл или каталог. В Java 1.4 и последующих версиях метод toURI() возвращает объект java.net.URI, который применяет схему file: для присваивания имени этому файлу. Преобразование из file в URI можно изменить на прямо противоположное, передав объект file: URI конструктору File().

exists(), canWrite(), canRead(), isFile(), isDirectory() и isHidden() выполняют очевидные тесты с конкретным File. length() возвращает длину файла. lastModified() возвращает время изменения файла, что следует применять только для сравнения с другими временны*ми* характеристиками файла, а не интерпретировать как какой-либо конкретный формат времени). setLastModified() позволяет задавать время модификации; setReadOnly() устанавливает какому-либо файлу или каталогу атрибут «только для чтения».

list() возвращает имена всех тех элементов в каталоге, которые не отвергнуты необязательным FilenameFilter. listFiles() возвращает массив объектов File, которые представляют все элементы в каталоге, не отвергнутые необязательным FilenameFilter или FileFilter. listRoots() возвращает массив объектов File, которые представляют все корневые каталоги в системе. Например, в системах Unix обычно существует только один корень, /. В системах Windows присутствуют разные корни для каждого имени дисковода, например *c:*\, *d:*\ и *e:*\.

mkdir() создает каталог, а mkdirs() создает все каталоги, описанные в объекте File. renameTo() переименовывает файл или каталог; delete() удаляет файл или каталог. До версии Java 1.2 класс File не предоставлял ни одного способа создания файла; эту задачу можно выполнить с помощью FileOutputStream. В Java 1.2 были добавлены два специальных метода для создания файла. Статический метод createTempFile() возвращает объект File, который ссылается на вновь созданный пустой файл с уникальным именем. Имя начинается с заданной приставки, которая должна быть длиной как минимум в три символа и заканчиваться заданным суффиксом. Одна из версий этого метода создает файл в указанном каталоге, а другая создает его во временном каталоге системы. Приложения могут использовать временные файлы в любых целях без опасения перезаписи файлов, принадлежащих другим приложениям. Другой метод создания файла в Java 1.2 – это createNewFile(). Этот метод экземпляра пытается создать новый, пустой файл с именем, указанным объектом File. Если это удается, он возвращает true. Если файл уже существует, то он возвращает false. Метод createNewFile() выполняется атомарно, поэтому он полезен для блокировки файла и других схем взаимного исключения. При работе с createTempFile() или createNewFile() рассмотрите возможность использования deleteOnExit(), чтобы запрашивать удаление файлов, когда виртуальная машина Java корректно завершает работу.




```
public class File implements Comparable, Serializable {
// Открытые конструкторы
   public File(String pathname);
1.4 public File(java.net.URI uri);
   public File(String parent, String child);
   public File(File parent, String child);
// Открытые константы
   public static final String pathSeparator;
   public static final char pathSeparatorChar;
   public static final String separator;
   public static final char separatorChar;
// Открытые методы класса
1.2 public static File createTempFile(String prefix, String suffix) throws IOException;
1.2 public static File createTempFile(String prefix, String suffix, File directory)
       throws IOException;
1.2 public static File[] listRoots();
// Методы доступа к свойствам (по имени свойства)
   public boolean isAbsolute();
1.2 public File getAbsoluteFile();
   public String getAbsolutePath();
1.2 public File getCanonicalFile() throws IOException;
1.1 public String getCanonicalPath() throws IOException;
   public boolean isDirectory();
   public boolean isFile();
1.2 public boolean isHidden();
   public String getName();
   public String getParent();
1.2 public File getParentFile();
   public String getPath();
// Открытые методы экземпляра
   public boolean canRead();
   public boolean canWrite();
1.2 public int compareTo(File pathname);
1.2 public boolean createNewFile() throws IOException;
   public boolean delete();
1.2 public void deleteOnExit();
   public boolean exists();
   public long lastModified();
   public long length();
   public String[] list();
   public String[] list(FilenameFilter filter);
1.2 public File[] listFiles();
1.2 public File[] listFiles(FilenameFilter filter);
1.2 public File[] listFiles(java.io.FileFilter filter);
   public boolean mkdir();
   public boolean mkdirs();
   public boolean renameTo(File dest);
1.2 public boolean setLastModified(long time);
1.2 public boolean setReadOnly();
1.4 public java.net.URI toURI();
1.2 public java.net.URL toURL() throws java.net.MalformedURLException;
```

```
// Методы, реализующие Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

FileDescriptor Java 1.0

java.io

Этот класс является платформо-независимым представлением низкоуровневого дескриптора открытого файла или сокета. Статические переменные `in`, `out` и `err` являются объектами `FileDescriptor`, которые представляют соответственно стандартные входные/выходные потоки и поток сообщений об ошибках. Для создания `FileDescriptor` не существует открытых конструкторов. Вы можете получить его при помощи метода `getFD()` классов `FileInputStream`, `FileOutputStream` или `RandomAccessFile`.

```
public final class FileDescriptor {
// Открытые конструкторы
   public FileDescriptor();
// Открытые константы
   public static final FileDescriptor err;
   public static final FileDescriptor in;
   public static final FileDescriptor out;
// Открытые методы экземпляра
1.1 public void sync() throws SyncFailedException;  // зависит от платформы
   public boolean valid();
}
```

Передается методам: `FileInputStream.FileInputStream()`, `FileOutputStream.FileOutputStream()`, `FileReader.FileReader()`, `FileWriter.FileWriter()`, `SecurityManager.{checkRead(), checkWrite()}`

Возвращается методами: `FileInputStream.getFD()`, `FileOutputStream.getFD()`, `RandomAccessFile.getFD()`, `java.net.DatagramSocketImpl.getFileDescriptor()`, `java.net.SocketImpl.getFileDescriptor()`

Экземпляры: `FileDescriptor.{err, in, out}`, `java.net.DatagramSocketImpl.fd`, `java.net.SocketImpl.fd`

FileFilter Java 1.2

java.io

Этот интерфейс определяет метод `accept()`, который фильтрует список файлов. Вы можете прочитать содержимое каталога, вызвав метод `listFiles()` объекта `File`, который представляет необходимый каталог. Если вам необходим отфильтрованный список, например список файлов, не содержащий подкаталогов, или список файлов, чьи имена заканчиваются на `.class`, то вы можете передать объект `FileFilter` для `listFiles()`. Для каждого элемента каталога объект `File` передается методу `accept()`. Если `accept()` возвращает `true`, то `File` включается в возвращаемое значение `listFiles()`. Если `accept()` возвращает `false`, то этот элемент не включается в список. `FileFilter`

введен в Java 1.2. Используйте FilenameFilter, если требуется совместимость с предыдущими версиями Java или если вы предпочитаете фильтровать имена файлов (то есть объекты String), а не объекты File.

```
public interface FileFilter {
// Открытые методы экземпляра
   public abstract boolean accept(File pathname);
}
```

Передается методам: File.listFiles()

FileInputStream

Java 1.0

java.io

Этот класс является подклассом InputStream, который считывает байты из файла, указанного по имени или посредством объекта File или FileDescriptor. read() считывает байт или массив байтов из файла. Он возвращает –1, когда достигнут конец файла. При чтении двоичных данных этот класс обычно используется совместно с BufferedInputStream и DataInputStream. Для прочтения текста его обычно применяют с InputStreamReader и BufferedReader. Вызовите close() для закрытия файла, когда ввод уже не нужен.

В Java 1.4 и последующих версиях применяйте getChannel() для получения объекта FileChannel, чтобы читать данные из базового файла с использованием нового API ввода/вывода java.nio и его подпакетов.


```
public class FileInputStream extends java.io.InputStream {
// Открытые конструкторы
   public FileInputStream(String name) throws FileNotFoundException;
   public FileInputStream(File file) throws FileNotFoundException;
   public FileInputStream(FileDescriptor fdObj);
// Открытые методы экземпляра
1.4 public java.nio.channels.FileChannel getChannel();
   public final FileDescriptor getFD() throws IOException;
// Открытые методы, замещающие методы класса InputStream
   public int available() throws IOException;                          // зависит от платформы
   public void close() throws IOException;
   public int read() throws IOException;                               // зависит от платформы
   public int read(byte[] b) throws IOException;
   public int read(byte[] b, int off, int len) throws IOException;
   public long skip(long n) throws IOException;                        // зависит от платформы
// Замещенные защищенные методы класса Object
   protected void finalize() throws IOException;
}
```

FilenameFilter

Java 1.0

java.io

Этот интерфейс определяет метод accept(), который должен быть реализован любым объектом, фильтрующим имена файлов, то есть выбирающим подмножество имен

файлов из списка имен файлов. В Java не реализованы стандартные классы FilenameFilter, но объекты, которые реализуют этот интерфейс, используются объектом java.awt.FileDialog и методом File.list(). Типичный объект FilenameFilter будет осуществлять проверку того, что указанный File представляет файл (а не каталог), является читаемым (и по возможности перезаписываемым), а его имя заканчивается на требуемое расширение.

```
public interface FilenameFilter {
// Открытые методы экземпляра
   public abstract boolean accept(File dir, String name);
}
```

Передается методам: java.awt.FileDialog.setFilenameFilter(), java.awt.peer.FileDialogPeer.setFilenameFilter(), File.{list(), listFiles()}

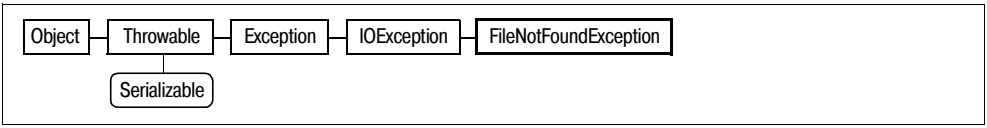
Возвращается методами: java.awt.FileDialog.getFilenameFilter()

FileNotFoundException

Java 1.0

java.io ***сериализуемое, проверяемое***

FileNotFoundException — это исключение IOException, которое свидетельствует о том, что указанный файл не найден.




```
public class FileNotFoundException extends IOException {
// Открытые конструкторы
   public FileNotFoundException();
   public FileNotFoundException(String s);
}
```

Генерируется методами: FileInputStream.FileInputStream(), FileOutputStream.FileOutputStream(), FileReader.FileReader(), RandomAccessFile.RandomAccessFile(), javax.imageio.stream.FileImageInputStream.FileImageInputStream(), javax.imageio.stream.FileImageOutputStream.FileImageOutputStream()

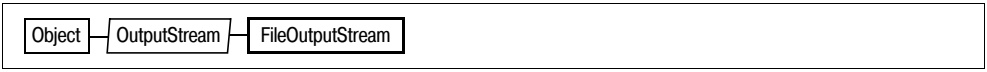
FileOutputStream

Java 1.0

java.io

Этот класс является подклассом OutputStream, который записывает данные в файл, указанный по имени или посредством объекта File или FileDescriptor. Если указанный файл уже существует, то FileOutputStream может быть сконфигурирован либо для перезаписи, либо для дополнения существующего файла. write() записывает байт или массив байтов в файл. Для записи двоичных данных этот класс применяется совместно с BufferedOutputStream и DataOutputStream. Для записи текста он обычно используется с PrintWriter, BufferedWriter и OutputStreamWriter либо применяется подходящий класс FileWriter. Используйте close() для закрытия FileOutputStream, если в него больше не будут записываться выходные данные.

В Java 1.4 и последующих версиях используйте `getChannel()` для получения объекта `FileChannel`, чтобы записать данные в базовый файл с помощью нового API ввода/вывода `java.nio` и его подпакетов.




```
public class FileOutputStream extends java.io.OutputStream {
// Открытые конструкторы
   public FileOutputStream(FileDescriptor fdObj);
   public FileOutputStream(File file) throws FileNotFoundException;
   public FileOutputStream(String name) throws FileNotFoundException;
1.1 public FileOutputStream(String name, boolean append) throws FileNotFoundException;
1.4 public FileOutputStream(File file, boolean append) throws FileNotFoundException;
// Открытые методы экземпляра
1.4 public java.nio.channels.FileChannel getChannel();
   public final FileDescriptor getFD() throws IOException;
// Открытые методы, замещающие методы класса OutputStream
   public void close() throws IOException;
   public void write(int b) throws IOException;                          // зависит от платформы
   public void write(byte[] b) throws IOException;
   public void write(byte[] b, int off, int len) throws IOException;
// Замещенные защищенные методы класса Object
   protected void finalize() throws IOException;
}
```

FilePermission

Java 1.2

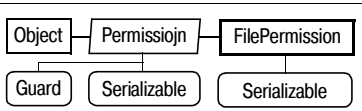
java.io ***сериализуемый***

Этот класс – `java.security.Permission`, который управляет доступом к локальной файловой системе. `FilePermission` имеет имя, или целевой объект (указывает на то, к какому файлу или файлам он относится), и список действий, которые могут быть выполнены с этим файлом или файлами (действия разделены запятыми). Поддерживаются следующие действия: чтение, запись, удаление и выполнение. Права на чтение и запись требуются любым методам, которые читают или записывают файл. Право на удаление необходимо `File.delete()`, а право на выполнение – `Runtime.exec()`.

Имя `FilePermission` может быть простым именем файла или каталога. Кроме того, `FilePermission` поддерживает использование определенных подстановочных знаков для указания права, которое применимо более чем к одному файлу. Если имя `FilePermission` является именем каталога, за которым следуют символы «/*» («*» в случае с платформами Windows), то оно указывает на все файлы в названном каталоге. Если имя является именем каталога, за которым следуют символы «/-» (либо «\-» для платформы Windows), то оно указывает на все файлы в каталоге и, рекурсивно, на все файлы во всех подкаталогах. Одна звездочка (*) указывает на все файлы в текущем каталоге, а одно тире (–) указывает на все файлы в текущем каталоге и его подкаталогах. Наконец, особое имя <<ALL FILES>> обозначает все файлы в файловой системе.

Приложениям не нужно использовать этот класс напрямую. Тем не менее он может понадобиться программистам, пишущим код системного уровня, и системным администраторам, отвечающим за конфигурацию системы безопасности. Будьте очень осмотрительны при предоставлении любых типов `FilePermission`. Ограничение досту-

па к файлам (в особенности доступа для записи) является одним из краеугольных камней механизма защиты применительно к ненадежному коду.



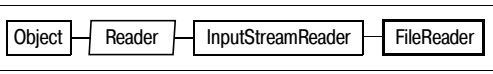

```
public final class FilePermission extends java.security.Permission implements Serializable {
// Открытые конструкторы
   public FilePermission(String path, String actions);
// Открытые методы, замещающие методы класса Permission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
   public boolean implies(java.security.Permission p);
   public java.security.PermissionCollection newPermissionCollection();
}
```

FileReader

Java 1.1

java.io

`FileReader` является удобным подклассом `InputStreamReader`, который полезен, когда вам необходимо прочесть из файла текст, а не двоичные данные. Вы создаете `FileReader`, указав файл для чтения в любой из трех возможных форм. Внутри конструктора `FileReader` создает `FileInputStream` для чтения байтов из указанного файла. Он применяет функциональность родительского класса `InputStreamReader` для перевода этих байтов из символов локальной кодировки в символы Unicode, используемые в Java. `FileReader` является тривиальным подклассом `InputStreamReader`, поэтому он не определяет какие-либо методы `read()` или другие собственные методы. Напротив, он наследует все методы из родительского класса. Если вам необходимо прочесть символы Unicode из файла, который использует какую-либо другую кодировку вместо кодировки по умолчанию, соответствующей данному региону (locale), то вы должны создать ваш собственный `InputStreamReader` для перевода байтов в символы.




```
public class FileReader extends InputStreamReader {
// Открытые конструкторы
   public FileReader(FileDescriptor fd);
   public FileReader(File file) throws FileNotFoundException;
   public FileReader(String fileName) throws FileNotFoundException;
}
```

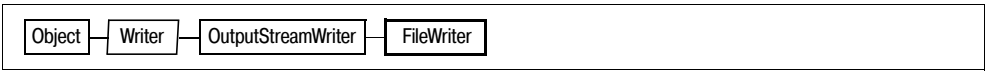
FileWriter

Java 1.1

java.io

`FileWriter` является удобным подклассом `OutputStreamWriter`, который полезен, когда вам необходимо записать в файл текст, а не двоичные данные. Вы создаете `FileWriter` путем указания файла, в который будет производиться запись. Вы можете опреде-

лить необходимость добавлять данные к концу существующего файла вместо того, чтобы перезаписывать этот файл. Класс FileWriter создает внутренний FileOutputStream для записи байтов в указанный файл. Он использует функциональность родительского класса OutputStreamWriter для перевода символов Unicode, записанных в поток, в байты с помощью кодировки по умолчанию, соответствующей данному региону. Если вам необходима кодировка, отличающаяся от кодировки по умолчанию, то вы не можете использовать FileWriter; в таком случае нужно создать собственный OutputStreamWriter и FileOutputStream. FileWriter является тривиальным подклассом OutputStreamWriter, поэтому он не определяет какие-либо собственные методы, а просто наследует их от родительского класса.




```
public class FileWriter extends OutputStreamWriter {
// Открытые конструкторы
   public FileWriter(File file) throws IOException;
   public FileWriter(FileDescriptor fd);
   public FileWriter(String fileName) throws IOException;
1.4 public FileWriter(File file, boolean append) throws IOException;
   public FileWriter(String fileName, boolean append) throws IOException;
}
```

FilterInputStream

Java 1.0

java.io

Этот класс предоставляет определения методов, которые требуются для фильтрации данных, полученных из InputStream, указанного при создании FilterInputStream. От него должен быть создан подкласс, реализующий какой-либо алгоритм фильтрации, поскольку непосредственное создание экземпляра данного класса невозможно. См. подклассы BufferedInputStream, DataInputStream и PushbackInputStream.


```
public class FilterInputStream extends java.io.InputStream {
// Защищенные конструкторы
   protected FilterInputStream(java.io.InputStream in);
// Открытые методы, замещающие методы класса InputStream
   public int available() throws IOException;
   public void close() throws IOException;
   public void mark(int readlimit);                                    // синхронизирован
   public boolean markSupported();
   public int read() throws IOException;
   public int read(byte[] b) throws IOException;
   public int read(byte[] b, int off, int len) throws IOException;
   public void reset() throws IOException;                             // синхронизирован
   public long skip(long n) throws IOException;
// Защищенные поля экземпляра
   protected java.io.InputStream in;
}
```

Подклассы: BufferedInputStream, java.io.DataInputStream, LineNumberInputStream, PushbackInputStream, java.security.DigestInputStream, java.util.zip.CheckedInputStream, java.util.zip.InflaterInputStream, javax.crypto.CipherInputStream, javax.swing.ProgressMonitorInputStream

FilterOutputStream

Java 1.0

java.io

Этот класс обеспечивает определения методов, необходимых для фильтрации данных, которые нужно записать в OutputStream, указываемый при создании FilterOutputStream. От него должен быть создан подкласс, реализующий какой-либо алгоритм фильтрации, поскольку непосредственное создание экземпляра данного класса невозможно. См. подклассы BufferedOutputStream, DataOutputStream.


```
public class FilterOutputStream extends java.io.OutputStream {
// Открытые конструкторы
   public FilterOutputStream(java.io.OutputStream out);
// Открытые методы, замещающие методы класса OutputStream
   public void close() throws IOException;
   public void flush() throws IOException;
   public void write(int b) throws IOException;
   public void write(byte[] b) throws IOException;
   public void write(byte[] b, int off, int len) throws IOException;
// Защищенные поля экземпляра
   protected java.io.OutputStream out;
}
```

Подклассы: BufferedOutputStream, java.io.DataOutputStream, PrintStream, java.security.DigestOutputStream, java.util.zip.CheckedOutputStream, java.util.zip.DeflaterOutputStream, javax.crypto.CipherOutputStream

FilterReader

Java 1.1

java.io

Этот абстрактный класс выступает в качестве родительского класса для входных потоков символов, которые считывают данные из какого-либо другого входного потока символов. С его помощью символы фильтруются каким-либо образом, а затем отфильтрованные данные возвращаются при вызове метода read(). FilterReader объявляется абстрактным, поэтому создать его экземпляр невозможно. Но ни один из его методов сам по себе не является абстрактным: они вызывают запрошенную операцию над входным потоком, передаваемым конструктору FilterReader(). Если создать экземпляр FilterReader, то можно обнаружить, что он является нулевым фильтром – он просто считывает символы из указанного входного потока и возвращает их без какой-либо фильтрации.

FilterReader реализует нулевой фильтр, поэтому он является идеальным родительским классом для классов, которые желают реализовать простые фильтры, но не хотят замещать все методы Reader. Для создания собственного отфильтрованного входного потока символов следует создать подкласс FilterReader и переопределить оба его метода read(), чтобы выполнять требуемую операцию фильтрации. Заметьте, что вы можете реализо-

вать один из методов read() посредством другого, то есть реализовать фильтрацию только один раз. Вспомните о том, что другие методы read(), определенные в классе Reader, реализуются посредством этих методов, поэтому у вас нет необходимости их замещать. В некоторых случаях понадобится заменить другие методы FilterReader и обеспечить методы или конструкторы, которые специфичны для вашего подкласса. FilterReader является аналогом, состоящим из потока символов, для FilterInputStream.


```
public abstract class FilterReader extends Reader {
// Защищенные конструкторы
   protected FilterReader(Reader in);
// Открытые методы, замещающие методы класса Reader
   public void close() throws IOException;
   public void mark(int readAheadLimit) throws IOException;
   public boolean markSupported();
   public int read() throws IOException;
   public int read(char[] cbuf, int off, int len) throws IOException;
   public boolean ready() throws IOException;
   public void reset() throws IOException;
   public long skip(long n) throws IOException;
// Защищенные поля экземпляра
   protected Reader in;
}
```

Подклассы: PushbackReader

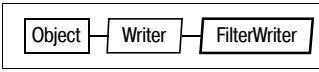
FilterWriter

Java 1.1

java.io

Этот абстрактный класс выступает в качестве родительского класса для выходных потоков символов, которые фильтруют записанные в них данные прежде, чем записать их в какой-либо другой выходной поток символов. FilterWriter объявляется абстрактным, чтобы нельзя было создать его экземпляр. Но ни один из его методов сам по себе не является абстрактным: они просто вызывают соответствующий метод в выходном потоке, который был передан конструктору FilterWriter. Если создать экземпляр объекта FilterWriter, то можно обнаружить, что он выступает в качестве нулевого фильтра, то есть просто передает записанные в него символы без какой-либо фильтрации.

FilterWriter реализует нулевой фильтр, поэтому он является идеальным родительским классом для классов, которые желают реализовать простые фильтры без необходимости замещать все методы Writer. Для создания собственного отфильтрованного выходного потока символов следует создать подкласс FilterWriter и заменить все его методы write(), чтобы выполнять требуемую операцию фильтрации. Заметьте, что вы можете реализовать два метода write() посредством третьего, то есть реализовать алгоритм фильтрации только один раз. В некоторых случаях необходимо заменить другие методы класса Writer и добавить дополнительные методы или конструкторы, которые специфичны для вашего подкласса. FilterWriter является аналогом, состоящим из потока символов, для FilterOutputStream.




```
public abstract class FilterWriter extends Writer {
// Защищенные конструкторы
   protected FilterWriter(Writer out);
// Открытые методы, замещающие методы класса Writer
   public void close() throws IOException;
   public void flush() throws IOException;
   public void write(int c) throws IOException;
   public void write(char[] cbuf, int off, int len) throws IOException;
   public void write(String str, int off, int len) throws IOException;
// Защищенные поля экземпляра
   protected Writer out;
}
```

InputStream

Java 1.0

java.io

Этот абстрактный класс является родительским классом всех входных потоков. Он определяет основные методы для чтения символов, предоставляемые всеми классами входящих потоков. `read()` считывает один байт или массив (или подмассив) байтов. Он возвращает считанный байт, количество считанных байтов или –1, если был достигнут конец файла. `skip()` пропускает указанное количество байтов ввода. `available()` возвращает количество байтов, которые могут быть прочитаны без блокировки. `close()` закрывает входной поток и освобождает все системные ресурсы, которые с ним ассоциированы. Поток не должен быть использован после вызова `close()`.

Если `markSupported()` возвращает `true` для данного `InputStream`, то такой поток поддерживает методы `mark()` и `reset()`. `mark()` запоминает текущую позицию во входном потоке, чтобы `reset()` мог вернуться к такой позиции (если только количество байтов, прочитанных между вызовами `mark()` и `reset()`, не превысит заданного предела). См. также `Reader`.

```
public abstract class InputStream {
// Открытые конструкторы
   public InputStream();
// Открытые методы экземпляра
   public int available() throws IOException;                          // константа
   public void close() throws IOException;                             // пустой
   public void mark(int readlimit);                                    // синхронизирован; пустой
   public boolean markSupported();                                     // константа
   public abstract int read() throws IOException;
   public int read(byte[] b) throws IOException;
   public int read(byte[] b, int off, int len) throws IOException;
   public void reset() throws IOException;                             // синхронизирован
   public long skip(long n) throws IOException;
}
```

Подклассы: `ByteArrayInputStream`, `FileInputStream`, `FilterInputStream`, `ObjectInputStream`, `PipedInputStream`, `SequenceInputStream`, `StringBufferInputStream`, `javax.sound.sampled.AudioInputStream`, `org.omg.CORBA.portable.InputStream`

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: `FilterInputStream.in`, `System.in`

InputStreamReader Java 1.1

java.io

Этот класс является входным потоком символов, который использует входной поток байтов в качестве источника данных. Он считывает байты из указанного InputStream и переводит их в символы Unicode в соответствии с конкретной кодировкой символов, зависящей от платформы и региона. Эта важная функциональная особенность, связанная с локализацией, реализована в Java 1.1 и последующих версиях. InputStreamReader поддерживает стандартные методы класса Reader. Он также имеет метод getEncoding(), который возвращает имя кодировки, используемой для перевода байтов в символы.

При создании InputStreamReader вы указываете InputStream, из которого InputStreamReader должен читать байты. Кроме того, вы можете задать название кодировки символов, которая используется этими байтами. Если вы не указываете имя кодировки, то InputStreamReader использует кодировку по умолчанию, соответствующую предопределенному региону. В Java 1.4 и последующих версиях этот класс использует средства преобразования набора символов из пакета java.nio.charset и позволяет явно указать Charset или CharsetDecoder, который следует использовать. В версиях, предшествующих 1.4, этот класс позволяет указывать имя кодировки набора символов.

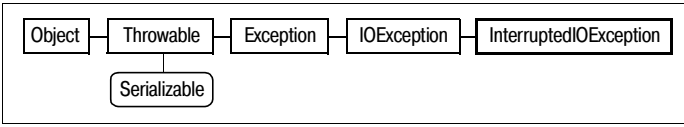

```
public class InputStreamReader extends Reader {
// Открытые конструкторы
   public InputStreamReader(java.io.InputStream in);
   public InputStreamReader(java.io.InputStream in, String charsetName)
        throws UnsupportedEncodingException;
1.4 public InputStreamReader(java.io.InputStream in, java.nio.charset.Charset cs);
1.4 public InputStreamReader(java.io.InputStream in, java.nio.charset.CharsetDecoder dec);
// Открытые методы экземпляра
   public String getEncoding();
// Открытые методы, замещающие методы класса Reader
   public void close() throws IOException;
   public int read() throws IOException;
   public int read(char[] cbuf, int offset, int length) throws IOException;
   public boolean ready() throws IOException;
}
```

Подклассы: FileReader

InterruptedIOException Java 1.0

java.io ***сериализуемое, проверяемое***

InterruptedIOEException — это исключение IOException, свидетельствующее о том, что операция ввода или вывода была прервана. Поле bytesTransferred содержит количество байтов, прочитанных или записанных до прерывания операции.




```
public class InterruptedIOException extends IOException {
// Открытые конструкторы
   public InterruptedIOException();
   public InterruptedIOException(String s);
// Открытые поля экземпляра
   public int bytesTransferred;
}
```

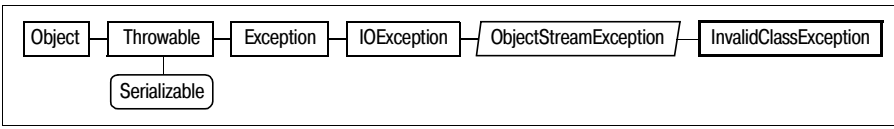
Подклассы: java.net.SocketTimeoutException

InvalidClassException

Java 1.1

java.io ***сериализуемое, проверяемое***

InvalidClassException свидетельствует о том, что механизм сериализации столкнулся с одной из нескольких возможных проблем, связанных с классом объекта, который сериализуется или десериализуется. Поле classname должно содержать имя рассматриваемого класса, а метод getMessage() замещается, чтобы возвращать имя класса вместе с сообщением.



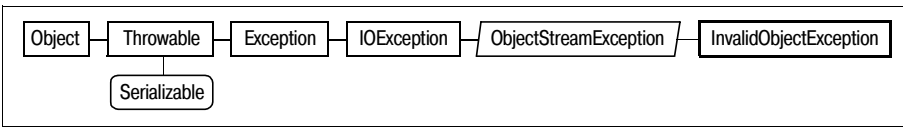

```
public class InvalidClassException extends ObjectStreamException {
// Открытые конструкторы
   public InvalidClassException(String reason);
   public InvalidClassException(String cname, String reason);
// Открытые методы, замещающие методы класса Throwable
   public String getMessage();
// Открытые поля экземпляра
   public String classname;
}
```

InvalidObjectException

Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение должно генерироваться методом validateObject() объекта, который реализует интерфейс ObjectInputValidation, если десериализуемый объект по какой-либо причине не проходит тест на подтверждение входных данных.




```
public class InvalidObjectException extends ObjectStreamException {
```

```
// Открытые конструкторы
   public InvalidObjectException(String reason);
}
```

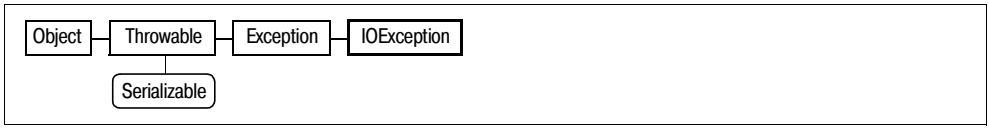
Генерируется методами: java.awt.font.TextAttribute.readResolve(), ObjectInputStream.registerValidation(), ObjectInputValidation.validateObject(), java.text.AttributedCharacterIterator.Attribute.readResolve(), java.text.DateFormat.Field.readResolve(), java.text.MessageFormat.Field.readResolve(), java.text.NumberFormat.Field.readResolve()

IOException

Java 1.0

java.io ***сериализуемое, проверяемое***

IOException свидетельствует о том, что во время ввода или вывода есть исключительное условие. В этом классе представлены несколько более специализированных подклассов. См. EOFException, FileNotFoundException, InterruptedIOException и UTFDataFormatException.




```
public class IOException extends Exception {
// Открытые конструкторы
   public IOException();
   public IOException(String s);
}
```

Подклассы: Классов слишком много, чтобы их перечислить.

Передается методам: java.awt.print.PrinterIOException.PrinterIOException()

Возвращается методами: java.awt.print.PrinterIOException.getIOException()

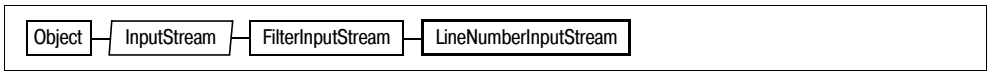
Генерируется методами: Методов слишком много, чтобы их перечислить.

LineNumberInputStream

Java 1.0, устарел в Java 1.1

java.io

Этот класс является FilterInputStream, который отслеживает количество прочитанных строк данных. getLineNumber() возвращает номер текущей строки; setLineNumber() устанавливает номер текущей строки. Последующим строкам присваиваются номера, начиная с этого номера. В Java 1.1 этот класс был признан устаревшим, поскольку он некорректно переводит байты в символы. Вместо него применяйте LineNumberReader.




```
public class LineNumberInputStream extends FilterInputStream {
// Открытые конструкторы
   public LineNumberInputStream(java.io.InputStream in);
// Открытые методы экземпляра
   public int getLineNumber();
   public void setLineNumber(int lineNumber);
```

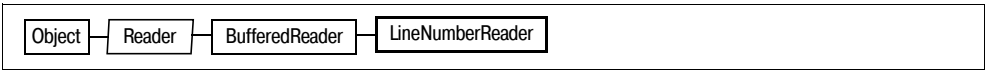
```
// Открытые методы, замещающие методы класса FilterInputStream
   public int available() throws IOException;
   public void mark(int readlimit);
   public int read() throws IOException;
   public int read(byte[] b, int off, int len) throws IOException;
   public void reset() throws IOException;
   public long skip(long n) throws IOException;
}
```

LineNumberReader

Java 1.1

java.io

Этот класс является входным потоком символов. Он отслеживает количество строк текста, прочитанных из потока. Поддерживаются обычные методы класса `Reader` и метод `readLine()`, введенный родительским классом. В дополнение к этим методам можно вызывать `getLineNumber()`, чтобы запросить количество строк, установленное на текущий момент. Кроме того, можно вызвать `setLineNumber()` для установки номера текущей строки. Последующие строки нумеруются последовательно, начиная с указанной отправной точки. Этот класс является аналогом, выраженным потоком символов, класса `LineNumberInputStream`, который был признан устаревшим в Java 1.1.



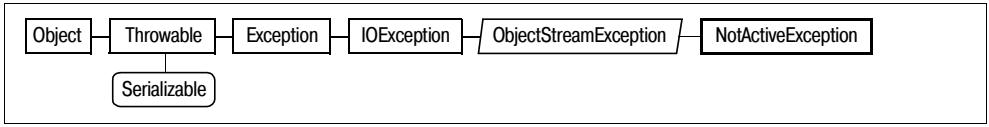

```
public class LineNumberReader extends BufferedReader {
// Открытые конструкторы
   public LineNumberReader(Reader in);
   public LineNumberReader(Reader in, int sz);
// Открытые методы экземпляра
   public int getLineNumber();
   public void setLineNumber(int lineNumber);
// Открытые методы, замещающие методы класса BufferedReader
   public void mark(int readAheadLimit) throws IOException;
   public int read() throws IOException;
   public int read(char[] cbuf, int off, int len) throws IOException;
   public String readLine() throws IOException;
   public void reset() throws IOException;
   public long skip(long n) throws IOException;
}
```

NotActiveException

Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение генерируется в нескольких случаях. Оно означает, что вызванный метод был вызван в неподходящее время или в неправильном контексте. Как правило, `ObjectOutputStream` или `ObjectInputStream` в настоящий момент неактивен, поэтому запрошенная операция не может быть выполнена.




```
public class NotActiveException extends ObjectStreamException {
// Открытые конструкторы
   public NotActiveException();
   public NotActiveException(String reason);
}
```

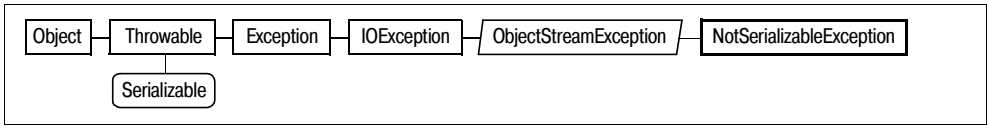
Генерируется методами: `ObjectInputStream.registerValidation()`

NotSerializableException

Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение свидетельствует о том, что объект не может быть сериализован. Оно генерируется в том случае, если происходит попытка сериализации экземпляра класса, который не реализует интерфейс `Serializable`. Имейте в виду, что оно выдается, если предпринимается попытка сериализовать объект `Serializable`, который ссылается на (или содержит) объект, не являющийся `Serializable`. Подкласс класса, являющегося `Serializable`, может предотвратить свою сериализацию путем генерации этого исключения из своих методов `writeObject()` и/или `readObject()`.




```
public class NotSerializableException extends ObjectStreamException {
// Открытые конструкторы
   public NotSerializableException();
   public NotSerializableException(String classname);
}
```

ObjectInput

Java 1.1

java.io

Этот интерфейс расширяет интерфейс `DataInput` и добавляет методы для десериализации объектов и чтения байтов и массивов байтов.


```
public interface ObjectInput extends DataInput {
// Открытые методы экземпляра
   public abstract int available() throws IOException;
   public abstract void close() throws IOException;
   public abstract int read() throws IOException;
   public abstract int read(byte[] b) throws IOException;
   public abstract int read(byte[] b, int off, int len) throws IOException;
   public abstract Object readObject() throws ClassNotFoundException,IOException;
```

```
   public abstract long skip(long n) throws IOException;
}
```

Реализации: ObjectInputStream

Передается методам: java.awt.datatransfer.DataFlavor.readExternal(), Externalizable.readExternal(), java.rmi.server.ObjID.read()

Возвращается методами: java.rmi.server.RemoteCall.getInputStream()

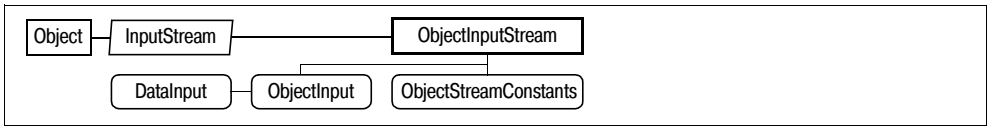
ObjectInputStream Java 1.1

java.io

ObjectInputStream десериализует объекты, массивы и другие значения из потока, который был ранее создан при помощи ObjectOutputStream. Метод readObject() десериализует объекты и массивы, которые затем должны быть приведены к подходящему типу; другие методы считывают значения примитивных данных из потока. Имейте в виду, что сериализовать или десериализовать можно только объекты, которые реализуют интерфейс Serializable или Externalizable.

Класс может реализовывать собственный закрытый метод readObject(ObjectInputStream), чтобы настроить способ своей десериализации. Если вы определяете такой метод, то в вашем распоряжении появляется несколько методов класса ObjectInputStream, которые вы можете использовать, чтобы сделать десериализацию объекта более легкой. defaultReadObject() является самым простым. Он считывает содержимое объекта точно так же, как это сделал бы ObjectInputStream. Если вы записали дополнительные данные до или после содержимого объекта по умолчанию, то вам следует прочитать эти данные до или после вызова defaultReadObject(). При работе с различными версиями или реализациями класса вам придется десериализовать набор полей, которые не совпадают с полями вашего класса. В этом случае создайте в своем классе статическое поле с именем serialPersistentFields. Его значением будет массив объектов ObjectStreamField, которые описывают поля, подлежащие десериализации. В этом случае ваш метод readObject() сможет вызвать readFields() для прочтения указанных полей из потока и возвращения их в объекте ObjectInputStream.GetField. Более подробная информация представлена в описании ObjectStreamField и ObjectInputStream.GetField. Наконец, вы можете вызвать registerValidation() из вашего метода readObject(). Этот метод регистрирует объект ObjectInputValidation (как правило, это объект, который десериализуется). Таким образом происходит уведомление о завершении десериализации целого дерева объектов, которое будет получено непосредственно перед завершением вызова метода readObject() класса ObjectInputStream.

Оставшиеся методы представляют собой разнообразные методы работы с потоками и несколько защищенных методов, предназначенных для использования подклассами, которые хотят настроить десериализацию ObjectInputStream.




```
public class ObjectInputStream extends java.io.InputStream
                         implements ObjectInput, ObjectStreamConstants {
// Открытые конструкторы
   public ObjectInputStream(java.io.InputStream in) throws IOException;
```

```
// Защищенные конструкторы
1.2 protected ObjectInputStream() throws IOException, SecurityException;
// Внутренние классы
1.2 public abstract static class GetField;
// Открытые методы экземпляра
   public void defaultReadObject() throws IOException,ClassNotFoundException;
1.2 public ObjectInputStream.GetField readFields() throws IOException,ClassNotFoundException;
1.4 public Object readUnshared() throws IOException,ClassNotFoundException;
   public void registerValidation(ObjectInputValidation obj, int prio)
        throws NotActiveException, InvalidObjectException;
// Методы, реализующие DataInput
   public boolean readBoolean() throws IOException;
   public byte readByte() throws IOException;
   public char readChar() throws IOException;
   public double readDouble() throws IOException;
   public float readFloat() throws IOException;
   public void readFully(byte[] buf) throws IOException;
   public void readFully(byte[] buf, int off, int len) throws IOException;
   public int readInt() throws IOException;
   public long readLong() throws IOException;
   public short readShort() throws IOException;
   public int readUnsignedByte() throws IOException;
   public int readUnsignedShort() throws IOException;
   public String readUTF() throws IOException;
   public int skipBytes(int len) throws IOException;
// Методы, реализующие ObjectInput
   public int available() throws IOException;
   public void close() throws IOException;
   public int read() throws IOException;
   public int read(byte[] buf, int off, int len) throws IOException;
   public final Object readObject() throws IOException, ClassNotFoundException;
// Защищенные методы экземпляра
   protected boolean enableResolveObject(boolean enable) throws SecurityException;
1.3 protected ObjectStreamClass readClassDescriptor() throws IOException, ClassNotFoundException;
1.2 protected Object readObjectOverride() throws IOException,
      ClassNotFoundException;  // константа
   protected void readStreamHeader() throws IOException,StreamCorruptedException;
   protected Class resolveClass(ObjectStreamClass desc) throws IOException,
      ClassNotFoundException;
   protected Object resolveObject(Object obj) throws IOException;
1.3 protected Class resolveProxyClass(String[] interfaces) throws IOException,
      ClassNotFoundException;
// Устаревшие открытые методы
# public String readLine() throws IOException;  // Реализует:DataInput
}
```

Передается методам:

java.beans.beancontext.BeanContextServicesSupport.bcsPreDeserializationHook(), java.beans.beancontext.BeanContextSupport.{bcsPreDeserializationHook(), deserialize(), readChildren()}, javax.rmi.CORBA.StubDelegate.readObject(), javax.swing.text.StyleContext.{readAttributes(), readAttributeSet()}

ObjectInputStream.GetField Java 1.2

java.io

Этот класс содержит значения именованных полей, прочитанных ObjectInputStream. Он дает программисту возможность жестко контролировать процесс десериализации и обычно используется при реализации объекта с набором полей, которые не соответствуют набору полей (и формату потока сериализации) оригинальной реализации объекта. Этот класс позволяет реализации класса изменяться без нарушения совместимости сериализации.

Для того чтобы использовать класс GetField, ваш класс должен реализовывать закрытый метод readObject(), который отвечает за специализированную десериализацию. Как правило, при использовании класса GetField указывается массив объектов ObjectStreamField как значение закрытого статического поля с именем serialPersistentFields. Этот массив указывает имена и типы всех полей, которые должны быть обнаружены при прочтении из потока сериализации. Если поле serialPersistentFields отсутствует, то массив объектов ObjectStreamField создается из фактических полей класса, за исключением полей static и transient.

В пределах метода readObject() вашего класса вызовите метод readFields(), принадлежащий ObjectInputStream(). Этот метод считывает значения всех полей из потока и сохраняет их в возвращаемом объекте ObjectInputStream.GetField. Объект GetField, по существу, является проекцией имен полей на их значения. Вы можете извлечь значения любых нужных вам полей, чтобы восстановить должное состояние объекта, который подвергается десериализации. Различные методы get() возвращают значения именованных полей указанных типов. Каждый метод принимает значение по умолчанию в качестве аргумента на тот случай, если в потоке сериализации не присутствовало какого-либо значения указанного поля. Например, это может произойти при десериализации объекта, записанного более ранней версией класса. Используйте метод defaulted() для определения того, содержит ли объект GetField значение для именованного поля. Если этот метод возвращает true, то именованное поле не имело значения в потоке, поэтому метод get() объекта GetField должен вернуть указанное значение по умолчанию. Метод getObjectStreamClass() объекта GetField возвращает объект ObjectStreamClass для объекта, который проходит десериализацию. Этот ObjectStreamClass может получить массив объектов ObjectStreamField для класса. См. также ObjectOutputStream.PutField.

```
public abstract static class ObjectInputStream.GetField {
// Открытые конструкторы
   public GetField();
// Открытые методы экземпляра
   public abstract boolean defaulted(String name) throws IOException;
   public abstract boolean get(String name, boolean val) throws IOException;
   public abstract byte get(String name, byte val) throws IOException;
   public abstract char get(String name, char val) throws IOException;
   public abstract short get(String name, short val) throws IOException;
   public abstract int get(String name, int val) throws IOException;
   public abstract long get(String name, long val) throws IOException;
   public abstract float get(String name, float val) throws IOException;
   public abstract double get(String name, double val) throws IOException;
   public abstract Object get(String name, Object val) throws IOException;
   public abstract ObjectStreamClass getObjectStreamClass();
}
```

Возвращается методами: ObjectInputStream.readFields()

ObjectInputValidation Java 1.1

java.io

Класс реализует этот интерфейс и определяет метод `validateObject()`, чтобы подтвердить правильность самого себя, когда он и все объекты, от которых он зависит, будут полностью десериализованы из `ObjectInputStream`. Однако метод `validateObject()` вызывается, если объект передается методу `ObjectInputStream.registerValidation()`; такая операция выполняется из метода `readObject()` объекта. Обратите внимание: если объект десериализуется как часть более крупного графа объекта, то его метод `validateObject()` не вызывается до тех пор, пока не будет прочитан весь граф; первоначальный вызов метода `ObjectInputStream.readObject()` будет готов выполнить возврат. `validateObject()` должен генерировать `InvalidObjectException`, если объект не проходит подтверждение. Это останавливает сериализацию объекта, а первоначальный вызов, направленный к `ObjectInputStream.readObject()`, завершается исключением `InvalidObjectException`.

```
public interface ObjectInputValidation {
// Открытые методы экземпляра
   public abstract void validateObject() throws InvalidObjectException;
}
```

Передается методам: `ObjectInputStream.registerValidation()`

ObjectOutput Java 1.1

java.io

Этот интерфейс расширяет интерфейс `DataOutput` и добавляет методы для сериализации объектов и записи байтов и массивов байтов.


```
public interface ObjectOutput extends DataOutput {
// Открытые методы экземпляра
   public abstract void close() throws IOException;
   public abstract void flush() throws IOException;
   public abstract void write(byte[] b) throws IOException;
   public abstract void write(int b) throws IOException;
   public abstract void write(byte[] b, int off, int len) throws IOException;
   public abstract void writeObject(Object obj) throws IOException;
}
```

Реализации: `ObjectOutputStream`

Передается методам: `java.awt.datatransfer.DataFlavor.writeExternal()`, `Externalizable.writeExternal()`, `ObjectOutputStream.PutField.write()`, `java.rmi.server.ObjID.write()`, `java.rmi.server.RemoteRef.getRefClass()`

Возвращается методами: `java.rmi.server.RemoteCall.{getOutputStream(), getResultStream()}`

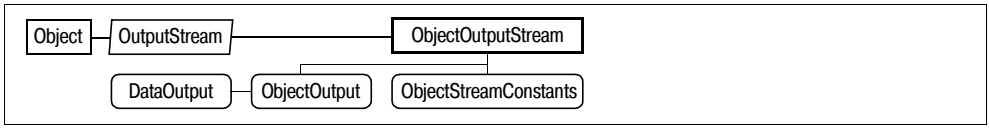
ObjectOutputStream Java 1.1

java.io

ObjectOutputStream сериализует объекты, массивы и другие значения в поток. Метод writeObject() сериализует объект или массив, а другие методы записывают в поток значения примитивных данных. Имейте в виду, что могут быть сериализованы только объекты, которые реализуют интерфейс Serializable или Externalizable.

Класс, который желает настроить сериализацию своих экземпляров, должен объявить закрытый метод writeObject(ObjectOutputStream). Этот метод вызывается при сериализации объекта и может использовать несколько дополнительных методов ObjectOutputStream. defaultWriteObject() выполняет ту же сериализацию, которая бы произошла, если бы не существовал метод writeObject(). Объект может вызвать этот метод, чтобы сериализовать самого себя, а затем вызвать другие методы ObjectOutputStream для записи дополнительных данных в поток сериализации. Конечно, класс должен определить подходящий метод readObject() для считывания дополнительных данных. При работе с различными версиями или реализациями класса вам, возможно, придется сериализовать набор полей, которые неточно совпадают с полями вашего класса. В этом случае придайте своему классу статическое поле с именем serialPersistentFields, чьим значением будет массив объектов ObjectStreamField, которые описывают поля, подлежащие сериализации. В вашем методе writeObject() вызовите putFields() для получения объекта ObjectOutputStream.PutField. Сохраните имена полей и значения в этом объекте, а затем вызовите writeFields() для их записи в поток сериализации. Более подробная информация представлена в описании ObjectStreamField и ObjectOutputStream.PutField.

Оставшиеся методы ObjectOutputStream являются разнообразными методами по работе с потоками и защищенными методами, которые предназначены для использования подклассами, желающими настроить свою сериализацию.




```
public class ObjectOutputStream extends java.io.OutputStream
       implements ObjectOutput, ObjectStreamConstants {
// Открытые конструкторы
   public ObjectOutputStream(java.io.OutputStream out) throws IOException;
// Защищенные конструкторы
1.2 protected ObjectOutputStream() throws IOException, SecurityException;
// Внутренние классы
1.2 public abstract static class PutField;
// Открытые методы экземпляра
   public void defaultWriteObject() throws IOException;
1.2 public ObjectOutputStream.PutField putFields() throws IOException;
   public void reset() throws IOException;
1.2 public void useProtocolVersion(int version) throws IOException;
1.2 public void writeFields() throws IOException;
1.4 public void writeUnshared(Object obj) throws IOException;
// Методы, реализующие DataOutput
   public void writeBoolean(boolean val) throws IOException;
   public void writeByte(int val) throws IOException;
   public void writeBytes(String str) throws IOException;
```

```
    public void writeChar(int val) throws IOException;
    public void writeChars(String str) throws IOException;
    public void writeDouble(double val) throws IOException;
    public void writeFloat(float val) throws IOException;
    public void writeInt(int val) throws IOException;
    public void writeLong(long val) throws IOException;
    public void writeShort(int val) throws IOException;
    public void writeUTF(String str) throws IOException;
// Методы, реализующие ObjectOutput
    public void close() throws IOException;
    public void flush() throws IOException;
    public void write(int val) throws IOException;
    public void write(byte[] buf) throws IOException;
    public void write(byte[] buf, int off, int len) throws IOException;
    public final void writeObject(Object obj) throws IOException;
// Защищенные методы экземпляра
    protected void annotateClass(Class cl) throws IOException;                    // пустой
1.3 protected void annotateProxyClass(Class cl) throws IOException;               // пустой
    protected void drain() throws IOException;
    protected boolean enableReplaceObject(boolean enable) throws SecurityException;
    protected Object replaceObject(Object obj) throws IOException;
1.3 protected void writeClassDescriptor(ObjectStreamClass desc) throws IOException;
1.2 protected void writeObjectOverride(Object obj) throws IOException;            // пустой
    protected void writeStreamHeader() throws IOException;
}
```

Передается методам: java.awt.AWTEventMulticaster.{save(), saveInternal()}, java.beans.beancontext.BeanContextServicesSupport.bcsPreSerializationHook(), java.beans.beancontext.BeanContextSupport.{bcsPreSerializationHook(), serialize(), writeChildren()}, javax.rmi.CORBA.StubDelegate.writeObject(), javax.swing.text.StyleContext.{writeAttributes(), writeAttributeSet()}

ObjectOutputStream.PutField Java 1.2

java.io

Этот класс содержит значения именованных полей и обеспечивает возможность их записи в ObjectOutputStream во время процесса сериализации объекта. Это дает программисту возможность четко контролировать процесс сериализации. Обычно bjectOutputStream.PutField используется, если набор полей, определенных классом, не соответствует набору полей (и формату потока сериализации), который определен оригинальной реализацией класса. Другими словами, ObjectOutputStream.PutField позволяет реализации класса изменяться без нарушения совместимости сериализации.

Для использования класса PutField обычно определяют закрытое статическое поле serialPersistentFields, которое ссылается на массив объектов ObjectStreamField. Этот массив определяет набор полей, записываемых в ObjectOutputStream, и таким образом определяет формат сериализации. Если вы не объявите поле serialPersistentFields, то набором полей будут все поля класса, за исключением полей static и transient.

В дополнение к полю serialPersistentFields ваш класс должен определить закрытый метод writeObject(), который отвечает за специализированную сериализацию вашего класса. В этом методе вызовите метод putFields(), принадлежащий ObjectOutputStream, для получения объекта ObjectOutputStream.PutField. Как только вы получите этот объект, используйте различные методы put() для указания имен и значений поля, которые следует записать. Набор именованных полей должен совпадать с полями,

которые указаны в serialPersistentFields. Вы можете указывать поля в любом порядке; класс PutField отвечает за то, чтобы они были записаны в правильном порядке. Как только вы указали значения всех полей, вызовите метод write() вашего объекта PutField, чтобы записать значения полей в поток сериализации.

Изменение специализированного процесса сериализации на противоположный представлено в описании ObjectInputStream.GetField.

```
public abstract static class ObjectOutputStream.PutField {
// Открытые конструкторы
   public PutField();
// Открытые методы экземпляра
   public abstract void put(String name, long val);
   public abstract void put(String name, int val);
   public abstract void put(String name, float val);
   public abstract void put(String name, Object val);
   public abstract void put(String name, double val);
   public abstract void put(String name, byte val);
   public abstract void put(String name, boolean val);
   public abstract void put(String name, short val);
   public abstract void put(String name, char val);
// Устаревшие открытые методы
# public abstract void write(ObjectOutput out) throws IOException;
}
```

Возвращается методами: ObjectOutputStream.putFields()

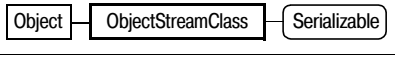
ObjectStreamClass

Java 1.1

java.io ***сериализуемый***

Этот класс представляет собой класс, который проходит сериализацию. Объект ObjectStreamClass содержит имя класса, уникальный идентификатор версии, имя и тип полей, которые составляют формат сериализации для класса. getSerialVersionUID() возвращает уникальный идентификатор версии класса. Он возвращает значение закрытого поля serialVersionUID класса или вычисленное значение, основанное на открытом API класса. В Java 1.2 и последующих версиях getFields() возвращает массив объектов ObjectStreamField, которые представляют имена и типы полей класса, подлежащие сериализации. getField() возвращает один объект ObjectStreamField, который представляет одно именованное поле. По умолчанию эти методы используют все поля класса за исключением тех, которые являются static или transient. И все же этот набор полей по умолчанию может быть изменен путем объявления в классе закрытого поля serialPersistentFields. Значением этого поля должен быть желаемый массив объектов ObjectStreamField.

Класс ObjectStreamClass не имеет конструктора; вам следует использовать статический метод lookup() для получения объекта ObjectStreamClass, предназначенного для данного объекта Class. Метод экземпляра forClass() выполняет противоположную операцию; он возвращает объект Class, который соответствует данному ObjectStreamClass. У большинства приложений никогда не возникает необходимости использовать этот класс.



```
public class ObjectStreamClass implements Serializable {
// Конструкторов нет
// Открытые константы
1.2 public static final ObjectStreamField[] NO_FIELDS;
// Открытые методы класса
   public static ObjectStreamClass lookup(Class cl);
// Открытые методы экземпляра
   public Class forClass();
1.2 public ObjectStreamField getField(String name);
1.2 public ObjectStreamField[] getFields();
   public String getName();
   public long getSerialVersionUID();
// Открытые методы, замещающие методы класса Object
   public String toString();
}
```

Передается методам:
`ObjectInputStream.resolveClass()`, `ObjectOutputStream.writeClassDescriptor()`

Возвращается методами: `ObjectInputStream.readClassDescriptor()`, `ObjectInputStream.GetField.getObjectStreamClass()`, `ObjectStreamClass.lookup()`

ObjectStreamConstants

Java 1.2

java.io

Этот интерфейс определяет различные константы, которые используются механизмом сериализации объектов Java. Двумя важными константами являются `PROTOCOL_VERSION_1` и `PROTOCOL_VERSION_2`, которые указывают, какую версию протокола сериализации следует использовать. В Java 1.2 вы можете передать любое из этих значений методу `useProtocolVersion()`, принадлежащему классу `ObjectOutputStream`. По умолчанию при сериализации объектов Java 1.2 использует версию 2 протокола, а Java 1.1 – версию 1. Подобно Java 1.1.7, Java 1.2 может десериализовать объекты, написанные с использованием любой версии протокола. Если вы хотите сериализовать объект таким образом, чтобы он мог быть прочитан версиями Java, предшествующими 1.1.7, используйте `PROTOCOL_VERSION_1`.

Другие константы, определяемые этим интерфейсом, являются низкоуровневыми значениями, которые используются протоколом сериализации. Вам нет необходимости использовать их, если только вы сами не собираетесь заново реализовать механизм сериализации.

```
public interface ObjectStreamConstants {
// Открытые константы
   public static final int baseWireHandle;                                    // =8257536
   public static final int PROTOCOL_VERSION_1;                                // =1
   public static final int PROTOCOL_VERSION_2;                                // =2
   public static final byte SC_BLOCK_DATA;                                    // =8
   public static final byte SC_EXTERNALIZABLE;                                // =4
   public static final byte SC_SERIALIZABLE;                                  // =2
   public static final byte SC_WRITE_METHOD;                                  // =1
   public static final short STREAM_MAGIC;                                    // =-21267
   public static final short STREAM_VERSION;                                  // =5
   public static final SerializablePermission SUBCLASS_IMPLEMENTATION_PERMISSION;
   public static final SerializablePermission SUBSTITUTION_PERMISSION;
   public static final byte TC_ARRAY;                                         // =117
   public static final byte TC_BASE;                                         // =112
```

```
   public static final byte TC_BLOCKDATA;                    // =119
   public static final byte TC_BLOCKDATALONG;                // =122
   public static final byte TC_CLASS;                        // =118
   public static final byte TC_CLASSDESC;                    // =114
   public static final byte TC_ENDBLOCKDATA;                 // =120
   public static final byte TC_EXCEPTION;                    // =123
1.3 public static final byte TC_LONGSTRING;                  // =124
   public static final byte TC_MAX;                          // =125
   public static final byte TC_NULL;                         // =112
   public static final byte TC_OBJECT;                       // =115
1.3 public static final byte TC_PROXYCLASSDESC;              // =125
   public static final byte TC_REFERENCE;                    // =113
   public static final byte TC_RESET;                        // =121
   public static final byte TC_STRING;                       // =116
}
```

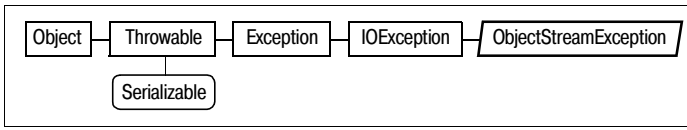
Реализации: ObjectInputStream, ObjectOutputStream

ObjectStreamException

Java 1.1

java.io ***сериализуемое, проверяемое***

Этот класс является родительским классом более специфичных типов исключений, которые могут возникнуть в процессе сериализации и десериализации объектов в классах ObjectOutputStream и ObjectInputStream.




```
public abstract class ObjectStreamException extends IOException {
// Защищенные конструкторы
   protected ObjectStreamException();
   protected ObjectStreamException(String classname);
}
```

Подклассы: InvalidClassException, InvalidObjectException, NotActiveException, NotSerializableException, OptionalDataException, StreamCorruptedException, WriteAbortedException

Генерируется методами: java.awt.AWTKeyStroke.readResolve(), java.awt.color.ICC_Profile.readResolve(), java.security.cert.Certificate.writeReplace(), java.security.cert.Certificate.CertificateRep.readResolve(), java.security.cert.CertPath.writeReplace(), java.security.cert.CertPath.CertPathRep.readResolve(), javax.print.attribute.EnumSyntax.readResolve()

ObjectStreamField

Java 1.2

java.io ***сравнимый***

Этот класс представляет именованное поле указанного типа (то есть указанного Class). Когда класс сериализует себя путем записи набора полей, которые отличаются от полей, используемых им в своей собственной реализации, то он определяет набор полей, которые будут записываться в массив объектов ObjectStreamField. Этот массив должен быть значением закрытого статического поля с именем serialPersis-

tentFields. Обычно методы этого класса используются механизмом сериализации. См. также ObjectOutputStream.PutField и ObjectInputStream.GetField.


```
public class ObjectStreamField implements Comparable {
// Открытые конструкторы
   public ObjectStreamField(String name, Class type);
1.4 public ObjectStreamField(String name, Class type, boolean unshared);
// Методы доступа к свойствам (по имени свойства)
   public String getName();
   public int getOffset();
   public boolean isPrimitive();
   public Class getType();
   public char getTypeCode();
   public String getTypeString();
1.4 public boolean isUnshared();
// Методы, реализующие Comparable
   public int compareTo(Object obj);
// Открытые методы, замещающие методы класса Object
   public String toString();
// Защищенные методы экземпляра
   protected void setOffset(int offset);
}
```

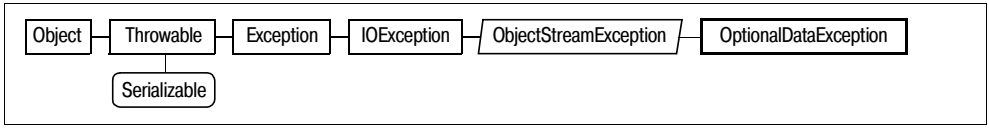
Возвращается методами: ObjectStreamClass.{getField(), getFields()}

Экземпляры: ObjectStreamClass.NO_FIELDS

OptionalDataException Java 1.1

java.io ***сериализуемое, проверяемое***

Генерируется методом readObject(), принадлежащим ObjectInputStream, когда он сталкивается с данными примитивного типа в том месте, где ожидает данные объекта. Несмотря на название исключения, эти данные не являются необязательными, а десериализация объекта прекращается.




```
public class OptionalDataException extends ObjectStreamException {
// Конструктор отсутствует
// Открытые поля экземпляра
   public boolean eof;
   public int length;
}
```

OutputStream Java 1.0

java.io

Этот абстрактный класс является родительским классом всех выходных потоков. Он определяет основные методы вывода, предоставляемые всеми классами выходных потоков. `write()` записывает один байт или массив (или подмассив) байтов. `flush()` выводит запись любого буферизованного вывода. `close()` закрывает поток и освобождает все системные ресурсы, которые с ним связаны. Поток не должен быть использован после вызова `close()`. См. также `Writer`.

```
public abstract class OutputStream {
// Открытые конструкторы
   public OutputStream();
// Открытые методы экземпляра
   public void close() throws IOException;                                    // пустой
   public void flush() throws IOException;                                    // пустой
   public abstract void write(int b) throws IOException;
   public void write(byte[] b) throws IOException;
   public void write(byte[] b, int off, int len) throws IOException;
}
```

Подклассы: `ByteArrayOutputStream`, `FileOutputStream`, `FilterOutputStream`, `ObjectOutputStream`, `PipedOutputStream`, `org.omg.CORBA.portable.OutputStream`

***Передается методам*:** Методов слишком много, чтобы их перечислить.

Возвращается методами: `Process.getOutputStream()`, `Runtime.getLocalizedOutputStream()`, `java.net.Socket.getOutputStream()`, `java.net.SocketImpl.getOutputStream()`, `java.net.URLConnection.getOutputStream()`, `java.nio.channels.Channels.newOutputStream()`, `java.rmi.server.LogStream.getOutputStream()`, `java.sql.Blob.setBinaryStream()`, `java.sql.Clob.setAsciiStream()`, `javax.print.StreamPrintService.getOutputStream()`, `javax.xml.transform.stream.StreamResult.getOutputStream()`

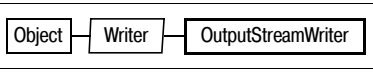
Экземпляры: `FilterOutputStream.out`

OutputStreamWriter Java 1.1

java.io

Этот класс является выходным потоком символов, который использует выходной поток байтов в качестве пункта назначения для своих данных. Когда символы записываются в `OutputStreamWriter`, он переводит их в байты в соответствии с конкретной кодировкой символов, специфичной для данного региона и/или платформы, а затем записывает эти байты в указанный `OutputStream`. Это важная функциональная особенность, связанная с локализацией, представлена в Java 1.1 и последующих версиях. `OutputStreamWriter` поддерживает обычные методы `Writer`. Он также имеет метод `getEncoding()`, который возвращает имя кодировки, используемой для перевода символов в байты.

При создании `OutputStreamWriter` вы указываете `OutputStream`, в который он записывает байты. Кроме того, вы можете указать название кодировки символов, которая должна использоваться для перевода символов в байты. Если вы не указываете имя кодировки, то `OutputStreamWriter` использует кодировку по умолчанию, существующую для предопределенного региона. В Java 1.4 и последующих версиях этот класс использует средства перевода набора символов из пакета `java.nio.charset` и позволяет явно указать необходимый `Charset` или `CharsetEncoder`. В версиях, предшествующих 1.4, класс позволяет указывать имя кодировки набора символов.



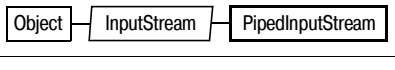

```
public class OutputStreamWriter extends Writer {
// Открытые конструкторы
   public OutputStreamWriter(java.io.OutputStream out);
   public OutputStreamWriter(java.io.OutputStream out, String charsetName)
        throws UnsupportedEncodingException;
1.4 public OutputStreamWriter(java.io.OutputStream out, java.nio.charset.CharsetEncoder enc);
1.4 public OutputStreamWriter(java.io.OutputStream out, java.nio.charset.Charset cs);
// Открытые методы экземпляра
   public String getEncoding();
// Открытые методы, замещающие методы класса Writer
   public void close() throws IOException;
   public void flush() throws IOException;
   public void write(int c) throws IOException;
   public void write(char[] cbuf, int off, int len) throws IOException;
   public void write(String str, int off, int len) throws IOException;
}
```

Подклассы: FileWriter

PipedInputStream Java 1.0

java.io

Этот класс – InputStream, который реализует одну половину канала и полезен при передаче данных между потоками выполнения (threads). PipedInputStream должен быть присоединен к объекту PipedOutputStream, который можно указать при создании PipedInputStream или при помощи метода connect(). Данные, прочитанные из объекта PipedInputStream, получены из PipedOutputStream, к которому он присоединен. В описании InputStream рассказано о низкоуровневых методах, предназначенных для прочтения данных из PipedInputStream. FilterInputStream может обеспечить интерфейс более высокого уровня для прочтения данных из PipedInputStream.




```
public class PipedInputStream extends java.io.InputStream {
// Открытые конструкторы
   public PipedInputStream();
   public PipedInputStream(PipedOutputStream src) throws IOException;
// Защищенные константы
1.1 protected static final int PIPE_SIZE;                                    // =1024
// Открытые методы экземпляра
   public void connect(PipedOutputStream src) throws IOException;
// Открытые методы, замещающие методы класса InputStream
   public int available() throws IOException;                              // синхронизирован
   public void close() throws IOException;
   public int read() throws IOException;                                   // синхронизирован
   public int read(byte[] b, int off, int len) throws IOException;         // синхронизирован
// Защищенные методы экземпляра
1.1 protected void receive(int b) throws IOException;                      // синхронизирован
// Защищенные поля экземпляра
1.1 protected byte[] buffer;
```

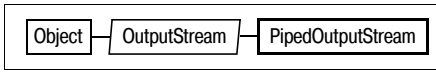
```
1.1 protected int in;
1.1 protected int out;
}
```

Передается методам: PipedOutputStream.{connect(), PipedOutputStream()}

PipedOutputStream Java 1.0

java.io

Этот класс – OutputStream, который реализует одну половину канала и полезен при взаимодействии потоков. PipedOutputStream должен быть присоединен к PipedInputStream, который можно указать при создании PipedOutputStream или при помощи метода connect(). Данные, записанные в PipedOutputStream, доступны для чтения в PipedInputStream. В описании OutputStream рассказано о низкоуровневых методах, предназначенных для записи данных в PipedOutputStream. FilterOutputStream может обеспечить интерфейс более высокого уровня для записи данных в PipedOutputStream.



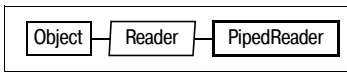

```
public class PipedOutputStream extends java.io.OutputStream {
// Открытые конструкторы
   public PipedOutputStream();
   public PipedOutputStream(PipedInputStream snk) throws IOException;
// Открытые методы экземпляра
   public void connect(PipedInputStream snk) throws IOException;          // синхронизирован
// Открытые методы, замещающие методы класса OutputStream
   public void close() throws IOException;
   public void flush() throws IOException;                                // синхронизирован
   public void write(int b) throws IOException;
   public void write(byte[] b, int off, int len) throws IOException;
}
```

Передается методам: PipedInputStream.{connect(), PipedInputStream()}

PipedReader Java 1.1

java.io

PipedReader – это выходной поток символов, считывающий символы из выходного потока символов PipedWriter, к которому он присоединен. PipedReader реализует одну половину канала. Он полезен при передаче данных между двумя потоками приложения. PipedReader не может быть использован до тех пор, пока он не будет присоединен к объекту PipedWriter, который может быть передан конструктору PipedReader() или методу connect(). PipedReader наследует большую часть методов родительского класса. См. также Reader. PipedReader является символьным аналогом класса PipedInputStream.




```
public class PipedReader extends Reader {
// Открытые конструкторы
   public PipedReader();
```

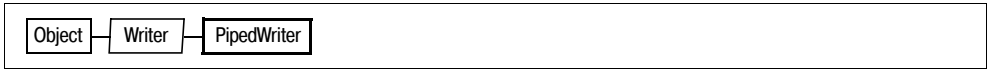
```
   public PipedReader(PipedWriter src) throws IOException;
// Открытые методы экземпляра
   public void connect(PipedWriter src) throws IOException;
// Открытые методы, замещающие методы класса Reader
   public void close() throws IOException;
1.2 public int read() throws IOException;                                    // синхронизирован
   public int read(char[] cbuf, int off, int len) throws IOException;        // синхронизирован
1.2 public boolean ready() throws IOException;                               // синхронизирован
}
```

Передается методам: PipedWriter.{connect(), PipedWriter()}

PipedWriter Java 1.1

java.io

PipedWriter — это выходной поток символов, который записывает символы во входящий поток символов PipedReader, к которому он присоединен. PipedWriter реализует одну половину канала и полезен при передаче данных между двумя потоками приложения. PipedWriter не может быть использован до тех пор, пока он не будет присоединен к объекту PipedReader, который может быть передан конструктору PipedWriter() или методу connect(). PipedWriter наследует большую часть методов родительского класса. См. также Writer. PipedWriter является символьным аналогом класса PipedOutputStream.




```
public class PipedWriter extends Writer {
// Открытые конструкторы
   public PipedWriter();
   public PipedWriter(PipedReader snk) throws IOException;
// Открытые методы экземпляра
   public void connect(PipedReader snk) throws IOException;  // синхронизирован
// Открытые методы, замещающие методы класса Writer
   public void close() throws IOException;
   public void flush() throws IOException;                   // синхронизирован
1.2 public void write(int c) throws IOException;
   public void write(char[] cbuf, int off, int len) throws IOException;
}
```

Передается методам: PipedReader.{connect(), PipedReader()}

PrintStream Java 1.0

java.io

Этот класс является FilterOutputStream, который реализует ряд методов, предназначенных для отображения текстового представления примитивных типов данных Java. Методы print() выводят стандартное текстовое представление каждого типа данных. Методы println() делают то же самое, но добавляют в конец символ новой строки. Каждый метод переводит примитивный тип Java в String и выводит итоговую строку. Когда Object передается print() или println(), он преобразуется в String путем вызова его метода toString(). PrintStream является типом OutputStream, который делает вывод текста наиболее легким. По этой причине он наиболее широко применяется

по сравнению с другими выходными потоками. Переменной System.out является объект класса PrintStream.

Обратите внимание, что в Java 1.0 этот класс обрабатывает символы Unicode неправильно; он не учитывает старшие 8 бит всех 16-битных символов. Таким образом, он работает только с символами Latin-1 (ISO8859-1). Хотя в Java 1.1 эта проблема была решена, PrintStream был вытеснен PrintWriter. Конструкторы этого класса признаны устаревшими, однако сам класс все еще используется стандартными выходными потоками System.out и System.err.

PrintStream и его замена, PrintWriter, выводят текстовые представления типов данных Java. Для вывода двоичных представлений данных используйте DataOutputStream.


```
public class PrintStream extends FilterOutputStream {
// Открытые конструкторы
   public PrintStream(java.io.OutputStream out);
   public PrintStream(java.io.OutputStream out, boolean autoFlush);
1.4 public PrintStream(java.io.OutputStream out, boolean autoFlush, String encoding)
        throws UnsupportedEncodingException;
// Замещенные методы экземпляра
   public boolean checkError();
   public void print(float f);
   public void print(long l);
   public void print(int i);
   public void print(double d);
   public void print(char[] s);
   public void print(String s);
   public void print(Object obj);
   public void print(char c);
   public void print(boolean b);
   public void println();
   public void println(float x);
   public void println(long x);
   public void println(int x);
   public void println(String x);
   public void println(Object x);
   public void println(double x);
   public void println(char[] x);
   public void println(char x);
   public void println(boolean x);
// Открытые методы, замещающие методы класса FilterOutputStream
   public void close();
   public void flush();
   public void write(int b);
   public void write(byte[] buf, int off, int len);
// Защищенные методы экземпляра
1.1 protected void setError();
}
```

Подклассы: java.rmi.server.LogStream

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.rmi.server.LogStream.getDefaultStream(), java.rmi.server.RemoteServer.getLog(), java.sql.DriverManager.getLogStream(), javax.swing.DebugGraphics.logStream()

Экземпляры: System.{err, out}

PrintWriter Java 1.1

java.io

Этот класс является выходным потоком символов, который реализует ряд методов print() и println(), выводящих текстовые представления примитивных значений и объектов.

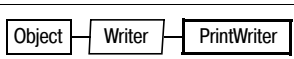
Когда вы создаете объект PrintWriter, вы указываете выходной поток символов или байтов, в которые он должен записывать свои символы. Кроме того, вы можете указать, будет ли автоматически сбрасываться поток PrintWriter при вызове println().

Если в качестве пункта назначения вы укажете выходной поток байтов, конструктор PrintWriter() автоматически создаст необходимый объект OutputStreamWriter для перевода символов в байты с использованием кодировки по умолчанию.

PrintWriter реализует обычные методы write(), flush() и close(), которые определяются всеми подклассами Writer. Чаще применяются методы print() и println(). Это методы более высокого уровня, каждый из которых переводит свой аргумент в строку перед тем, как его выводить. Кроме того, println() может ограничивать строку и сбрасывать буфер после распечатки своего аргумента.

Методы PrintWriter никогда не генерируют исключений. Вместо этого при возникновении ошибки они устанавливают внутренний флаг. Этот флаг можно проверить, вызвав checkError(). checkError() сначала сбрасывает внутренний поток, а затем возвращает true, если произошло какое-либо исключение во время записи значений в этот поток. Как только произошла ошибка в объекте PrintWriter, все последующие запросы к checkError() будут возвращать true; нет никакой возможности сбросить флаг ошибки.

PrintWriter является символьным аналогом класса PrintStream, который он замещает. Обычно вы можете тривиально заменить любые объекты PrintStream в программе на объекты PrintWriter. Это особенно важно для многоязыковых программ. Единственный приемлемый вариант использования класса PrintStream – это применение стандартных выходных потоков System.out и System.err. См. PrintStream на предмет более детальной информации.




```
public class PrintWriter extends Writer {
// Открытые конструкторы
   public PrintWriter(java.io.OutputStream out);
   public PrintWriter(Writer out);
   public PrintWriter(java.io.OutputStream out, boolean autoFlush);
   public PrintWriter(Writer out, boolean autoFlush);
// Открытые методы экземпляра
   public boolean checkError();
   public void print(int i);
   public void print(long l);
   public void print(char c);
   public void print(boolean b);
   public void print(float f);
   public void print(double d);
   public void print(char[] s);
   public void print(Object obj);
   public void print(String s);
```

```
   public void println();
   public void println(int x);
   public void println(long x);
   public void println(boolean x);
   public void println(char x);
   public void println(String x);
   public void println(Object x);
   public void println(char[] x);
   public void println(float x);
   public void println(double x);
// Открытые методы, замещающие методы класса Writer
   public void close();
   public void flush();
   public void write(char[] buf);
   public void write(String s);
   public void write(int c);
   public void write(String s, int off, int len);
   public void write(char[] buf, int off, int len);
// Защищенные методы экземпляра
   protected void setError();
// Защищенные поля экземпляра
1.2 protected Writer out;
}
```

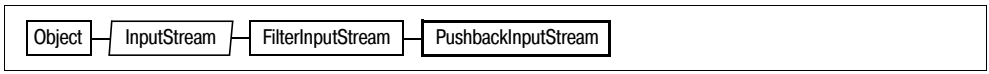
Передается методам: java.awt.Component.list(), java.awt.Container.list(), Throwable.printStackTrace(), java.security.cert.CertPathBuilderException.printStackTrace(), java.security.cert.CertPathValidatorException.printStackTrace(), java.security.cert.CertStoreException.printStackTrace(), java.sql.DriverManager.setLogWriter(), java.util.Properties.list(), javax.naming.NamingException.printStackTrace(), javax.sql.ConnectionPoolDataSource.setLogWriter(), javax.sql.DataSource.setLogWriter(), javax.sql.XADataSource.setLogWriter(), javax.xml.transform.TransformerException.printStackTrace()

Возвращается методами: java.sql.DriverManager.getLogWriter(), javax.sql.ConnectionPoolDataSource.getLogWriter(), javax.sql.DataSource.getLogWriter(), javax.sql.XADataSource.getLogWriter()

PushbackInputStream Java 1.0

java.io

Этот класс – FilterInputStream, который реализует однобайтный буфер возврата в поток или (начиная с Java 1.1) буфер возврата в поток указанной длины. Методы unread() возвращают байты обратно в поток; эти байты являются первыми байтами, которые прочитываются последующим вызовом метода read(). Этот класс может быть полезен при написании парсеров (программ синтаксического анализа, parsers). См. также PushbackReader.



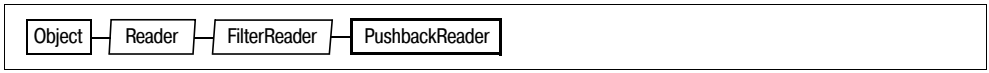

```
public class PushbackInputStream extends FilterInputStream {
// Открытые конструкторы
   public PushbackInputStream(java.io.InputStream in);
1.1 public PushbackInputStream(java.io.InputStream in, int size);
```

```
// Открытые методы экземпляра
   public void unread(int b) throws IOException;
1.1 public void unread(byte[] b) throws IOException;
1.1 public void unread(byte[] b, int off, int len) throws IOException;
// Открытые методы, замещающие методы класса FilterInputStream
   public int available() throws IOException;
1.2 public void close() throws IOException;                          // синхронизирован
   public boolean markSupported();                                   // константа
   public int read() throws IOException;
   public int read(byte[] b, int off, int len) throws IOException;
1.2 public long skip(long n) throws IOException;
// Защищенные поля экземпляра
1.1 protected byte[] buf;
1.1 protected int pos;
}
```

PushbackReader Java 1.1

java.io

Этот класс является входным потоком символов, который использует другой входной поток в качестве источника входных данных и добавляет возможность возвращать символы обратно в поток. Эта функциональная особенность часто оказывается полезной при написании парсеров. При создании потока PushbackReader вы указываете поток, из которого будет производиться считывание. Кроме того, можно указать размер буфера возврата в поток, то есть количество символов, которое может быть возвращено обратно в поток (unread). Если вы не укажете размер этого буфера, то размер по умолчанию будет равен одному символу. PushbackReader наследует или замещает все стандартные методы Reader и добавляет три метода unread(), которые возвращают обратно в поток один символ, массив символов или часть массива символов. Этот класс является символьным аналогом класса PushbackInputStream.




```
public class PushbackReader extends FilterReader {
// Открытые конструкторы
   public PushbackReader(Reader in);
   public PushbackReader(Reader in, int size);
// Открытые методы экземпляра
   public void unread(int c) throws IOException;
   public void unread(char[] cbuf) throws IOException;
   public void unread(char[] cbuf, int off, int len) throws IOException;
// Открытые методы, замещающие методы класса FilterReader
   public void close() throws IOException;
1.2 public void mark(int readAheadLimit) throws IOException;
   public boolean markSupported();                                   // константа
   public int read() throws IOException;
   public int read(char[] cbuf, int off, int len) throws IOException;
   public boolean ready() throws IOException;
1.2 public void reset() throws IOException;
}
```

RandomAccessFile Java 1.0

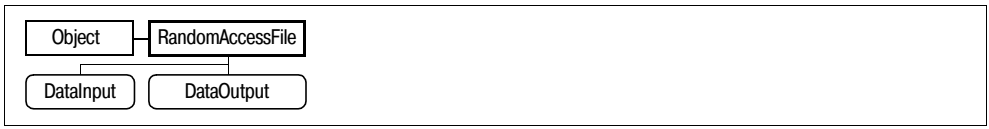
java.io

Этот класс позволяет записывать и считывать произвольные байты, текст и примитивные типы данных Java из любого или в любое место файла. Так как этот класс предоставляет произвольный, а не последовательный доступ к файлам, то он не является ни подклассом `InputStream`, ни подклассом `OutputStream`; он предоставляет абсолютно независимый способ чтения и записи данных из файлов и в файлы. `RandomAccessFile` реализует те же интерфейсы, что и `DataInputStream` и `DataOutputStream`. Таким образом, он определяет те же методы для чтения и записи данных, что и вышеупомянутые классы.

Метод `seek()` обеспечивает произвольный доступ к файлу; он используется для выбора того места в файле, в которое будут записаны данные или из которого они будут прочитаны. Различные методы записи и чтения обновляют позицию в файле, чтобы последовательность операций записи или чтения могла быть выполнена на непрерывной части файла без необходимости вызова метода `seek()` перед каждой операцией записи или чтения.

Для файла, который предназначен только для чтения, аргумент `mode` к методам конструктора должен быть «r», а для файла, который предназначен для записи (и, возможно, также для чтения) – «rw». Для аргумента `mode` в Java 1.4 и последующих версиях также разрешены два других значения. Режим «rwd» открывает файл для чтения и записи. Если файл находится в локальной файловой системе, то каждое обновление содержимого файла должно быть синхронно записано в базовый файл. Режим «rws» похож на предыдущий, но требует синхронных обновлений как содержимого файла, так и его «метаданных» (например, времени доступа к файлу). Использование режима «rws» может потребовать того, чтобы метаданные файла модифицировались при каждом чтении файла.

В Java 1.4 и последующих версиях используйте метод `getChannel()` для получения объекта `FileChannel`, который можно задействовать для доступа к файлу с применением нового API ввода/вывода `java.nio` и его подпакетов. Если `RandomAccessFile` был открыт в режиме «r», то `FileChannel` разрешит только чтение. В остальных случаях он разрешит как запись, так и чтение.




```
public class RandomAccessFile implements DataInput,
      DataOutput {
// Открытые конструкторы
   public RandomAccessFile(File file, String mode) throws FileNotFoundException;
   public RandomAccessFile(String name, String mode) throws FileNotFoundException;
// Открытые методы экземпляра
   public void close() throws IOException;                          // зависит от платформы
1.4 public final java.nio.channels.FileChannel getChannel();
   public final FileDescriptor getFD() throws IOException;
   public long getFilePointer() throws IOException;                 // зависит от платформы
   public long length() throws IOException;                         // зависит от платформы
   public int read() throws IOException;                            // зависит от платформы
   public int read(byte[] b) throws IOException;
```

```
   public int read(byte[] b, int off, int len) throws IOException;
   public void seek(long pos) throws IOException;                    // зависит от платформы
1.2 public void setLength(long newLength) throws IOException;        // зависит от платформы
// Методы, реализующие DataInput
   public final boolean readBoolean() throws IOException;
   public final byte readByte() throws IOException;
   public final char readChar() throws IOException;
   public final double readDouble() throws IOException;
   public final float readFloat() throws IOException;
   public final void readFully(byte[] b) throws IOException;
   public final void readFully(byte[] b, int off, int len) throws IOException;
   public final int readInt() throws IOException;
   public final String readLine() throws IOException;
   public final long readLong() throws IOException;
   public final short readShort() throws IOException;
   public final int readUnsignedByte() throws IOException;
   public final int readUnsignedShort() throws IOException;
   public final String readUTF() throws IOException;
   public int skipBytes(int n) throws IOException;
// Методы, реализующие DataOutput
   public void write(int b) throws IOException;                      // зависит от платформы
   public void write(byte[] b) throws IOException;
   public void write(byte[] b, int off, int len) throws IOException;
   public final void writeBoolean(boolean v) throws IOException;
   public final void writeByte(int v) throws IOException;
   public final void writeBytes(String s) throws IOException;
   public final void writeChar(int v) throws IOException;
   public final void writeChars(String s) throws IOException;
   public final void writeDouble(double v) throws IOException;
   public final void writeFloat(float v) throws IOException;
   public final void writeInt(int v) throws IOException;
   public final void writeLong(long v) throws IOException;
   public final void writeShort(int v) throws IOException;
   public final void writeUTF(String str) throws IOException;
}
```

Передается методам: `javax.imageio.stream.FileImageInputStream.FileImageInputStream()`, `javax.imageio.stream.FileImageOutputStream.FileImageOutputStream()`

Reader

Java 1.1

java.io

Этот абстрактный класс является родительским классом для всех входных потоков символов. Он представляет собой аналог `InputStream`, который является родительским классом всех входных потоков байтов. `Reader` определяет базовые методы, предоставляемые всеми входными потоками символов. `read()` возвращает один символ или массив (или подмассив) символов, при необходимости проводя блокировку; он возвращает –1, если был достигнут конец потока. `ready()` возвращает `true`, если есть символы, доступные для чтения. Если `ready()` возвращает `true`, то последующий вызов метода`read()` не вызовет блокировки. `close()` закрывает входной поток символов. `skip()` пропускает указанное количество символов во входном потоке. Если `markSupported()` возвращает `true`, то `mark()` отмечает позицию в потоке и при необходимости создает буфер упреждающей выборки указанного размера. Последующие запросы к `reset()` восстанавливают поток в отмеченную позицию при условии, если они имеют

место в пределах указанного лимита упреждающей выборки. Отметим, что не все типы потоков поддерживают функциональность вида «отметить-и-сбросить». Для создания подкласса Reader вам нужно лишь реализовать версию метода read() с тремя аргументами и метод close(). Тем не менее многие подклассы реализуют дополнительные методы.

```
public abstract class Reader {
// Защищенные конструкторы
   protected Reader();
   protected Reader(Object lock);
// Открытые методы экземпляра
   public abstract void close() throws IOException;
   public void mark(int readAheadLimit) throws IOException;
   public boolean markSupported();                                    // константа
   public int read() throws IOException;
   public int read(char[] cbuf) throws IOException;
   public abstract int read(char[] cbuf, int off, int len) throws IOException;
   public boolean ready() throws IOException;                         // константа
   public void reset() throws IOException;
   public long skip(long n) throws IOException;
// Защищенные поля экземпляра
   protected Object lock;
}
```

Подклассы: BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.awt.datatransfer.DataFlavor.getReaderForText(), java.nio.channels.Channels.newReader(), java.sql.Clob.getCharacterStream(), java.sql.ResultSet.getCharacterStream(), java.sql.SQLInput.readCharacterStream(), javax.print.Doc.getReaderForText(), javax.print.SimpleDoc.getReaderForText(), javax.xml.transform.stream.StreamSource.getReader(), org.xml.sax.InputSource.getCharacterStream()

Экземпляры: FilterReader.in

SequenceInputStream Java 1.0

java.io

Этот класс обеспечивает способ эффективного сцепления данных из двух или более входных потоков. Он обеспечивает интерфейс InputStream для ряда объектов InputStream. Данные считываются из потоков в том порядке, в каком указаны потоки. Когда достигнут конец одного потока, данные автоматически считываются из следующего потока. Например, этот класс может пригодиться при реализации возможности вставки (include) файлов в каком-либо парсере.


```
public class SequenceInputStream extends java.io.InputStream {
// Открытые конструкторы
   public SequenceInputStream(java.util.Enumeration e);
```

```
   public SequenceInputStream(java.io.InputStream s1, java.io.InputStream s2);
// Открытые методы, замещающие методы класса InputStream
1.1 public int available() throws IOException;
   public void close() throws IOException;
   public int read() throws IOException;
   public int read(byte[] b, int off, int len) throws IOException;
}
```

Serializable Java 1.1

java.io ***сериализуемый***

Интерфейс Serializable не определяет никаких методов или констант. Класс должен реализовывать этот интерфейс, чтобы обозначить, что он позволяет сериализовать или же десериализовать себя при помощи ObjectOutputStream.writeObject() и ObjectInputStream.readObject().

Объекты, требующие специальной обработки во время сериализации или десериализации, могут реализовывать один или оба следующих метода. Обратите внимание, что эти методы не являются частью интерфейса Serializable:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws
                                                    ception,ClassNotFoundException;
```

Как правило, метод writeObject() выполняет всю необходимую очистку или подготовку к сериализации, вызывает метод defaultWriteObject(), принадлежащий ObjectOutputStream, для сериализации нерезидентных полей класса и при необходимости записывает какие-либо дополнительные данные. Аналогично, метод readObject() вызывает метод defaultReadObject(), принадлежащий ObjectInputStream, считывает какие-либо дополнительные данные, записанные соответствующим методом writeObject(), и выполняет любую дополнительную инициализацию, требуемую объектом. Метод readObject() может регистрировать объект ObjectInputValidation в целях подтверждения целостности объекта в тот момент, когда он будет полностью десериализован.

```
public interface Serializable {
}
```

Реализации: Классов слишком много, чтобы их перечислить.

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: org.omg.CORBA.ValueBaseHolder.value

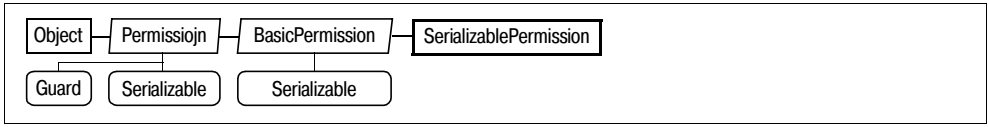
SerializablePermission Java 1.2

java.io ***сериализуемый***

Этот класс — java.security.Permission. Он управляет использованием определенных особенностей сериализации. Объекты SerializablePermission имеют имя, или целевой объект, но у них нет списка операций. Имя «enableSubclassImplementation» представляет разрешение на сериализацию или десериализацию объектов с использованием подклассов ObjectOutputStream и ObjectInputStream. Эта функция защищена правом доступа, поскольку злонамеренный программный код может определить подклассы потока объектов, которые неправильно сериализуют и десериализуют объекты.

Еще одно имя, поддерживаемое SerializablePermission, – «enableSubstitution», которое представляет для одного объекта право замены на другой во время сериализации или десериализации. Методам ObjectOutputStream.enableReplaceObject() и ObjectInputStream.enableResolveObject() необходимо право доступа такого типа.

У приложений никогда не возникает необходимости использовать этот класс. Его могут применять программисты, пишущие код системного уровня, а системные администраторы, отвечающие за конфигурацию политики безопасности, должны быть с ним знакомы.




```
public final class SerializablePermission extends java.security.BasicPermission {
// Открытые конструкторы
   public SerializablePermission(String name);
   public SerializablePermission(String name, String actions);
}
```

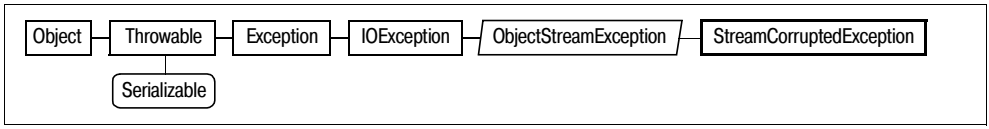
Экземпляры: ObjectStreamConstants.{SUBCLASS_IMPLEMENTATION_PERMISSION, SUBSTITUTION_PERMISSION}

StreamCorruptedException

Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение свидетельствует, что поток данных, который считывается ObjectInputStream, искажен и не содержит достоверных данных по сериализованному объекту.




```
public class StreamCorruptedException extends ObjectStreamException {
// Открытые конструкторы
   public StreamCorruptedException();
   public StreamCorruptedException(String reason);
}
```

Генерируется методами: ObjectInputStream.readStreamHeader(), java.rmi.server.RemoteCall.getResultStream()

StreamTokenizer

Java 1.0

java.io

Этот класс выполняет лексический анализ указанного входного потока и разбивает ввод на токены (tokens). Он может быть чрезвычайно полезен при написании простых парсеров. nextToken() возвращает следующий токен в потоке; это будет одна из констант, определенных классом (которые представляют конец файла, конец строки, число с плавающей запятой и слово), или же значение символа. pushBack() возвращает токен обратно в поток, чтобы он был возвращен при следующем вызове метода

nextToken(). Открытые переменные sval и nval содержат строковые и числовые значения того токена, который был прочитан самым последним. Они применимы в том случае, если возвращенный токен является TT_WORD или TT_NUMBER. lineno() возвращает номер текущей строки.

Оставшиеся методы позволяют указать, как будут распознаваться токены. wordChars() задает диапазон символов, которые должны трактоваться как часть слова. whitespaceChars() задает диапазон символов, которые служат для разграничения токенов. ordinaryChars() и ordinaryChar() указывают символы, которые никогда не являются частью токенов и должны возвращаться в том состоянии, в котором они находятся. resetSyntax() делает все символы обычными. eolIsSignificant() указывает, должен ли игнорироваться символ конца строки. Если эти символы не игнорируются, то для концов строк возвращается константа TT_EOL; в других случаях они обрабатываются как неотображаемые символы. commentChar() определяет символ, который начинает комментарий, длящийся до конца строки. Никакие символы в комментариях не возвращаются. slashStarComments() и slashSlashComments() определяют, будет ли StreamTokenizer распознавать комментарии в стиле C и C++. Если да, то никакая часть комментария не возвращается как токен. quoteChar() указывает символ, используемый для разграничения строк. Когда обрабатывается строковый токен, символ кавычек возвращается как значение токена, а тело строки сохраняется в переменной sval. lowerCaseMode() указывает на необходимость перевода токенов TT_WORD в символы, которые представлены только нижним регистром, перед тем как они будут сохранены в sval. parseNumbers() указывает, что StreamTokenizer должен распознавать и возвращать токены удвоенной точности, содержащие числа с плавающей точкой.

```
public class StreamTokenizer {
// Открытые конструкторы
# public StreamTokenizer(java.io.InputStream is);
1.1 public StreamTokenizer(Reader r);
// Открытые константы
   public static final int TT_EOF;                                    // =-1
   public static final int TT_EOL;                                    // =10
   public static final int TT_NUMBER;                                 // =-2
   public static final int TT_WORD;                                   // =-3
// Открытые методы экземпляра
   public void commentChar(int ch);
   public void eolIsSignificant(boolean flag);
   public int lineno();
   public void lowerCaseMode(boolean fl);
   public int nextToken() throws IOException;
   public void ordinaryChar(int ch);
   public void ordinaryChars(int low, int hi);
   public void parseNumbers();
   public void pushBack();
   public void quoteChar(int ch);
   public void resetSyntax();
   public void slashSlashComments(boolean flag);
   public void slashStarComments(boolean flag);
   public void whitespaceChars(int low, int hi);
   public void wordChars(int low, int hi);
// Открытые методы, замещающие методы класса Object
   public String toString();
// Открытые поля экземпляра
   public double nval;
   public String sval;
   public int ttype;
}
```

StringBufferInputStream

Java 1.0; устарел в java 1.1

java.io

Этот класс является подклассом InputStream, в котором входными байтами являются символы указанного объекта String. Этот класс некорректно переводит символы StringBuffer в байты, поэтому в Java 1.1 он признан устаревшим. Для перевода символов в байты служит StringReader, а для чтения байтов из массива – ByteArrayInputStream.

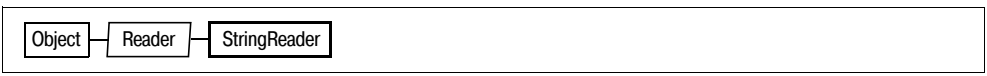

```
public class StringBufferInputStream extends java.io.InputStream {
// Открытые конструкторы
   public StringBufferInputStream(String s);
// Открытые методы, замещающие методы класса InputStream
   public int available();                                          // синхронизирован
   public int read();                                               // синхронизирован
   public int read(byte[] b, int off, int len);                     // синхронизирован
   public void reset();                                             // синхронизирован
   public long skip(long n);                                        // синхронизирован
// Защищенные поля экземпляра
   protected String buffer;
   protected int count;
   protected int pos;
}
```

StringReader

Java 1.1

java.io

Этот класс является входным потоком символов, который использует объект String в качестве источника возвращаемых символов. Когда вы создаете StringReader, укажите String, из которого будет производиться считывание. StringReader определяет обычные методы Reader и поддерживает mark() и reset(). Если reset() вызывается до вызова mark(), то поток устанавливается в начало указанной строки. StringReader является символьным аналогом класса StringBufferInputStream, который признан устаревшим в Java 1.1. StringReader также подобен CharArrayReader.




```
public class StringReader extends Reader {
// Открытые конструкторы
   public StringReader(String s);
// Открытые методы, замещающие методы класса Reader
   public void close();
   public void mark(int readAheadLimit) throws IOException;
   public boolean markSupported();                                  // константа
   public int read() throws IOException;
   public int read(char[] cbuf, int off, int len) throws IOException;
   public boolean ready() throws IOException;
   public void reset() throws IOException;
   public long skip(long ns) throws IOException;
}
```

StringWriter Java 1.1

java.io

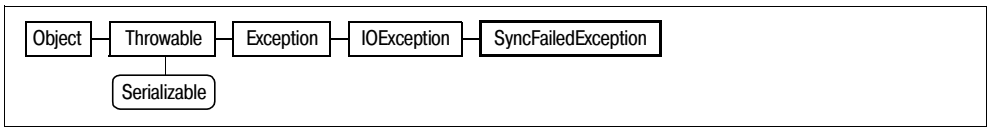
Этот класс является выходным потоком символов, который использует внутренний объект StringBuffer в качестве пункта назначения для символов, записанных в поток. При создании StringWriter вы можете указать внутренний размер StringBuffer. Однако вы не указываете сам StringBuffer; управление этим объектом происходит внутри StringWriter, а по мере необходимости StringBuffer увеличивается, чтобы вмещать записываемые в него символы. StringWriter определяет стандартные методы write(), flush() и close(), которые определяются всеми подклассами Writer, а также два метода, необходимые для получения символов, записанных во внутренний буфер потока. toString() возвращает содержимое внутреннего буфера в виде объекта String, а getBuffer() возвращает сам буфер. Отметим, что getBuffer() возвращает ссылку на реальный внутренний буфер, а не на его копию, а любые изменения, которые вы внесете в буфер, будут отражены в последующих запросах к toString(). StringWriter довольно похож на CharArrayWriter, но не имеет байтового аналога.


```
public class StringWriter extends Writer {
// Открытые конструкторы
   public StringWriter();
   public StringWriter(int initialSize);
// Открытые методы экземпляра
   public StringBuffer getBuffer();
// Открытые методы, замещающие методы класса Writer
   public void close() throws IOException;                    // пустой
   public void flush();                                       // пустой
   public void write(int c);
   public void write(String str);
   public void write(String str, int off, int len);
   public void write(char[] cbuf, int off, int len);
// Открытые методы, замещающие методы класса Object
   public String toString();
}
```

SyncFailedException Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение свидетельствует о том, что запрос к FileDescriptor.sync() не был завершен успешно.




```
public class SyncFailedException extends IOException {
// Открытые конструкторы
   public SyncFailedException(String desc);
}
```

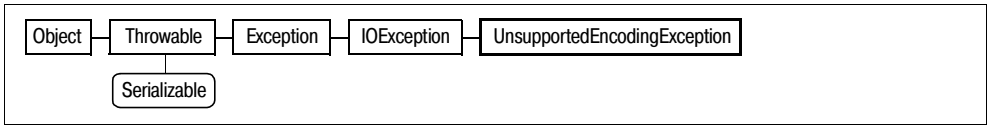
Генерируется методами: `FileDescriptor.sync()`

UnsupportedEncodingException

Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение свидетельствует о том, что запрошенная кодировка символов не поддерживается текущей виртуальной машиной Java.




```
public class UnsupportedEncodingException extends IOException {
// Открытые конструкторы
   public UnsupportedEncodingException();
   public UnsupportedEncodingException(String s);
}
```

Генерируется методами:

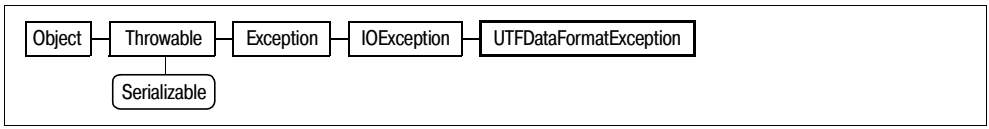
`ByteArrayOutputStream.toString()`, `InputStreamReader.InputStreamReader()`, `OutputStreamWriter.OutputStreamWriter()`, `PrintStream.PrintStream()`, `String.{getBytes(), String()}`, `java.net.URLDecoder.decode()`, `java.net.URLEncoder.encode()`, `java.util.logging.Handler.setEncoding()`, `java.util.logging.StreamHandler.setEncoding()`

UTFDataFormatException

Java 1.0

java.io ***сериализуемое, проверяемое***

Это исключение – `IOException`. Оно свидетельствует о том, что искаженная строка UTF-8 была обнаружена классом, который реализует интерфейс `DataInput`. UTF-8 является преобразовательным форматом, совместимым с ASCII. Он используется для хранения и передачи Unicode-текста.




```
public class UTFDataFormatException extends IOException {
// Открытые конструкторы
   public UTFDataFormatException();
   public UTFDataFormatException(String s);
}
```

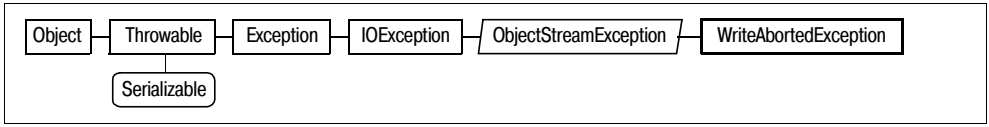
WriteAbortedException

Java 1.1

java.io ***сериализуемое, проверяемое***

Это исключение генерируется при прочтении потока данных, который является неполным, поскольку во время его записи было сгенерировано исключение. Поле `detail` может содержать исключение, которое прервало создание выходного потока. В Java 1.4

и последующих версиях это исключение может быть также получено при помощи стандартного метода Throwable getCause(). Метод getMessage() был замещен, чтобы включать сообщение этого исключения detail, если таковое имеется.




```
public class WriteAbortedException extends ObjectStreamException {
// Открытые конструкторы
   public WriteAbortedException(String s, Exception ex);
// Открытые методы, замещающие методы класса Throwable
1.4 public Throwable getCause();
   public String getMessage();
// Открытые поля экземпляра
   public Exception detail;
}
```

Writer Java 1.1

java.io

Данный абстрактный класс является родительским классом всех выходных потоков символов. Это аналог OutputStream, который является родительским классом всех выходных потоков байтов. Writer определяет базовые методы write(), flush() и close(), предоставляемые всеми выходными потоками символов. Пять версий метода write() записывают один символ, массив, подмассив символов, строку или подстроку в пункт назначения потока. Наиболее общая версия этого метода записывает указанную часть массива символов. Это абстрактная версия, которая должна быть реализована всеми подклассами. По умолчанию другие методы write() реализуются сообразно с этой абстрактной версией. Метод flush() является еще одним абстрактным методом, который должны реализовывать все подклассы. Любой вывод, буферизированный потоком, записывается в его пункт назначения. Если такой пункт назначения сам является выходным потоком байтов или символов, то он также должен вызывать метод flush(), принадлежащий потоку пункта назначения. Метод close() тоже является абстрактным. Подкласс должен реализовать этот метод, чтобы он сбрасывал и затем закрывал текущий поток, а также закрывал любой поток пункта назначения, к которому он присоединен. Как только поток закрыт, любые последующий запросы к write() или flush() должны генерировать IOException.

```
public abstract class Writer {
// Защищенные конструкторы
   protected Writer();
   protected Writer(Object lock);
// Открытые методы экземпляра
   public abstract void close() throws IOException;
   public abstract void flush() throws IOException;
   public void write(String str) throws IOException;
   public void write(char[] cbuf) throws IOException;
   public void write(int c) throws IOException;
   public void write(String str, int off, int len) throws IOException;
   public abstract void write(char[] cbuf, int off, int len) throws IOException;
```

```
// Защищенные поля экземпляра
   protected Object lock;
}
```

Подклассы: BufferedWriter, CharArrayWriter, FilterWriter, OutputStreamWriter, PipedWriter, PrintWriter, StringWriter

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.nio.channels.Channels.newWriter(), java.sql.Clob.setCharacterStream(), javax.swing.text.AbstractWriter.getWriter(), javax.xml.transform.stream.StreamResult.getWriter()

Экземпляры: FilterWriter.out, PrintWriter.out

Глава 11

java.lang, java.lang.ref и java.lang.reflect

В этой главе описывается пакет java.lang. Он содержит основные классы и интерфейсы, которые являются неотъемлемой частью платформы и языка программирования Java. Здесь также описываются два специализированных подпакета. Первый, java.lang.ref, определяет классы-ссылки, которые ссылаются на объекты, не мешая при этом сборщику мусора утилизировать эти объекты. Второй, java.lang.reflect, позволяет вызывать методы, запрашивать и устанавливать значения полей произвольного класса.

Пакет java.lang

Java 1.0

Пакет java.lang содержит самые главные классы языка Java. Класс Object является родительским классом всех классов Java и находится на вершине иерархии классов. Class описывает произвольный класс Java. С каждым загруженным классом связан только один объект типа Class.

Классы Boolean, Character, Byte, Short, Integer, Long, Float и Double являются обертками (wrappers) для простых типов данных Java. Они позволяют обращаться с простыми типами как с объектами. В этих классах также есть удобные методы для преобразования типов и другие вспомогательные методы. Аналогичным образом класс Void представляет тип void возвращаемого значения, но в нем не определены собственные методы. String и StringBuffer представляют строки. String является неизменяемым типом, тогда как строка типа StringBuffer может меняться. В Java 1.4 классы String и StringBuffer реализуют интерфейс CharSequence, поэтому можно легко управлять экземплярами этих двух классов посредством общего API. Строки и обертки для простых типов реализуют интерфейс Comparable, который поддерживает упорядочение экземпляров этих классов и алгоритмы сортировки и поиска (например, java.util.Arrays и java.util.Collections). Интерфейс Cloneable служит признаком, по которому определяется, может ли метод Object.clone() создавать копии данного объекта.

Класс Math (а также StrictMath в Java 1.3) определяет статические методы, выполняющие различные математические операции над числами с плавающей точкой.

Класс Thread предназначен для поддержки нескольких потоков управления, выполняющихся в одном интерпретаторе Java. Интерфейс Runnable реализуется объектами, в которых есть метод run(), служащий телом потока.

Класс System предоставляет низкоуровневые системные методы, а Runtime – аналогичные высокоуровневые методы, в том числе и exec(), который вместе с классом Process определяет платформо-зависимый API для запуска внешних процессов.

Throwable является базовым классом в иерархии исключений и ошибок. Объекты этого класса используются вместе с операторами throw и catch. В пакете java.lang определено несколько потомков класса Throwable. Exception и Error являются базовыми классами для всех исключений и ошибок. RuntimeException является специальным классом «непроверяемых исключений», которые не нужно указывать в операторе throws при объявлении метода. В Java 1.4 класс Throwable был переписан; в него была добавлена поддержка сцепленных исключений (chain exceptions) и возможность получить трассировку стека исключения в виде массива объектов StackTraceElement.

Интерфейсы

```
public interface CharSequence;
public interface Cloneable;
public interface Comparable;
public interface Runnable;
```

Классы

```
public class Object;
   └ public final class Boolean implements Serializable;
   └ public final class Character implements Comparable, Serializable;
   └ public static class Character.Subset;
      └ public static final class Character.UnicodeBlock extends Character.Subset;
   └ public final class Class implements Serializable;
   └ public abstract class ClassLoader;
   └ public final class Compiler;
   └ public final class Math;
   └ public abstract class Number implements Serializable;
      └ public final class Byte extends Number implements Comparable;
      └ public final class Double extends Number implements Comparable;
      └ public final class Float extends Number implements Comparable;
      └ public final class Integer extends Number implements Comparable;
      └ public final class Long extends Number implements Comparable;
      └ public final class Short extends Number implements Comparable;
   └ public class Package;
   └ public abstract class Process;
   └ public class Runtime;
   └ public class SecurityManager;
   └ public final class StackTraceElement implements Serializable;
   └ public final class StrictMath;
   └ public final class String implements CharSequence, Comparable, Serializable;
   └ public final class StringBuffer implements CharSequence, Serializable;
   └ public final class System;
   └ public class Thread implements Runnable;
   └ public class ThreadGroup;
   └ public class ThreadLocal;
      └ public class InheritableThreadLocal extends ThreadLocal;
   └ public class Throwable implements Serializable;
   └ public final class Void;
public final class RuntimePermission extends java.security.BasicPermission;
```

Исключения

```
public class Exception extends Throwable;
   └ public class ClassNotFoundException extends Exception;
   └ public class CloneNotSupportedException extends Exception;
   └ public class IllegalAccessException extends Exception;
   └ public class InstantiationException extends Exception;
   └ public class InterruptedException extends Exception;
   └ public class NoSuchFieldException extends Exception;
   └ public class NoSuchMethodException extends Exception;
   └ public class RuntimeException extends Exception;
      └ public class ArithmeticException extends RuntimeException;
      └ public class ArrayStoreException extends RuntimeException;
      └ public class ClassCastException extends RuntimeException;
      └ public class IllegalArgumentException extends RuntimeException;
         └ public class IllegalThreadStateException extends IllegalArgumentException;
         └ public class NumberFormatException extends IllegalArgumentException;
      └ public class IllegalMonitorStateException extends RuntimeException;
      └ public class IllegalStateException extends RuntimeException;
      └ public class IndexOutOfBoundsException extends RuntimeException;
         └ public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException;
         └ public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException;
      └ public class NegativeArraySizeException extends RuntimeException;
      └ public class NullPointerException extends RuntimeException;
      └ public class SecurityException extends RuntimeException;
      └ public class UnsupportedOperationException extends RuntimeException;
```

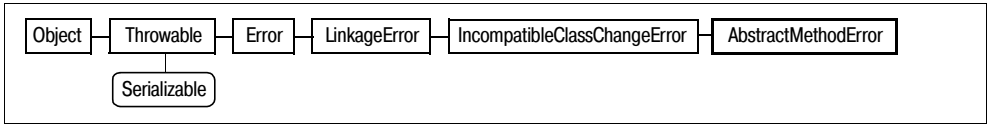
Ошибки

```
public class Error extends Throwable;
   └ public class AssertionError extends Error;
   └ public class LinkageError extends Error;
      └ public class ClassCircularityError extends LinkageError;
      └ public class ClassFormatError extends LinkageError;
         └ public class UnsupportedClassVersionError extends ClassFormatError;
      └ public class ExceptionInInitializerError extends LinkageError;
      └ public class IncompatibleClassChangeError extends LinkageError;
         └ public class AbstractMethodError extends IncompatibleClassChangeError;
         └ public class IllegalAccessError extends IncompatibleClassChangeError;
         └ public class InstantiationError extends IncompatibleClassChangeError;
         └ public class NoSuchFieldError extends IncompatibleClassChangeError;
         └ public class NoSuchMethodError extends IncompatibleClassChangeError;
      └ public class NoClassDefFoundError extends LinkageError;
      └ public class UnsatisfiedLinkError extends LinkageError;
      └ public class VerifyError extends LinkageError;
   └ public class ThreadDeath extends Error;
   └ public abstract class VirtualMachineError extends Error;
   └ public class InternalError extends VirtualMachineError;
   └ public class OutOfMemoryError extends VirtualMachineError;
   └ public class StackOverflowError extends VirtualMachineError;
   └ public class UnknownError extends VirtualMachineError;
```

AbstractMethodError Java 1.0

java.lang *сериализуемый, ошибка*

Ошибка этого типа возникает при попытке вызова абстрактного метода.



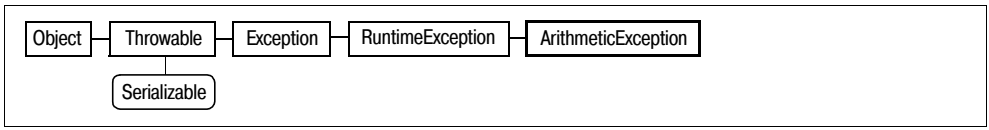

```
public class AbstractMethodError extends IncompatibleClassChangeError {
// Открытые конструкторы
   public AbstractMethodError();
   public AbstractMethodError(String s);
}
```

ArithmeticException

Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение RuntimeException сообщает об особой ситуации, возникшей при выполнении арифметического действия (например, при делении на нуль).



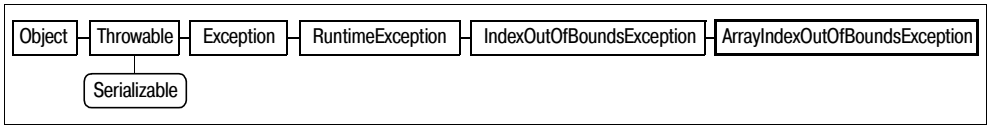

```
public class ArithmeticException extends RuntimeException {
// Открытые конструкторы
   public ArithmeticException();
   public ArithmeticException(String s);
}
```

ArrayIndexOutOfBoundsException

Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение свидетельствует о том, что индекс элемента массива меньше нуля либо больше или равен размеру массива.




```
public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException {
// Открытые конструкторы
   public ArrayIndexOutOfBoundsException();
   public ArrayIndexOutOfBoundsException(String s);
   public ArrayIndexOutOfBoundsException(int index);
}
```

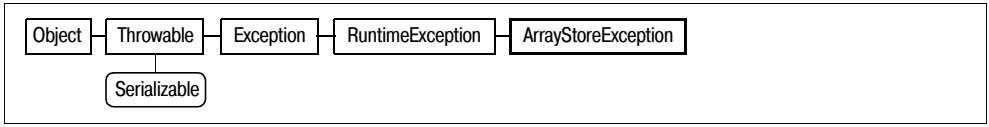
Генерируется методами: Методов слишком много, чтобы их перечислить.

ArrayStoreException

Java 1.0

java.lang *сериализуемое, непроверяемое*

Это исключение возникает при попытке записать в массив объект недопустимого типа.



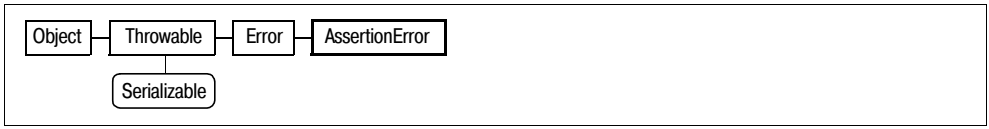

```
public class ArrayStoreException extends RuntimeException {
// Открытые конструкторы
   public ArrayStoreException();
   public ArrayStoreException(String s);
}
```

AssertionError

Java 1.4

java.lang *сериализуемый, ошибка*

Данное исключение генерируется при ошибке в утверждении (`assertion`). Это происходит тогда, когда утверждения включены, а выражение, стоящее после оператора `assert`, не возвращает `true`. Если утверждение ошибочно, а в операторе `assert` стоит второе выражение, отделенное от первого двоеточием, то вычисляется второе выражение и возвращенное значение передается конструктору `AssertionError()`, который преобразует его в строку и использует как текст сообщения об ошибке.




```
public class AssertionError extends Error {
// Открытые конструкторы
   public AssertionError();
   public AssertionError(long detailMessage);
   public AssertionError(float detailMessage);
   public AssertionError(double detailMessage);
   public AssertionError(int detailMessage);
   public AssertionError(Object detailMessage);
   public AssertionError(boolean detailMessage);
   public AssertionError(char detailMessage);
}
```

Boolean

Java 1.0

java.lang *сериализуемый*

Этот класс представляет собой неизменяемую обертку для булева типа. Обратите внимание на то, что константы `TRUE` и `FALSE` являются объектами типа `Boolean`, а не булевыми значениями `true` и `false`. В Java 1.1 этот класс определяет константу типа `Class`, которой представлен логический тип. Метод `booleanValue()` возвращает булево значение, содержащееся в объекте `Boolean`. Метод класса `getBoolean()` извлекает бу-

лево значение указанного свойства из списка системных свойств. Метод класса valueOf() анализирует строку и возвращает объект типа Boolean, который она представляет. В Java 1.4 добавлены два новых метода класса для преобразования значений простейшего логического типа в объекты Boolean и String.


```
public final class Boolean implements Serializable {
// Открытые конструкторы
   public Boolean(String s);
   public Boolean(boolean value);
// Открытые константы
   public static final Boolean FALSE;
   public static final Boolean TRUE;
1.1 public static final Class TYPE;
// Открытые методы класса
   public static boolean getBoolean(String name);
1.4 public static String toString(boolean b);
1.4 public static Boolean valueOf(boolean b);
   public static Boolean valueOf(String s);
// Открытые методы экземпляра
   public boolean booleanValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: javax.swing.DefaultDesktopManager.setWasIcon()

Возвращается методами: Boolean.valueOf(), javax.swing.filechooser.FileSystemView.isTraversable(), javax.swing.filechooser.FileView.isTraversable(), javax.swing.plaf.basic.BasicFileChooserUI.BasicFileView.isHidden()

Экземпляры: java.awt.font.TextAttribute.{RUN_DIRECTION_LTR, RUN_DIRECTION_RTL, STRIKETHROUGH_ON, SWAP_COLORS_ON}, Boolean.FALSE, TRUE}

Byte

Java 1.1

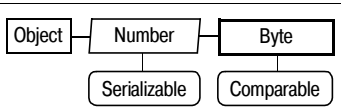
java.lang ***сериализуемый, сравнимый***

Этот класс является оберткой для простого типа byte. В нем определены полезные константы для минимального и максимального значений этого типа, а также константа типа Class, которой представлен тип byte. Кроме того, в этом классе содержатся различные методы для преобразования значений Byte в строки и другие целочисленные значения, а также методы для обратного преобразования.

Большая часть статических методов этого класса предназначена для преобразования объекта String в объект Byte или значение типа byte: четыре варианта метода parseByte() и valueOf() преобразуют указанную строку в число (может быть задано основание системы счисления) и возвращают его в виде Byte или byte. Метод decode() анализирует десятичное, восьмеричное или шестнадцатеричное число, содержащееся в строке, и возвращает его в виде Byte. Если строка начинается с «0x» или «#», она интерпрети-

руется как шестнадцатеричное число. Если она начинается с «0», то интерпретируется как восьмеричное число. В остальных случаях число считается десятичным.

Обратите внимание, что в этом классе есть два варианта метода toString(). Первый, статический, преобразует значение типа byte в строку; второй преобразует в строку объект Byte. Остальные методы приводят Byte к простым целочисленным типам.




```
public final class Byte extends Number implements Comparable {
// Открытые конструкторы
   public Byte(byte value);
   public Byte(String s) throws NumberFormatException;
// Открытые константы
   public static final byte MAX_VALUE;                                    // =127
   public static final byte MIN_VALUE;                                    // =-128
   public static final Class TYPE;
// Открытые методы класса
   public static Byte decode(String nm) throws NumberFormatException;
   public static byte parseByte(String s) throws NumberFormatException;
   public static byte parseByte(String s, int radix) throws NumberFormatException;
   public static String toString(byte b);
   public static Byte valueOf(String s) throws NumberFormatException;
   public static Byte valueOf(String s, int radix) throws NumberFormatException;
// Открытые методы экземпляра
1.2 public int compareTo(Byte anotherByte);
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы Number
   public byte byteValue();
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
   public short shortValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: Byte.compareTo()

Возвращается методами: Byte.{decode(), valueOf()}

Character

Java 1.0

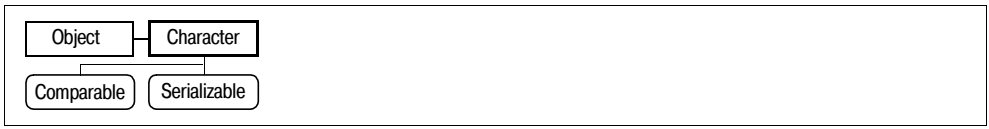
java.lang ***сериализуемый, сравнимый***

Этот класс является неизменяемой оберткой для простого типа char. Метод charValue() возвращает значение типа char объекта Character. Метод compareTo() реализует интерфейс Comparable, поэтому объекты Character можно сравнивать и сортировать. Но наибольший интерес представляют статические методы этого класса: они прове-

ряют принадлежность символьных значений категориям, определенным в стандарте Unicode. (Для использования некоторых методов необходимо знание этого стандарта.) Статические методы, начинающиеся с «is», проверяют, принадлежит ли символ данной категории.

Из этих методов чаще всего используются isDigit(), isLetter(), isWhitespace(), isUpperCase() и isLowerCase(). Заметьте, что данные методы работают с любыми символами в формате Unicode, а не только с латинскими буквами и арабскими цифрами из таблицы символов ASCII. Метод getType() возвращает константу, которая определят категорию символа. getDirectionality() возвращает константу DIRECTIONALITY_, определяющую так называемую «категорию направления» символа.

Кроме методов для определения категории символа, в этом классе также определены статические методы для преобразования символов. toUpperCase() возвращает эквивалент указанного символа в верхнем регистре (или исходный символ, если символ представлен в верхнем регистре или не имеет эквивалента в верхнем регистре). toLowerCase() преобразует символ в нижний регистр. digit() возвращает целочисленный эквивалент данного символа в указанной системе счисления (можно указывать основание; например для шестнадцатеричной системы укажите 16). Этот метод работает с символами Unicode, обозначающими цифры и латинские буквы (a-z и A-Z) таблицы ASCII. forDigit() возвращает символ ASCII, соответствующий указанному значению (0-35) с заданным основанием системы счисления. Аналогичный метод getNumericValue() работает также с символами Unicode, включая недесятичные цифры (например, римские). Наконец, статический метод toString() возвращает строку длиной 1, содержащую указанный символ.




```
public final class Character implements Comparable, Serializable {
// Открытые конструкторы
   public Character(char value);
// Открытые константы
1.1 public static final byte COMBINING_SPACING_MARK;                    // =8
1.1 public static final byte CONNECTOR_PUNCTUATION;                     // =23
1.1 public static final byte CONTROL;                                   // =15
1.1 public static final byte CURRENCY_SYMBOL;                           // =26
1.1 public static final byte DASH_PUNCTUATION;                          // =20
1.1 public static final byte DECIMAL_DIGIT_NUMBER;                      // =9
1.1 public static final byte ENCLOSING_MARK;                            // =7
1.1 public static final byte END_PUNCTUATION;                           // =22
1.4 public static final byte FINAL_QUOTE_PUNCTUATION;                   // =30
1.1 public static final byte FORMAT;                                    // =16
1.4 public static final byte INITIAL_QUOTE_PUNCTUATION;                 // =29
1.1 public static final byte LETTER_NUMBER;                             // =10
1.1 public static final byte LINE_SEPARATOR;                            // =13
1.1 public static final byte LOWERCASE_LETTER;                          // =2
1.1 public static final byte MATH_SYMBOL;                               // =25
   public static final int MAX_RADIX;                                   // =36
   public static final char MAX_VALUE;                                  // ='\uFFFF'
   public static final int MIN_RADIX;                                   // =2
   public static final char MIN_VALUE;                                  // ='\u0000'
1.1 public static final byte MODIFIER_LETTER;                           // =4
```

```
1.1 public static final byte MODIFIER_SYMBOL;                                    // =27
1.1 public static final byte NON_SPACING_MARK;                                   // =6
1.1 public static final byte OTHER_LETTER;                                       // =5
1.1 public static final byte OTHER_NUMBER;                                       // =11
1.1 public static final byte OTHER_PUNCTUATION;                                  // =24
1.1 public static final byte OTHER_SYMBOL;                                       // =28
1.1 public static final byte PARAGRAPH_SEPARATOR;                                // =14
1.1 public static final byte PRIVATE_USE;                                        // =18
1.1 public static final byte SPACE_SEPARATOR;                                    // =12
1.1 public static final byte START_PUNCTUATION;                                  // =21
1.1 public static final byte SURROGATE;                                          // =19
1.1 public static final byte TITLECASE_LETTER;                                   // =3
1.1 public static final Class TYPE;
1.1 public static final byte UNASSIGNED;                                         // =0
1.1 public static final byte UPPERCASE_LETTER;                                   // =1
1.4 public static final byte DIRECTIONALITY_ARABIC_NUMBER;                       // =6
1.4 public static final byte DIRECTIONALITY_BOUNDARY_NEUTRAL;                    // =9
1.4 public static final byte DIRECTIONALITY_COMMON_NUMBER_SEPARATOR;             // =7
1.4 public static final byte DIRECTIONALITY_EUROPEAN_NUMBER;                     // =3
1.4 public static final byte DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR;           // =4
1.4 public static final byte DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR;          // =5
1.4 public static final byte DIRECTIONALITY_LEFT_TO_RIGHT;                       // =0
1.4 public static final byte DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING;             // =14
1.4 public static final byte DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE;              // =15
1.4 public static final byte DIRECTIONALITY_NONSPACING_MARK;                     // =8
1.4 public static final byte DIRECTIONALITY_OTHER_NEUTRALS;                      // =13
1.4 public static final byte DIRECTIONALITY_PARAGRAPH_SEPARATOR;                 // =10
1.4 public static final byte DIRECTIONALITY_POP_DIRECTIONAL_FORMAT;              // =18
1.4 public static final byte DIRECTIONALITY_RIGHT_TO_LEFT;                       // =1
1.4 public static final byte DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC;                // =2
1.4 public static final byte DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING;             // =16
1.4 public static final byte DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE;              // =17
1.4 public static final byte DIRECTIONALITY_SEGMENT_SEPARATOR;                   // =11
1.4 public static final byte DIRECTIONALITY_UNDEFINED;                           // =-1
1.4 public static final byte DIRECTIONALITY_WHITESPACE;                          // =12
// Внутренние классы
1.2 public static class Subset;
1.2 public static final class UnicodeBlock extends Character.Subset;
// Открытые методы класса
   public static int digit(char ch, int radix);
   public static char forDigit(int digit, int radix);
1.4 public static byte getDirectionality(char c);
1.1 public static int getNumericValue(char ch);
1.1 public static int getType(char ch);
   public static boolean isDefined(char ch);
   public static boolean isDigit(char ch);
1.1 public static boolean isIdentifierIgnorable(char ch);
1.1 public static boolean isISOControl(char ch);
1.1 public static boolean isJavaIdentifierPart(char ch);
1.1 public static boolean isJavaIdentifierStart(char ch);
   public static boolean isLetter(char ch);
   public static boolean isLetterOrDigit(char ch);
   public static boolean isLowerCase(char ch);
1.4 public static boolean isMirrored(char c);
1.1 public static boolean isSpaceChar(char ch);
   public static boolean isTitleCase(char ch);
```

```
1.1 public static boolean isUnicodeIdentifierPart(char ch);
1.1 public static boolean isUnicodeIdentifierStart(char ch);
   public static boolean isUpperCase(char ch);
1.1 public static boolean isWhitespace(char ch);
   public static char toLowerCase(char ch);
1.4 public static String toString(char c);
   public static char toTitleCase(char ch);
   public static char toUpperCase(char ch);
// Открытые методы экземпляра
   public char charValue();
1.2 public int compareTo(Character anotherCharacter);
// Реализация методов интерфейса Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
// Открытые методы, объявленные устаревшими
# public static boolean isJavaLetter(char ch);
# public static boolean isJavaLetterOrDigit(char ch);
# public static boolean isSpace(char ch);
}
```

Передается методам: java.awt.AWTKeyStroke.getAWTKeyStroke(), Character.compareTo(), javax.swing.KeyStroke.getKeyStroke()

Character.Subset

Java 1.2

java.lang

Этот класс представляет именованное подмножество набора символов Unicode. Метод toString() возвращает имя этого подмножества. Это базовый класс, предназначенный для наследования. В частности, в нем нет методов для составления списка из членов подмножества или для проверки на принадлежность этому подмножеству. См. также Character.UnicodeBlock.

```
public static class Character.Subset {
// Защищенные конструкторы
   protected Subset(String name);
// Открытые методы, замещающие методы класса Object
   public final boolean equals(Object obj);
   public final int hashCode();
   public final String toString();
}
```

Подклассы: java.awt.im.InputSubset, Character.UnicodeBlock

Передается методам: java.awt.im.InputContext.setCharacterSubsets(), java.awt.im.spi.InputMethod.setCharacterSubsets()

Character.UnicodeBlock

Java 1.2

java.lang

Этот потомок класса Character.Subset определяет набор констант, представляющих собой именованные подмножества набора символов Unicode. Подмножества и их имена являются блоками символов, определенных в спецификации Unicode (см. по

адресу *http://www.unicode.org/)*. В Java 1.4 этот класс приведен в соответствие с новым стандартом Unicode; в нем определены новые константы. Статический метод `of()` принимает аргумент типа `Character` и возвращает объект блока (`Character.UnicodeBlock`), которому принадлежит этот символ, или `null`, если он не принадлежит ни одному из определенных блоков. Этот метод удобен, если нужно определить, к какому алфавиту принадлежит неизвестный символ.

```
public static final class Character.UnicodeBlock extends Character.Subset {
// Конструктор отсутствует
// Открытые константы
   public static final Character.UnicodeBlock ALPHABETIC_PRESENTATION_FORMS;
   public static final Character.UnicodeBlock ARABIC;
   public static final Character.UnicodeBlock ARABIC_PRESENTATION_FORMS_A;
   public static final Character.UnicodeBlock ARABIC_PRESENTATION_FORMS_B;
   public static final Character.UnicodeBlock ARMENIAN;
   public static final Character.UnicodeBlock ARROWS;
   public static final Character.UnicodeBlock BASIC_LATIN;
   public static final Character.UnicodeBlock BENGALI;
   public static final Character.UnicodeBlock BLOCK_ELEMENTS;
   public static final Character.UnicodeBlock BOPOMOFO;
1.4 public static final Character.UnicodeBlock BOPOMOFO_EXTENDED;
   public static final Character.UnicodeBlock BOX_DRAWING;
1.4 public static final Character.UnicodeBlock BRAILLE_PATTERNS;
1.4 public static final Character.UnicodeBlock CHEROKEE;
   public static final Character.UnicodeBlock CJK_COMPATIBILITY;
   public static final Character.UnicodeBlock CJK_COMPATIBILITY_FORMS;
   public static final Character.UnicodeBlock CJK_COMPATIBILITY_IDEOGRAPHS;
1.4 public static final Character.UnicodeBlock CJK_RADICALS_SUPPLEMENT;
   public static final Character.UnicodeBlock CJK_SYMBOLS_AND_PUNCTUATION;
   public static final Character.UnicodeBlock CJK_UNIFIED_IDEOGRAPHS;
1.4 public static final Character.UnicodeBlock CJK_UNIFIED_IDEOGRAPHS_EXTENSION_A;
   public static final Character.UnicodeBlock COMBINING_DIACRITICAL_MARKS;
   public static final Character.UnicodeBlock COMBINING_HALF_MARKS;
   public static final Character.UnicodeBlock COMBINING_MARKS_FOR_SYMBOLS;
   public static final Character.UnicodeBlock CONTROL_PICTURES;
   public static final Character.UnicodeBlock CURRENCY_SYMBOLS;
   public static final Character.UnicodeBlock CYRILLIC;
   public static final Character.UnicodeBlock DEVANAGARI;
   public static final Character.UnicodeBlock DINGBATS;
   public static final Character.UnicodeBlock ENCLOSED_ALPHANUMERICS;
   public static final Character.UnicodeBlock ENCLOSED_CJK_LETTERS_AND_MONTHS;
1.4 public static final Character.UnicodeBlock ETHIOPIC;
   public static final Character.UnicodeBlock GENERAL_PUNCTUATION;
   public static final Character.UnicodeBlock GEOMETRIC_SHAPES;
   public static final Character.UnicodeBlock GEORGIAN;
   public static final Character.UnicodeBlock GREEK;
   public static final Character.UnicodeBlock GREEK_EXTENDED;
   public static final Character.UnicodeBlock GUJARATI;
   public static final Character.UnicodeBlock GURMUKHI;
   public static final Character.UnicodeBlock HALFWIDTH_AND_FULLWIDTH_FORMS;
   public static final Character.UnicodeBlock HANGUL_COMPATIBILITY_JAMO;
   public static final Character.UnicodeBlock HANGUL_JAMO;
   public static final Character.UnicodeBlock HANGUL_SYLLABLES;
   public static final Character.UnicodeBlock HEBREW;
   public static final Character.UnicodeBlock HIRAGANA;
1.4 public static final Character.UnicodeBlock IDEOGRAPHIC_DESCRIPTION_CHARACTERS;
```

```
   public static final Character.UnicodeBlock IPA_EXTENSIONS;
   public static final Character.UnicodeBlock KANBUN;
1.4 public static final Character.UnicodeBlock KANGXI_RADICALS;
   public static final Character.UnicodeBlock KANNADA;
   public static final Character.UnicodeBlock KATAKANA;
1.4 public static final Character.UnicodeBlock KHMER;
   public static final Character.UnicodeBlock LAO;
   public static final Character.UnicodeBlock LATIN_1_SUPPLEMENT;
   public static final Character.UnicodeBlock LATIN_EXTENDED_A;
   public static final Character.UnicodeBlock LATIN_EXTENDED_ADDITIONAL;
   public static final Character.UnicodeBlock LATIN_EXTENDED_B;
   public static final Character.UnicodeBlock LETTERLIKE_SYMBOLS;
   public static final Character.UnicodeBlock MALAYALAM;
   public static final Character.UnicodeBlock MATHEMATICAL_OPERATORS;
   public static final Character.UnicodeBlock MISCELLANEOUS_SYMBOLS;
   public static final Character.UnicodeBlock MISCELLANEOUS_TECHNICAL;
1.4 public static final Character.UnicodeBlock MONGOLIAN;
1.4 public static final Character.UnicodeBlock MYANMAR;
   public static final Character.UnicodeBlock NUMBER_FORMS;
1.4 public static final Character.UnicodeBlock OGHAM;
   public static final Character.UnicodeBlock OPTICAL_CHARACTER_RECOGNITION;
   public static final Character.UnicodeBlock ORIYA;
   public static final Character.UnicodeBlock PRIVATE_USE_AREA;
1.4 public static final Character.UnicodeBlock RUNIC;
1.4 public static final Character.UnicodeBlock SINHALA;
   public static final Character.UnicodeBlock SMALL_FORM_VARIANTS;
   public static final Character.UnicodeBlock SPACING_MODIFIER_LETTERS;
   public static final Character.UnicodeBlock SPECIALS;
   public static final Character.UnicodeBlock SUPERSCRIPTS_AND_SUBSCRIPTS;
   public static final Character.UnicodeBlock SURROGATES_AREA;
1.4 public static final Character.UnicodeBlock SYRIAC;
   public static final Character.UnicodeBlock TAMIL;
   public static final Character.UnicodeBlock TELUGU;
1.4 public static final Character.UnicodeBlock THAANA;
   public static final Character.UnicodeBlock THAI;
   public static final Character.UnicodeBlock TIBETAN;
1.4 public static final Character.UnicodeBlock UNIFIED_CANADIAN_ABORIGINAL_SYLLABICS;
1.4 public static final Character.UnicodeBlock YI_RADICALS;
1.4 public static final Character.UnicodeBlock YI_SYLLABLES;
// Открытые методы класса
   public static Character.UnicodeBlock of(char c);
}
```

Возвращается методами: `Character.UnicodeBlock.of()`

Экземпляры: Полей слишком много, чтобы их перечислить.

CharSequence Java 1.4

java.lang

Этот интерфейс содержит простой API для чтения последовательностей символов. Его реализуют классы `String`, `StringBuffer` и `java.nio.CharBuffer` ядра платформы Java. Метод `charAt()` возвращает символ, находящийся на указанной позиции в последовательности. `length()` возвращает количество символов в последовательности. Метод `subSequence()` возвращает подпоследовательность символов (объект `CharSequence`) с указанным начальным индексом (включительно) и конечным индексом, не включая

этот символ. Наконец, метод `toString()` возвращает последовательность в виде объекта `String`.

Обратите внимание, что в реализациях интерфейса `CharSequence` нет универсальных методов `equals()` и `hashCode()`. Как правило, не существует возможности сравнить два объекта `CharSequence` и добавить несколько объектов в одно множество или хеш-таблицу, если это не объекты одного класса.

```
public interface CharSequence {
// Открытые методы экземпляра
   public abstract char charAt(int index);
   public abstract int length();
   public abstract CharSequence subSequence(int start, int end);
   public abstract String toString();
}
```

Реализации: `String`, `StringBuffer`, `java.nio.CharBuffer`

Передается методам: `java.nio.CharBuffer.wrap()`, `java.nio.charset.CharsetEncoder.canEncode()`, `java.util.regex.Matcher.reset()`, `java.util.regex.Pattern.{matcher(), matches(), split()}`

Возвращается методами: `CharSequence.subSequence()`, `String.subSequence()`, `StringBuffer.subSequence()`, `java.nio.CharBuffer.subSequence()`

Class

Java 1.0

java.lang ***сериализуемый***

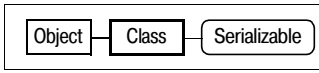
Этим классом представлен какой-нибудь класс или интерфейс Java, а в Java 1.1 – любой тип. Для каждого класса, загруженного в ВМ Java, создается свой объект `Class`, а в Java 1.1 создаются специальные объекты `Class`, представляющие простые типы. Эти специальные объекты присвоены константам `TYPE`, определенным в классах `Boolean`, `Integer` и других обертках для простых типов. В Java 1.1 массивы также представлены объектами `Class`.

У этого класса нет конструктора. Получить объект `Class` можно с помощью вызова метода `getClass()` из любого экземпляра требуемого класса. И самое главное, класс можно динамически загрузить, передав статическому методу `Class.forName()` полное имя класса (то есть имя пакета плюс имя класса). Этот метод загружает указанный класс (если он еще не загружен) в интерпретатор Java и возвращает соответствующий ему объект `Class`. Кроме того, классы можно загружать с помощью объекта `ClassLoader`.

Метод `newInstance()` создает экземпляр указанного класса; это позволяет создавать экземпляр динамически загруженного класса, если нельзя использовать для него ключевое слово `new`. Заметьте, что данный метод работает только тогда, когда у требуемого класса есть конструктор без аргументов. Метод `newInstance()` класса `java.lang.reflect.Constructor` является более мощным средством для создания экземпляров динамически загруженных классов.

Метод `getName()` возвращает имя класса, а `getSuperclass()` возвращает имя родительского класса. Метод `isInterface()` проверяет, представляет ли объект `Class` интерфейс, а `getInterfaces()` возвращает массив интерфейсов, которые реализованы данным классом. В Java 1.2 и последующих версиях метод `getPackage()` возвращает объект `Package`, представляющий пакет, в котором содержится класс. `getProtectionDomain()` возвращает объект `java.security.ProtectionDomain`, с которым связан данный класс. Различные методы `get-()` и `is-()` возвращают дополнительную информацию

о данном классе; вместе с классами пакета java.lang.reflect они являются частью Java Reflection API.




```
public final class Class implements Serializable {
// Конструктор отсутствует
// Открытые методы класса
   public static Class forName(String className) throws ClassNotFoundException;
1.2 public static Class forName(String name, boolean initialize, ClassLoader loader)
       throws ClassNotFoundException;
// Методы доступа к свойствам (по именам свойств)
1.1 public boolean isArray();                                    // зависит от платформы
1.1 public Class[] getClasses();
   public ClassLoader getClassLoader();
1.1 public Class getComponentType();                             // зависит от платформы
1.1 public java.lang.reflect.Constructor[] getConstructors() throws SecurityException
1.1 public Class[] getDeclaredClasses() throws SecurityException;
1.1 public java.lang.reflect.Constructor[] getDeclaredConstructors() throws SecurityException;
1.1 public java.lang.reflect.Field[] getDeclaredFields() throws SecurityException;
1.1 public java.lang.reflect.Method[] getDeclaredMethods() throws SecurityException;
1.1 public Class getDeclaringClass();                            // зависит от платформы
1.1 public java.lang.reflect.Field[] getFields() throws SecurityException;
   public boolean isInterface();                                 // зависит от платформы
   public Class[] getInterfaces();                               // зависит от платформы
1.1 public java.lang.reflect.Method[] getMethods() throws SecurityException;
1.1 public int getModifiers();                                   // зависит от платформы
   public String getName();                                      // зависит от платформы
1.2 public Package getPackage();
1.1 public boolean isPrimitive();                                // зависит от платформы
1.2 public java.security.ProtectionDomain getProtectionDomain();
1.1 public Object[] getSigners();                                // зависит от платформы
   public Class getSuperclass();                                 // зависит от платформы
// Открытые методы экземпляра
1.4 public boolean desiredAssertionStatus();
1.1 public java.lang.reflect.Constructor getConstructor(Class[] parameterTypes)
       throws NoSuchMethodException, SecurityException;
1.1 public java.lang.reflect.Constructor getDeclaredConstructor(Class[] parameterTypes)
      throws NoSuchMethodException, SecurityException;
1.1 public java.lang.reflect.Field getDeclaredField(String name)
       throws NoSuchFieldException, SecurityException;
1.1 public java.lang.reflect.Method getDeclaredMethod(String name, Class[] parameterTypes)
      throws NoSuchMethodException, SecurityException;
1.1 public java.lang.reflect.Field getField(String name) throws NoSuchFieldException,
      SecurityException;
1.1 public java.lang.reflect.Method getMethod(String name, Class[] parameterTypes)
      throws NoSuchMethodException, SecurityException;
1.1 public java.net.URL getResource(String name);
1.1 public java.io.InputStream getResourceAsStream(String name);
1.1 public boolean isAssignableFrom(Class cls);                   // зависит от платформы
1.1 public boolean isInstance(Object obj);                       // зависит от платформы
   public Object newInstance() throws InstantiationException,IllegalAccessException;
```

```
// Открытые методы, замещающие методы класса Object
   public String toString();
}
```

Передается методами: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

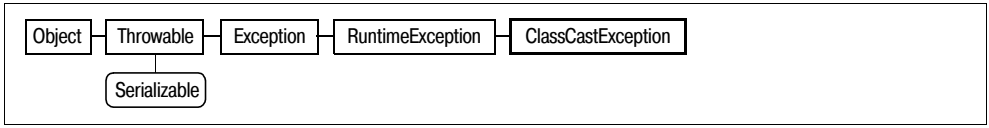
Экземпляры: Полей слишком много, чтобы их перечислить.

ClassCastException

Java 1.0

java.lang ***сериализуемое, непроверяемое***

Данное исключение возникает при приведении объекта к недопустимому типу, если объект не является экземпляром этого типа.




```
public class ClassCastException extends RuntimeException {
// Открытые конструкторы
   public ClassCastException();
   public ClassCastException(String s);
}
```

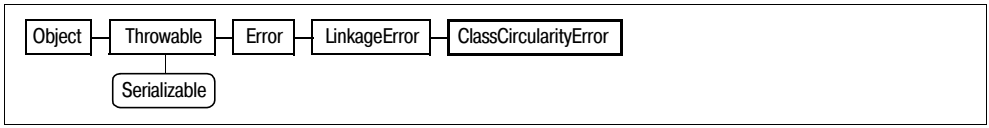
Генерируется методами: `javax.rmi.PortableRemoteObject.narrow()`, `javax.rmi.CORBA.PortableRemoteObjectDelegate.narrow()`, `org.xml.sax.helpers.ParserFactory.makeParser()`

ClassCircularityError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, если при инициализации класса обнаружена циклическая зависимость.



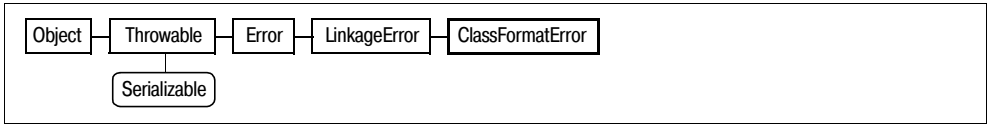

```
public class ClassCircularityError extends LinkageError {
// Открытые конструкторы
   public ClassCircularityError();
   public ClassCircularityError(String s);
}
```

ClassFormatError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, если двоичный файл класса имеет неправильный формат.




```
public class ClassFormatError extends LinkageError {
// Открытые конструкторы
   public ClassFormatError();
   public ClassFormatError(String s);
}
```

Подклассы: UnsupportedClassVersionError

Генерируется методами: ClassLoader.defineClass()

ClassLoader

Java 1.0

java.lang

Этот абстрактный класс является родительским классом объектов, которые могут загружать классы Java в виртуальную машину. С помощью объекта ClassLoader можно динамически загружать класс, вызвав метод loadClass() с полным именем класса в качестве параметра. С помощью методов getResource() и getResourceAsStream() можно получить ресурс, связанный с классом. В приложениях, как правило, не требуется использовать ClassLoader непосредственно. Вместо этого для динамической загрузки классов и ресурсов применяются методы Class.forName() и Class.getResource(). Они используют объект ClassLoader, который загрузил данное приложение.

Чтобы загрузить классы по сети или из источника, не находящегося на заданном пути к классам, нужно задействовать потомок объекта ClassLoader, предназначенный для загрузки данных из этого источника. Класс java.net.URLClassLoader подходит для такой задачи почти во всех приложениях. Создавать потомок ClassLoader приходится очень редко. В таких случаях подкласс должен расширить класс java.security.SecureClassLoader и заменить метод findClass(). Этот метод должен найти последовательность байтов, составляющую указанный класс, а затем передать ее методу defineClass() и возвратить полученный объект типа Class. В Java 1.2 и последующих версиях метод findClass() должен также определить пакет (объект Package), связанный с классом, если он еще не определен. Для этого он может задействовать методы getPackage() и definePackage(). В подклассах ClassLoader также должны быть замещены методы findResource() и findResources(), чтобы сделать доступными открытые методы getResource() и getResources().

В Java 1.4 и последующих версиях можно указать, следует ли включать утверждения (assertions) в классах, загружаемых с помощью ClassLoader. Метод setDefaultAssertionStatus() включает или выключает утверждения для всех загруженных классов. Методы setPackageAssertionStatus() и setClassAssertionStatus() позволяют изменять состояние утверждений для указанного пакета или класса. Наконец, clearAssertionStatus() устанавливает состояние по умолчанию в false и сбрасывает все утверждения для всех указанных пакетов и классов.

```
public abstract class ClassLoader {
// Защищенные конструкторы
   protected ClassLoader();
1.2 protected ClassLoader(ClassLoader parent);
// Открытые методы класса
```

```
1.2 public static ClassLoader getSystemClassLoader();
1.1 public static java.net.URL getSystemResource(String name);
1.1 public static java.io.InputStream getSystemResourceAsStream(String name);
1.2 public static java.util.Enumeration getSystemResources(String name) throws java.io.IOException;
// Открытые методы экземпляра
1.4 public void clearAssertionStatus();                                    // синхронизирован
1.2 public final ClassLoader getParent();
1.1 public java.net.URL getResource(String name);
1.1 public java.io.InputStream getResourceAsStream(String name);
1.2 public final java.util.Enumeration getResources(String name) throws java.io.IOException;
1.1 public Class loadClass(String name) throws ClassNotFoundException;
1.4 public void setClassAssertionStatus(String className, boolean enabled);     // синхронизирован
1.4 public void setDefaultAssertionStatus(boolean enabled);                     // синхронизирован
1.4 public void setPackageAssertionStatus(String packageName, boolean enabled);   // синхронизирован
// Защищенные методы экземпляра
1.1 protected final Class defineClass(String name, byte[] b, int off, int len)
                                                                  throws ClassFormatError;
1.2 protected final Class defineClass(String name, byte[] b, int off, int len,
                    java.security.ProtectionDomain protectionDomain) throws ClassFormatError;
1.2 protected Package definePackage(String name, String specTitle, String specVersion,
     String specVendor, String implTitle, String implVersion, String implVendor, java.net.URL
     sealBase) throws IllegalArgumentException;
1.2 protected Class findClass(String name) throws ClassNotFoundException;
1.2 protected String findLibrary(String libname);                                  // константа
1.1 protected final Class findLoadedClass(String name);                  // зависит от платформы
1.2 protected java.net.URL findResource(String name);                              // константа
1.2 protected java.util.Enumeration findResources(String name) throws java.io.IOException;
   protected final Class findSystemClass(String name) throws ClassNotFoundException;
1.2 protected Package getPackage(String name);
1.2 protected Package[] getPackages();
   protected Class loadClass(String name, boolean resolve) throws ClassNotFoundException;
                                                                                   // синхронизирован
   protected final void resolveClass(Class c);
1.1 protected final void setSigners(Class c, Object[] signers);
// Устаревшие защищенные методы
# protected final Class defineClass(byte[] b, int off, int len) throws ClassFormatError;
}
```

Подклассы: java.security.SecureClassLoader

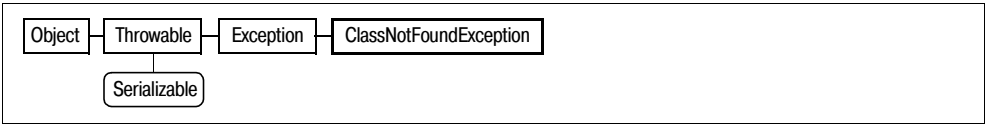
Передается методами: Методов слишком много, чтобы их перечислить.

Возвращается методами: Class.getClassLoader(), ClassLoader.{getParent(), getSystemClassLoader()}, SecurityManager.currentClassLoader(), Thread.getContextClassLoader(), java.rmi.server.RMIClassLoader.getClassLoader(), java.rmi.server.RMIClassLoaderSpi.getClassLoader(), java.security.ProtectionDomain.getClassLoader()

ClassNotFoundException Java 1.0

java.lang ***сериализуемое, проверяемое***

Данное исключение возникает, если не найден класс, который нужно загрузить. Если исключение этого типа вызвано исключениями более низкого уровня, можно запросить эти исключения методом getException() или более новым getCause().




```
public class ClassNotFoundException extends Exception {
// Открытые конструкторы
   public ClassNotFoundException();
   public ClassNotFoundException(String s);
1.2 public ClassNotFoundException(String s, Throwable ex);
// Открытые методы экземпляра
1.2 public Throwable getException();                                    // по умолчанию:null
// Открытые методы, замещающие методы класса Throwable
1.4 public Throwable getCause();                                        // по умолчанию:null
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

Cloneable

Java 1.0

java.lang ***клонируемый***

Данный интерфейс не определяет методов или переменных. Он служит признаком того, что класс, реализующий этот интерфейс, может быть клонирован (скопирован) с помощью метода `clone()` класса `Object`. Применение `clone()` для объекта, который не реализует данный интерфейс (и не замещает `clone()` своей версией этого метода), вызывает исключение `CloneNotSupportedException`.

```
public interface Cloneable {
}
```

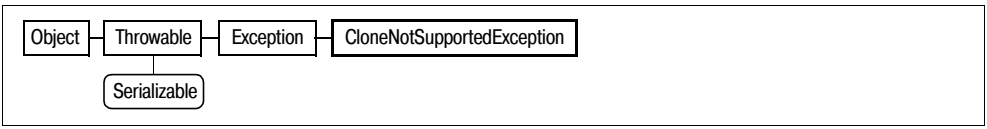
Реализации: Классов слишком много, чтобы их перечислить.

CloneNotSupportedException

Java 1.0

java.lang ***сериализуемое, проверяемое***

Это исключение генерируется, когда метод `clone()` вызывается для объекта, который не реализует интерфейс `Cloneable`.




```
public class CloneNotSupportedException extends Exception {
// Открытые конструкторы
   public CloneNotSupportedException();
   public CloneNotSupportedException(String s);
}
```

Подклассы: `java.rmi.server.ServerCloneException`

Генерируется методами: Методов слишком много, чтобы их перечислить.

Comparable Java 1.2

java.lang ***сравнимый***

В данном интерфейсе определен единственный метод – compareTo(), предназначенный для сравнения двух объектов и определения их относительного порядка, установленного для этого типа объектов. Любой универсальный класс, объекты которого можно упорядочить, должен реализовать этот интерфейс. Тогда в нем будут доступны достаточно мощные методы, такие как java.util.Collections.sort() и java.util.Arrays.binarySearch(). В Java 1.2 многие ключевые классы Java API были модифицированы – в них был реализован этот интерфейс.

Метод compareTo() сравнивает данный объект с объектом, переданным ему в качестве аргумента. Предполагается, что аргумент – того же типа, что и сам объект. Если это не так, вызывается исключение ClassCastException. Если этот объект меньше или равен аргументу или будет стоять перед ним в отсортированном списке, то compareTo() должен возвратить отрицательное число. Если объект больше аргумента или будет стоять в отсортированном списке после него, то этот метод должен возвращать положительное число. Если compareTo() возвращает 0, значит метод equals() должен возвратить true. В противном случае такие объекты Comparable не могут быть задействованы в классах java.util.TreeSet и java.util.TreeMap.

В описании java.util.Comparator представлено, как определить правило упорядочения объектов, которые не реализуют интерфейс Comparable, и как изменить правило, определенное в классе Comparable.

```
public interface Comparable {
// Открытые методы экземпляра
   public abstract int compareTo(Object o);
}
```

Реализации: Классов слишком много, чтобы их перечислить.

Передается методам: javax.imageio.metadata.IIOMetadataFormatImpl.addObjectValue(), javax.swing.SpinnerDateModel.{setEnd(), setStart(), SpinnerDateModel()}, javax.swing.SpinnerNumberModel.{setMaximum(), setMinimum(), SpinnerNumberModel()}, javax.swing.text.InternationalFormatter.{setMaximum(), setMinimum()}

Возвращается методами: javax.imageio.metadata.IIOMetadataFormat.{getObjectMaxValue(), getObjectMinValue()}, javax.imageio.metadata.IIOMetadataFormatImpl.{getObjectMaxValue(), getObjectMinValue()}, javax.swing.SpinnerDateModel.{getEnd(), getStart()}, javax.swing.SpinnerNumberModel.{getMaximum(), getMinimum()}, javax.swing.text.InternationalFormatter.{getMaximum(), getMinimum()}

Compiler Java 1.0

java.lang

Статические методы этого класса предоставляют интерфейс с динамическим компилятором Java (JIT), преобразующим байт-код в машинный код. Этот компилятор используется интерпретатором Java. Если ВМ не использует JIT, эти методы не производят никаких действий. Метод compileClass() посылает запрос на компиляцию всех классов с указанным именем. Если компиляция прошла успешно, возвращается true. В противном случае, а также, если компилятор JIT не доступен в системе, возвращается false. Методы enable() и disable() соответственно включают и выключают динамическую компиляцию. Метод command() запрашивает у компилятора выполне-

ние специальных операций; это дает возможность производителю ВМ реализовывать нестандартные расширения. Стандартные операции не определены.

```
public final class Compiler {
// Конструктор отсутствует
// Открытые методы класса
   public static Object command(Object any);                    // зависит от платформы
   public static boolean compileClass(Class clazz);             // зависит от платформы
   public static boolean compileClasses(String string);         // зависит от платформы
   public static void disable();                                // зависит от платформы
   public static void enable();                                 // зависит от платформы
}
```

Double

Java 1.0

java.lang — ***сериализуемый, сравнимый***

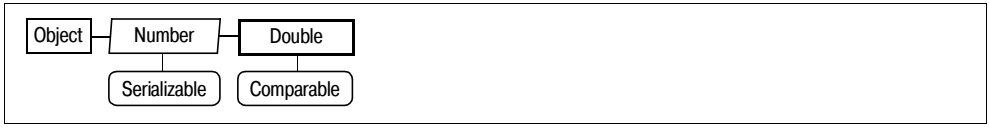
Этот класс представляет собой неизменяемую обертку для простого типа `double`. Метод `doubleValue()` возвращает значение типа `double`, хранящееся в объекте `Double`. Кроме того, есть еще несколько методов, возвращающих значение из этого объекта в виде других простых типов. Данные методы замещают методы класса `Number` и заканчиваются суффиксом «`Value`».

В этом классе есть несколько полезных констант и статических методов для проверки значений типа `double`. `MIN_VALUE` и `MAX_VALUE` хранят минимальное (ближайшее к нулю) и максимальное значения типа `double`, которые могут присутствовать в этом объекте. `POSITIVE_INFINITY` и `NEGATIVE_INFINITY` хранят значения для бесконечностей, а `NaN` – специальное значение «нечисло» типа `double`. Метод `isInfinite()` проверяет, является ли значение, хранящееся в объекте `Double`, бесконечным. Аналогично `isNaN()` проверяет, хранится ли в нем «нечисло»; эту проверку нельзя выполнить простым сравнением, потому что константа `NaN` не может быть равна ни одному другому значению, в том числе самой себе.

Статический метод `parseDouble()` преобразует `String` в `double`. Статический метод `valueOf()` преобразует `String` в `Double` и, по сути, эквивалентен конструктору `Double()` с аргументом типа `String`. Статический метод и метод экземпляра `toString()` преобразуют `double` или `Double` в `String`. Более гибкий механизм преобразования и форматирования реализован в классе `java.text.NumberFormat`.

Метод `compareTo()` дает возможность сравнивать объекты `Double` и применяется при их упорядочении и сортировке. Статический метод `compare()` выполняет те же действия. Он возвращает те же значения, что и `Comparable.compareTo()`, но работает с простыми типами, а не с объектами, и применяется при упорядочении и сортировке массивов типа `double`.

Методы `doubleToLongBits()`, `doubleToRawLongBits()` и `longBitsToDouble()` позволяют работать непосредственно с битовым представлением типа `double`, определенным в стандарте IEEE 754. В обычных приложениях они применяются очень редко.




```
public final class Double extends Number implements Comparable {
// Открытые конструкторы
```

```
   public Double(String s) throws NumberFormatException;
   public Double(double value);
// Открытые константы
   public static final double MAX_VALUE;                          // =1.7976931348623157E308
   public static final double MIN_VALUE;                          // =4.9E-324
   public static final double NaN;                                // =NaN
   public static final double NEGATIVE_INFINITY;                  // =-Infinity
   public static final double POSITIVE_INFINITY;                  // =Infinity
1.1 public static final Class TYPE;
// Открытые методы класса
1.4 public static int compare(double d1, double d2);
   public static long doubleToLongBits(double value);             // зависит от платформы
1.3 public static long doubleToRawLongBits(double value);         // зависит от платформы
   public static boolean isInfinite(double v);
   public static boolean isNaN(double v);
   public static double longBitsToDouble(long bits);              // зависит от платформы
1.2 public static double parseDouble(String s) throws NumberFormatException;
   public static String toString(double d);
   public static Double valueOf(String s) throws NumberFormatException;
// Открытые методы экземпляра
1.2 public int compareTo(Double anotherDouble);
   public boolean isInfinite();
   public boolean isNaN();
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы Number
1.1 public byte byteValue();
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
1.1 public short shortValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: `Double.compareTo()`

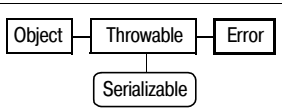
Возвращается методами: `Double.valueOf()`

Error

Java 1.0

java.lang ***сериализуемый, ошибка***

Этот класс является вершиной иерархии ошибок в Java. Подклассы `Error`, в отличие от подклассов `Exception`, не должны «захватываться». Обычно они вызывают завершение программы. Подклассы `Error` не нужно указывать в операторе `throws` при определении метода. Этот класс наследует методы класса `Throwable`, но не имеет собственных методов. В каждом конструкторе класса `Error` вызывается только соответствующий конструктор `Throwable()`. См. также `Throwable`.




```
public class Error extends Throwable {
// Открытые конструкторы
   public Error();
1.4 public Error(Throwable cause);
   public Error(String message);
1.4 public Error(String message, Throwable cause);
}
```

Подклассы: java.awt.AWTError, AssertionError, LinkageError, ThreadDeath, VirtualMachineError, java.nio.charset.CoderMalfunctionError, javax.xml.parsers.FactoryConfigurationError, javax.xml.transform.TransformerFactoryConfigurationError

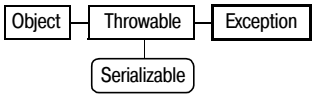
Передается методам: java.rmi.ServerError.ServerError()

Exception

Java 1.0

java.lang ***сериализуемое, проверяемое***

Этот класс является вершиной иерархии исключений в Java. Появление исключения свидетельствует о возникновении особого состояния, которое нужно специально обработать, чтобы избежать аварийного завершения программы. Исключения можно «захватывать» и обрабатывать. Исключение, которое не является потомком RuntimeException, должно быть объявлено в операторе throws любого метода, который может его вызывать. Этот класс наследует методы Throwable, но не определяет собственных методов. Все его конструкторы содержат только вызов соответствующих конструкторов Throwable(). См. также Throwable.




```
public class Exception extends Throwable {
// Открытые конструкторы
   public Exception();
1.4 public Exception(Throwable cause);
   public Exception(String message);
1.4 public Exception(String message, Throwable cause);
}
```

Подклассы: Классов слишком много, чтобы их перечислить.

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.awt.event.InvocationEvent.getException(), java.security.PrivilegedActionException.getException(), javax.xml.parsers.FactoryConfigurationError.getException(), javax.xml.transform.TransformerFactoryConfigurationError.getException(), org.omg.CORBA.Environment.exception(), org.xml.sax.SAXException.getException()

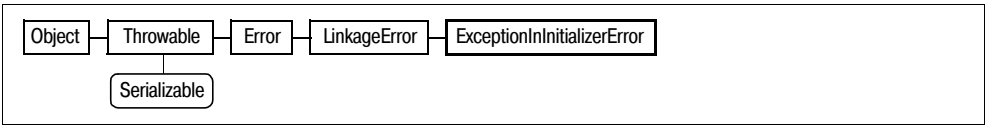
Генерируется методами: java.awt.im.spi.InputMethodDescriptor.createInputMethod(), java.beans.Expression.getValue(), java.beans.Statement.execute(), java.rmi.server.RemoteCall.executeCall(), java.rmi.server.RemoteRef.invoke(), java.rmi.server.Skeleton.dispatch(), java.security.PrivilegedExceptionAction.run(), javax.naming.spi.DirectoryManager.getObjectInstance(), javax.naming.spi.DirObjectFactory.getObjectInstance(), javax.naming.spi.NamingManager.getObjectInstance(), javax.naming.spi.ObjectFactory.getObjectInstance()

Экземпляры: java.io.WriteAbortedException.detail, java.rmi.server.ServerCloneException.detail

ExceptionInInitializerError Java 1.1

java.lang ***сериализуемый, ошибка***

Эта ошибка вызывается виртуальной машиной Java, когда возникает исключение в статическом инициализаторе класса. Чтобы получить объект Throwable, который был сгенерирован инициализатором, применяйте метод getException(). В Java 1.4 и последующих версиях этот метод был заменен более универсальным методом getCause() класса Throwable.

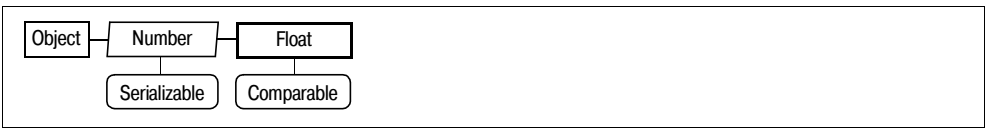



```
public class ExceptionInInitializerError extends LinkageError {
// Открытые конструкторы
   public ExceptionInInitializerError();
   public ExceptionInInitializerError(String s);
   public ExceptionInInitializerError(Throwable thrown);
// Открытые методы экземпляра
   public Throwable getException();                          // по умолчанию: null
// Открытые методы, замещающие методы класса Throwable
1.4 public Throwable getCause();                             // по умолчанию: null
}
```

Float Java 1.0

java.lang ***сериализуемый, сравнимый***

Этот класс является неизменяемой оберткой для простого типа float. Метод floatValue() возвращает число с плавающей точкой, хранящееся в объекте Float. Кроме того, есть методы, возвращающие значение объекта Float в виде других простых типов. Этот класс очень похож на Double и имеет такой же набор методов и констант. Подробности представлены в описании Double.



```
public final class Float extends Number
       implements Comparable {
// Открытые конструкторы
   public Float(String s) throws NumberFormatException;
   public Float(float value);
   public Float(double value);
// Открытые константы
   public static final float MAX_VALUE;                                    // =3.4028235E38
   public static final float MIN_VALUE;                                    // =1.4E-45
   public static final float NaN;                                          // =NaN
   public static final float NEGATIVE_INFINITY;                            // =-Infinity
   public static final float POSITIVE_INFINITY;                            // =Infinity
1.1 public static final Class TYPE;
// Открытые методы класса
1.4 public static int compare(float f1, float f2);
   public static int floatToIntBits(float value);                          // зависит от платформы
1.3 public static int floatToRawIntBits(float value);                       // зависит от платформы
   public static float intBitsToFloat(int bits);                           // зависит от платформы
   public static boolean isInfinite(float v);
   public static boolean isNaN(float v);
1.2 public static float parseFloat(String s) throws NumberFormatException;
   public static String toString(float f);
   public static Float valueOf(String s) throws NumberFormatException;
// Открытые методы экземпляра
1.2 public int compareTo(Float anotherFloat);
   public boolean isInfinite();
   public boolean isNaN();
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы Number
1.1 public byte byteValue();
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
1.1 public short shortValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: `Float.compareTo()`

Возвращается методами: `Float.valueOf()`

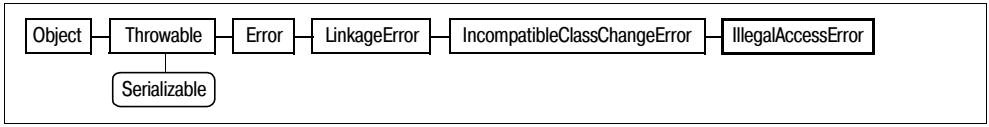
Экземпляры: Полей слишком много, чтобы их перечислить.

IllegalAccessError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает при попытке использования недоступного класса, метода или поля.

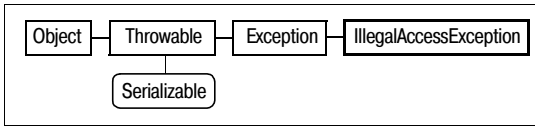



```
public class IllegalAccessError extends IncompatibleClassChangeError {
// Открытые конструкторы
   public IllegalAccessError();
   public IllegalAccessError(String s);
}
```

IllegalAccessException Java 1.0

java.lang ***сериализуемое, проверяемое***

Это исключение возникает, если класс или инициализатор не доступен. Оно вызывается методом `Class.newInstance()`.



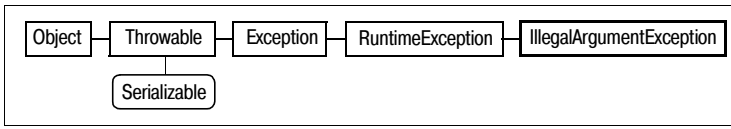

```
public class IllegalAccessException extends Exception {
// Открытые конструкторы
   public IllegalAccessException();
   public IllegalAccessException(String s);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

IllegalArgumentException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает при недопустимом аргументе метода. См. описание его подклассов `IllegalThreadStateException` и `NumberFormatException`.




```
public class IllegalArgumentException extends RuntimeException {
// Открытые конструкторы
   public IllegalArgumentException();
   public IllegalArgumentException(String s);
}
```

Подклассы: `IllegalThreadStateException`, `NumberFormatException`, `java.nio.channels.IllegalSelectorException`, `java.nio.channels.UnresolvedAddressException`, `java.nio.channels.UnsupportedAddressTypeException`, `java.nio.charset.IllegalCharsetNameException`,

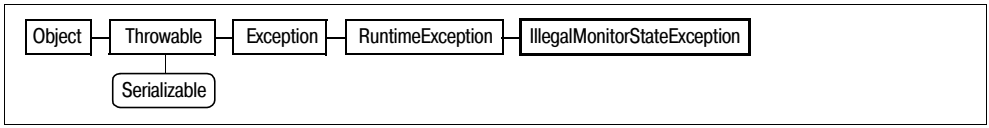
java.nio.charset.UnsupportedCharsetException, java.security.InvalidParameterException, java.util.regex.PatternSyntaxException

Генерируется методами: Методов слишком много, чтобы их перечислить.

IllegalMonitorStateException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает при недопустимом состоянии монитора синхронизации. Оно вызывается методами `notify()` и `wait()` класса `Object`, используемыми для синхронизации потоков.

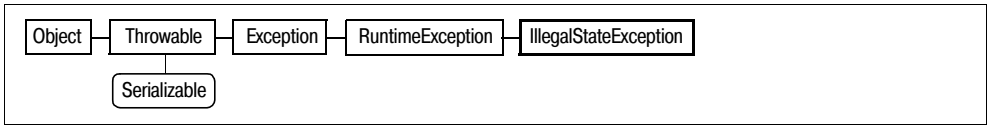



```
public class IllegalMonitorStateException extends RuntimeException {
// Открытые конструкторы
   public IllegalMonitorStateException();
   public IllegalMonitorStateException(String s);
}
```

IllegalStateException Java 1.1

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает, если вызван метод объекта, который в данный момент находится в состоянии, не подходящем для выполнения требуемой операции.




```
public class IllegalStateException extends RuntimeException {
// Открытые конструкторы
   public IllegalStateException();
   public IllegalStateException(String s);
}
```

Подклассы:

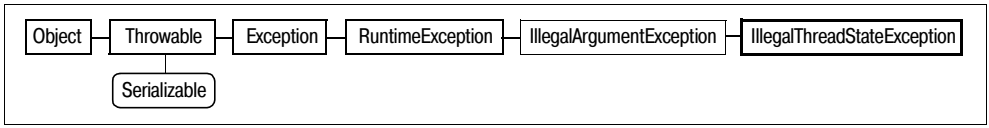
java.awt.IllegalComponentStateException, java.awt.dnd.InvalidDnDOperationException, java.nio.InvalidMarkException, java.nio.channels.AlreadyConnectedException, java.nio.channels.CancelledKeyException, java.nio.channels.ClosedSelectorException, java.nio.channels.ConnectionPendingException, java.nio.channels.IllegalBlockingModeException, java.nio.channels.NoConnectionPendingException, java.nio.channels.NonReadableChannelException, java.nio.channels.NonWritableChannelException, java.nio.channels.NotYetBoundException, java.nio.channels.NotYetConnectedException, java.nio.channels.OverlappingFileLockException

Генерируется методами: Методов слишком много, чтобы их перечислить.

IllegalThreadStateException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает, когда поток не в состоянии успешно выполнить операцию.

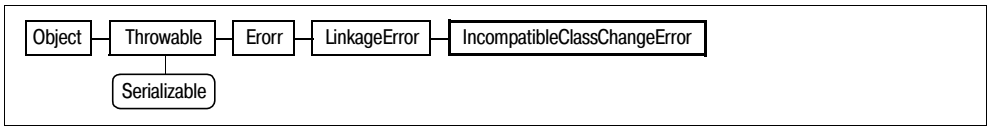



```
public class IllegalThreadStateException extends IllegalArgumentException {
// Открытые конструкторы
   public IllegalThreadStateException();
   public IllegalThreadStateException(String s);
}
```

IncompatibleClassChangeError Java 1.0

java.lang ***сериализуемый, ошибка***

Этот класс является родительским классом для группы ошибок такого типа, которые возникают при недопустимом использовании существующего класса.



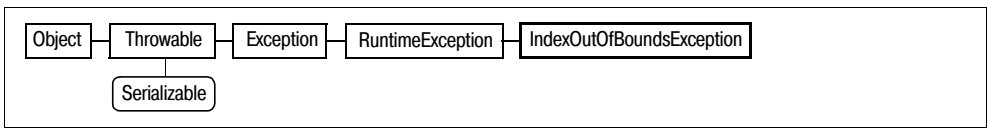

```
public class IncompatibleClassChangeError extends LinkageError {
// Открытые конструкторы
   public IncompatibleClassChangeError();
   public IncompatibleClassChangeError(String s);
}
```

Подклассы: AbstractMethodError, IllegalAccessError, InstantiationError, NoSuchFieldError, NoSuchMethodError

IndexOutOfBoundsException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает, если индекс находится за пределами допустимых значений. См. также описание подклассов ArrayIndexOutOfBoundsException и StringIndexOutOfBoundsException.




```
public class IndexOutOfBoundsException extends RuntimeException {
// Открытые конструкторы
   public IndexOutOfBoundsException();
```

```
   public IndexOutOfBoundsException(String s);
}
```

Подклассы: ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException

Генерируется методами: java.awt.Toolkit.createCustomCursor(), java.awt.print.Book.{getPageFormat(), getPrintable(), setPage()}, java.awt.print.Pageable.{getPageFormat(), getPrintable()}

InheritableThreadLocal

Java 1.2

java.lang

Этот класс содержит локальное значение потока, наследуемое дочерними потоками. Описание таких значений представлено в описании класса ThreadLocal. Заметьте, что наследование (inheritance) в имени этого класса означает именно наследование в потоках, а не в классах.

Понять принцип работы этого класса поможет следующий пример. Пусть в приложении определен объект InheritableThreadLocal, а какой-нибудь родительский поток имеет локальное значение, хранящееся в этом объекте. Когда поток создает новый поток (дочерний), объект InheritableThreadLocal автоматически обновляется, поэтому новый поток имеет такое же локальное значение, как и родительский поток. Заметьте, что значения, связанные с дочерним и родительским потоками, независимы. Если дочерний поток изменяет свое значение в результате вызова метода set() объекта InheritableThreadLocal, то значение, связанное с родительским потоком, меняться не будет.

По умолчанию дочерний поток наследует значение родительского потока, не изменяя его. Но можно заменить метод childValue() так, чтобы дочернему потоку передавалась та или иная функция от значения родительского потока.

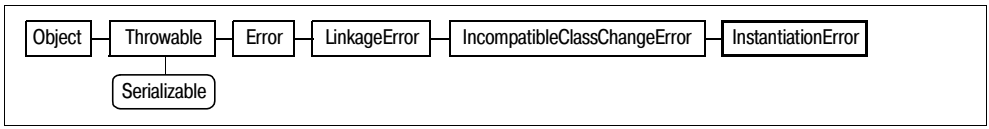

```
public class InheritableThreadLocal extends ThreadLocal {
// Открытые конструкторы
   public InheritableThreadLocal();
// Защищенные методы экземпляра
   protected Object childValue(Object parentValue);
}
```

InstantiationError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает при попытке создать экземпляр интерфейса или абстрактного класса.




```
public class InstantiationError extends IncompatibleClassChangeError {
// Открытые конструкторы
```

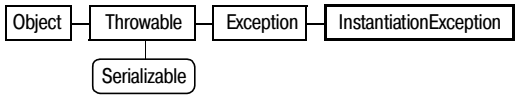
```
   public InstantiationError();
   public InstantiationError(String s);
}
```

InstantiationException

Java 1.0

java.lang ***сериализуемое, проверяемое***

Это исключение возникает при попытке создать экземпляр интерфейса или абстрактного класса.




```
public class InstantiationException extends Exception {
// Открытые конструкторы
   public InstantiationException();
   public InstantiationException(String s);
}
```

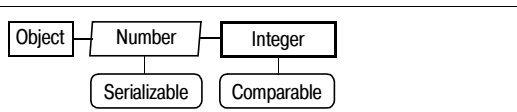
Генерируется методами: `Class.newInstance()`, `java.lang.reflect.Constructor.newInstance()`, `javax.swing.UIManager.setLookAndFeel()`, `org.xml.sax.helpers.ParserFactory.makeParser()`

Integer

Java 1.0

java.lang ***сериализуемый, сравнимый***

Этот класс является неизменяемой оберткой для простого типа данных `int`. Этот класс также содержит полезные константы для минимального и максимального значений и методы для преобразования к другим типам. Методы `parseInt()` и `valueOf()` преобразуют строку в тип `int` или в `Integer` соответственно. Каждый из них может принимать аргумент `radix`, определяющий основание системы счисления исходного значения. Метод `decode()` также преобразует `String` в `Integer`. Он предполагает, что число шестнадцатеричное, если строка начинается с «0X» или «0x»; восьмеричное, если строка начинается с «0»; десятичное в остальных случаях. `toString()` производит обратное преобразование, а статический вариант этого метода в качестве аргумента получает основание системы счисления. `toBinaryString()`, `toOctalString()` и `toHexString()` преобразуют `int` в строку, используя системы счисления с основаниями 2, 8 и 16 соответственно. Эти методы рассматривают целочисленное значение как беззнаковое. Другие методы возвращают значение объекта `Integer` в виде значений простых типов. Наконец, методы `getInteger()` возвращают целочисленное значение указанного свойства из списка системных свойств или его значение по умолчанию.




```
public final class Integer extends Number
       implements Comparable {
// Открытые конструкторы
```

```
   public Integer(int value);
   public Integer(String s) throws NumberFormatException;
// Открытые константы
   public static final int MAX_VALUE;                                    // =2147483647
   public static final int MIN_VALUE;                                    // =-2147483648
1.1 public static final Class TYPE;
// Открытые методы класса
1.1 public static Integer decode(String nm) throws NumberFormatException;
   public static Integer getInteger(String nm);
   public static Integer getInteger(String nm, int val);
   public static Integer getInteger(String nm, Integer val);
   public static int parseInt(String s) throws NumberFormatException;
   public static int parseInt(String s, int radix) throws NumberFormatException;
   public static String toBinaryString(int i);
   public static String toHexString(int i);
   public static String toOctalString(int i);
   public static String toString(int i);
   public static String toString(int i, int radix);
   public static Integer valueOf(String s) throws NumberFormatException;
   public static Integer valueOf(String s, int radix) throws NumberFormatException;
// Открытые методы экземпляра
1.2 public int compareTo(Integer anotherInteger);
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы Number
1.1 public byte byteValue();
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
1.1 public short shortValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: Integer.{compareTo(), getInteger()}, javax.swing.JInternalFrame.setLayer()

Возвращается методами: Integer.{decode(), getInteger(), valueOf()}, javax.swing.JLayeredPane.getObjectForLayer()

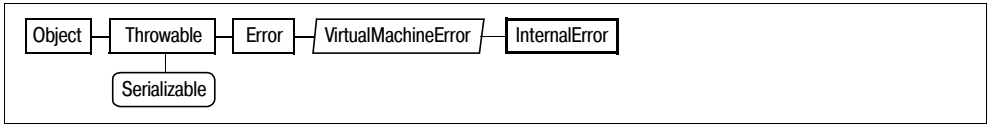
Экземпляры: java.awt.font.TextAttribute.{SUPERSCRIPT_SUB, SUPERSCRIPT_SUPER, UNDERLINE_LOW_DASHED, UNDERLINE_LOW_DOTTED, UNDERLINE_LOW_GRAY, UNDERLINE_LOW_ONE_PIXEL, UNDERLINE_LOW_TWO_PIXEL, UNDERLINE_ON}, javax.swing.JLayeredPane.{DEFAULT_LAYER, DRAG_LAYER, FRAME_CONTENT_LAYER, MODAL_LAYER, PALETTE_LAYER, POPUP_LAYER}

InternalError

Java 1.0

java.lang ***сериализуемый, ошибка***

Возникает при появлении внутренней ошибки в работе интерпретатора Java.



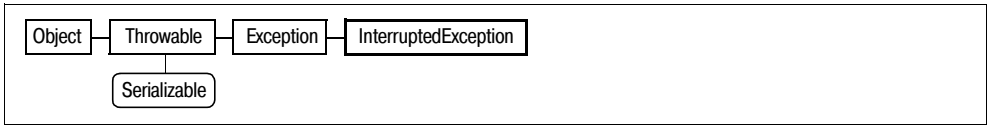

```
public class InternalError extends VirtualMachineError {
// Открытые конструкторы
   public InternalError();
   public InternalError(String s);
}
```

InterruptedException

Java 1.0

java.lang ***сериализуемое, проверяемое***

Это исключение возникает при прерывании потока.




```
public class InterruptedException extends Exception {
// Открытые конструкторы
   public InterruptedException();
   public InterruptedException(String s);
}
```

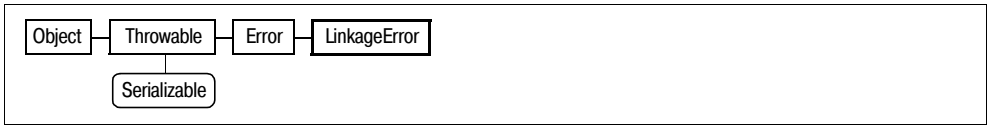
Генерируется методами: Методов слишком много, чтобы их перечислить.

LinkageError

Java 1.0

java.lang ***сериализуемый, ошибка***

Этот класс является родительским классом для группы ошибок. Они сообщают о возникновении осложнений при компоновке класса или установлении зависимостей между классами.




```
public class LinkageError extends Error {
// Открытые конструкторы
   public LinkageError();
   public LinkageError(String s);
}
```

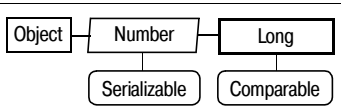
Подклассы: ClassCircularityError, ClassFormatError, ExceptionInInitializerError, IncompatibleClassChangeError, NoClassDefFoundError, UnsatisfiedLinkError, VerifyError

Long

Java 1.0

java.lang ***сериализуемый, сравнимый***

Этот класс является неизменяемой оберткой для простого типа `long`. В нем содержатся полезные константы для минимального и максимального значений и методы для преобразования к другим типам. Методы `parseLong()` и `valueOf()` преобразуют строку к типам `long` или `Long` соответственно. Они могут принимать аргумент `radix`, определяющий основание системы счисления исходного значения. `toString()` выполняет обратное преобразование и может принимать аргумент `radix`. Методы `toBinaryString()`, `toOctalString()` и `toHexString()` конвертируют `long` в строку, используя системы счисления с основаниями 2, 8 или 16 соответственно. Эти методы рассматривают `long` как беззнаковое целое. Другие методы возвращают значение, хранящееся в объекте `Long` в виде значения простого типа. Наконец, метод `getLong()` возвращает значение указанного свойства в виде длинного целого или его значение по умолчанию.




```
public final class Long extends Number implements Comparable {
// Открытые конструкторы
   public Long(long value);
   public Long(String s) throws NumberFormatException;
// Открытые константы
   public static final long MAX_VALUE;                          // =9223372036854775807
   public static final long MIN_VALUE;                          // =-9223372036854775808
1.1 public static final Class TYPE;
// Открытые методы класса
1.2 public static Long decode(String nm) throws NumberFormatException;
   public static Long getLong(String nm);
   public static Long getLong(String nm, long val);
   public static Long getLong(String nm, Long val);
   public static long parseLong(String s) throws NumberFormatException;
   public static long parseLong(String s, int radix) throws NumberFormatException;
   public static String toBinaryString(long i);
   public static String toHexString(long i);
   public static String toOctalString(long i);
   public static String toString(long i);
   public static String toString(long i, int radix);
   public static Long valueOf(String s) throws NumberFormatException;
   public static Long valueOf(String s, int radix) throws NumberFormatException;
// Открытые методы экземпляра
1.2 public int compareTo(Long anotherLong);
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы Number
1.1 public byte byteValue();
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
1.1 public short shortValue();
// Открытые методы, замещающие методы класса Object
```

```
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Передается методам: Long.{compareTo(), getLong()}

Возвращается методами: Long.{decode(), getLong(), valueOf()}

Math Java 1.0

java.lang

Этот класс определяет математические константы e и π и статические методы для тригонометрических функций, возведения в степень и для других операций над числами с плавающей точкой. Они эквивалентны функциям из файла *<math.h>* языка C. В этом классе есть методы для вычисления минимального и максимального значений и для генерации псевдослучайных чисел.

Большинство методов класса Math работают с числами типов float и double. Обратите внимание, что эти значения могут быть приблизительными. Поскольку различные реализации Java используют механизм вычислений с плавающей точкой базовой платформы, не стоит ожидать, что методы класса Math будут возвращать одни и те же значения на всех платформах. Другими словами, результаты одних и тех же операций в разных реализациях могут отличаться младшими значащими битами. В Java 1.3 для обеспечения строгой независимости приложения от платформы можно применять класс StrictMath.

```
public final class Math {
// Конструктор отсутствует
// Открытые константы
    public static final double E;                                    // =2.718281828459045
    public static final double PI;                                   // =3.141592653589793
// Открытые методы класса
    public static int abs(int a);                                    // strictfp
    public static long abs(long a);                                  // strictfp
    public static float abs(float a);                                // strictfp
    public static double abs(double a);                              // strictfp
    public static double acos(double a);                             // strictfp
    public static double asin(double a);                             // strictfp
    public static double atan(double a);                             // strictfp
    public static double atan2(double y, double x);                  // strictfp
    public static double ceil(double a);                             // strictfp
    public static double cos(double a);                              // strictfp
    public static double exp(double a);                              // strictfp
    public static double floor(double a);                            // strictfp
    public static double IEEEremainder(double f1, double f2);       // strictfp
    public static double log(double a);                              // strictfp
    public static int max(int a, int b);                             // strictfp
    public static long max(long a, long b);                          // strictfp
    public static float max(float a, float b);                       // strictfp
    public static double max(double a, double b);                    // strictfp
    public static int min(int a, int b);                             // strictfp
    public static long min(long a, long b);                          // strictfp
    public static float min(float a, float b);                       // strictfp
    public static double min(double a, double b);                    // strictfp
    public static double pow(double a, double b);                    // strictfp
```

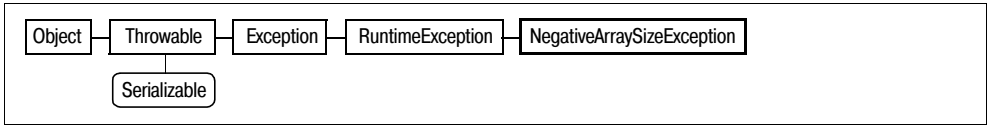
```
   public static double random();                                    // strictfp
   public static double rint(double a);                              // strictfp
   public static int round(float a);                                 // strictfp
   public static long round(double a);                               // strictfp
   public static double sin(double a);                               // strictfp
   public static double sqrt(double a);                              // strictfp
   public static double tan(double a);                               // strictfp
1.2 public static double toDegrees(double angrad);                   // strictfp
1.2 public static double toRadians(double angdeg);                   // strictfp
}
```

NegativeArraySizeException

Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает при попытке разместить в памяти массив с количеством элементов, меньшим нуля.




```
public class NegativeArraySizeException extends RuntimeException {
// Открытые конструкторы
   public NegativeArraySizeException();
   public NegativeArraySizeException(String s);
}
```

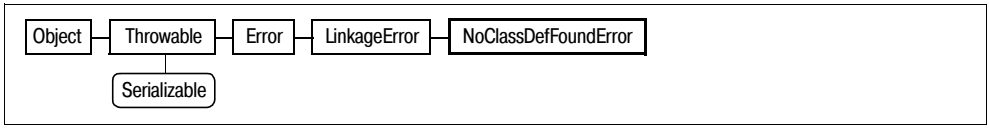
Генерируется методами: `java.lang.reflect.Array.newInstance()`

NoClassDefFoundError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, если не найдено определение указанного класса.



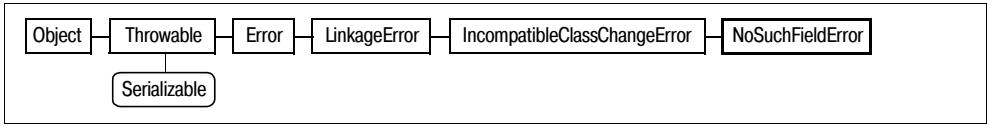

```
public class NoClassDefFoundError extends LinkageError {
// Открытые конструкторы
   public NoClassDefFoundError();
   public NoClassDefFoundError(String s);
}
```

NoSuchFieldError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, если в указанном классе нет данного поля.

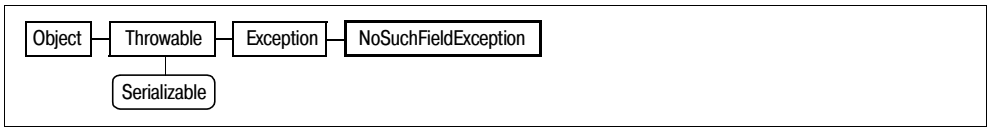



```
public class NoSuchFieldError extends IncompatibleClassChangeError {
// Открытые конструкторы
   public NoSuchFieldError();
   public NoSuchFieldError(String s);
}
```

NoSuchFieldException Java 1.1

java.lang ***сериализуемое, проверяемое***

Это исключение сигнализирует, что указанное поле не найдено в заданном классе.



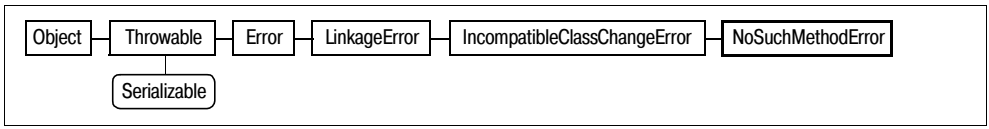

```
public class NoSuchFieldException extends Exception {
// Открытые конструкторы
   public NoSuchFieldException();
   public NoSuchFieldException(String s);
}
```

Генерируется методами: `Class.{getDeclaredField(), getField()}`

NoSuchMethodError Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, если указанный метод не найден.

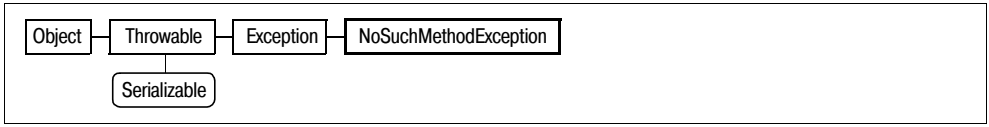



```
public class NoSuchMethodError extends IncompatibleClassChangeError {
// Открытые конструкторы
   public NoSuchMethodError();
   public NoSuchMethodError(String s);
}
```

NoSuchMethodException Java 1.0

java.lang ***сериализуемое, проверяемое***

Это исключение возникает, если в указанном классе нет данного метода.




```
public class NoSuchMethodException extends Exception {
// Открытые конструкторы
   public NoSuchMethodException();
   public NoSuchMethodException(String s);
}
```

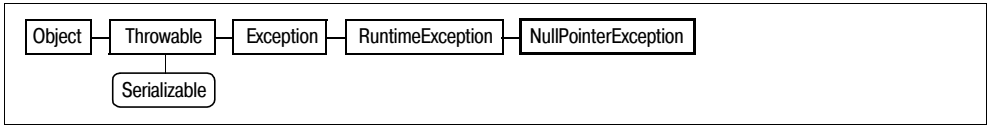
Генерируется методами: `Class.{getConstructor(), getDeclaredConstructor(), getDeclaredMethod(), getMethod()}`

NullPointerException

Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает при попытке обратиться к полю или вызвать метод объекта, который имеет значение null.




```
public class NullPointerException extends RuntimeException {
// Открытые конструкторы
   public NullPointerException();
   public NullPointerException(String s);
}
```

Генерируется методами: `java.awt.print.PrinterJob.setPageable(), org.xml.sax.helpers.ParserFactory.makeParser()`

Number

Java 1.0

java.lang ***сериализуемый***

Этот абстрактный класс является родительским классом для классов `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. В нем определены функции для преобразования, реализованные в этих классах.


```
public abstract class Number implements Serializable {
// Открытые конструкторы
   public Number();
// Открытые методы экземпляра
1.1 public byte byteValue();
   public abstract double doubleValue();
   public abstract float floatValue();
   public abstract int intValue();
```

```
   public abstract long longValue();
1.1 public short shortValue();
}
```

Подклассы: Byte, Double, Float, Integer, Long, Short, java.math.BigDecimal, java.math.BigInteger

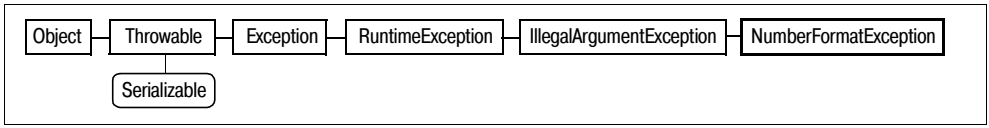
Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

NumberFormatException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение свидетельствует о недопустимом формате числа.




```
public class NumberFormatException extends IllegalArgumentException {
// Открытые конструкторы
   public NumberFormatException();
   public NumberFormatException(String s);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

Object Java 1.0

java.lang

Это базовый класс в Java. Все классы являются потомками Object, поэтому во всех объектах можно вызывать открытые и защищенные методы этого класса. Если класс реализует интерфейс Cloneable, то метод clone() выполняет побайтное копирование объекта (Object) этого класса. getClass() возвращает объект типа Class, который связан с любым объектом Object. Методы notify(), notifyAll() и wait() применяются для синхронизации потоков в данном объекте.

Некоторые методы класса Object должны замещаться в его подклассах. Например, подкласс должен переопределить метод toString(), чтобы его объект можно было использовать в качестве операнда для оператора конкатенации строк и с методом PrintWriter.println(). Кроме того, определение метода toString() для всех объектов облегчает отладку.

Реализация по умолчанию метода equals() использует оператор == для проверки тождественности ссылок. Многие потомки класса Object замещают этот метод, чтобы сравнивать отдельные поля объектов, а не их ссылки (то есть производится проверка эквивалентности объектов, а не равенства ссылок). Некоторые классы, особенно классы, замещающие equals(), могут замещать метод hashCode(), вычисляющий хешкод, который применяется для организации хранения экземпляров в структуре данных Hashtable.

Класс, который кроме памяти использует другие системные ресурсы (например, дескрипторы файлов или графические контексты оконного интерфейса), должен заме-

щать метод finalize(), чтобы освобождать эти ресурсы, когда ссылок на объект больше нет и должна быть произведена сборка мусора.

```
public class Object {
// Открытые конструкторы
   public Object();                                                    // пустой
// Открытые методы экземпляра
   public boolean equals(Object obj);
   public final Class getClass();                                      // зависит от платформы
   public int hashCode();                                              // зависит от платформы
   public final void notify();                                         // зависит от платформы
   public final void notifyAll();                                      // зависит от платформы
   public String toString();
   public final void wait() throws InterruptedException;
   public final void wait(long timeout) throws InterruptedException;   // зависит от платформы
   public final void wait(long timeout, int nanos) throws InterruptedException;
// Защищенные методы экземпляра
   protected Object clone() throws CloneNotSupportedException;         // зависит от платформы
   protected void finalize() throws Throwable;                         // пустой
}
```

Подклассы: Классов слишком много, чтобы их перечислить.

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

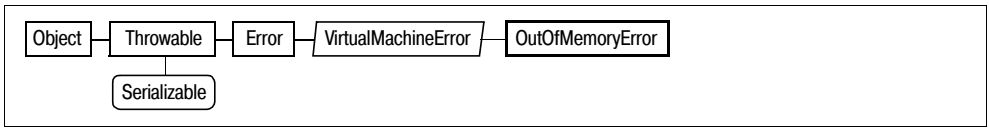
Экземпляры: Полей слишком много, чтобы их перечислить.

OutOfMemoryError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, когда интерпретатор испытывает нехватку памяти, а сборщик мусора не может освободить ни одного блока памяти.




```
public class OutOfMemoryError extends VirtualMachineError {
// Открытые конструкторы
   public OutOfMemoryError();
   public OutOfMemoryError(String s);
}
```

Package

Java 1.2

java.lang

Этот класс представляет пакет Java. Объект Package для данного класса можно получить с помощью вызова метода getPackage() объекта Class. Статический метод Package.getPackage() возвращает объект Package для указанного пакета, если этот пакет был загружен текущим загрузчиком классов. Аналогично, статический метод Package.getPackages() возвращает все объекты, загруженные текущим загрузчиком классов. Обратите внимание, что объект Package не определен, если ни один класс из соот-

ветствующего пакета не был загружен. Несмотря на то что можно получить объект Package для указанного класса, массив классов (объектов Class) из данного пакета получить нельзя.

Если классы, составляющие пакет, находятся в файле JAR, манифест которого содержит соответствующие атрибуты, то с помощью объекта Package можно получить заглавие, имя автора и версию спецификации и реализации пакета; все шесть значений являются строками. Строка версии спецификации имеет специальный формат. Она состоит из нескольких чисел, разделенных точками. Каждое число может начинаться с нуля (нулей), но при этом оно не рассматривается как восьмеричное. Чем больше числа, тем более поздней считается версия. Метод isCompatibleWith() вызывает getSpecificationVersion(), чтобы определить версию спецификации, а затем сравнивает ее со строкой версии, переданной в качестве аргумента. Если версия спецификации пакета больше или равна версии в этой строке, isCompatibleWith() возвращает true. С помощью этого метода можно проверять, подходит ли для приложения версия пакета, содержащего, как правило, стандартное расширение.

Пакеты можно «запечатывать» (seal). В таком случае все классы этого пакета должны быть из одного файла JAR. Если пакет запечатан, метод isSealed() без аргументов возвращает true. Тот же метод с одним аргументом возвращает true, если указанный URL ссылается на файл JAR, из которого загружен пакет.

```
public class Package {
// Конструктор отсутствует
// Открытые методы класса
   public static Package getPackage(String name);
   public static Package[] getPackages();
// Методы доступа к свойствам (по именам свойств)
   public String getРеализации:Title();
   public String getРеализации:Vendor();
   public String getРеализации:Version();
   public String getName();
   public boolean isSealed();
   public boolean isSealed(java.net.URL url);
   public String getSpecificationTitle();
   public String getSpecificationVendor();
   public String getSpecificationVersion();
// Открытые методы экземпляра
   public boolean isCompatibleWith(String desired) throws NumberFormatException;
// Открытые методы, замещающие методы класса Object
   public int hashCode();
   public String toString();
}
```

Возвращается методами: Class.getPackage(), ClassLoader.{definePackage(), getPackage(), getPackages()}, Package.{getPackage(), getPackages()}, java.net.URLClassLoader.definePackage()

Process Java 1.0

java.lang

Этот класс описывает процесс, выполняемый вне интерпретатора Java. Обратите внимание, что класс Process сильно отличается от Thread; класс Process является абстрактным и не может применяться в экземплярах. Чтобы запустить процесс, вызовите один из методов Runtime.exec(), возвращающих соответствующий объект Process.

Метод waitFor() приостанавливает выполнение, пока процесс не завершится. exitValue() возвращает код завершения процесса. destroy() уничтожает процесс. Метод getErrorStream() возвращает входной поток (объект InputStream), из которого можно прочитать любые данные, посылаемые процессом в стандартный поток ошибок. getOutputStream() возвращает выходной поток (объект OutputStream), посредством которого можно посылать данные в стандартный входной поток процесса.

```
public abstract class Process {
// Открытые конструкторы
   public Process();
// Методы доступа к свойствам (по именам свойств)
   public abstract java.io.InputStream getErrorStream();
   public abstract java.io.InputStream getInputStream();
   public abstract java.io.OutputStream getOutputStream();
// Открытые методы экземпляра
   public abstract void destroy();
   public abstract int exitValue();
   public abstract int waitFor() throws InterruptedException;
}
```

Возвращается методами: Runtime.exec()

Runnable

Java 1.0

java.lang ***запускаемый***

Этот интерфейс определяет метод run(), который нужно использовать вместе с классом Thread. Любой класс, который реализует этот интерфейс, может предоставить тело потока. Подробности представлены в описании класса Thread.

```
public interface Runnable {
// Открытые методы экземпляра
   public abstract void run();
}
```

Реализации: java.awt.image.renderable.RenderableImageProducer, Thread, java.util.TimerTask, javax.swing.text.AsyncBoxView.ChildState

Передается методам: java.awt.EventQueue.{invokeAndWait(), invokeLater()}, java.awt.event.InvocationEvent.InvocationEvent(), Thread.Thread(), javax.swing.SwingUtilities.{invokeAndWait(), invokeLater()}, javax.swing.text.AbstractDocument.render(), javax.swing.text.Document.render(), javax.swing.text.LayoutQueue.addTask()

Возвращается методами: javax.swing.text.LayoutQueue.waitForWork()

Экземпляры: java.awt.event.InvocationEvent.runnable

Runtime

Java 1.0

java.lang

Этот класс инкапсулирует некоторые платформо-зависимые системные функции. Статический метод getRuntime() возвращает объект Runtime для текущей платформы; этот объект может выполнять системные функции независимо от платформы.

Метод exit() завершает интерпретатор Java с указанным кодом завершения. Этот метод обычно вызывается так: System.exit(). В Java 1.3 метод addShutdownHook() регист-

рирует поток (объект Thread), который выполняется при закрытии виртуальной машины – либо после вызова метода exit(), либо после прерывания, инициированного пользователем (например, после нажатия Ctrl-C). Цель «ловушки» (hook) – выполнить необходимую «уборку»: например закрыть сетевые подключения, удалить временные файлы и т. п. Методом addShutdownHook() можно зарегистрировать любое количество «ловушек». Прежде чем завершить свою работу, интерпретатор запускает все потоки-«ловушки» и позволяет им выполняться одновременно. Любая «ловушка» должна быстро выполнить уборку, не задерживая процесс завершения работы. Чтобы удалить ловушку, прежде чем она будет запущена, нужно вызвать метод removeShutdownHook(). Чтобы принудительно завершить работу интерпретатора без запуска ловушек, вызовите метод halt().

Метод exec() запускает новый процесс, который будет выполняться вне интерпретатора. Обратите внимание, что любой процесс, выполняемый вне интерпретатора, может быть системо-зависимым.

Метод freeMemory() возвращает приблизительное значение объема свободной памяти, а totalMemory() – объем памяти, доступный интерпретатору Java. Метод gc() заставляет сборщик мусора работать синхронно; это поможет высвободить дополнительную память. Аналогично, runFinalization() приводит к немедленному вызову методов finalize() объектов, на которые нет ни одной ссылки. Таким образом высвобождаются системные ресурсы, занятые такими объектами.

Метод load() загружает динамическую библиотеку, для которой указаны путь и имя файла. loadLibrary() загружает динамическую библиотеку с заданным именем; библиотека разыскивается по системным путям поиска. Как правило, такие библиотеки содержат методы для данной платформы в виде ее родного кода.

Методы traceInstructions() и traceMethodCalls() включают и выключают трассировку, выполняемую интерпретатором. Эти методы применяются для отладки или оптимизации приложения. В спецификации не определено, каким образом виртуальная машина будет выводить трассировочную информацию; виртуальным машинам не требуется поддерживать эту функцию.

Обратите внимание, что некоторые методы класса Runtime чаще вызываются посредством статических методов класса System.

```
public class Runtime {
// Конструктор отсутствует
// Открытые методы класса
   public static Runtime getRuntime();
// Открытые методы экземпляра
1.3 public void addShutdownHook(Thread hook);
1.4 public int availableProcessors();                                    // зависит от платформы
   public Process exec(String[] cmdarray) throws java.io.IOException;
   public Process exec(String command) throws java.io.IOException;
   public Process exec(String cmd, String[] envp) throws java.io.IOException;
   public Process exec(String[] cmdarray, String[] envp) throws java.io.IOException;
1.3 public Process exec(String[] cmdarray, String[] envp, java.io.File dir)
                                                    throws java.io.IOException;
1.3 public Process exec(String command, String[] envp, java.io.File dir) throws java.io.IOException;
   public void exit(int status);
   public long freeMemory();                                          // зависит от платформы
   public void gc();                                                  // зависит от платформы
1.3 public void halt(int status);
   public void load(String filename);
   public void loadLibrary(String libname);
```

```
1.4 public long maxMemory();              // зависит от платформы
1.3 public boolean removeShutdownHook(Thread hook);
   public void runFinalization();
   public long totalMemory();                                    // зависит от платформы
   public void traceInstructions(boolean on);                    // зависит от платформы
   public void traceMethodCalls(boolean on);                     // зависит от платформы
// Устаревшие открытые методы
# public java.io.InputStream getLocalizedInputStream(java.io.InputStream in);
# public java.io.OutputStream getLocalizedOutputStream(java.io.OutputStream out);
1.1# public static void runFinalizersOnExit(boolean value);
}
```

Возвращается методами: `Runtime.getRuntime()`

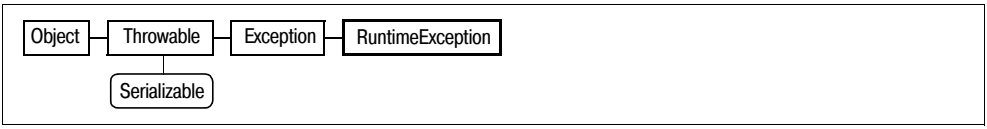
RuntimeException

Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение не применяется напрямую, но оно служит базовым классом для группы исключений времени выполнения (runtime exception), которые не нужно объявлять в определении метода с помощью оператора `throws`. Поскольку эти исключения могут возникнуть в любом методе Java, объявлять их не требуется. Такой подход помогает избежать громоздкости кода.

Этот класс наследует методы класса `Throwable`, но не объявляет собственных методов. Каждый конструктор `RuntimeException` просто вызывает соответствующие конструкторы `Exception()` и `Throwable()`. См. также `Throwable`.




```
public class RuntimeException extends Exception {
// Открытые конструкторы
   public RuntimeException();
1.4 public RuntimeException(Throwable cause);
   public RuntimeException(String message);
1.4 public RuntimeException(String message, Throwable cause);
}
```

Подклассы: Классов слишком много, чтобы их перечислить.

RuntimePermission

Java 1.2

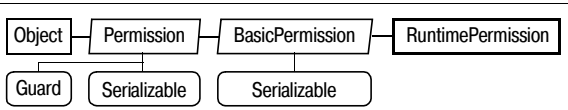
java.lang ***сериализуемый***

Данный класс является классом `java.security.Permission`, который предоставляет доступ к различным важным системным средствам. `RuntimePermission` содержит имя (или target), представляющее средство, доступ к которому запрашивается или предоставляется. Имя «exitVM» означает право на вызов `System.exit()`, а имя «accessClassInPackage.java.lang» означает право на чтение классов из пакета `java.lang`. В имени `RuntimePermission` может присутствовать суффикс «.*» как признак множественного выбора. Например, имя «accessClassInPackage.java.*» связано с разрешением на чтение классов из любого пакета, начинающегося на «java.». В отличие от некоторых

классов Permission объект RuntimePermission не использует строки со списком действий; здесь достаточно одного имени права.

Поддерживаются следующие права RuntimePermssion: «accessClassInPackage.*package*», «accessDeclaredMembers», «createClassLoader», «createSecurityManager», «defineClassInPackage.*package*», «exitVM», «getClassLoader», «getProtectionDomain», «loadLibrary.*library_name*», «modifyThread», «modifyThreadGroup», «queuePrintJob», «readFileDescriptor», «set-ContextClassLoader», «setFactory», «setIO», «setSecurityManager», «stopThread» и «writeFileDescriptor».

Системным администраторам, настраивающим системные политики, нужно знать эти имена и действия, на которые предоставляется разрешение. Кроме того, они должны оценивать риск, связанный с предоставлением тех или иных прав. Эти классы могут потребоваться системным программистам, а программисты, разрабатывающие приложения, не должны применять их напрямую.

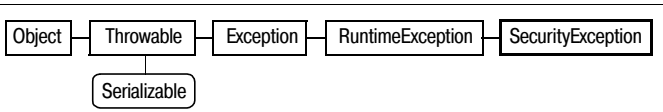



```
public final class RuntimePermission extends java.security.BasicPermission {
// Открытые конструкторы
   public RuntimePermission(String name);
   public RuntimePermission(String name, String actions);
}
```

SecurityException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает, если операция не разрешена в целях обеспечения безопасности.




```
public class SecurityException extends RuntimeException {
// Открытые конструкторы
   public SecurityException();
   public SecurityException(String s);
}
```

Подклассы: java.rmi.RMISecurityException, java.security.AccessControlException

Генерируется методами: Методов слишком много, чтобы их перечислить.

SecurityManager Java 1.0

java.lang

Этот класс определяет методы, которые необходимы для реализации политики безопасности, контролирующей выполнение ненадежных (untrusted) программ. До вы-

полнения потенциально опасной операции Java вызывает методы текущего объекта SecurityManager, чтобы определить, разрешена ли эта операция. Данные методы генерируют исключение SecurityException, если операция не разрешена. Как правило, приложениям не нужно применять SecurityManager. Он используется только веб-броузерами, программами просмотра апплетов и другими программами, которые запускают ненадежные приложения в контролируемой среде.

В версиях Java, предшествующих 1.2, этот класс был абстрактным. Все методы check(), реализованные по умолчанию, всегда вызывают исключение SecurityException. Механизм безопасности Java был переработан в версии 1.2. Теперь этот класс не является абстрактным, а его методы, реализованные по умолчанию, вполне работоспособны, поэтому создавать потомки этого класса приходится очень редко. Теперь, если операция разрешена, вызов метода нормально завершается, а если нет, то генерируется исключение SecurityException. Метод checkPermission() вызывает метод checkPermission() системного объекта java.security.AccessController. В Java 1.2 и последующих версиях все остальные методы с приставкой check- объекта SecurityManager реализованы на основе checkPermission().

```
public class SecurityManager {
// Открытые конструкторы
   public SecurityManager();
// Методы доступа к свойствам (по именам свойств)
   public Object getSecurityContext();                    // по умолчанию:AccessControlContext
1.1 public ThreadGroup getThreadGroup();
// Открытые методы экземпляра
   public void checkAccept(String host, int port);
   public void checkAccess(ThreadGroup g);
   public void checkAccess(Thread t);
1.1 public void checkAwtEventQueueAccess();
   public void checkConnect(String host, int port);
   public void checkConnect(String host, int port, Object context);
   public void checkCreateClassLoader();
   public void checkDelete(String file);
   public void checkExec(String cmd);
   public void checkExit(int status);
   public void checkLink(String lib);
   public void checkListen(int port);
1.1 public void checkMemberAccess(Class clazz, int which);
1.1 public void checkMulticast(java.net.InetAddress maddr);
   public void checkPackageAccess(String pkg);
   public void checkPackageDefinition(String pkg);
1.2 public void checkPermission(java.security.Permission perm);
1.2 public void checkPermission(java.security.Permission perm, Object context);
1.1 public void checkPrintJobAccess();
   public void checkPropertiesAccess();
   public void checkPropertyAccess(String key);
   public void checkRead(String file);
   public void checkRead(java.io.FileDescriptor fd);
   public void checkRead(String file, Object context);
1.1 public void checkSecurityAccess(String target);
   public void checkSetFactory();
1.1 public void checkSystemClipboardAccess();
   public boolean checkTopLevelWindow(Object window);
   public void checkWrite(java.io.FileDescriptor fd);
   public void checkWrite(String file);
// Защищенные методы экземпляра
   protected Class[] getClassContext();                   // зависит от платформы
```

```
// Устаревшие открытые методы
1.1# public void checkMulticast(java.net.InetAddress maddr, byte ttl);
# public boolean getInCheck();                                    // по умолчанию:false
// Устаревшие защищенные методы
# protected int classDepth(String name);                          // зависит от платформы
# protected int classLoaderDepth();
# protected ClassLoader currentClassLoader();
1.1# protected Class currentLoadedClass();
# protected boolean inClass(String name);
# protected boolean inClassLoader();
// Устаревшие защищенные поля
# protected boolean inCheck;
}
```

Подклассы: `java.rmi.RMISecurityManager`

Передается методам: `System.setSecurityManager()`

Возвращается методами: `System.getSecurityManager()`

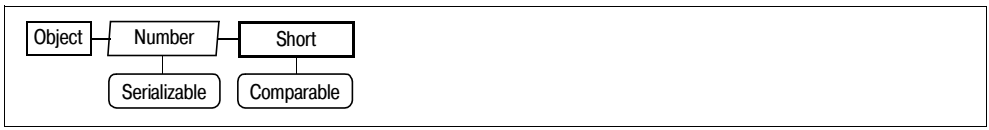
Short Java 1.1

java.lang ***сериализуемый, сравнимый***

Этот класс предоставляет обертку для простого типа `short`. В нем определены константы для минимального и максимального значений, которые могут храниться в переменных типа `short`, и константа типа `Class`, представляющая тип `short`. В этом классе есть различные методы для конвертирования значений объекта `Short` в строки, методы для обратного преобразования, а также методы для преобразования в другие числовые типы.

Большая часть статических методов этого класса конвертирует `String` в объект `Short` или значение типа `short`; четыре варианта методов `parseShort()` и `valueOf()` извлекают число из указанной строки и возвращают его в этих двух форматах. Метод `decode()` преобразует строку в число типа `Short` по основанию 10, 8 или 16. Если строка начинается с «0x» или «#», то число, записанное в ней, рассматривается как шестнадцатеричное, а если с «0» – как восьмеричное. В остальных случаях число считается десятичным.

Обратите внимание, что в этом классе есть два различных метода `toString()`. Первый метод – статический – преобразует значение типа `short` в `string`. Второй метод конвертирует объект `Short` в строку. Большая часть оставшихся методов преобразует `Short` в значения простых целочисленных типов.




```
public final class Short extends Number implements Comparable {
// Открытые конструкторы
   public Short(short value);
   public Short(String s) throws NumberFormatException;
// Открытые константы
   public static final short MAX_VALUE;                                  // =32767
   public static final short MIN_VALUE;                                  // =-32768
   public static final Class TYPE;
```

```
// Открытые методы класса
   public static Short decode(String nm) throws NumberFormatException;
   public static short parseShort(String s) throws NumberFormatException;
   public static short parseShort(String s, int radix) throws NumberFormatException;
   public static String toString(short s);
   public static Short valueOf(String s) throws NumberFormatException;
   public static Short valueOf(String s, int radix) throws NumberFormatException;
// Открытые методы экземпляра
1.2 public int compareTo(Short anotherShort);
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы Number
   public byte byteValue();
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
   public short shortValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: `Short.compareTo()`

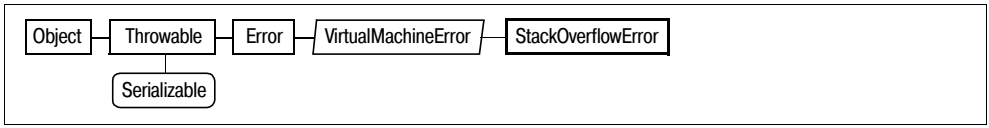
Возвращается методами: `Short.{decode(), valueOf()}`

StackOverflowError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает при переполнении стека интерпретатора Java.




```
public class StackOverflowError extends VirtualMachineError {
// Открытые конструкторы
   public StackOverflowError();
   public StackOverflowError(String s);
}
```

StackTraceElement

Java 1.4

java.lang ***сериализуемый***

Массив экземпляров этого класса возвращается методом `Throwable.getStackTrace()`. Каждый экземпляр представляет один фрейм трассировки стека, связанного с объектом исключения или ошибки. Методы `getClassName()` и `getMethodName()` возвращают имя класса (включая имя пакета) и имя метода, фрагмент которого представлен фреймом стека. Если файл класса содержит необходимую информацию, методы `getFileName()` и `getLineNumber()` возвращают имя исходного файла и номер строки, связанной с фреймом. `getFileName()` возвращает `null`, если указанный метод является

собственным методом Java (зависит от платформы) и для него не доступна информация об исходном файле и номере строки.


```
public final class StackTraceElement implements Serializable {
// Конструктор отсутствует
// Методы доступа к свойствам (по именам свойств)
   public String getClassName();
   public String getFileName();
   public int getLineNumber();
   public String getMethodName();
   public boolean isЗависит от платформыMethod();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: `Throwable.setStackTrace()`

Возвращается методами: `Throwable.getStackTrace()`

StrictMath

Java 1.3

java.lang

Этот класс аналогичен `Math`, но на его методы наложено дополнительное требование: они должны строго реализовывать определенные алгоритмы. Методы класса `StrictMath` должны выдавать однозначный результат с точностью до последнего значащего бита для всех возможных аргументов. Если не требуется строгая тождественность результатов для любой платформы, то для увеличения производительности можно применять класс `Math`.

```
public final class StrictMath {
// Конструктор отсутствует
// Открытые константы
   public static final double E;                              // =2.718281828459045
   public static final double PI;                             // =3.141592653589793
// Открытые методы класса
   public static int abs(int a);                              // strictfp
   public static long abs(long a);                            // strictfp
   public static float abs(float a);                          // strictfp
   public static double abs(double a);                        // strictfp
   public static double acos(double a);                       // зависит от платформы; strictfp
   public static double asin(double a);                       // зависит от платформы; strictfp
   public static double atan(double a);                       // зависит от платформы; strictfp
   public static double atan2(double y, double x);            // зависит от платформы; strictfp
   public static double ceil(double a);                       // зависит от платформы; strictfp
   public static double cos(double a);                        // зависит от платформы; strictfp
   public static double exp(double a);                        // зависит от платформы; strictfp
   public static double floor(double a);                      // зависит от платформы; strictfp
   public static double IEEEremainder(double f1, double f2); // зависит от платформы; strictfp
   public static double log(double a);                        // зависит от платформы; strictfp
   public static int max(int a, int b);                       // strictfp
```

```
    public static long max(long a, long b);                    // strictfp
    public static float max(float a, float b);                 // strictfp
    public static double max(double a, double b);              // strictfp
    public static int min(int a, int b);                       // strictfp
    public static long min(long a, long b);                    // strictfp
    public static float min(float a, float b);                 // strictfp
    public static double min(double a, double b);              // strictfp
    public static double pow(double a, double b);              // зависит от платформы; strictfp
    public static double random();                             // strictfp
    public static double rint(double a);                       // зависит от платформы; strictfp
    public static int round(float a);                          // strictfp
    public static long round(double a);                        // strictfp
    public static double sin(double a);                        // зависит от платформы; strictfp
    public static double sqrt(double a);                       // зависит от платформы; strictfp
    public static double tan(double a);                        // зависит от платформы; strictfp
    public static double toDegrees(double angrad);             // strictfp
    public static double toRadians(double angdeg);             // strictfp
}
```

String

Java 1.0

java.lang ***сериализуемый, сравнимый***

Класс `String` хранит строку символов, доступную только для чтения. Объект `String` создается компилятором Java для каждой строки, заключенной в двойные кавычки; как правило, такой способ создания объекта удобнее, чем вызов конструктора. Статические методы-фабрики `valueOf()` создают новые объекты `String`, содержащие текстовое представление простых типов Java. Также есть методы `valueOf()`, `copyValueOf()` и конструкторы `String()`, которые создают объект `String`, хранящий копию текста, содержащегося в массиве или части массива. Все три группы методов выполняют одинаковые функции. Конструктор `String()` можно применять для создания массива или подмассива байтовых значений. При этом нужно точно указать имя набора символов (или кодировки), который будет задействован при декодировании этих значений в символы. Кроме того, по умолчанию можно использовать основную кодировку базовой системы (имена наборов символов представлены в описании `java.nio.charset.Charset`).

Метод `length()` возвращает количество символов в строке, а `charAt()` – символ из строки. Эти методы можно использовать в цикле при прохождении всей строки. Для получения массива байтов, который содержит закодированные символы строки (в системной или в другой кодировке), применяйте метод `getBytes()`.

В этом классе определено много методов для сравнения строк и подстрок. `equals()` возвращает `true`, если два объекта `String` содержат одинаковый текст, а `equalsIgnoreCase()` возвращает `true`, если две строки равны без учета регистра. В Java 1.4 метод `contentEquals()` сравнивает строку с содержимым указанного объекта `StringBuffer` и возвращает `true`, если они содержат одинаковый текст. Методы `startsWith()` и `endsWith()` возвращают `true`, если строка начинается с указанного префикса или заканчивается указанным суффиксом соответственно. Вариант метода `startsWith()` с двумя аргументами позволяет указать позицию в строке, на которой должна находиться префиксная подстрока. Метод `regionMatches()` – это обобщенный вариант метода `startsWith()`. Он возвращает `true`, если подстрока указанной строки совпадает с подстрокой, начинающейся на заданной позиции строки. Вариант этого метода, принимающий пять аргументов, позволяет выполнить такое сравнение без учета регистра.

Наконец, matches() сравнивает строку с паттерном регулярного выражения. Он будет рассмотрен ниже.

compareTo() – еще один метод для сравнения строк, но он предназначен скорее для определения порядка двух строк, нежели для проверки их равенства. compareTo() реализует интерфейс Comparable и позволяет сортировать списки и массивы строк. Подробности представлены в описании Comparable. Метод compareToIgnoreCase() аналогичен compareTo(), но при проверке не учитывает регистр. Константа CASE_INSENSITIVE_ORDER – это объект Comparator для сортировки строк без учета регистра. (Интерфейс java.util.Comparator аналогичен интерфейсу Comparable, но разрешает определять порядок объектов, отличающийся от порядка, заданного по умолчанию в Comparable.) Методы compareTo(), compareToIgnoreCase() и объект CASE_INSENSITIVE_ORDER типа Comparator могут упорядочивать строки только по порядку номеров символов в кодировке Unicode. В некоторых языках такой порядок отличается от алфавитного. Общая методика сортировки строк представлена в описании класса java.text.Collator.

Методы indexOf() и lastIndexOf() проводят поиск указанного символа или подстроки в прямом и обратном направлениях. Они возвращают позицию найденной подстроки или символа либо –1, если вхождений не найдено. Вариант с одним аргументом начинает поиск с начала или с конца строки, а с двумя аргументами – от указанной позиции.

Метод substring() возвращает подстроку данной строки от символа с начальным индексом (включительно) до символа с конечным индексом (не входит в подстроку). Кроме того, есть версия с одним аргументом, которая возвращает все символы от указанной позиции и до конца строки. В Java 1.4 класс String реализует интерфейс CharSequence и определяет методы subSequence(), работающие так же, как метод substring() с двумя аргументами, но возвращающие подстроку в виде объекта CharSequence, а не String.

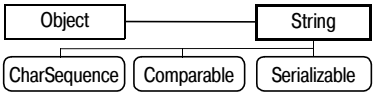
Несколько методов этого класса возвращают новую строку, содержащую модифицированный текст исходной строки, которая не изменяется. Метод replace() создает новую строку, в которой все вхождения указанного символа заменены другим символом. Более универсальные методы, replaceAll() и replaceFirst(), используют паттерн регулярного выражения. Они описаны ниже. Методы toUpperCase() и toLowerCase() возвращают новую строку, в которой все символы представлены в верхнем или нижнем регистре. Эти методы для изменения регистра могут принимать необязательный аргумент Locale для изменения регистра с учетом региональных настроек (locale). trim() – это вспомогательный метод, возвращающий новую строку, в которой удалены все пробельные символы (whitespaces) в начале и конце строки. concat() возвращает новую строку, состоящую из данной строки и присоединенной строки. Но обычно конкатенация строк проводится с помощью оператора +.

Обратите внимание, что объект String нельзя изменять: в нем нет метода setCharAt(), изменяющего содержимое строки. Методы, возвращающие объект String, не изменяют исходную строку, а возвращают новый объект String, содержащий измененную копию оригинального текста. Чтобы манипулировать содержимым строки, используйте StringBuffer или вызывайте методы toCharArray() и getChars(), преобразующие строку в массив символов.

В Java 1.4 предоставлена поддержка сравнения текста с паттернами регулярных выражений. Метод matches() возвращает true, если данная строка в точности соответствует паттерну, указанному в аргументе. Методы replaceAll() и replaceFirst() создают новую строку, в которой соответственно все вхождения или первое вхождение подстроки, соответствующей указанному паттерну, заменяются заданной строкой. Методы split() возвращают массив подстрок данной строки, сформированный путем

разделения строки в местах, соответствующих регулярным выражениям. В этом классе для удобства определены все методы, использующие регулярные выражения. Они просто вызывают одноименные методы из пакета `java.util.regex`. Дополнительная информация представлена в описаниях классов `Pattern` и `Matcher` этого пакета.

Многие программы используют строки так же часто, как и простые типы Java. Поскольку тип `String` представляет объект, а не простой тип, то в общем случае для сравнения двух строк нельзя использовать оператор `==`. Несмотря на то что строки этого типа неизменяемы, вместо этого оператора необходимо использовать более медленный метод `equals()`. В программах, которые производят много сравнений строк, метод `intern()` осуществляет более быстрое сравнение. Класс `String` поддерживает набор объектов `String`, включающих все строковые литералы, заключенные в двойные кавычки, и все строки программы, обрабатываемые в процессе компиляции. Метод `intern()` производит поиск строки в этом наборе или добавляет в него новую строку. Он ищет в наборе строку, полностью совпадающую с данной строкой, и возвращает ее. Если такая строка не найдена, то она записывается в этот набор и возвращается методом. Таким образом, можно сравнивать любые строки, возвращаемые методом `intern()`, применяя операторы `==` и `!=` вместо метода `equals()`. Значение, возвращаемое `intern()`, можно сравнивать с любой строковой константой с помощью операторов `==` и `!=`.




```
public final class String implements CharSequence, Comparable, Serializable {
// Открытые конструкторы
   public String();
   public String(StringBuffer buffer);
1.1 public String(byte[] bytes);
   public String(String original);
   public String(char[] value);
# public String(byte[] ascii, int hibyte);
1.1 public String(byte[] bytes, String charsetName) throws java.io.UnsupportedEncodingException;
   public String(char[] value, int offset, int count);
1.1 public String(byte[] bytes, int offset, int length);
# public String(byte[] ascii, int hibyte, int offset, int count);
1.1 public String(byte[] bytes, int offset, int length, String charsetName)
        throws java.io.UnsupportedEncodingException;
// Открытые константы
1.2 public static final java.util.Comparator CASE_INSENSITIVE_ORDER;
// Открытые методы класса
   public static String copyValueOf(char[] data);
   public static String copyValueOf(char[] data, int offset, int count);
   public static String valueOf(long l);
   public static String valueOf(int i);
   public static String valueOf(double d);
   public static String valueOf(float f);
   public static String valueOf(Object obj);
   public static String valueOf(char[] data);
   public static String valueOf(char c);
   public static String valueOf(boolean b);
   public static String valueOf(char[] data, int offset, int count);
```

```
// Открытые методы экземпляра
   public char charAt(int index);                                    // Реализует:CharSequence
   public int compareTo(String anotherString);
1.2 public int compareToIgnoreCase(String str);
   public String concat(String str);
1.4 public boolean contentEquals(StringBuffer sb);
   public boolean endsWith(String suffix);
   public boolean equalsIgnoreCase(String anotherString);
1.1 public byte[] getBytes();
1.1 public byte[] getBytes(String charsetName) throws java.io.UnsupportedEncodingException;
   public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);
   public int indexOf(String str);
   public int indexOf(int ch);
   public int indexOf(String str, int fromIndex);
   public int indexOf(int ch, int fromIndex);
   public String intern();                                            // зависит от платформы
   public int lastIndexOf(String str);
   public int lastIndexOf(int ch);
   public int lastIndexOf(String str, int fromIndex);
   public int lastIndexOf(int ch, int fromIndex);
   public int length();                                              // Реализует:CharSequence
1.4 public boolean matches(String regex);
   public boolean regionMatches(int toffset, String other, int ooffset, int len);
   public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len);
   public String replace(char oldChar, char newChar);
1.4 public String replaceAll(String regex, String replacement);
1.4 public String replaceFirst(String regex, String replacement);
1.4 public String[] split(String regex);
1.4 public String[] split(String regex, int limit);
   public boolean startsWith(String prefix);
   public boolean startsWith(String prefix, int toffset);
   public String substring(int beginIndex);
   public String substring(int beginIndex, int endIndex);
   public char[] toCharArray();
   public String toLowerCase();
1.1 public String toLowerCase(java.util.Locale locale);
   public String toString();                                         // Реализует:CharSequence
   public String toUpperCase();
1.1 public String toUpperCase(java.util.Locale locale);
   public String trim();
// Реализация методов интерфейса CharSequence
   public char charAt(int index);
   public int length();
1.4 public CharSequence subSequence(int beginIndex, int endIndex);
   public String toString();
// Реализация методов из Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object anObject);
   public int hashCode();
// Устаревшие открытые методы
# public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin);
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: Полей слишком много, чтобы их перечислить.

StringBuffer

Java 1.0

java.lang *сериализуемый*

Этот класс хранит изменяемую строку символов, которая при необходимости может увеличиваться или уменьшаться. Поскольку строка изменяемая, этот класс можно применять для обработки текста, которую не может выполнить класс `String`. Благодаря переменному размеру строки и реализованным в этом классе методам, он удобнее массива `char[]`. Создать объект `StringBuffer` можно с помощью конструктора `StringBuffer()`. Конструктору можно передать объект `String`, содержащий текст, который будет помещен в буфер. Если конструктор вызывается без параметров, создается пустой буфер. Если известно максимальное количество символов, которые будет содержать буфер, укажите в конструкторе первоначальную емкость этого буфера.

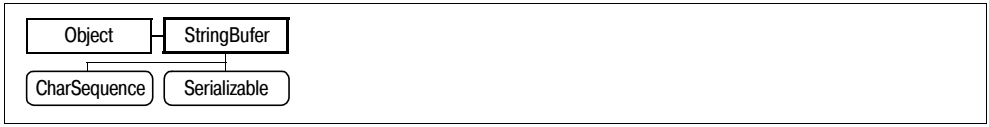
Метод `charAt()` запрашивает символ с указанным индексом, а методы `setCharAt()` и `deleteCharAt()` соответственно записывают или удаляют символ. Чтобы получить длину буфера, используйте `length()`, а чтобы установить длину — `setLength()`. При этом часть буфера может отбрасываться или, наоборот, при необходимости в буфер могут быть добавлены null-символы (**«\u0000»**). `capacity()` возвращает максимальное количество символов, которые можно поместить в объект `StringBuffer`, не вызвав перераспределение памяти для его внутреннего буфера. Если ожидается существенное увеличение длины буфера и известна примерная максимальная длина, то можно заранее выделить достаточный объем памяти для внутреннего буфера с помощью метода `ensureCapacity()`.

Для добавления текста к концу буфера применяйте методы `append()`. Метод `insert()` вставляет текст в буфер на указанную позицию. Обратите внимание, что этим методам можно передавать не только строки, значения простых типов, массивы символов, но и произвольные объекты. Их значения предварительно преобразуются в строки. Метод `delete()` удаляет символы из указанного диапазона буфера, а `replace()` заменяет диапазон символов строкой из заданного объекта `String`.

Метод `substring()` преобразует часть строки из `StringBuffer` в объект `String`. Два варианта этого метода работают точно так же, как одноименные методы класса `String`. В Java 1.4 `StringBuffer` реализует интерфейс `CharSequence` и, следовательно, определяет метод `subSequence()`, который аналогичен `substring()`, но возвращает объект типа `CharSequence`. Кроме того, в Java 1.4 добавлены методы `indexOf()` и `lastIndexOf()`, которые ищут соответственно в прямом и обратном направлениях вхождение указанной строки `String`. При этом можно задать индекс, относительно которого будет проведен поиск. Эти методы возвращают индекс вхождения или –1, если вхождение не найдено. См. также одноименные методы класса `String`, на основе которых созданы данные методы.

Для получения содержимого `StringBuffer` в виде объекта `String` применяйте метод `toString()`. Метод `getChars()` извлекает символы из указанного диапазона и сохраняет их в заданной позиции указанного массива символов.

Конкатенация строк в Java выполняется с помощью оператора + и реализуется с помощью метода `append()` класса `StringBuffer`. После того как над строкой выполнены необходимые действия в объекте `StringBuffer`, ее можно конвертировать в объект `String`. Метод `StringBuffer.toString()` обычно не копирует массив символов. Вместо этого новый объект `String` и объект `StringBuffer` используют один массив, а новая копия создается только при дальнейшем изменении строки в `StringBuffer`.




```
public final class StringBuffer implements CharSequence, Serializable {
// Открытые конструкторы
   public StringBuffer();
   public StringBuffer(int length);
   public StringBuffer(String str);
// Открытые методы экземпляра
   public StringBuffer append(Object obj);                                    // синхронизирован
   public StringBuffer append(boolean b);
   public StringBuffer append(char c);                                        // синхронизирован
   public StringBuffer append(char[] str);                                    // синхронизирован
1.4 public StringBuffer append(StringBuffer sb);                              // синхронизирован
   public StringBuffer append(String str);                                    // синхронизирован
   public StringBuffer append(float f);
   public StringBuffer append(long l);
   public StringBuffer append(int i);
   public StringBuffer append(double d);
   public StringBuffer append(char[] str, int offset, int len);               // синхронизирован
   public int capacity();                                                     // синхронизирован
   public char charAt(int index);  // Реализует:CharSequence синхронизирован
1.2 public StringBuffer delete(int start, int end);                           // синхронизирован
1.2 public StringBuffer deleteCharAt(int index);                              // синхронизирован
   public void ensureCapacity(int minimumCapacity);                           // синхронизирован
   public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin); // синхронизирован
1.4 public int indexOf(String str);
1.4 public int indexOf(String str, int fromIndex);                            // синхронизирован
   public StringBuffer insert(int offset, boolean b);
   public StringBuffer insert(int offset, char[] str);                        // синхронизирован
   public StringBuffer insert(int offset, Object obj);                        // синхронизирован
   public StringBuffer insert(int offset, String str);                        // синхронизирован
   public StringBuffer insert(int offset, char c);                            // синхронизирован
   public StringBuffer insert(int offset, float f);
   public StringBuffer insert(int offset, double d);
   public StringBuffer insert(int offset, int i);
   public StringBuffer insert(int offset, long l);
1.2 public StringBuffer insert(int index, char[] str, int offset, int len);  // синхронизирован
1.4 public int lastIndexOf(String str);                                       // синхронизирован
1.4 public int lastIndexOf(String str, int fromIndex);                        // синхронизирован
   public int length();                                  // Реализует:CharSequence синхронизирован
1.2 public StringBuffer replace(int start, int end, String str);              // синхронизирован
   public StringBuffer reverse();                                             // синхронизирован
   public void setCharAt(int index, char ch);                                 // синхронизирован
   public void setLength(int newLength);                                      // синхронизирован
1.2 public String substring(int start);                                       // синхронизирован
1.2 public String substring(int start, int end);                              // синхронизирован
   public String toString();                                          // Реализует:CharSequence
// Реализация методов интерфейса CharSequence
   public char charAt(int index);                                             // синхронизирован
   public int length();                                                       // синхронизирован
1.4 public CharSequence subSequence(int start, int end);
   public String toString();
}
```

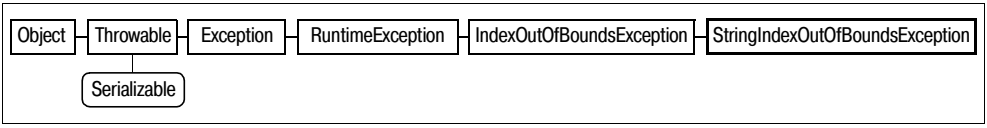
Передается методами: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

StringIndexOutOfBoundsException Java 1.0

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает, если индекс, используемый для доступа к символу строки `String` или `StringBuffer`, слишком большой или отрицательный.




```
public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException {
// Открытые конструкторы
   public StringIndexOutOfBoundsException();
   public StringIndexOutOfBoundsException(int index);
   public StringIndexOutOfBoundsException(String s);
}
```

System Java 1.0

java.lang

В этом классе определен платформо-независимый интерфейс для системных функций, в том числе системные свойства и потоки. Все методы и переменные этого класса статические, а экземпляр этого класса создать нельзя. Поскольку методы, определенные в этом классе, являются низкоуровневыми системными методами, многие из них требуют специального разрешения и не могут применяться ненадежными программами.

Метод `getProperty()` ищет указанное свойство в списке системных свойств и, если это свойство не найдено, возвращает значение по умолчанию (если оно задано). `getProperties()` возвращает полный список свойств. `setProperties()` присваивает указанному объекту `Properties` список системных свойств. В Java 1.2 и последующих версиях метод `setProperty()` устанавливает значение системного свойства. В следующей таблице приведены свойства, которые всегда определены. Ненадежные программы могут не иметь доступа для чтения некоторых из этих свойств. Дополнительные свойства можно определить с помощью ключа `-D` командной строки интерпретатора Java.

Имя свойства	Описание
`java.home`	Каталог, в котором установлен пакет Java
`java.class.path`	Каталог, из которого загружаются классы
`java.specification.version`	Версия спецификации Java API (Java 1.2)
`java.specification.vendor`	Поставщик спецификации Java API (Java 1.2)
`java.specification.name`	Имя спецификации Java API (Java 1.2)
`java.version`	Версия реализации Java API
`java.vendor`	Поставщик реализации Java API

Имя свойства	Описание
java.vendor.url	URL поставщика реализации Java API
java.vm.specification.version	Версия спецификации ВМ Java (Java 1.2)
java.vm.specification.vendor	Поставщик спецификации ВМ Java (Java 1.2)
java.vm.specification.name	Имя спецификации ВМ Java (Java 1.2)
java.vm.version	Версия реализации ВМ Java (Java 1.2)
java.vm.vendor	Версия поставщика ВМ Java (Java 1.2)
java.vm.name	Имя реализации ВМ Java (Java 1.2)
java.class.version	Версия формата файлов класса Java
os.name	Имя базовой операционной системы
os.arch	Архитектура базовой операционной системы
os.version	Версия базовой операционной системы
file.separator	Символ-разделитель имен каталогов текущей платформы
line.separator	Символ(ы) перевода строки текущей платформы
user.name	Имя текущего пользователя
user.home	Домашний каталог текущего пользователя
user.dir	Текущий рабочий каталог

Поля `in`, `out` и `err` содержат стандартные входной, выходной потоки системы и поток ошибок соответственно. Эти поля часто применяются при вызове таких методов, как `System.out.println()`. В Java 1.1 методы `setIn()`, `setOut()` и `setErr()` позволяют перенаправить эти потоки.

В классе `System` есть еще несколько полезных методов. `exit()` завершает работу ВМ Java. `arraycopy()` выполняет эффективное копирование части массива-источника в массив-приемник. Метод `currentTimeMillis()` возвращает текущее время в миллисекундах относительно полуночи 1 января 1970 года по GMT. `gc()` вызывает полную уборку мусора, а `runFinalization()` заставляет сборщик мусора выполнить закрытие (finalization) всех готовых для этого объектов. В приложении обычно не требуется вызывать данные методы, но они могут быть полезны при оценке производительности кода с помощью `currentTimeMillis()`. Метод `identityHashCode()` вычисляет хеш-код объекта точно так же, как метод `Object.hashCode()` по умолчанию. Его вызов не зависит от того, как был замещен метод `hashCode()`. Методы `load()` и `loadLibrary()` читают библиотеки, используемые в данной системе. `mapLibraryName()` конвертирует системонезависимое имя библиотеки в системо-зависимое имя файла библиотеки. Наконец, методы `getSecurityManager()` и `setSecurityManager()` соответственно получают и устанавливают объект `SecurityManager`, отвечающий за политику безопасности системы.

См. также класс `Runtime`, в котором определено несколько других методов, осуществляющих низкоуровневый доступ к системным свойствам.

```
public final class System {
// Конструктор отсутствует
// Открытые константы
   public static final java.io.PrintStream err;
   public static final java.io.InputStream in;
   public static final java.io.PrintStream out;
// Открытые методы класса
```

```
   public static void arraycopy(Object src, int srcPos, Object dest, int destPos,
                                int length);                          // зависит от платформы
   public static long currentTimeMillis();                            // зависит от платформы
   public static void exit(int status);
   public static void gc();
   public static java.util.Properties getProperties();
   public static String getProperty(String key);
   public static String getProperty(String key, String def);
   public static SecurityManager getSecurityManager();
1.1 public static int identityHashCode(Object x);                     // зависит от платформы
   public static void load(String filename);
   public static void loadLibrary(String libname);
1.2 public static String mapLibraryName(String libname);              // зависит от платформы
   public static void runFinalization();
1.1 public static void setErr(java.io.PrintStream err);
1.1 public static void setIn(java.io.InputStream in);
1.1 public static void setOut(java.io.PrintStream out);
   public static void setProperties(java.util.Properties props);
1.2 public static String setProperty(String key, String value);
   public static void setSecurityManager(SecurityManager s);
// Устаревшие открытые методы
# public static String getenv(String name);
1.1# public static void runFinalizersOnExit(boolean value);
}
```

Thread

Java 1.0

java.lang ***запускаемый***

Этот класс инкапсулирует всю необходимую информацию об одном потоке управления, выполняющемся в интерпретаторе Java. Для создания потока нужно либо передать конструктору `Thread` объект `Runnable` (то есть объект, реализующий метод `run()` интерфейса `Runnable`), либо наследовать `Thread`, определив собственный метод `run()`. Метод `run()` объекта `Thread` или указанного объекта `Runnable` представляет собой тело потока. Он начинает выполняться, когда вызывается метод `start()` объекта `Thread`. Поток выполняется до возвращения из метода `run()`. Метод `isAlive()` возвращает `true`, если поток начал выполнение, а метод `run()` еще не завершился.

Статические методы этого класса работают с выполняющимся в данный момент потоком. `currentThread()` возвращает объект `Thread` программы, выполняемой в данный момент. `sleep()` останавливает текущий поток на указанный промежуток времени. Метод `yield()` забирает управление у текущего потока и передает его любому потоку с таким же приоритетом, ожидающему выполнения. `holdsLock()` проверяет, заблокирован ли текущий поток синхронизационным методом или оператором; этот метод доступен в Java 1.4 и часто применяется с оператором `assert`.

Методы экземпляра могут вызываться одним потоком для выполнения действия с другим потоком. Метод `checkAccess()` проверяет, имеет ли выполняющийся поток разрешение на изменение объекта `Thread`. Если разрешения нет, вызывается исключение `SecurityException`. Метод `join()` ждет, пока поток не будет уничтожен. `interrupt()` прерывает ожидание потока (исключением `InterruptedException`) или устанавливает флаг прерывания для потока, не находящегося в режиме ожидания. Поток может выполнить проверку своего флага прерывания с помощью статического метода `interrupted()`, а проверку флага другого потока – методом `isInterrupted()`. При вызове метода `interrupted()` автоматически очищается флаг прерывания, а при вызове `isIn-`

terrupted() — нет. Методы sleep() и interrupt() похожи на методы wait() и notify(), определенные в классе Object. Вызов метода wait() блокирует текущий поток, пока из другого потока не будет вызван метод notify() данного объекта.

setName() определяет имя для потока (делать это не обязательно). setPriority() устанавливает приоритет потока. Поток с более высоким приоритетом выполняется до потока с более низким приоритетом. В Java не оговаривается поведение нескольких потоков с одинаковым приоритетом. В некоторых системах производится распределение процессорного времени между потоками, которые выполняются на одном процессоре. На других системах текущий поток может подавлять другие потоки с таким же приоритетом, пока не будет вызван метод yield(). Метод setDaemon() устанавливает флаг типа boolean, определяющий, является ли данный поток демоном. ВМ Java работает до тех пор, пока не завершится последний поток, не являющийся демоном. Для получения группы потоков (объект ThreadGroup), к которой относится текущий поток, нужно вызвать метод getThreadGroup(). В Java 1.2 и последующих версиях с помощью метода setContextClassLoader() можно указать объект ClassLoader, который будет применяться для загрузки классов, необходимых данному потоку.

Методы suspend(), resume() и stop() соответственно приостанавливают, возобновляют и завершают выполнение потока. Но эти методы считаются устаревшими (deprecated), потому что применять их небезопасно: они могут привести к взаимной блокировке (deadlock). Чтобы поток можно было останавливать, используйте флаг, который периодически проверяется потоком; поток должен завершиться, если флаг установлен.

В Java 1.4 и последующих версиях конструктору Thread() с четырьмя аргументами можно передать размер стека для потока. Как правило, больший размер уменьшает вероятность переполнения стека, а меньший размер стека сокращает объем памяти, требуемой для одного потока, и позволяет одновременно существовать нескольким потокам. В зависимости от реализации этот аргумент интерпретируется по-разному; в некоторых случаях он даже игнорируется.


```
public class Thread implements Runnable {
// Открытые конструкторы
   public Thread();
   public Thread(String name);
   public Thread(Runnable target);
   public Thread(Runnable target, String name);
   public Thread(ThreadGroup group, String name);
   public Thread(ThreadGroup group, Runnable target);
   public Thread(ThreadGroup group, Runnable target, String name);
1.4 public Thread(ThreadGroup group, Runnable target, String name, long stackSize);
// Открытые константы
   public static final int MAX_PRIORITY;                                    // =10
   public static final int MIN_PRIORITY;                                    // =1
   public static final int NORM_PRIORITY;                                   // =5
// Открытые методы класса
   public static int activeCount();
   public static Thread currentThread();                    // зависит от платформы
   public static void dumpStack();
   public static int enumerate(Thread[] tarray);
1.4 public static boolean holdsLock(Object obj);            // зависит от платформы
   public static boolean interrupted();
```

```
   public static void sleep(long millis) throws InterruptedException;      // зависит от платформы
   public static void sleep(long millis, int nanos) throws InterruptedException;
   public static void yield();                                             // зависит от платформы
// Методы доступа к свойствам (по именам свойств)
   public final boolean isAlive();                    // зависит от платформы; по умолчанию: false
1.2 public ClassLoader getContextClassLoader();
1.2 public void setContextClassLoader(ClassLoader cl);
   public final boolean isDaemon();                                          // по умолчанию: false
   public final void setDaemon(boolean on);
   public boolean isInterrupted();                                           // по умолчанию: false
   public final String getName();                                          // по умолчанию: "Thread-1"
   public final void setName(String name);
   public final int getPriority();                                             // по умолчанию: 5
   public final void setPriority(int newPriority);
   public final ThreadGroup getThreadGroup();
// Открытые методы экземпляра
   public final void checkAccess();
   public void destroy();
   public void interrupt();
   public final void join() throws InterruptedException;
   public final void join(long millis) throws InterruptedException;            // синхронизирован
   public final void join(long millis, int nanos) throws InterruptedException; // синхронизирован
   public void start();                              // зависит от платформы;  синхронизирован
// Реализация методов интерфейса Runnable
   public void run();
// Открытые методы, замещающие методы класса Object
   public String toString();
// Устаревшие открытые методы
# public int countStackFrames();                                              // зависит от платформы
# public final void resume();
# public final void stop();
# public final void stop(Throwable obj);                                       // синхронизирован
# public final void suspend();
}
```

Передается методами: `Runtime.{addShutdownHook(), removeShutdownHook()}`, `SecurityManager.checkAccess()`, `Thread.enumerate()`, `ThreadGroup.{enumerate(), uncaughtException()}`

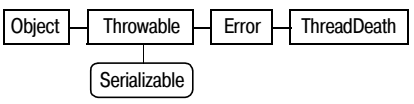
Возвращается методами: `Thread.currentThread()`, `javax.swing.text.AbstractDocument.getCurrentWriter()`

ThreadDeath

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка сообщает, что поток должен завершиться. Она возникает в потоке, когда вызывается метод `Thread.stop()` этого потока. Данная ошибка отличается от других типов ошибок: когда она возникает, поток останавливается; при этом не выводится сообщение об ошибке, а интерпретатор не завершает работу. Можно «захватить» ошибку типа `ThreadDeath` и выполнить необходимые действия по очистке, но потом нужно заново вызвать эту ошибку, чтобы поток действительно остановился.



```
public class ThreadDeath extends Error {
// Открытые конструкторы
   public ThreadDeath();
}
```

ThreadGroup Java 1.0

java.lang

Этот класс представляет группу потоков и позволяет работать со всеми потоками этой группы одновременно. В группе ThreadGroup могут содержаться объекты Thread и другие объекты ThreadGroup. Все объекты ThreadGroup создаются как дочерние объекты иной группы ThreadGroup, образуя иерархию объектов ThreadGroup. Для получения родительского объекта ThreadGroup применяйте метод getParent(). Методы activeCount(), activeGroupCount() и различные варианты метода enumerate() позволяют составить список дочерних объектов Thread и ThreadGroup. Большинству приложений можно ограничиться лишь системной группой потоков, используемой по умолчанию. В программах системного уровня и в серверных приложениях, которые применяют большое количество потоков, можно для удобства создавать собственные объекты ThreadGroup.

Метод interrupt() одновременно прерывает все потоки в группе. setMaxPriority() определяет максимальный приоритет, который может иметь поток из данной группы. Если текущий поток не имеет соответствующего доступа, то этот метод вызывает исключение SecurityException. Метод uncaughtException() содержит код, который будет выполнен при аварийном завершении потока после возникновения непроверяемого исключения. Поведение этого метода можно изменить в потомках класса ThreadGroup.

```
public class ThreadGroup {
// Открытые конструкторы
   public ThreadGroup(String name);
   public ThreadGroup(ThreadGroup parent, String name);
// Методы доступа к свойствам (по именам свойств)
   public final boolean isDaemon();
   public final void setDaemon(boolean daemon);
1.1 public boolean isDestroyed();                                    // синхронизирован
   public final int getMaxPriority();
   public final void setMaxPriority(int pri);
   public final String getName();
   public final ThreadGroup getParent();
// Открытые методы экземпляра
   public int activeCount();
   public int activeGroupCount();
   public final void checkAccess();
   public final void destroy();
   public int enumerate(ThreadGroup[] list);
   public int enumerate(Thread[] list);
   public int enumerate(Thread[] list, boolean recurse);
   public int enumerate(ThreadGroup[] list, boolean recurse);
1.2 public final void interrupt();
   public void list();
   public final boolean parentOf(ThreadGroup g);
   public void uncaughtException(Thread t, Throwable e);
// Открытые методы, замещающие методы класса Object
   public String toString();
// Устаревшие открытые методы
1.1# public boolean allowThreadSuspension(boolean b);
```

```
# public final void resume();
# public final void stop();
# public final void suspend();
}
```

Передается методам: SecurityManager.checkAccess(), Thread.Thread(), ThreadGroup.{enumerate(), parentOf(), ThreadGroup()}

Возвращается методами: SecurityManager.getThreadGroup(), Thread.getThreadGroup(), ThreadGroup.getParent()

ThreadLocal Java 1.2

java.lang

С помощью этого класса можно легко создавать локальные переменные потока. Статическое поле класса может иметь только одно значение для всех объектов класса. Если в классе объявлено нестатическое поле экземпляра класса, то каждый объект имеет собственную копию этой переменной. Класс ThreadLocal соединяет в себе возможности этих видов полей. Если в объекте ThreadLocal объявлено статическое поле, то для каждого потока в данном объекте хранится свое значение. Объекты из одного потока получают одно и то же значение при вызове метода get() объекта ThreadLocal. Объектам разных потоков этот метод возвращает разные значения.

Метод set() устанавливает значение, хранимое в объекте ThreadLocal текущего потока, а get() возвращает это значение. Обратите внимание, что получить значение, хранящееся в объекте ThreadLocal какого-нибудь потока, можно только вызовом метода get(). Чтобы понять принцип действия объекта ThreadLocal, можно рассматривать его как хеш-таблицу или объект java.util.Map, ставящий в соответствие объекту Thread произвольное значение. Вызов set() создает связь между текущим потоком (Thread.currentThread()) и указанным значением. При вызове метода get() сначала проводится поиск текущего потока, а затем с помощью хеш-таблицы находится значение, связанное с текущим потоком.

Если поток вызывает метод get() в первый раз, а локальное значение потока не было установлено методом set(), то get() вызывает защищенный метод initialValue(), чтобы получить начальное значение, которое и будет возвращено. Реализация метода initialValue() по умолчанию возвращает null, но его можно заменить в подклассах.

См. также класс InheritableThreadLocal, который позволяет дочерним потокам наследовать локальное значение родительского потока.

```
public class ThreadLocal {
// Открытые конструкторы
   public ThreadLocal();
// Открытые методы экземпляра
   public Object get();
   public void set(Object value);
// Защищенные методы экземпляра
   protected Object initialValue();  // константа
}
```

Подклассы: InheritableThreadLocal

Throwable

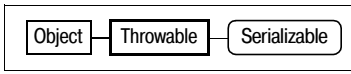
Java 1.0

java.lang *сериализуемый*

Данный класс – вершина иерархии исключений и ошибок Java. Все исключения и ошибки являются потомками Throwable. Метод getMessage() возвращает сообщение, связанное с данным исключением или ошибкой. Метод getLocalizedMessage(), реализованный по умолчанию, просто вызывает getMessage(), но этот метод можно замещать в подклассах, чтобы сообщение об ошибке было локализовано согласно региональным установкам системы.

Часто объекты Exception и Error создаются в результате исключения или ошибки в API низкого уровня. В Java 1.4 и последующих версиях любой объект типа Throwable может иметь объект-«причину», которой определяется вызвавший его объект Throwable более низкого уровня. В этом случае нужно передать объект-«причину» конструктору Throwable() или методу initCause(). Когда перехватывается объект Throwable, метод getCause() позволяет получить объект Throwable, инициировавший его.

Все объекты Throwable несут информацию о связанном с ними стеке выполнения. Эта информация появляется при создании объекта Throwable. Если объект будет «вызван» не там, где он был создан, или будет перенаправлен, то для захвата текущего стека выполнения можно применить метод fillInStackTrace(). printStackTrace() направляет текущий стек выполнения в текстовом виде указанному объекту PrintWriter, PrintStream или потоку System.err. В Java 1.4 информацию такого рода можно получить с помощью метода getStackTrace(), возвращающего массив объектов StackTraceElement, которые описывают стек выполнения.




```
public class Throwable implements Serializable {
// Открытые конструкторы
   public Throwable();
   public Throwable(String message);
1.4 public Throwable(Throwable cause);
1.4 public Throwable(String message, Throwable cause);
// Методы доступа к свойствам (по именам свойств)
1.4 public Throwable getCause();                                    // по умолчанию:null
1.1 public String getLocalizedMessage();                            // по умолчанию:null
   public String getMessage();                                      // по умолчанию:null
1.4 public StackTraceElement[] getStackTrace();
1.4 public void setStackTrace(StackTraceElement[] stackTrace);
// Открытые методы экземпляра
   public Throwable fillInStackTrace();              // зависит от платформы; синхронизирован
1.4 public Throwable initCause(Throwable cause);                    // синхронизирован
   public void printStackTrace();
   public void printStackTrace(java.io.PrintStream s);
1.1 public void printStackTrace(java.io.PrintWriter s);
// Открытые методы, замещающие методы класса Object
   public String toString();
}
```

Подклассы: Error, Exception

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Генерируется методами: `java.awt.Cursor.finalize()`, `java.awt.Font.finalize()`, `java.awt.Frame.finalize()`, `java.awt.Window.finalize()`, `Object.finalize()`, `java.lang.reflect.InvocationHandler.invoke()`, `javax.imageio.spi.ServiceRegistry.finalize()`, `javax.imageio.stream.ImageInputStreamImpl.finalize()`, `javax.swing.text.AbstractDocument.AbstractElement.finalize()`

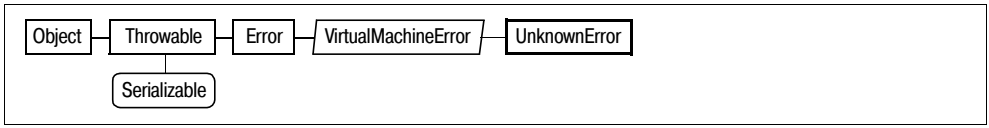
Экземпляры: `java.rmi.RemoteException.detail`, `java.rmi.activation.ActivationException.detail`, `javax.naming.NamingException.rootException`, `org.omg.CORBA.portable.UnknownException.originalException`

UnknownError

Java 1.0

java.lang ***сериализуемый, ошибка***

Неизвестная ошибка на уровне виртуальной машины Java.



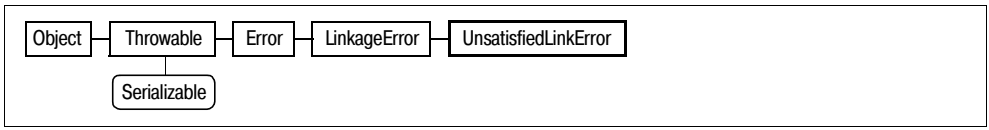

```
public class UnknownError extends VirtualMachineError {
// Открытые конструкторы
   public UnknownError();
   public UnknownError(String s);
}
```

UnsatisfiedLinkError

Java 1.0

java.lang ***сериализуемый, ошибка***

Эта ошибка возникает, когда Java не может установить все связи в загруженном классе.



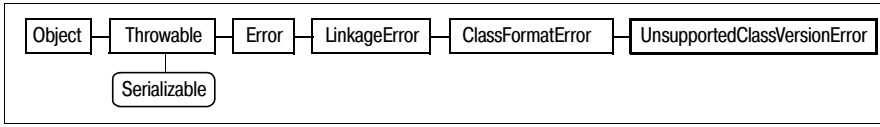

```
public class UnsatisfiedLinkError extends LinkageError {
// Открытые конструкторы
   public UnsatisfiedLinkError();
   public UnsatisfiedLinkError(String s);
}
```

UnsupportedClassVersionError

Java 1.2

java.lang ***сериализуемый, ошибка***

Все классы Java содержат номер версии, определяющей формат файла класса. Эта ошибка возникает, когда виртуальная машина Java предпринимает попытку прочесть файл класса с номером версии, которую она не поддерживает.



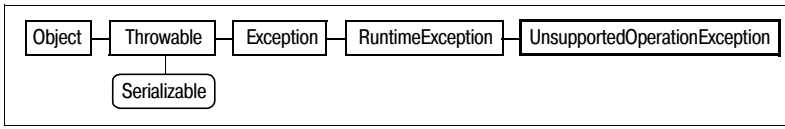

```
public class UnsupportedClassVersionError extends ClassFormatError {
// Открытые конструкторы
   public UnsupportedClassVersionError();
   public UnsupportedClassVersionError(String s);
}
```

UnsupportedOperationException

Java 1.2

java.lang ***сериализуемое, непроверяемое***

Это исключение возникает, когда вызванный метод не поддерживается и не выполняет никаких действий, кроме вызова этого исключения. Чаще всего оно применяется в коллекциях Java из пакета `java.util`. Неизменяемые коллекции генерируют это исключение при вызове таких методов, как `add()` или `delete()`.




```
public class UnsupportedOperationException extends RuntimeException {
// Открытые конструкторы
   public UnsupportedOperationException();
   public UnsupportedOperationException(String message);
}
```

Подклассы: `java.awt.HeadlessException`, `java.nio.ReadOnlyBufferException`

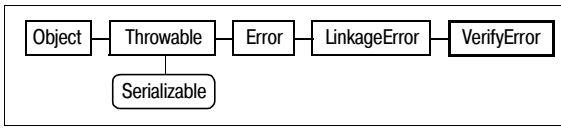
Генерируется методами: `java.awt.Toolkit.{getLockingKeyState()`, `setLockingKeyState()}`, `javax.imageio.ImageReadParam.setSourceRenderSize()`

VerifyError

Java 1.0

java.lang ***сериализумый, ошибка***

Эта ошибка возникает, если класс не прошел процедуру верификации байт-кода.

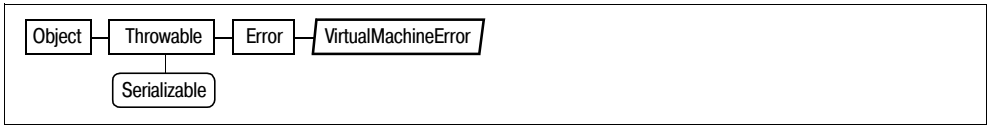



```
public class VerifyError extends LinkageError {
// Открытые конструкторы
   public VerifyError();
   public VerifyError(String s);
}
```

VirtualMachineError Java 1.0

java.lang ***сериализуемый, ошибка***

Эта абстрактная ошибка служит родительским классом для группы ошибок, связанных с ВМ Java. См. InternalError, UnknownError, OutOfMemoryError и StackOverflowError.




```
public abstract class VirtualMachineError extends Error {
// Открытые конструкторы
   public VirtualMachineError();
   public VirtualMachineError(String s);
}
```

Подклассы: InternalError, OutOfMemoryError, StackOverflowError, UnknownError

Void Java 1.1

java.lang

Экземпляры класса Void создавать нельзя. Он служит лишь для хранения статического поля TYPE – константы типа Class, представляющей тип void.

```
public final class Void {
// Конструктор отсутствует
// Открытые константы
   public static final Class TYPE;
}
```

Пакет java.lang.ref Java 1.2

В пакете java.lang.ref определены классы, позволяющие программам на Java взаимодействовать со сборщиком мусора. Класс Reference представляет собой косвенную ссылку на произвольный объект – так называемый *объект ссылки* (*referent*). Три потомка класса Reference – SoftReference, WeakReference и PhantomReference – по-разному взаимодействуют со сборщиком мусора. Подробнее они рассмотрены ниже. Класс ReferenceQueue представляет собой связанный список объектов Reference. Любой объект Reference может быть связан с объектом ReferenceQueue. Объект Reference ставится в очередь ReferenceQueue после того, как сборщик мусора определяет, что объект ссылки стал недостижимым. (Точный уровень «недостижимости» зависит от используемого типа Reference.) Приложение может осуществлять контроль над ReferenceQueue, чтобы определить, когда объекты ссылки получат новый статус достижимости.

С помощью механизмов, предоставленных этим пакетом, можно организовать кэш, который увеличивается или уменьшается в зависимости от размера свободной системной памяти. Можно также реализовать хеш-таблицу, связывающую информацию произвольного характера с произвольными объектами, но не запрещающую освобождение памяти уже неиспользуемых объектов. Но это механизмы низкого уровня, а пакет java.lang.ref обычно не используется в приложениях напрямую. Вместо

этого применяются высокоуровневые средства, основанные на классах из этого пакета. См., например, java.util.WeakHashMap.

Классы

```
public abstract class Reference;
   └ public class PhantomReference extends Reference;
   └ public class SoftReference extends Reference;
   └ public class WeakReference extends Reference;
public class ReferenceQueue;
```

PhantomReference Java 1.2

java.lang.ref

Этот класс представляет собой ссылку на объект, которая не запрещает закрытие объекта (finalization) сборщиком мусора. Как только сборщик мусора обнаруживает, что на объект больше нет жестких ссылок и ссылок типа SoftReference и WeakReference и объект еще не закрыт, сборщик мусора ставит объект PhantomReference в очередь ReferenceQueue, указанную при создании PhantomReference. Это событие служит признаком того, что объект был закрыт, и предоставляется последняя возможность выполнить «уборку», если в ней есть необходимость.

Чтобы запретить восстановление объекта ссылки PhantomReference, метод get() возвращает null во всех случаях – до и после того, как PhantomReference поставлен в очередь. Тем не менее PhantomReference при постановке в очередь не очищается автоматически, поэтому при его удалении из очереди ReferenceQueue нужно вызвать метод clear(), чтобы был утилизирован сам объект PhantomReference.

Этот класс предоставляет более гибкий механизм для очистки объектов по сравнению с методом finalize(). Для его применения нужно наследовать класс PhantomReference и определить метод, выполняющий необходимую очистку. Поскольку метод get() объекта PhantomReference всегда возвращает null, подкласс должен хранить все данные, необходимые для операции очистки.


```
public class PhantomReference extends java.lang.ref.Reference {
// Открытые конструкторы
   public PhantomReference(Object referent, ReferenceQueue q);
// Открытые методы, замещающие методы Reference
   public Object get();                                          // константа
}
```

Reference Java 1.2

java.lang.ref

Этот абстрактный класс хранит один из видов косвенной ссылки. Метод get() возвращает объект ссылки, если она не была явно очищена методом clear() или неявно очищена сборщиком мусора. У класса Reference есть три подкласса. При различных условиях сборщик мусора по-разному обрабатывает каждый из них и очищает их ссылки.

В каждом из подклассов Reference определен конструктор, позволяющий связывать с объектом ReferenceQueue объект ReferenceQueue. Сборщик мусора помещает объекты

Reference в связанные с ними очереди (ReferenceQueue), чтобы посылать уведомления о состоянии объекта ссылки. Метод isEnqueued() проверяет, помещен ли объект Reference в связанную с ним очередь ReferenceQueue, а enqueue() помещает его в очередь. Метод enqueue() возвращает false, если объект Reference не связан ни с одним объектом ReferenceQueue или уже был поставлен в очередь.

```
public abstract class Reference {
// Конструктор отсутствует
// Открытые методы экземпляра
   public void clear();
   public boolean enqueue();
   public Object get();
   public boolean isEnqueued();
}
```

Подклассы: PhantomReference, SoftReference, WeakReference

Возвращается методами: ReferenceQueue.{poll(), remove()}

ReferenceQueue

Java 1.2

java.lang.ref

Этот класс содержит очередь (или связный список) объектов Reference. Объект Reference помещается в очередь после того, как сборщик мусора определит, что объект ссылки больше не доступен. Этот класс служит в качестве системы уведомлений об изменении степени доступности объекта. Метод poll() возвращает первый объект Reference в очереди; он возвращает null, если очередь пуста. Метод remove() возвращает первый элемент очереди, или, если очередь пуста, ждет, пока объект Reference будет поставлен в очередь. При необходимости можно создавать любое количество объектов ReferenceQueue. Можно определить очередь ReferenceQueue для объекта Reference, передав его методу SoftReference() или WeakReference() либо конструктору PhantomReference().

Объект ReferenceQueue должен использовать объекты PhantomReference. Кроме того, он может применять SoftReference и WeakReference. Метод get() этих классов возвращает null, если объект ссылки больше не доступен в достаточной степени.

```
public class ReferenceQueue {
// Открытые конструкторы
   public ReferenceQueue();
// Открытые методы экземпляра
   public java.lang.ref.Reference poll();
   public java.lang.ref.Reference remove() throws InterruptedException;
   public java.lang.ref.Reference remove(long timeout) throws IllegalArgumentException,
      InterruptedException;
}
```

Передается методам: PhantomReference.PhantomReference(), SoftReference.SoftReference(), WeakReference.WeakReference()

SoftReference

Java 1.2

java.lang.ref

Этот класс представляет собой «мягкую» ссылку на объект. Ссылка SoftReference не очищается, пока на объект ссылается хотя бы одна «жесткая» (прямая) ссылка. Если объект ссылки больше не применяется (на него не осталось «жестких» ссылок), сбор-

щик мусора в любой момент может очистить SoftReference. Но сборщик мусора не будет очищать ссылку SoftReference, пока не определит, что системная память почти полностью исчерпана. В частности, ВМ Java никогда не вызовет ошибку OutOfMemoryError без предварительной очистки всех «мягких» ссылок и освобождения занимаемой ими памяти. ВМ может (но не обязана) очищать «мягкие» ссылки в порядке их последнего использования.

Если ссылка SoftReference связана с объектом ReferenceQueue, то сборщик мусора ставит ее в очередь через некоторое время после того, как ссылка была очищена.

Класс SoftReference особенно полезен при реализации системы кэширования объектов, размер которых не фиксирован, а увеличивается или уменьшается в зависимости от размера доступной памяти.


```
public class SoftReference extends java.lang.ref.Reference {
// Открытые конструкторы
   public SoftReference(Object referent);
   public SoftReference(Object referent, ReferenceQueue q);
// Открытые методы, замещающие методы Reference
   public Object get();
}
```

WeakReference Java 1.2

java.lang.ref

Этот класс ссылается на объект, не запрещая сборщику мусора закрывать его и освобождать память. Когда сборщик мусора определяет, что на объект больше нет «жестких» (прямых) ссылок и ссылок типа SoftReference, он очищает WeakReference и помечает объект ссылки для последующего закрытия. Спустя некоторое время объект WeakReference ставится в связанную с ним очередь ReferenceQueue (если такая существует) для уведомления о том, что объект ссылки освобожден.

WeakReference применяется в классе java.util.WeakHashMap для реализации хеш-таблицы, не запрещающей утилизацию объекта ключа. WeakHashMap полезен при связывании любой информации с объектом, не запрещая при этом освобождение памяти объекта.


```
public class WeakReference extends java.lang.ref.Reference {
// Открытые конструкторы
   public WeakReference(Object referent);
   public WeakReference(Object referent, ReferenceQueue q);
}
```

Пакет java.lang.reflect Java 1.1

Пакет java.lang.reflect содержит классы и интерфейсы, которые вместе с java.lang.Class составляют Java Reflection API.

Классы Constructor, Field и Method представляют конструкторы, поля и методы класса. Поскольку все эти типы представляют члены класса, каждый из них реализует интерфейс Member, определяющий минимальный набор методов, которые можно вызывать для любого члена класса. Эти классы позволяют получить информацию о членах класса, вызовах методов и доступе к полям.

Модификаторы членов класса представлены битовыми флагами. Класс Modifier определяет методы, с помощью которых можно интерпретировать значения этих флагов. В классе Array определены статические методы для создания массивов, чтения и записи их элементов.

В Java 1.3 класс Proxy позволяет осуществить динамическое создание новых классов Java, реализующих определенный набор интерфейсов. Когда метод интерфейса вызывается в экземпляре такого класса, этот вызов направляется объекту InvocationHandler.

Интерфейсы

```
public interface InvocationHandler;
public interface Member;
```

Классы

```
public class AccessibleObject;
   └ public final class Constructor extends AccessibleObject implements Member;
   └ public final class Field extends AccessibleObject implements Member;
   └ public final class Method extends AccessibleObject implements Member;
public final class Array;
public class Modifier;
public class Proxy implements Serializable;
public final class ReflectPermission extends java.security.BasicPermission;
```

Исключения

```
public class InvocationTargetException extends Exception;
public class UndeclaredThrowableException extends RuntimeException;
```

AccessibleObject Java 1.2

java.lang.reflect

Этот класс является потомком классов Method, Constructor и Field; его методы предоставляют надежным (trusted) приложениям механизм для работы с закрытыми (private), защищенными (protected) и доступными по умолчанию членами класса, к которым нельзя обратиться с помощью других средств Reflection API. Этот класс появился в Java 1.2; в Java 1.1 классы Method, Constructor и Field являются прямыми потомками класса Object.

Чтобы использовать пакет java.lang.reflect для доступа к члену класса, который нельзя осуществить обычным способом, нужно вызвать метод setAccessible(), передав ему true. Если данная программа имеет разрешение ReflectPermission («suppressAccessChecks»), то к этому члену класса можно обращаться как к открытому (public). С помощью статического варианта метода setAccessible() удобно устанавливать флаг доступа к массиву членов, но он производит проверку безопасности один раз.

```
public class AccessibleObject {
// Защищенные конструкторы
   protected AcessibleObjects()
// Открытые методы класса
   public static void setAccessible(AccessibleObject[] array, boolean flag) throws SecurityException;
```

```
// Открытые методы экземпляра
   public boolean isAccessible();
   public void setAccessible(boolean flag) throws SecurityException;
}
```

Подклассы: Constructor, Field, Method

Передается методам: AccessibleObject.setAccessible()

Array Java 1.1

java.lang.reflect

Методы этого класса позволяют устанавливать и запрашивать значения элементов массива, определять его длину и создавать новые экземпляры массива. Обратите внимание, что класс Array может работать только с массивами, а не с типами массивов; типы данных Java, в том числе типы массивов, представлены классом java.lang.Class. В отличие от классов Field, Method и Constructor, которые представляют члены классов, класс Array представляет значение, поэтому Array существенно отличается от остальных классов этого пакета (за исключением некоторого внешнего подобия). Самая главная особенность состоит в том, что все методы класса Array являются статическими и применяются к любым массивам, а не только к конкретному полю, методу или конструктору.

Метод get() возвращает значение указанного элемента данного массива в виде Object. Если элементы массива – простые типы, это значение преобразуется к типу объекта-обертки. Чтобы получить элементы массива в виде определенного простого типа, можно использовать getInt() и подобные методы. Метод set() и его разновидности для примитивных типов выполняют обратное действие. Метод getLength() возвращает длину массива.

Методы newInstance() предназначены для создания новых массивов. Одна из разновидностей данного метода принимает в качестве параметра количество элементов массива и их тип. Другой вариант этого метода создает многомерные массивы. Кроме типа элементов, ему передается массив чисел. Длиной этого массива определяется количество измерений создаваемого массива, а каждый элемент указывает размер каждого измерения.

```
public final class Array {
// Конструктор отсутствует
// Открытые методы класса
   public static Object get(Object array, int index) throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
   public static boolean getBoolean(Object array, int index)
        throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
   public static byte getByte(Object array, int index) throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
   public static char getChar(Object array, int index) throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
   public static double getDouble(Object array, int index)
        throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
   public static float getFloat(Object array, int index) throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
   public static int getInt(Object array, int index) throws IllegalArgumentException,
      ArrayIndexOutOfBoundsException;                                  // зависит от платформы
```

```
    public static int getLength(Object array) throws
             IllegalArgumentException;                                    // зависит от платформы
    public static long getLong(Object array, int index) throws IllegalArgumentException,
       ArrayIndexOutOfBoundsException;                                    // зависит от платформы
    public static short getShort(Object array, int index) throws IllegalArgumentException,
       ArrayIndexOutOfBoundsException;                                    // зависит от платформы
    public static Object newInstance(Class componentType, int length)
                     throws NegativeArraySizeException;
    public static Object newInstance(Class componentType, int[] dimensions)
                     throws IllegalArgumentException, NegativeArraySizeException;
    public static void set(Object array, int index, Object value)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setBoolean(Object array, int index, boolean z)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setByte(Object array, int index, byte b)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setChar(Object array, int index, char c)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setDouble(Object array, int index, double d)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setFloat(Object array, int index, float f)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setInt(Object array, int index, int i)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setLong(Object array, int index, long l)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
    public static void setShort(Object array, int index, short s)
         throws IllegalArgumentException, ArrayIndexOutOfBoundsException;  // зависит от платформы
}
```

Constructor

Java 1.1

java.lang.reflect

Этот класс представляет конструктор определенного класса. Можно получить экземпляр `Constructor`, вызвав метод `getConstructor()` или аналогичные методы класса `java.lang.Class`. `Constructor` реализует интерфейс `Member`, поэтому с помощью методов этого интерфейса можно получить имя, модификаторы и класс конструктора. Кроме того, методы `getParameterTypes()` и `getExceptionTypes()` возвращают полезную информацию о конструкторе.

С помощью вышеперечисленных методов можно не только получить информацию о конструкторе, но и создать экземпляр его класса с помощью метода `newInstance()`, передав ему массив аргументов конструктора. Если все аргументы конструктора являются простыми типами, их нужно привести к типам их оберток, прежде чем передавать методу `newInstance()`. Если конструктор генерирует исключение, объект `Throwable` этого исключения упаковывается в объект `InvocationTargetException`, который генерируется методом `newInstance()`. Заметьте, что область применения метода `newInstance()` данного класса намного шире, чем у одноименного метода класса `java.lang.Class`, поскольку он может передавать конструктору аргументы.

```
public final class Constructor extends AccessibleObject implements Member {
// Конструктор отсутствует
// Открытые методы экземпляра
   public Class[] getExceptionTypes();
   public Class[] getParameterTypes();
   public Object newInstance(Object[] initargs) throws InstantiationException,
      IllegalAccessException, IllegalArgumentException, InvocationTargetException;
// Реализация методов интерфейса Member
   public Class getDeclaringClass();
   public int getModifiers();
   public String getName();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Возвращается методами: Class.{getConstructor(), getConstructors(), etDeclaredConstructor(), getDeclaredConstructors()}

Field

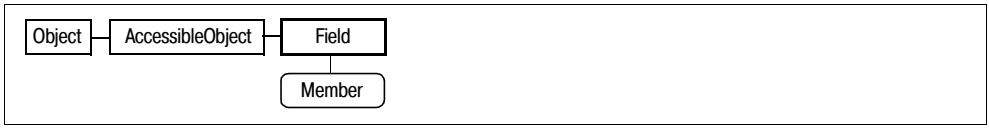
Java 1.1

java.lang.reflect

Этот класс представляет поле класса. Можно получить экземпляр Field, вызвав метод getField() или аналогичные методы класса java.lang.Class. Field реализует интерфейс Member, поэтому при получении объекта Field можно вызывать объекты getName(), getModifiers() и getDeclaringClass(), чтобы определить имя, модификаторы и класс данного поля. Кроме того, метод getType() возвращает тип поля.

Метод set() устанавливает значение данного поля в указанном объекте. (Конечно, если представленное поле является статическим, объект указывать не нужно.) Если это поле простого типа, то его значение можно указать в объекте-обертке Boolean, Integer и т. п. или установить с помощью методов setBoolean(), setInt() и подобных методов.

Метод get() запрашивает значение данного поля указанного объекта и возвращает его значение в виде Object. Остальные методы запрашивают значение поля и возвращают его в виде значения простого типа.




```
public final class Field extends AccessibleObject implements Member {
// Конструктор отсутствует
// Открытые методы экземпляра
   public Object get(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public boolean getBoolean(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public byte getByte(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public char getChar(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public double getDouble(Object obj) throws IllegalArgumentException,IllegalAccessException;
   public float getFloat(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public int getInt(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public long getLong(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public short getShort(Object obj) throws IllegalArgumentException, IllegalAccessException;
   public Class getType();
```

```
   public void set(Object obj, Object value) throws IllegalArgumentException, IllegalAccessException;
   public void setBoolean(Object obj, boolean z) throws IllegalArgumentException,IllegalAccessException;
   public void setByte(Object obj, byte b) throws IllegalArgumentException, IllegalAccessException;
   public void setChar(Object obj, char c) throws IllegalArgumentException, IllegalAccessException;
   public void setDouble(Object obj, double d) throws IllegalArgumentException, IllegalAccessException;
   public void setFloat(Object obj, float f) throws IllegalArgumentException,IllegalAccessException;
   public void setInt(Object obj, int i) throws IllegalArgumentException, IllegalAccessException;
   public void setLong(Object obj, long l) throws IllegalArgumentException,IllegalAccessException;
   public void setShort(Object obj, short s) throws IllegalArgumentException,IllegalAccessException;
// Реализация методов интерфейса Member
   public Class getDeclaringClass();
   public int getModifiers();
   public String getName();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Возвращается методами: Class.{getDeclaredField(), getDeclaredFields(), getField(), getFields()}

InvocationHandler

Java 1.3

java.lang.reflect

В этом интерфейсе определен один метод invoke(), который вызывается при каждом вызове метода динамически созданного объекта Proxy. Каждый объект Proxy связан с объектом InvocationHandler, который указывается при создании экземпляра Proxy. Все вызовы методов объекта Proxy преобразуются в вызовы метода invoke() класса InvocationHandler.

Первый аргумент invoke() – это объект Proxy, для которого вызван метод. Второй аргумент – это объект Method, представляющий вызванный метод. Для определения интерфейса, в котором объявлен метод, нужно вызвать getDeclaringClass(). Это может быть родительский класс одного из указанных интерфейсов и даже класс java.lang.Object, если вызываемый метод – toString(), hashCode() или другой метод класса Object. Третий аргумент метода invoke() – массив аргументов метода. Все аргументы простых типов передаются в виде своих объектов-оберток (Boolean, Integer, Double).

Значение, возвращаемое invoke(), возвращается вызываемым из объекта Proxy методом и должно быть соответствующего типа. Если метод объекта Proxy возвращает простой тип, то invoke() должен возвратить экземпляр класса-обертки для этого типа. Метод invoke() может вызвать любое непроверяемое исключение (то есть исключение во время выполнения) или проверяемое исключение, объявленное методом объекта Proxy. Если invoke() генерирует проверяемое исключение, которое не объявлено в методе объекта proxy, то это исключение запаковывается в объект непроверяемого исключения UndeclaredThrowableException, которое и будет вызвано.

```
public interface InvocationHandler {
// Открытые методы экземпляра
   public abstract Object invoke(Object proxy, Method method, Object[] args) throws Throwable;
}
```

Реализации: java.beans.EventHandler

Передается методам: Proxy.{newProxyInstance(), Proxy()}

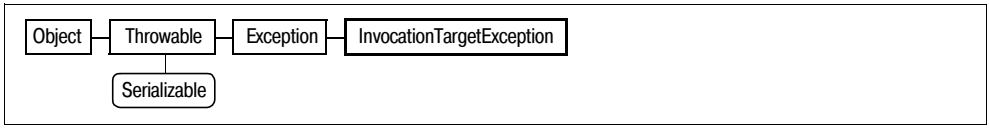
Возвращается методами: Proxy.getInvocationHandler()

Экземпляры: Proxy.h

InvocationTargetException Java 1.1

java.lang.reflect ***сериализуемое, проверяемое***

Объект этого класса генерируется методами Method.invoke() и Constructor.newInstance(), когда в вызываемом ими методе или конструкторе возникает исключение. Класс InvocationTargetException служит оберткой для возникшего исключения; этот объект можно получить методом getTargetException(). В Java 1.4 и последующих версиях все исключения могут таким образом «сцепляться», а метод getTargetException() заменен более универсальным методом getCause().




```
public class InvocationTargetException extends Exception {
// Открытые конструкторы
   public InvocationTargetException(Throwable target);
   public InvocationTargetException(Throwable target, String s);
// Защищенные конструкторы
   protected InvocationTargetException();
// Открытые методы экземпляра
   public Throwable getTargetException();
// Открытые методы, замещающие методы класса Throwable
1.4 public Throwable getCause();
}
```

Генерируется методами: java.awt.EventQueue.invokeAndWait(), Constructor.newInstance(), Method.invoke(), javax.swing.SwingUtilities.invokeAndWait()

Member Java 1.1

java.lang.reflect

Этот интерфейс определяет методы, которые применяются для всех членов класса (полей, методов и конструкторов). getName() возвращает имя члена, getModifiers() — его модификаторы, а getDeclaringClass() — объект Class, представляющий класс, к которому принадлежит данный член.

```
public interface Member {
// Открытые константы
   public static final int DECLARED;                                    // =1
   public static final int PUBLIC;                                      // =0
// Открытые методы экземпляра
   public abstract Class getDeclaringClass();
   public abstract int getModifiers();
   public abstract String getName();
}
```

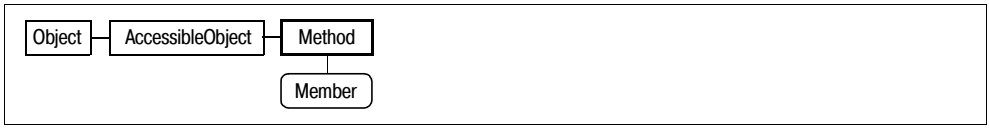
Реализации: Constructor, Field, Method

Method Java 1.1

java.lang.reflect

Этот класс представляет определенный метод. Экземпляр класса `Method` можно получить, вызвав `getMethod()` и аналогичные методы класса `java.lang.Class`. Класс `Method` реализует интерфейс `Member`, поэтому методы этого интерфейса можно задействовать для получения имени данного метода, его модификаторов и его класса. Кроме того, `getReturnType()`, `getParameterTypes()` и `getExceptionTypes()` возвращают полезную информацию о представленном методе.

Что еще более важно, `invoke()` позволяет вызывать метод, представленный объектом `Method`, с указанным массивом аргументов. Если какой-нибудь аргумент является простым типом, то он должен быть предварительно приведен к типу соответствующего класса-обертки. Если данный метод является методом экземпляра (то есть он не статический), то этот экземпляр также нужно передать методу `invoke()`. Если значение, возвращаемое методом `invoke()`, принадлежит простому типу, то оно предварительно приводится к типу соответствующего класса-обертки. Если в вызываемом методе возникает исключение, то соответствующий объект `Throwable` заключается в объект `InvocationTargetException`, который генерируется методом `invoke()`.




```
public final class Method extends AccessibleObject implements Member {
// Конструктор отсутствует
// Открытые методы экземпляра
   public Class[] getExceptionTypes();
   public Class[] getParameterTypes();
   public Class getReturnType();
   public Object invoke(Object obj, Object[] args) throws IllegalAccessException,
      IllegalArgumentException, InvocationTargetException;
// Реализация методов интерфейса Member
   public Class getDeclaringClass();
   public int getModifiers();
   public String getName();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: `java.beans.EventSetDescriptor.{getAddListenerMethod(), getGetListenerMethod(), getListenerMethods(), getRemoveListenerMethod()}, java.beans.IndexedPropertyDescriptor.{getIndexedReadMethod(), getIndexedWriteMethod()}, java.beans.MethodDescriptor.getMethod(), java.beans.PropertyDescriptor.{getReadMethod(), getWriteMethod()}, Class.{getDeclaredMethod(), getDeclaredMethods(), getMethod(), getMethods()}`

Modifier Java 1.1

java.lang.reflect

В этом классе определено несколько констант и статических методов для интерпретации значений типа `int`, возвращаемых методом `getModifiers()` классов `Field`, `Method` и `Constructor`. Методы `isPublic()`, `isAbstract()` и им подобные возвращают `true`, если в соответствующее значение включен указанный модификатор; в противном случае возвращается `false`. Константы, определенные в этом классе, соответствуют различным битовым флагам, используемым для определения значений модификаторов. Составив свою алгебру логики, эти константы можно применять для проверки наличия модификаторов.

```
public class Modifier {
// Открытые конструкторы
   public Modifier();
// Открытые константы
   public static final int ABSTRACT;                                    // =1024
   public static final int FINAL;                                       // =16
   public static final int INTERFACE;                                   // =512
   public static final int NATIVE;                                      // =256
   public static final int PRIVATE;                                     // =2
   public static final int PROTECTED;                                   // =4
   public static final int PUBLIC;                                      // =1
   public static final int STATIC;                                      // =8
1.2 public static final int STRICT;                                     // =2048
   public static final int SYNCHRONIZED;                                // =32
   public static final int TRANSIENT;                                   // =128
   public static final int VOLATILE;                                    // =64
// Открытые методы класса
   public static boolean isAbstract(int mod);
   public static boolean isFinal(int mod);
   public static boolean isInterface(int mod);
   public static boolean isNATIVE(int mod);
   public static boolean isPrivate(int mod);
   public static boolean isProtected(int mod);
   public static boolean isPublic(int mod);
   public static boolean isStatic(int mod);
1.2 public static boolean isStrict(int mod);
   public static boolean isSynchronized(int mod);
   public static boolean isTransient(int mod);
   public static boolean isVolatile(int mod);
   public static String toString(int mod);
}
```

Proxy Java 1.3

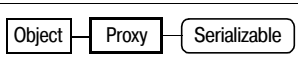
java.lang.reflect ***сериализуемый***

Этот класс предоставляет простые, но достаточно мощные средства API для динамического создания *класса-заместителя* (*proxy class*). Класс-заместитель реализует указанный список интерфейсов и направляет объекту-обработчику вызовы методов этих интерфейсов.

Статический метод `getProxyClass()` динамически создает новый объект типа `Class`, который реализует все интерфейсы, входящие в указанный массив `Class[]`. Созданный

класс определяется контекстом указанного объекта ClassLoader. Class, возвращаемый методом getProxyClass(), является потомком Proxy. Все классы, динамически созданные этим методом, имеют единственный открытый конструктор с одним аргументом типа InvocationHandler. Экземпляр динамического класса-заместителя можно создать, вызвав его конструктор с помощью класса Constructor. Но удобнее совместить вызов getProxyClass() с вызовом конструктора методом newProxyInstance(): в этом случае определяется класс-заместитель и создается его экземпляр.

Все экземпляры динамического класса-заместителя связаны с объектами InvocationHandler. Все вызовы метода этого класса преобразуются в вызовы метода invoke() объекта InvocationHandler, который обрабатывает вызовы подходящим способом. Статический метод isProxyClass() возвращает true, если указанный объект Class является динамическим классом-заместителем.




```
public class Proxy implements Serializable {
// Защищенные конструкторы
   protected Proxy(InvocationHandler h);
// Открытые методы класса
   public static InvocationHandler getInvocationHandler(Object proxy)
        throws IllegalArgumentException;
   public static Class getProxyClass(ClassLoader loader, Class[] interfaces)
        throws IllegalArgumentException;
   public static boolean isProxyClass(Class cl);
   public static Object newProxyInstance(ClassLoader loader, Class[] interfaces,
        InvocationHandler h) throws IllegalArgumentException;
// Защищенные поля экземпляра
   protected InvocationHandler h;
}
```

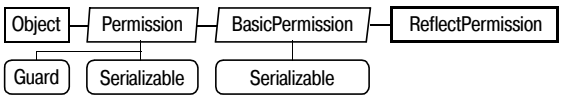
ReflectPermission

Java 1.2

java.lang.reflect ***сериализуемый***

Этот класс представляет собой разрешение типа java.security.Permission, отвечающее за доступ к закрытым, защищенным и открытым методам, конструкторам и полям, предоставляемый Java Reflection API. В Java 1.2 единственное имя («целевой объект»), соответствующее ReflectPermission, – это «suppressAccessChecks». Данное разрешение необходимо при вызове метода setAccessible() класса AccessibleObject. В отличие от некоторых потомков Permission, класс ReflectPermission не применяет список действий. См. также AccessibleObject.

Системные администраторы, производящие настройку политик безопасности, должны знать, как применять этот класс, а разработчикам приложений следует избегать его прямого использования.

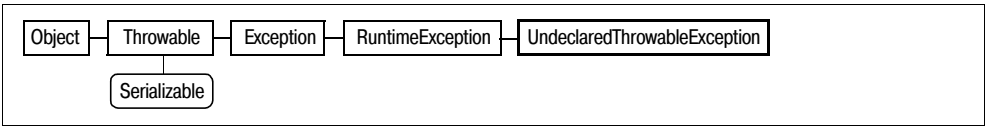


```
public final class ReflectPermission extends java.security.BasicPermission {
// Открытые конструкторы
   public ReflectPermission(String name);
   public ReflectPermission(String name, String actions);
}
```

UndeclaredThrowableException Java 1.3

java.lang.reflect ***сериализуемое, непроверяемое***

Это исключение вызывается методом объекта Proxy, если метод invoke() объекта InvocationHandler, связанного с данным Proxy, генерирует проверяемое исключение, не объявленное в вызываемом методе. Этот класс служит оберткой непроверяемого исключения для проверяемых исключений. Чтобы получить контролируемое исключение, вызванное методом invoke(), применяйте getUndeclaredThrowable(). В Java 1.4 и последующих версиях все исключения могут «сцепляться» таким образом, а метод getUndeclaredThrowable() заменен более универсальным методом getCause().




```
public class UndeclaredThrowableException extends RuntimeException {
// Открытые конструкторы
   public UndeclaredThrowableException(Throwable undeclaredThrowable);
   public UndeclaredThrowableException(Throwable undeclaredThrowable, String s);
// Открытые методы экземпляра
   public Throwable getUndeclaredThrowable();
// Открытые методы, замещающие методы класса Throwable
1.4 public Throwable getCause();
}
```

Глава 12

java.math

Пакет java.math

Java 1.1

Пакет `java.math`, впервые появившийся в Java 1.1, содержит классы для арифметики целых чисел и чисел с плавающей точкой произвольной точности. Целые числа произвольной длины применяются в криптографии, а числа с плавающей точкой произвольной точности нужны для финансовых приложений, в которых необходимо избегать ошибок при округлении.

Классы:

```
public class BigDecimal extends Number implements Comparable;
public class BigInteger extends Number implements Comparable;
```

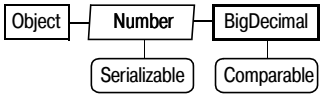
BigDecimal

Java 1.1

java.math *сериализуемый, сравнимый*

Этот класс является потомком `java.lang.Number` и представляет число с плавающей точкой произвольной длины и точности. Его методы дублируют стандартные арифметические операции Java. Метод `compareTo()` сравнивает значения двух объектов `BigDecimal` и возвращает результат сравнения в виде чисел –1, 0 или 1.

Объект `BigDecimal` представляет десятичную дробь в виде целого числа произвольной длины и масштаба (scale), который представляет количество десятичных разрядов после точки. При работе со значениями `BigDecimal` можно явно указать нужную точность (то есть количество десятичных разрядов). Кроме того, при выполнении операции, отбрасывающей разряды (например, при делении), требуется указать тип округления, которому подвергается первый разряд слева от отбрасываемых разрядов. В этом классе определено восемь констант, соответствующих различным типам округления. Поскольку в этом классе можно работать с числами произвольной точности и явно указывать точность и вид округления, его можно применять при выполнении операций над числами, представляющими денежные суммы, а также в других случаях, когда нельзя допускать ошибки при округлении.




```
public class BigDecimal extends Number
       implements Comparable {
// Открытые конструкторы
   public BigDecimal(BigInteger val);
   public BigDecimal(String val);
   public BigDecimal(double val);
   public BigDecimal(BigInteger unscaledVal, int scale);
// Открытые константы
   public static final int ROUND_CEILING;                                    // =2
   public static final int ROUND_DOWN;                                       // =1
   public static final int ROUND_FLOOR;                                      // =3
   public static final int ROUND_HALF_DOWN;                                  // =5
   public static final int ROUND_HALF_EVEN;                                  // =6
   public static final int ROUND_HALF_UP;                                    // =4
   public static final int ROUND_UNNECESSARY;                                // =7
   public static final int ROUND_UP;                                         // =0
// Открытые методы класса
   public static BigDecimal valueOf(long val);
   public static BigDecimal valueOf(long unscaledVal, int scale);
// Открытые методы экземпляра
   public BigDecimal abs();
   public BigDecimal add(BigDecimal val);
   public int compareTo(BigDecimal val);
   public BigDecimal divide(BigDecimal val, int roundingMode);
   public BigDecimal divide(BigDecimal val, int scale, int roundingMode);
   public BigDecimal max(BigDecimal val);
   public BigDecimal min(BigDecimal val);
   public BigDecimal movePointLeft(int n);
   public BigDecimal movePointRight(int n);
   public BigDecimal multiply(BigDecimal val);
   public BigDecimal negate();
   public int scale();
   public BigDecimal setScale(int scale);
   public BigDecimal setScale(int scale, int roundingMode);
   public int signum();
   public BigDecimal subtract(BigDecimal val);
   public BigInteger toBigInteger();
1.2 public BigInteger unscaledValue();
// Методы, реализующие методы класса Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы класса Number
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
// Открытые методы, замещающие Object
   public boolean equals(Object x);
   public int hashCode();
   public String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

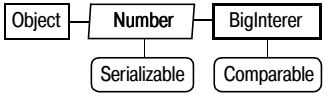
Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: `org.omg.CORBA.FixedHolder.value`

BigInteger Java 1.1

java.math ***сериализуемый, сравнимый***

Этот класс является потомком `java.lang.Number` и представляет целые числа произвольной длины, которая не ограничена 64 битами, как в типе `long`. Класс `BigInteger` содержит методы, дублирующие стандартные арифметические и побитовые операции Java. Метод `compareTo()` сравнивает два объекта `BigInteger` и возвращает результат сравнения в виде чисел –1, 0 или 1. Методы `gcd()`, `modPow()`, `modInverse()` и `isProbablePrime()` выполняют специальные операции и обычно применяются криптографическими и связанными с ними алгоритмами.




```
public class BigInteger extends Number
       implements Comparable {
// Открытые конструкторы
   public BigInteger(byte[] val);
   public BigInteger(String val);
   public BigInteger(String val, int radix);
   public BigInteger(int signum, byte[] magnitude);
   public BigInteger(int numBits, java.util.Random rnd);
   public BigInteger(int bitLength, int certainty, java.util.Random rnd);
// Открытые константы
1.2 public static final BigInteger ONE;
1.2 public static final BigInteger ZERO;
// Открытые методы класса
1.4 public static BigInteger probablePrime(int bitLength, java.util.Random rnd);
   public static BigInteger valueOf(long val);
// Открытые методы экземпляра
   public BigInteger abs();
   public BigInteger add(BigInteger val);
   public BigInteger and(BigInteger val);
   public BigInteger andNot(BigInteger val);
   public int bitCount();
   public int bitLength();
   public BigInteger clearBit(int n);
   public int compareTo(BigInteger val);
   public BigInteger divide(BigInteger val);
   public BigInteger[] divideAndRemainder(BigInteger val);
   public BigInteger flipBit(int n);
   public BigInteger gcd(BigInteger val);
   public int getLowestSetBit();
   public boolean isProbablePrime(int certainty);
   public BigInteger max(BigInteger val);
   public BigInteger min(BigInteger val);
```

```
   public BigInteger mod(BigInteger m);
   public BigInteger modInverse(BigInteger m);
   public BigInteger modPow(BigInteger exponent, BigInteger m);
   public BigInteger multiply(BigInteger val);
   public BigInteger negate();
   public BigInteger not();
   public BigInteger or(BigInteger val);
   public BigInteger pow(int exponent);
   public BigInteger remainder(BigInteger val);
   public BigInteger setBit(int n);
   public BigInteger shiftLeft(int n);
   public BigInteger shiftRight(int n);
   public int signum();
   public BigInteger subtract(BigInteger val);
   public boolean testBit(int n);
   public byte[] toByteArray();
   public String toString(int radix);
   public BigInteger xor(BigInteger val);
// Методы, реализующие Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие методы класса Number
   public double doubleValue();
   public float floatValue();
   public int intValue();
   public long longValue();
// Открытые методы, замещающие методы класса Object
   public boolean equals(Object x);
   public int hashCode();
   public String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: `BigInteger.{ONE,ZERO}, java.security.spec.RSAKeyGenParameterSpec.{F0, F4}`

Глава 13

java.net

Пакет java.net

Java 1.0

Пакет `java.net` содержит мощную и гибкую инфраструктуру для решения сетевых задач. В следующих разделах содержится краткое описание наиболее распространенных классов. Нужно отметить, что в Java 1.4 появились новые пакеты `java.nio` и `java.nio.channels`, реализующие новый API ввода/вывода. Их можно применять для организации высокоэффективной работы в сети без образования блокировок. См. также описание пакета `javax.net.ssl`, который содержит классы для организации сетевой защиты с использованием SSL.

Класс `URL` представляет унифицированный указатель информационного ресурса (URL). Он обеспечивает простой интерфейс для сетевого подключения: объект, на который ссылается URL, можно загрузить с помощью одного запроса; кроме того, можно открыть потоки для чтения или записи этого объекта. Из данного объекта `URL` можно получить объект более высокого уровня `URLConnection`. Класс `URLConnection` содержит дополнительные методы, позволяющие работать с URL на более сложном уровне. В Java 1.4 добавлен класс `URI`; в нем содержатся мощные средства для работы с URI- и URL-строками, но в нем нет собственных сетевых средств.

Если нужно не просто загрузить объект, на который ссылается URL, а произвести более сложные операции, то следует применять класс `Socket`. Этот класс позволяет подключиться к указанному порту заданного хоста в Интернете и прочитать данные с помощью классов `InputStream` и `OutputStream` пакета `java.io`. Для создания сервера, к которому будут подключаться клиенты, можно задействовать класс `ServerSocket`. Классы `Socket` и `ServerSocket` применяют класс `InetAddress`, представляющий интернет-адрес. В Java 1.4 добавлены потомки этого класса – подклассы `Inet4Address` и `Inet6Address`, представляющие адреса, которые соответствуют версиям 4 и 6 протокола IP. Кроме того, в Java 1.4 есть класс `SocketAddress`, который на более высоком уровне представляет сетевой адрес без привязки к протоколу. Потомок этого класса `InetSocketAddress` инкапсулирует `InetAddress` и номер порта.

Пакет `java.net` позволяет выполнять сетевые задачи с помощью объектов `DatagramPacket`, которые можно посылать и получать по сети через объект `DatagramSocket`. Класс `MulticastSocket` расширяет `DatagramSocket`, добавляя поддержку многоадресной (multicast) передачи.

Интерфейсы

```
public interface ContentHandlerFactory;
public interface DatagramSocketImplFactory;
public interface FileNameMap;
public interface SocketImplFactory;
public interface SocketOptions;
public interface URLStreamHandlerFactory;
```

Классы

```
public abstract class Authenticator;
public abstract class ContentHandler;
public final class DatagramPacket;
public class DatagramSocket;
   └ public class MulticastSocket extends DatagramSocket;
public abstract class DatagramSocketImpl implements SocketOptions;
public class InetAddress implements Serializable;
   └ public final class Inet4Address extends InetAddress;
   └ public final class Inet6Address extends InetAddress;
public final class NetPermission extends java.security.BasicPermission;
public final class NetworkInterface;
public final class PasswordAuthentication;
public class ServerSocket;
public class Socket;
public abstract class SocketAddress implements Serializable;
   └ public class InetSocketAddress extends SocketAddress;
public abstract class SocketImpl implements SocketOptions;
public final class SocketPermission extends java.security.Permission implements Serializable;
public final class URI implements Comparable, Serializable;
public final class URL implements Serializable;
public class URLClassLoader extends java.security.SecureClassLoader;
public abstract class URLConnection;
   └ public abstract class HttpURLConnection extends URLConnection;
   └ public abstract class JarURLConnection extends URLConnection;
public class URLDecoder;
public class URLEncoder;
public abstract class URLStreamHandler;
```

Исключения

```
public class MalformedURLException extends java.io.IOException;
public class ProtocolException extends java.io.IOException;
public class SocketException extends java.io.IOException;
   └ public class BindException extends SocketException;
   └ public class ConnectException extends SocketException;
   └ public class NoRouteToHostException extends SocketException;
   └ public class PortUnreachableException extends SocketException;
public class SocketTimeoutException extends java.io.InterruptedIOException;
public class UnknownHostException extends java.io.IOException;
public class UnknownServiceException extends java.io.IOException;
public class URISyntaxException extends Exception;
```

Authenticator Java 1.2

java.net

В этом абстрактном классе определен настраиваемый механизм для запроса и предоставления пароля URL. Статический метод setDefault() устанавливает системный объект Authenticator, который может получать от пользователя требуемую аутентификационную информацию (в консольном или графическом интерфейсе по выбору). Метод setDefault() может быть вызван только один раз; последующие вызовы игнорируются. Для вызова этого метода необходимо иметь право NetPermission.

Когда приложение или среда выполнения Java запрашивают пароль (например, на чтение указанного URL), вызывается метод requestPasswordAuthentication(), которому в качестве аргументов передаются хост, порт и текст приглашения пользователю. Этот метод ищет системный объект Authenticator, используемый по умолчанию, и вызывает его метод getPasswordAuthentication(). Вызов метода requestPasswordAuthentication() требует соответствующего права типа NetPermission.

Класс Authenticator абстрактный; исходная реализация его метода getPasswordAuthentication() всегда возвращает null. Чтобы создать Authenticator, нужно заменить этот метод таким образом, чтобы он выводил приглашение ввести пароль и имя пользователя и возвращал полученную информацию в виде объекта PasswordAuthentication. В реализации метода getPasswordAuthentication() можно применять различные варианты метода getRequesting(), чтобы установить сторону, запрашивающую пароль, и получить рекомендуемый текст приглашения. В Java 1.4 метод requestPasswordAuthentication() был переопределен, и теперь он позволяет указать имя хоста, запрашивающего пароль. Также был добавлен метод экземпляра getRequestingHost().

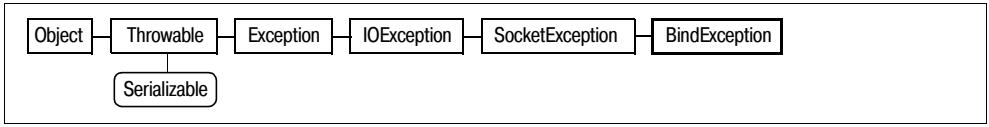
```
public abstract class Authenticator {
// Открытые конструкторы
   public Authenticator();
// Открытые методы класса
   public static PasswordAuthentication requestPasswordAuthentication(InetAddress addr, int port,
               String protocol, String prompt, String scheme);
1.4 public static PasswordAuthentication requestPasswordAuthentication(String host, InetAddress
addr, int port, String protocol, String prompt, String scheme);
   public static void setDefault(Authenticator a);                          // синхронизирован
// Защищенные методы экземпляра
   protected PasswordAuthentication getPasswordAuthentication();            // константа
1.4 protected final String getRequestingHost();
   protected final int getRequestingPort();
   protected final String getRequestingPrompt();
   protected final String getRequestingProtocol();
   protected final String getRequestingScheme();
   protected final InetAddress getRequestingSite();
}
```

Передается методам: Authenticator.setDefault()

BindException Java 1.1

java.net ***сериализуемое, проверяемое***

Это исключение сообщает о том, что сокет не удается связать с локальным адресом и портом. Часто оно возникает из-за того, что порт уже используется.

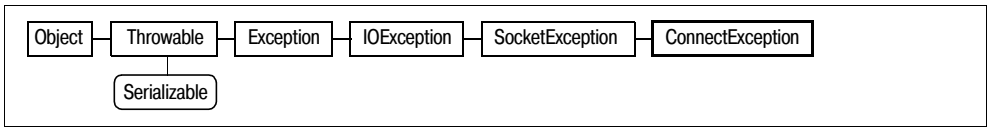



```
public class BindException extends SocketException {
// Открытые конструкторы
   public BindException();
   public BindException(String msg);
}
```

ConnectException Java 1.1

java.net ***сериализуемое, проверяемое***

Это исключение возникает, когда сокет не может соединиться с удаленным адресом или портом. При этом удаленный хост достижим, но не отвечает – возможно, по причине того, что на нем нет процесса, слушающего данный порт.




```
public class ConnectException extends SocketException {
// Открытые конструкторы
   public ConnectException();
   public ConnectException(String msg);
}
```

ContentHandler Java 1.0

java.net

В этом абстрактном классе определены методы, читающие данные из объекта `URLConnection` и возвращающие объект, содержащий эти данные. Все потомки этого класса, которые реализуют его методы, должны выполнять обработку данных (например, MIME). Приложения никогда напрямую не создают объекты `ContentHandler`; при необходимости они создаются зарегистрированным объектом `ContentHandlerFactory`. В приложениях не следует непосредственно вызывать методы класса `ContentHandler`; вместо этого нужно задействовать `URL.getContent()` и `URLConnection.getContent()`. Создавать потомки класса `ContentHandler` приходится только при написании веб-броузера или подобного приложения, которое должно анализировать данные.

```
public abstract class ContentHandler {
// Открытые конструкторы
   public ContentHandler();
// Открытые методы экземпляра
   public abstract Object getContent(URLConnection urlc) throws java.io.IOException;
1.3 public Object getContent(URLConnection urlc, Class[] classes) throws java.io.IOException;
}
```

Возвращается методами: `ContentHandlerFactory.createContentHandler()`

ContentHandlerFactory Java 1.0

java.net

В этом интерфейсе определен метод, который должен создавать и возвращать соответствующий объект ContentHandler для указанного типа MIME. Системный интерфейс ContentHandlerFactory можно указать с помощью метода URLConnection.setContentHandlerFactory(). В обычных приложениях никогда не требуется применять или реализовывать этот интерфейс.

```
public interface ContentHandlerFactory {
// Открытые методы экземпляра
   public abstract java.net.ContentHandler createContentHandler(String mimetype);
}
```

Передается методам: URLConnection.setContentHandlerFactory

DatagramPacket Java 1.0

java.net

Этот класс реализует пакет данных, который можно принять или отправить по сети через сокет DatagramSocket. Создать объект DatagramPacket для отправляемого пакета можно с помощью одного из конструкторов, принимающих в качестве аргумента сетевой адрес. Чтобы создать DatagramPacket для принимаемого пакета, используйте конструкторы без аргумента, определяющего сетевой адрес. Метод receive() класса DatagramSocket ожидает прихода данных и сохраняет их в объекте DatagramPacket, созданном вышеуказанным способом. Содержимое пакета и его отправителя можно узнать с помощью методов экземпляра класса DatagramPacket.

В Java 1.4 в этот класс добавлены новые конструкторы и методы для поддержки абстракции SocketAddress сетевого адреса.

```
public final class DatagramPacket {синхронизирован
// Открытые конструкторы
   public DatagramPacket(byte[] buf, int length);
1.4 public DatagramPacket(byte[] buf, int length, SocketAddress address) throws SocketException;
1.2 public DatagramPacket(byte[] buf, int offset, int length);
   public DatagramPacket(byte[] buf, int length, InetAddress address, int port);
1.4 public DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)
                                  throws SocketException;
1.2 public DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port);
// Методы доступа к свойствам (по именам свойств)
   public InetAddress getAddress();                          // синхронизирован
1.1 public void setAddress(InetAddress iaddr);               // синхронизирован
   public byte[] getData();                                  // синхронизирован
1.1 public void setData(byte[] buf);                         // синхронизирован
1.2 public void setData(byte[] buf, int offset, int length); // синхронизирован
   public int getLength();                                   // синхронизирован
1.1 public void setLength(int length);                       // синхронизирован
1.2 public int getOffset();                                  // синхронизирован
   public int getPort();                                     // синхронизирован
1.1 public void setPort(int iport);                          // синхронизирован
1.4 public SocketAddress getSocketAddress();                 // синхронизирован
1.4 public void setSocketAddress(SocketAddress address);     // синхронизирован
}
```

Передается методам: DatagramSocket.{receive(), send()}, DatagramSocketImpl.{peekData(), receive(), send()}, MulticastSocket.send()

DatagramSocket Java 1.0

java.net

Этот класс определяет сокет, который может получать и отправлять ненадежные дейтаграммы по сети, используя протокол UDP. *Дейтаграмма* – это низкоуровневый сетевой интерфейс: она представляет собой массив байтов, пересылаемый по сети. Дейтаграмма не реализует никакого протокола обмена данными, основанного на потоках, и не устанавливает соединение между получателем и отправителем. Пакеты дейтаграммы считаются ненадежными, потому что протокол не предпринимает попытки проверить, дошли ли они по назначению, и должным образом прореагировать в случае неудачи. Таким образом, не гарантируется, что пакеты придут в место назначения в том же порядке, в каком они были отправлены, и что они вообще придут. С другой стороны, функциональная простота этого протокола обуславливает высокую скорость передачи дейтаграммы. См. также классы Socket и URL, предоставляющие высокоуровневые интерфейсы для выполнения сетевых задач. Рассматриваемый класс появился в Java 1.0, а в Java 1.4 в него была добавлена возможность указывать локальный и удаленный адрес с помощью класса SocketAddress, не зависящего от протокола.

Метод send() отправляет пакет DatagramPacket через сокет. Пакет должен содержать адрес получателя. Метод receive() ожидает прибытия данных в сокет и сохраняет их вместе с адресом получателя в указанном объекте DatagramPacket. Метод close() закрывает сокет и освобождает локальный порт. После вызова close() объект DatagramSocket можно применять только для вызова метода isClosed(), который возвращает true, если сокет был закрыт.

Каждый раз, когда пакет отправляется или принимается, система безопасности должна проверить, имеет ли данная программа разрешение на отправление или получение данных из указанного хоста. Если в Java 1.2 и последующих версиях требуется отправить или принять несколько пакетов из одного хоста, то можно задействовать метод connect(), чтобы указать хост, с которым идет обмен данными. При этом проверка безопасности производится только один раз, но до вызова метода disconnect() сокету не разрешается связываться ни с каким другим хостом. Применяйте методы getRemoteSocketAddress(), getInetAddress() и getPort(), чтобы получить сетевой адрес, с которым соединен данный сокет, если он вообще соединен. Определить это можно с помощью метода isConnected().

По умолчанию DatagramSocket посылает данные через локальный адрес, назначенный системой. При необходимости можно связать сокет с указанным локальным адресом. Для этого нужно применить один из конструкторов с аргументами или связать DatagramSocket с локальным адресом SocketAddress с помощью метода bind(). Метод isBound() позволяет определить, связан ли сокет с каким-нибудь адресом, а получить его локальный адрес можно с помощью getLocalSocketAddress() или getLocalAddress() и getLocalPort().

Этот класс определяет несколько пар методов get/set, устанавливающих или запрашивающих различные «параметры сокета», которые учитываются при пересылке дейтаграммы. setSoTimeout() устанавливает время в миллисекундах, в течение которого метод receive() ожидает прибытия пакета. По прошествии этого времени он вызывает исключение InterruptedIOException. Для бесконечного ожидания укажите 0 миллисекунд. С помощью методов setSendBufferSize() и setReceiveBufferSize() мож-

но указать системе рекомендуемый размер соответствующего сетевого буфера. Методы setBroadcast(), setReuseAddress() и setTrafficClass() устанавливают более сложные параметры сокета, использование которых требует от программиста глубокого понимания сетевых протоколов, а их описание выходит за рамки нашего справочника.

В Java 1.4 и выше метод getChannel() возвращает объект java.nio.channels.DatagramChannel, связанный с данным объектом DatagramSocket. В сокетах, созданных с помощью одного из конструкторов DatagramSocket(), этот метод всегда возвращает null, а в сокетах, которые были созданы объектом DatagramChannel и принадлежат ему, этот метод возвращает соответствующее значение.

```
public class DatagramSocket {
// Открытые конструкторы
   public DatagramSocket() throws SocketException;
1.4 public DatagramSocket(SocketAddress bindaddr) throws SocketException;
   public DatagramSocket(int port) throws SocketException;
1.1 public DatagramSocket(int port, InetAddress laddr) throws SocketException;
// Защищенные конструкторы
1.4 protected DatagramSocket(DatagramSocketImpl impl);
// Открытые методы класса
1.3 public static void setDatagramSocketImplFactory(DatagramSocketImplFactory fac)
        throws java.io.IOException;                                    // синхронизирован
// Методы доступа к свойствам (по именам свойств)
1.4 public boolean isBound();                                          // по умолчанию:true
1.4 public boolean getBroadcast() throws SocketException;   // синхронизирован; по умолчанию:true
1.4 public void setBroadcast(boolean on) throws SocketException;       // синхронизирован
1.4 public java.nio.channels.DatagramChannel getChannel();       // константа; по умолчанию:null
1.4 public boolean isClosed();                                         // по умолчанию:false
1.4 public boolean isConnected();                                      // по умолчанию:false
1.2 public InetAddress getInetAddress();                               // по умолчанию:null
1.1 public InetAddress getLocalAddress();                        // по умолчанию:Inet4Address
   public int getLocalPort();
1.4 public SocketAddress getLocalSocketAddress();              // по умолчанию:InetSocketAddress
1.2 public int getPort();                                              // по умолчанию:-1
1.2 public int getReceiveBufferSize() throws
                              SocketException;          // синхронизирован; по умолчанию:32767
1.2 public void setReceiveBufferSize(int size) throws SocketException;  // синхронизирован
1.4 public SocketAddress getRemoteSocketAddress();                     // по умолчанию:null
1.4 public boolean getReuseAddress() throws
                               SocketException;         // синхронизирован; по умолчанию:false
1.4 public void setReuseAddress(boolean on) throws SocketException;     // синхронизирован
1.2 public int getSendBufferSize() throws SocketException;  // синхронизирован; по умолчанию:32767
1.2 public void setSendBufferSize(int size) throws SocketException;     // синхронизирован
1.1 public int getSoTimeout() throws SocketException;          // синхронизирован; по умолчанию:0
1.1 public void setSoTimeout(int timeout) throws SocketException;       // синхронизирован
1.4 public int getTrafficClass() throws SocketException;       // синхронизирован; по умолчанию:0
1.4 public void setTrafficClass(int tc) throws SocketException;         // синхронизирован
// Открытые методы экземпляра
1.4 public void bind(SocketAddress addr) throws SocketException;        // синхронизирован
   public void close();
1.4 public void connect(SocketAddress addr) throws SocketException;
1.2 public void connect(InetAddress address, int port);
1.2 public void disconnect();
   public void receive(DatagramPacket p) throws java.io.IOException;    // синхронизирован
   public void send(DatagramPacket p) throws java.io.IOException;
}
```

Подклассы: MulticastSocket

Возвращается методами: java.nio.channels.DatagramChannel.socket()

DatagramSocketImpl Java 1.1

java.net

В этом абстрактном классе определены методы, необходимые для реализации обмена данными посредством дейтаграмм и многоадресных сокетов. Системные программисты могут создавать потомки этого класса при реализации дейтаграмм или многоадресных сокетов в нестандартном сетевом окружении, например при наличии брандмауэра или при использовании нестандартного транспортного протокола. В обычных приложениях не требуется применять этот класс или создавать его потомков.


```
public abstract class DatagramSocketImpl implements SocketOptions {
// Открытые конструкторы
   public DatagramSocketImpl();
// Защищенные методы экземпляра
   protected abstract void bind(int lport, InetAddress laddr) throws SocketException;
   protected abstract void close();
1.4 protected void connect(InetAddress address, int port) throws SocketException;   // пустой
   protected abstract void create() throws SocketException;
1.4 protected void disconnect();                                                    // пустой
   protected java.io.FileDescriptor getFileDescriptor();
   protected int getLocalPort();
1.2 protected abstract int getTimeToLive() throws java.io.IOException;
   protected abstract void join(InetAddress inetaddr) throws java.io.IOException;
1.4 protected abstract void joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)
        throws java.io.IOException;
   protected abstract void leave(InetAddress inetaddr) throws java.io.IOException;
1.4 protected abstract void leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf)
        throws java.io.IOException;
   protected abstract int peek(InetAddress i) throws java.io.IOException;
1.4 protected abstract int peekData(DatagramPacket p) throws java.io.IOException;
   protected abstract void receive(DatagramPacket p) throws java.io.IOException;
   protected abstract void send(DatagramPacket p) throws java.io.IOException;
1.2 protected abstract void setTimeToLive(int ttl) throws java.io.IOException;
// Защищенные поля экземпляра
   protected java.io.FileDescriptor fd;
   protected int localPort;
// Устаревшие защищенные методы
# protected abstract byte getTTL() throws java.io.IOException;
# protected abstract void setTTL(byte ttl) throws java.io.IOException;
}
```

Передается методам: DatagramSocket.DatagramSocket()

Возвращается методами: DatagramSocketImplFactory.createDatagramSocketImpl()

DatagramSocketImplFactory Java 1.3

java.net

Этот интерфейс определяет метод, который должен создавать объекты DatagramSocketImpl. Можно зарегистрировать экземпляр этого интерфейса-фабрики с помощью статического метода setDatagramSocketImplFactory() класса DatagramSocket. Данный интерфейс никогда не следует применять или реализовывать на уровне приложений.

```
public interface DatagramSocketImplFactory {
// Открытые методы экземпляра
   public abstract DatagramSocketImpl createDatagramSocketImpl();
}
```

Передается методам: DatagramSocket.setDatagramSocketImplFactory()

FileNameMap Java 1.1

java.net

В этом интерфейсе определен единственный метод, который предназначен для получения типа формата MIME, в котором представлены данные файла, на основе имени этого файла. Поле fileNameMap класса URLConnection ссылается на объект, реализующий этот интерфейс. Правило установления соответствия между именем файла и его типом, реализованное в таком объекте, используется статическим методом URLConnection.guessContentTypeFromName().

```
public interface FileNameMap {
// Открытые методы экземпляра
   public abstract String getContentTypeFor(String fileName);
}
```

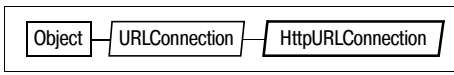
Передается методам: URLConnection.setFileNameMap()

Возвращается методами: URLConnection.getFileNameMap()

HttpURLConnection Java 1.1

java.net

Этот класс конкретизирует URLConnection. Экземпляр данного класса возвращается методом openConnection() объекта URL, использующего протокол HTTP. Многочисленные константы, определенные в этом классе, являются кодами состояния, которые возвращаются HTTP-серверами. Метод setRequestMethod() указывает тип запроса HTTP. Содержимое этого запроса должно быть передано через поток OutputStream, возвращаемый методом getOutputStream() родительского класса. После того как послан HTTP-запрос, метод getResponseCode() возвращает целое число, содержащее ответный код сервера, а getResponseMessage() возвращает ответное сообщение сервера. Метод disconnect() закрывает соединение. С помощью статического метода setFollowRedirects() можно разрешить автоматическую переадресацию для соединений URL, использующих протокол HTTP. Для успешного использования этого класса требуется знание протокола HTTP.



```
public abstract class HttpURLConnection extends URLConnection {
// Защищенные конструкторы
   protected HttpURLConnection(URL u);
// Открытые константы
   public static final int HTTP_ACCEPTED;                                    // =202
   public static final int HTTP_BAD_GATEWAY;                                 // =502
   public static final int HTTP_BAD_METHOD;                                  // =405
   public static final int HTTP_BAD_REQUEST;                                 // =400
   public static final int HTTP_CLIENT_TIMEOUT;                              // =408
   public static final int HTTP_CONFLICT;                                    // =409
   public static final int HTTP_CREATED;                                     // =201
   public static final int HTTP_ENTITY_TOO_LARGE;                            // =413
   public static final int HTTP_FORBIDDEN;                                   // =403
   public static final int HTTP_GATEWAY_TIMEOUT;                             // =504
   public static final int HTTP_GONE;                                        // =410
   public static final int HTTP_INTERNAL_ERROR;                              // =500
   public static final int HTTP_LENGTH_REQUIRED;                             // =411
   public static final int HTTP_MOVED_PERM;                                  // =301
   public static final int HTTP_MOVED_TEMP;                                  // =302
   public static final int HTTP_MULT_CHOICE;                                 // =300
   public static final int HTTP_NO_CONTENT;                                  // =204
   public static final int HTTP_NOT_ACCEPTABLE;                              // =406
   public static final int HTTP_NOT_AUTHORITATIVE;                           // =203
   public static final int HTTP_NOT_FOUND;                                   // =404
1.3 public static final int HTTP_NOT_IMPLEMENTED;                            // =501
   public static final int HTTP_NOT_MODIFIED;                                // =304
   public static final int HTTP_OK;                                          // =200
   public static final int HTTP_PARTIAL;                                     // =206
   public static final int HTTP_PAYMENT_REQUIRED;                            // =402
   public static final int HTTP_PRECON_FAILED;                               // =412
   public static final int HTTP_PROXY_AUTH;                                  // =407
   public static final int HTTP_REQ_TOO_LONG;                                // =414
   public static final int HTTP_RESET;                                       // =205
   public static final int HTTP_SEE_OTHER;                                   // =303
   public static final int HTTP_UNAUTHORIZED;                                // =401
   public static final int HTTP_UNAVAILABLE;                                 // =503
   public static final int HTTP_UNSUPPORTED_TYPE;                            // =415
   public static final int HTTP_USE_PROXY;                                   // =305
   public static final int HTTP_VERSION;                                     // =505
// Открытые методы класса
   public static boolean getFollowRedirects();
   public static void setFollowRedirects(boolean set);
// Методы доступа к свойствам (по именам свойств)
1.2 public java.io.InputStream getErrorStream();                             // константа
1.3 public boolean getInstanceFollowRedirects();
1.3 public void setInstanceFollowRedirects(boolean followRedirects);
1.2 public java.security.Permission getPermission() throws java.io.IOException;    // Замещает:URLConnection
   public String getRequestMethod();
   public void setRequestMethod(String method) throws ProtocolException;
   public int getResponseCode() throws java.io.IOException;                  // константа
   public String getResponseMessage() throws java.io.IOException;
// Открытые методы экземпляра
   public abstract void disconnect();
   public abstract boolean usingProxy();
// Открытые методы, замещающие URLConnection
1.3 public long getHeaderFieldDate(String name, long Default);
```

```
// Защищенные поля экземпляра
1.3 protected boolean instanceFollowRedirects;
   protected String method;
   protected int responseCode;
   protected String responseMessage;
// Устаревшие открытые поля
# public static final int HTTP_SERVER_ERROR;  // =500
}
```

Подклассы: javax.net.ssl.HttpsURLConnection

Inet4Address

Java 1.4

java.net ***сериализуемый***

В классе Inet4Address унаследованные методы приспособлены для работы с Internet Protocol версии 4 (IPv4). В этом классе нет конструктора. Создать его экземпляр можно с помощью статических методов класса InetAddress, возвращающих объекты Inet4Address или Inet6Address соответственно.

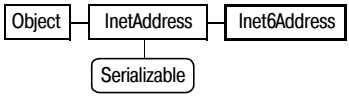

```
public final class Inet4Address extends InetAddress {
// Конструктор отсутствует
// Открытые методы, замещающие InetAddress
   public boolean equals(Object obj);
   public byte[] getAddress();
   public String getHostAddress();
   public int hashCode();
   public boolean isAnyLocalAddress();
   public boolean isLinkLocalAddress();
   public boolean isLoopbackAddress();
   public boolean isMCGlobal();
   public boolean isMCLinkLocal();
   public boolean isMCNodeLocal();  // константа
   public boolean isMCOrgLocal();
   public boolean isMCSiteLocal();
   public boolean isMulticastAddress();
   public boolean isSiteLocalAddress();
}
```

Inet6Address

Java 1.4

java.net ***сериализуемый***

В классе Inet6Address унаследованные методы приспособлены для использования с Internet Protocol версии 6 (IPv6). Полное описание адресов этого типа представлено в RFC 2373. В этом классе нет конструктора. Создать его экземпляр можно с помощью статических методов класса InetAddress, возвращающих объекты Inet4Address или Inet6Address соответственно.




```
public final class Inet6Address extends InetAddress {
// Конструктор отсутствует
// Открытые методы экземпляра
   public boolean isIPv4CompatibleAddress();
// Открытые методы, замещающие InetAddress
   public boolean equals(Object obj);
   public byte[] getAddress();
   public String getHostAddress();
   public int hashCode();
   public boolean isAnyLocalAddress();
   public boolean isLinkLocalAddress();
   public boolean isLoopbackAddress();
   public boolean isMCGlobal();
   public boolean isMCLinkLocal();
   public boolean isMCNodeLocal();
   public boolean isMCOrgLocal();
   public boolean isMCSiteLocal();
   public boolean isMulticastAddress();
   public boolean isSiteLocalAddress();
}
```

InetAddress

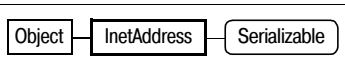
Java 1.0

java.net ***сериализуемый***

Этот класс представляет IP-адрес. Класс не имеет открытого конструктора, но поддерживает методы-фабрики, возвращающие объекты `InetAddress`. Метод `getLocalHost()` возвращает `InetAddress`, содержащий IP-адрес локального компьютера. `getByName()` возвращает `InetAddress` с адресом указанного хоста. `getAllByName()` возвращает массив объектов `InetAddress`, представляющих все доступные адреса хоста с указанным именем. Метод `getByAddress()` возвращает объект `InetAddress`, представляющий IP-адрес, который указан в виде массива байтовых значений.

После того как получен объект `InetAddress`, можно воспользоваться его методами экземпляра для получения различного рода информации об этом объекте. Два наиболее важных метода – это `getHostName()`, возвращающий имя хоста, и `getAddress()`, возвращающий IP-адрес в виде массива байтовых значений, в котором первым элементом является старший байт адреса. Метод `getHostAddress()` возвращает IP-адрес в виде строки. Различные методы, начинающиеся на `is-`, определяют принадлежность адреса определенным категориям. Методы, начинающиеся на `isMC-`, связаны с групповыми адресами (multicast addresses).

Этот класс изначально появился в Java 1.0, а в Java 1.4 в него было добавлено много методов, где также были определены потомки этого класса `Inet4Address` и `Inet6Address`, представляющие адреса IPv4 и IPv6 (4 и 6 версии) соответственно.



```
public class InetAddress implements Serializable {
// Конструктор отсутствует
// Открытые методы класса
   public static InetAddress[] getAllByName(String host)
         throws java.net.UnknownHostException;
1.4 public static InetAddress getByAddress(byte[] addr)
         throws java.net.UnknownHostException;
1.4 public static InetAddress getByAddress(String host, byte[] addr)
         throws java.net.UnknownHostException;
   public static InetAddress getByName(String host) throws java.net.UnknownHostException;
   public static InetAddress getLocalHost() throws java.net.UnknownHostException; // синхронизирован
// Методы доступа к свойствам (по именам свойств)
   public byte[] getAddress();                                          // константа
1.4 public boolean isAnyLocalAddress();                                 // константа
1.4 public String getCanonicalHostName();
   public String getHostAddress();                                      // константа
   public String getHostName();
1.4 public boolean isLinkLocalAddress();                                // константа
1.4 public boolean isLoopbackAddress();                                 // константа
1.4 public boolean isMCGlobal();                                        // константа
1.4 public boolean isMCLinkLocal();                                     // константа
1.4 public boolean isMCNodeLocal();                                     // константа
1.4 public boolean isMCOrgLocal();                                      // константа
1.4 public boolean isMCSiteLocal();                                     // константа
1.1 public boolean isMulticastAddress();                                // константа
1.4 public boolean isSiteLocalAddress();                                // константа
// Открытые методы, замещающие Object
   public boolean equals(Object obj);                                   // константа
   public int hashCode();                                               // константа
   public String toString();
}
```

Подклассы: `Inet4Address`, `Inet6Address`

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

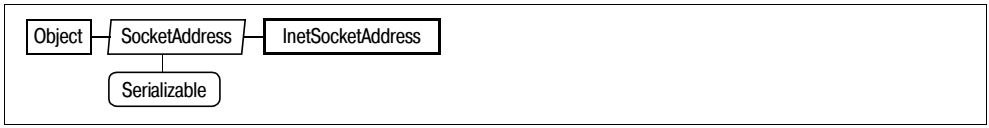
Экземпляры: `SocketImpl.address`

InetSocketAddress

Java 1.4

java.net ***сериализуемый***

Класс `InetSocketAddress` представляет IP-адрес и номер порта. В конструкторах можно указать IP-адрес в виде объекта `InetAddress` или в виде имени хоста; если этот параметр опустить, то будет применяться группа адресов (например, серверного сокета).




```
public class InetSocketAddress extends SocketAddress {
// Открытые конструкторы
   public InetSocketAddress(int port);
   public InetSocketAddress(InetAddress addr, int port);
```

```
    public InetSocketAddress(String hostname, int port);
// Открытые методы экземпляра
    public final InetAddress getAddress();
    public final String getHostName();
    public final int getPort();
    public final boolean isUnresolved();
// Открытые методы, замещающие Object
    public final boolean equals(Object obj);
    public final int hashCode();
    public String toString();
}
```

JarURLConnection

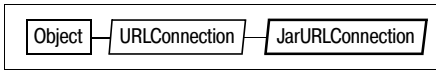
Java 1.2

java.net

Этот класс конкретизирует `URLConnection` и представляет соединение с объектом типа `jar:` URL. Это сложный URL, содержащий адрес архива `jar` и (необязательно) ссылку на файл или каталог внутри архива. В строке `jar:` URL для разделения пути к архиву и имени файла внутри JAR применяется символ «!». Обратите внимание, что в `jar:` URL содержится имя еще одного протокола, который извлекает сам файл JAR. Примеры:

```
jar:http://my.jar.com/my.jar!/                    // Весь архив JAR
jar:file:/usr/java/lib/my.jar!/com/jar/           // Каталог внутри архива JAR
jar:ftp://ftp.jar.com/pub/my.jar!/com/jar/Jar.class // Файл внутри архива
```

Чтобы получить `JarURLConnection`, определите требуемый объект `URL` для `jar:` URL, далее методом `openConnection()` откройте соединение с этим URL и приведите полученный объект `URLConnection` к типу `JarURLConnection`. Различные методы, определенные в классе `JarURLConnection`, позволяют читать файл манифеста для всего архива или для отдельных элементов этого архива. В данных методах применяются различные классы из пакета `java.util.jar`.

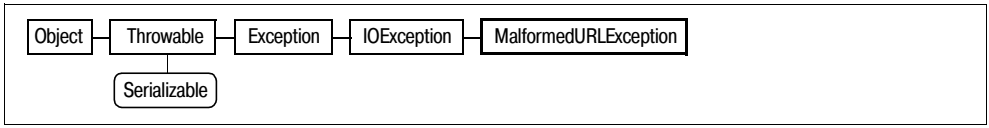



```
public abstract class JarURLConnection extends URLConnection {
// Защищенные конструкторы
    protected JarURLConnection(URL url) throws MalformedURLException;
// Методы доступа к свойствам (по именам свойств)
    public java.util.jar.Attributes getAttributes() throws java.io.IOException;
    public java.security.cert.Certificate[] getCertificates() throws java.io.IOException;
    public String getEntryName();
    public java.util.jar.JarEntry getJarEntry() throws java.io.IOException;
    public abstract java.util.jar.JarFile getJarFile() throws java.io.IOException;
    public URL getJarFileURL();
    public java.util.jar.Attributes getMainAttributes() throws java.io.IOException;
    public java.util.jar.Manifest getManifest() throws java.io.IOException;
// Защищенные поля экземпляра
    protected URLConnection jarFileURLConnection;
}
```

MalformedURLException Java 1.0

java.net ***сериализуемое, проверяемое***

Это исключение возникает при передаче методу ссылки URL, анализ которой завершился неудачей.



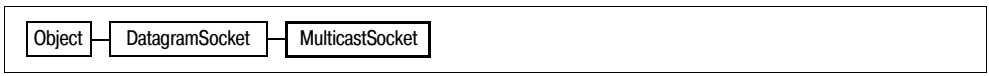

```
public class MalformedURLException extends java.io.IOException {
// Открытые конструкторы
   public MalformedURLException();
   public MalformedURLException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

MulticastSocket Java 1.1

java.net

Этот класс является потомком DatagramSocket и может отправлять или получать многоадресные пакеты UDP. Он дополняет класс DatagramSocket методами joinGroup() и leaveGroup(), с помощью которых можно войти в группу широковещания или покинуть ее. Методы setInterface() и setNetworkInterface() позволяют указать объекты InetAddress или NetworkInterface соответственно, которые должны использовать исходящие многоадресные пакеты. Такая возможность применяется на серверах или компьютерах, которые имеют несколько Интернет-адресов или сетевых интерфейсов. Метод setLoopbackMode() позволяет определить, будет ли многоадресный пакет, исходящий из данного сокета, отправлен обратно в этот сокет. С аргументом true этот метод запрашивает отмену пакетов обратной связи (но не обязательно отменяет), поэтому для данного метода больше подходит название «setLoopbackModeDisabled()».




```
public class MulticastSocket extends DatagramSocket {
// Открытые конструкторы
   public MulticastSocket() throws java.io.IOException;
1.4 public MulticastSocket(SocketAddress bindaddr) throws java.io.IOException;
   public MulticastSocket(int port) throws java.io.IOException;
// Методы доступа к свойствам (по именам свойств)
   public InetAddress getInterface() throws SocketException;          // по умолчанию:Inet4Address
   public void setInterface(InetAddress inf) throws SocketException;
1.4 public boolean getLoopbackMode() throws SocketException;                  // по умолчанию:true
1.4 public void setLoopbackMode(boolean disable) throws SocketException;
1.4 public NetworkInterface getNetworkInterface() throws SocketException;
1.4 public void setNetworkInterface(NetworkInterface netIf) throws SocketException;
1.2 public int getTimeToLive() throws java.io.IOException;                    // по умолчанию:1
1.2 public void setTimeToLive(int ttl) throws java.io.IOException;
// Открытые методы экземпляра
```

```
   public void joinGroup(InetAddress mcastaddr) throws java.io.IOException;
1.4 public void joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)
        throws java.io.IOException;
   public void leaveGroup(InetAddress mcastaddr) throws java.io.IOException;
1.4 public void leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf)
        throws java.io.IOException;
// Устаревшие открытые методы
# public byte getTTL() throws java.io.IOException;                          // по умолчанию: 1
# public void send(DatagramPacket p, byte ttl) throws java.io.IOException;
# public void setTTL(byte ttl) throws java.io.IOException;
}
```

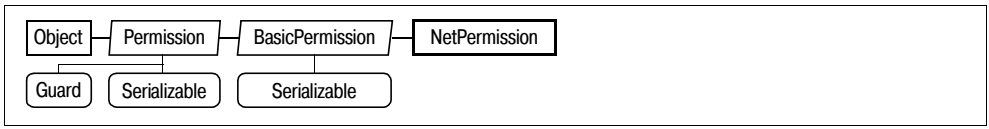
NetPermission

Java 1.2

java.net ***сериализуемый***

Этот класс является потомком `java.security.Permission` и представляет различные права, которые требуются для сетевых средств Java, основанных на URL. См. также описание класса `SocketPermission`, который представляет право на низкоуровневые сетевые операции. Право `NetPermission` определяется только именем; с ним не связан список действий. В Java 1.2 определено три целевых объекта этого права: `setDefaultAuthenticator` нужен для вызова `Authenticator.setDefault()`; `requestPasswordAuthentication` требуется при вызове `Authenticator.requestPasswordAuthentication()`; `specifyStreamHandler` нужен для явной передачи объекта `URLStreamHandler` конструктору `URL()`. Если в качестве имени целевого объекта указан групповой символ «*», применяются все три целевых объекта.

Системные администраторы, которые настраивают политики безопасности, должны знать этот класс и предоставляемые им права. Этот класс может понадобиться системным программистам, а разработчики приложений никогда не должны явно его использовать.




```
public final class NetPermission extends java.security.BasicPermission {
// Открытые конструкторы
   public NetPermission(String name);
   public NetPermission(String name, String actions);
}
```

NetworkInterface

Java 1.4

java.net

Экземпляры этого класса представляют сетевые интерфейсы локальной машины. Методы `getName()` и `getDisplayName()` возвращают имя интерфейса, а `getInetAddresses()` возвращает объект `java.util.Enumeration`, содержащий Интернет-адреса для этого интерфейса. Объект `NetworkInterface` можно получить с помощью одного из статических методов, определенных в этом классе. `getNetworkInterfaces()` возвращает перечень всех интерфейсов локального хоста. Как правило, этот класс применяется только в сетевых программах с расширенными возможностями.

```
public final class NetworkInterface {
// Конструктор отсутствует
// Открытые методы класса
   public static NetworkInterface getByInetAddress(InetAddress addr) throws SocketException;
                                                                          // зависит от платформы
   public static NetworkInterface getByName(String name) throws SocketException;
                                                                          // зависит от платформы
   public static java.util.Enumeration getNetworkInterfaces() throws SocketException;
// Открытые методы экземпляра
   public String getDisplayName();
   public java.util.Enumeration getInetAddresses();
   public String getName();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: `DatagramSocketImpl.{joinGroup(), leaveGroup()}`, `MulticastSocket.{joinGroup(), leaveGroup(), setNetworkInterface()}`

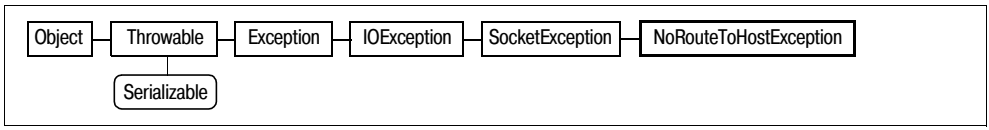
Возвращается методами: `MulticastSocket.getNetworkInterface()`, `NetworkInterface.{getByInetAddress(), getByName()}`

NoRouteToHostException

Java 1.1

java.net ***сериализуемое, проверяемое***

Это исключение возникает, когда сокет не может установить соединение с удаленным хостом. Как правило, причиной этого является разрыв связи в сети между локальной машиной и удаленным хостом или брандмауэр, защищающий удаленный хост.




```
public class NoRouteToHostException extends SocketException {
// Открытые конструкторы
   public NoRouteToHostException();
   public NoRouteToHostException(String msg);
}
```

PasswordAuthentication

Java 1.2

java.net

Этот простой неизменяемый класс инкапсулирует имя пользователя и пароль. Пароль хранится в виде массива символов, а не в виде объекта `String`, поэтому после использования содержимое массива можно удалить (в целях безопасности). Следует заметить, что конструктор `PasswordAuthentication()` создает копию указанного массива символов с паролем, а `getPassword()` возвращает ссылку на внутренний массив объекта.

Если в приложении определен объект `Authenticator`, то в его методе `getPasswordAuthentication()` нужно создавать и возвращать объект `PasswordAuthentication`. Этот класс применяется в системном программировании при реализации `URLStreamHandler` или

при использовании других средств для взаимодействия с сетевым сервером, запрашивающим пароль. Получить объект PasswordAuthentication, содержащий имя пользователя и пароль, можно с помощью статического метода Authenticator.requestPasswordAuthentication().

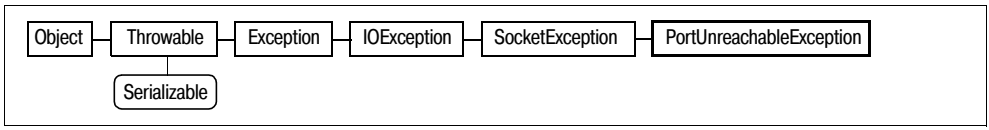
```
public final class PasswordAuthentication {
// Открытые конструкторы
   public PasswordAuthentication(String userName, char[] password);
// Открытые методы экземпляра
   public char[] getPassword();
   public String getUserName();
}
```

Возвращается методами: Authenticator.{getPasswordAuthentication(), requestPasswordAuthentication()}

PortUnreachableException Java 1.4

java.net ***сериализуемое, проверяемое***

Исключение этого типа может возникнуть в результате вызовов метода send() или receive() объекта DatagramSocket, если при попытке соединения с помощью метода connect() данного сокета было получено сообщение ICMP «port unreachable» («порт недоступен»).

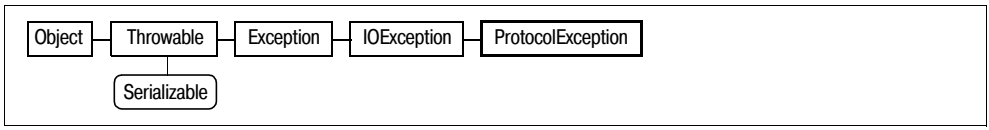



```
public class PortUnreachableException extends SocketException {
// Открытые конструкторы
   public PortUnreachableException();
   public PortUnreachableException(String msg);
}
```

ProtocolException Java 1.0

java.net ***сериализуемое, проверяемое***

Это исключение возникает при ошибке протокола в классе Socket.




```
public class ProtocolException extends java.io.IOException {
// Открытые конструкторы
   public ProtocolException();
   public ProtocolException(String host);
}
```

Генерируется методами: HttpURLConnection.setRequestMethod()

ServerSocket Java 1.0

java.net

Этот класс применяется в серверах, принимающих клиентские соединения. Прежде чем использовать ServerSocket, его нужно связать с адресом локальной сети, по которому он будет ожидать входящие соединения. Все конструкторы ServerSocket(), кроме конструктора без аргументов, создают серверный сокет и связывают его с указанным локальным портом, который будет прослушиваться. При этом можно указать количество незавершенных соединений – количество запросов на клиентские соединения, которые можно поставить в очередь. После превышения этого количества запросы на соединение будут отклоняться.

В Java 1.4 и последующих версиях конструктор ServerSocket() без аргументов создает свободный сокет. Это позволяет связать сокет с помощью метода bind(), который принимает в качестве параметра объект SocketAddress, а не номер порта. Кроме того, можно вызывать метод setReuseAddress(), который выполняет полезные действия, если сокет еще не связан. Чтобы определить, связан ли сокет, используйте метод isBound(). Если сокет связан, можно задействовать метод getLocalSocketAddress() или методы getLocalPort() и getInetAddress(), чтобы получить локальный адрес, с которым связан сокет.

После того как объект ServerSocket связан, можно вызвать метод accept(), чтобы прослушивать указанный порт и блокировать его, пока не будет получен запрос на клиентское соединение. Затем можно принять соединение методом accept(). При этом будет создан объект Socket, который может применяться сервером для общения с клиентом. Обычно серверы запускают новый поток, обрабатывающий сообщение с клиентом, и снова вызывают accept(), чтобы начать ожидание другого соединения.

В классе ServerSocket определено несколько методов, устанавливающих параметры сокета. Метод setSoTimeout() принимает аргумент в миллисекундах – время, в течение которого метод accept() должен ожидать соединение. По истечении этого времени вызывается исключение InterruptedIOException. Если аргумент равен 0, то время ожидания не ограничено. Метод setReceiveBufferSize() устанавливает дополнительный параметр – требуемый размер внутреннего буфера для принимаемых данных объекта Socket, возвращенного методом accept(). Это всего лишь рекомендуемый размер, и система может его проигнорировать. Еще один дополнительный параметр устанавливает метод setReuseAddress(); он определяет, что метод bind() должен завершиться успешно, даже если соответствующий локальный адрес в этот момент номинально используется сокетом, находящимся в процессе закрытия.

Как и все остальные сокеты, объект ServerSocket должен быть закрыт методом close(), когда он больше не нужен. После этого объект ServerSocket можно применять только для вызова метода isClosed(). Этот метод возвращает true, если сокет закрыт.

Метод getChannel() является связующим звеном между классом ServerSocket и классом java.nio.channels.ServerSocketChannel. Он возвращает объект ServerSocketChannel, связанный с данным сокетом ServerSocket, если такой объект существует. Но этот метод возвращает null для всех сокетов, которые были созданы с помощью конструкторов ServerSocket(). Если был создан объект ServerSocketChannel, а ServerSocket получен методами этого объекта, то с помощью метода getChannel() можно связать этот сокет с родительским каналом.

```
public class ServerSocket {
// Открытые конструкторы
1.4 public ServerSocket() throws java.io.IOException;
    public ServerSocket(int port) throws java.io.IOException;
```

```
   public ServerSocket(int port, int backlog) throws java.io.IOException;
1.1 public ServerSocket(int port, int backlog, InetAddress bindAddr) throws java.io.IOException;
// Открытые методы класса
   public static void setSocketFactory(SocketImplFactory fac) throws java.io.IOException;
                                                                          // синхронизирован
// Методы доступа к свойствам (по именам свойств)
1.4 public boolean isBound();                                          // по умолчанию:false
1.4 public java.nio.channels.ServerSocketChannel getChannel();   // константа; по умолчанию:null
1.4 public boolean isClosed();                                         // по умолчанию:false
   public InetAddress getInetAddress();                                 // по умолчанию:null
   public int getLocalPort();                                             // по умолчанию:-1
1.4 public SocketAddress getLocalSocketAddress();                       // по умолчанию:null
1.4 public int getReceiveBufferSize() throws SocketException;
                                                  // синхронизирован; по умолчанию:32767
1.4 public void setReceiveBufferSize(int size) throws SocketException;      // синхронизирован
1.4 public boolean getReuseAddress() throws SocketException;              // по умолчанию:true
1.4 public void setReuseAddress(boolean on) throws SocketException;
1.1 public int getSoTimeout() throws java.io.IOException;     // синхронизирован; по умолчанию:0
1.1 public void setSoTimeout(int timeout) throws SocketException;          // синхронизирован
// Открытые методы экземпляра
   public Socket accept() throws java.io.IOException;
1.4 public void bind(SocketAddress endpoint) throws java.io.IOException;
1.4 public void bind(SocketAddress endpoint, int backlog) throws java.io.IOException;
   public void close() throws java.io.IOException;
// Открытые методы, замещающие Object
   public String toString();
// Защищенные методы экземпляра
1.1 protected final void implAccept(Socket s) throws java.io.IOException;
}
```

Подклассы: javax.net.ssl.SSLServerSocket

Возвращается методами: java.nio.channels.ServerSocketChannel.socket(), java.rmi.server.RMIServerSocketFactory.createServerSocket(), java.rmi.server.RMISocketFactory.createServerSocket(), javax.net.ServerSocketFactory.createServerSocket()

Socket Java 1.0

java.net

Этот класс реализует сокет, предназначенный для потокового общения через сеть. См. также класс URL, реализующий сетевой интерфейс высокого уровня, и класс DatagramSocket, предоставляющий низкоуровневый интерфейс.

Прежде чем использовать сокет для сетевого сообщения, его нужно связать с локальным адресом и подключить к удаленному адресу. Связывание и соединение с удаленным адресом производятся автоматически при вызове любого конструктора Socket(), за исключением конструктора без аргументов. Этим конструкторам можно передать имя удаленного компьютера или объект InetAddress, и нужно обязательно передать номер порта, к которому будет произведено подключение. В двух конструкторах можно указать локальный адрес в виде объекта InetAddress и номер порта, с которым будет связан сокет. Как правило, в обычных приложениях не требуется указывать локальный адрес. В таком случае можно использовать конструктор Socket() с двумя аргументами; конструктор сам выберет локальный порт, с которым будет связан сокет.

Конструктор `Socket()` без аргументов отличается от остальных конструкторов: он создает сокет, не связанный с локальным адресом и не подключенный к удаленному адресу. В Java 1.4 и последующих версиях можно явно вызвать методы `bind()` и `connect()`, чтобы связать сокет и создать соединение. Такой подход применяется, когда нужно установить параметры сокета (описаны ниже) до связывания сокета и создания соединения. Метод `bind()` принимает в качестве параметра объект `SocketAddress`, описывающий локальный адрес, с которым нужно связать сокет. Методу `connect()` передается объект того же типа, но содержащий удаленный адрес. Есть вариант метода `connect()`, принимающий в качестве аргумента время ожидания: если по прошествии этого времени соединение не установлено, метод `connect()` вызывает исключение `IOException`. См. также описание `ServerSocket`, в котором представлена техника создания серверного приложения, принимающего клиентские запросы на соединение.

Чтобы определить, связан ли сокет и установлено ли соединение, используйте методы `isBound()` и `isConnected()`. С помощью метода `getInetAddress()` можно определить IP-адрес машины и номер порта, к которому подключен сокет. В Java 1.4 и последующих версиях метод `getRemoteSocketAddress()` возвращает удаленный адрес в виде объекта `SocketAddress`. Аналогично, с помощью методов `getLocalAddress()`, `getLocalPort()` и `getLocalSocketAddress()` можно получить локальный адрес, с которым связан сокет.

После того как объект `Socket` связан и для него установлено соединение, можно использовать методы `getInputStream()` и `getOutputStream()`, чтобы получить потоки `InputStream` и `OutputStream`. Затем эти объекты можно использовать для общения с удаленным хостом. Здесь эти потоки используются так же, как при файловом вводе/выводе. Когда работа с сокетом завершена, нужно вызвать метод `close()` для закрытия сокета. После этого нельзя повторно использовать сокет – вызывать метод `connect()` и другие методы, кроме `isClosed()`. Поскольку при работе с сетью велика вероятность возникновения исключений, метод `close()` обычно применяют в операторе `finally` блока `try/catch`, чтобы в любых ситуациях сокет гарантированно закрывался. Обратите внимание, что сам метод `close()` тоже может вызвать исключение `IOException`. Возможно, этот метод придется поместить в собственный блок `try`. В Java 1.3 и последующих версиях с помощью методов `shutdownInput()` и `shutdownOutput()` можно закрыть входные и выходные каналы связи, не закрывая при этом сокет. В Java 1.4 с помощью методов `isInputShutdown()` и `isOutputShutdown()` можно проверить, закрыты ли каналы.

В классе `Socket` определены несколько методов, устанавливающих (и запрашивающих) параметры сокета, которые влияют на его поведение на низком уровне. С помощью методов `setSendBufferSize()` и `setReceiveBufferSize()` можно указать системе рекомендуемый размер буферов сокета. Метод `setSoTimeout()` устанавливает время (в миллисекундах), в течение которого метод `read()` входного потока, возвращенного методом `getInputStream()`, будет ожидать получения данных; по истечении времени ожидания будет вызвано исключение `InterruptedIOException`. По умолчанию этот параметр равен `0`, при этом значении время блокировки потока не ограничено. Метод `setSoLinger()` определяет действие сокета после закрытия, если остались данные, которые не были переданы (то есть он включает или выключает задержку). Если задержка включена, метод `close()` блокируется на указанный промежуток времени, пока предпринимается попытка передать оставшиеся данные. Метод `setTcpNoDelay()` с аргументом `true` устанавливает режим, в котором данные должны отправляться через сокет всегда, когда это возможно, а не ожидать, пока пакет TCP заполнится до конца. В Java 1.3 метод `setKeepAlive()` разрешает или запрещает периодический обмен управляющими сообщениями через соединение сокета в то время, когда не выполняется передача данных. Протокол keepalive позволяет в случае аварийного отказа сервера определить, закрылся ли при этом сокет или нет. В Java 1.4, передав методу

setOOBInline() значение true, можно получать данные из другого диапазона, которые всегда передаются в сокет через входной поток (по умолчанию такие данные отбрасываются). Этот режим можно применять для получения данных, которые были отправлены методом sendUrgentData(). В Java 1.4 также добавлен метод setReuseAddress(), который нужно вызывать до того, как сокет будет связан с локальным адресом, чтобы разрешить сокету связываться с портом, который номинально еще используется закрывающимся сокетом. Метод setTrafficClass() также появился в Java 1.4. Он устанавливает поле «traffic class» данного сокета. Применение этого метода требует глубоких знаний протокола IP.

Метод getChannel() является связующим звеном между данным сокетом и классом java.nio.channels.SocketChannel (новый API ввода/вывода). Он возвращает объект SocketChannel, связанный с сокетом. Необходимо заметить, что этот метод всегда возвращает null, если данный сокет создан одним из конструкторов Socket(). Если был создан объект SocketChannel и данный сокет получен методами этого объекта, то метод getChannel() может связать сокет с породившим его каналом.

```
public class Socket {
// Открытые конструкторы
1.1 public Socket();
   public Socket(InetAddress address, int port) throws java.io.IOException;
   public Socket(String host, int port) throws java.net.UnknownHostException, java.io.IOException;
# public Socket(InetAddress host, int port, boolean stream) throws java.io.IOException;
# public Socket(String host, int port, boolean stream) throws java.io.IOException;
1.1 public Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
                                                              throws java.io.IOException;
1.1 public Socket(String host, int port, InetAddress localAddr, int localPort)
                                                              throws java.io.IOException;
// Защищенные конструкторы
1.1 protected Socket(SocketImpl impl) throws SocketException;
// Открытые методы класса
   public static void setSocketImplFactory(SocketImplFactory fac) throws java.io.IOException;
                                                                    // синхронизирован
// Методы доступа к свойствам (по именам свойств)
1.4 public boolean isBound();                                      // по умолчанию:false
1.4 public java.nio.channels.SocketChannel getChannel();      // константа; по умолчанию:null
1.4 public boolean isClosed();                                     // по умолчанию:false
1.4 public boolean isConnected();                                  // по умолчанию:false
   public InetAddress getInetAddress();                             // по умолчанию:null
1.4 public boolean isInputShutdown();                              // по умолчанию:false
   public java.io.InputStream getInputStream() throws java.io.IOException;
1.3 public boolean getKeepAlive() throws SocketException;          // по умолчанию:false
1.3 public void setKeepAlive(boolean on) throws SocketException;
1.1 public InetAddress getLocalAddress();                    // по умолчанию:Inet4Address
   public int getLocalPort();                                       // по умолчанию:-1
1.4 public SocketAddress getLocalSocketAddress();                  // по умолчанию:null
1.4 public boolean getOOBInline() throws SocketException;          // по умолчанию:false
1.4 public void setOOBInline(boolean on) throws SocketException;
1.4 public boolean isOutputShutdown();                             // по умолчанию:false
   public java.io.OutputStream getOutputStream() throws java.io.IOException;
   public int getPort();                                            // по умолчанию:0
1.2 public int getReceiveBufferSize() throws SocketException;
                                                  // синхронизирован; по умолчанию:32767
1.2 public void setReceiveBufferSize(int size) throws SocketException;     // синхронизирован
1.4 public SocketAddress getRemoteSocketAddress();                 // по умолчанию:null
1.4 public boolean getReuseAddress() throws SocketException;       // по умолчанию:false
```

```
1.4 public void setReuseAddress(boolean on) throws SocketException;
1.2 public int getSendBufferSize() throws SocketException;  // синхронизирован; по умолчанию:32767
1.2 public void setSendBufferSize(int size) throws SocketException;            // синхронизирован
1.1 public int getSoLinger() throws SocketException;                           // по умолчанию:-1
1.1 public int getSoTimeout() throws SocketException;          // синхронизирован; по умолчанию:0
1.1 public void setSoTimeout(int timeout) throws SocketException;              // синхронизирован
1.1 public boolean getTcpNoDelay() throws SocketException;                  // по умолчанию:false
1.1 public void setTcpNoDelay(boolean on) throws SocketException;
1.4 public int getTrafficClass() throws SocketException;                        // по умолчанию:0
1.4 public void setTrafficClass(int tc) throws SocketException;
// Открытые методы экземпляра
1.4 public void bind(SocketAddress bindpoint) throws java.io.IOException;
   public void close() throws java.io.IOException;                             // синхронизирован
1.4 public void connect(SocketAddress endpoint) throws java.io.IOException;
1.4 public void connect(SocketAddress endpoint, int timeout) throws java.io.IOException;
1.4 public void sendUrgentData(int data) throws java.io.IOException;
1.1 public void setSoLinger(boolean on, int linger) throws SocketException;
1.3 public void shutdownInput() throws java.io.IOException;
1.3 public void shutdownOutput() throws java.io.IOException;
// Открытые методы, замещающие Object
   public String toString();
}
```

Подклассы: javax.net.ssl.SSLSocket

Передается методам: ServerSocket.implAccept(), javax.net.ssl.SSLSocketFactory.createSocket(), javax.net.ssl.X509KeyManager.{chooseClientAlias(), chooseServerAlias()}

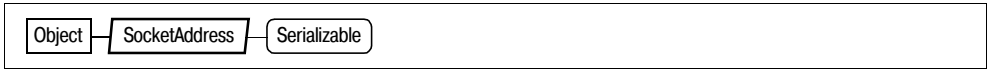
Возвращается методами: ServerSocket.accept(), java.nio.channels.SocketChannel.socket(), java.rmi.server.RMIClientSocketFactory.createSocket(), java.rmi.server.RMISocketFactory.createSocket(), javax.net.SocketFactory.createSocket(), javax.net.ssl.SSLSocketFactory.createSocket()

SocketAddress

Java 1.4

java.net ***сериализуемый***

Этот абстрактный класс является общим представлением сетевого адреса сокета. Единственный потомок в Java API, конкретизирующий этот класс, – это InetSocketAddress, который представляет Интернет-адрес и номер порта. См. также InetSocketAddress.




```
public abstract class SocketAddress implements Serializable {
// Открытые конструкторы
   public SocketAddress();
}
```

Подклассы: InetSocketAddress

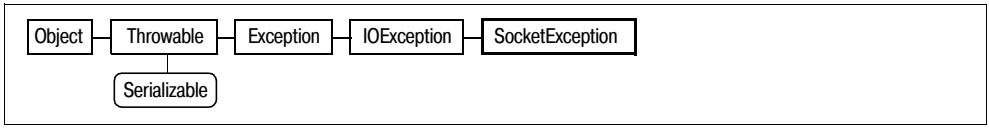
Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: DatagramPacket.getSocketAddress(), DatagramSocket.{getLocalSocketAddress(), getRemoteSocketAddress()}, ServerSocket.getLocalSocketAddress(), Socket.{getLocalSocketAddress(), getRemoteSocketAddress()}, java.nio.channels.DatagramChannel.receive()

SocketException Java 1.0

java.net *сериализуемое, проверяемое*

Это исключение свидетельствует о возникновении исключительной ситуации во время использования сокета.




```
public class SocketException extends java.io.IOException {
// Открытые конструкторы
   public SocketException();
   public SocketException(String msg);
}
```

Подклассы: BindException, java.net.ConnectException, NoRouteToHostException, PortUnreachableException

Генерируется методами: Методов слишком много, чтобы их перечислить.

SocketImpl Java 1.0

java.net

В этом абстрактном классе определены методы, которые должны реализовать передачу данных через сокет. Разные потомки этого класса могут реализовать его для работы в различных условиях (например, под защитой брандмауэра). Эти реализации сокета применяются классом ServerSocket. В обычных приложениях никогда не приходится использовать этот класс или создавать его потомков.


```
public abstract class SocketImpl implements SocketOptions {
// Открытые конструкторы
   public SocketImpl();
// Открытые методы, замещающие Object
   public String toString();
// Защищенные методы экземпляра
   protected abstract void accept(SocketImpl s) throws java.io.IOException;
   protected abstract int available() throws java.io.IOException;
   protected abstract void bind(InetAddress host, int port) throws java.io.IOException;
   protected abstract void close() throws java.io.IOException;
   protected abstract void connect(String host, int port) throws java.io.IOException;
   protected abstract void connect(InetAddress address, int port) throws java.io.IOException;
1.4 protected abstract void connect(SocketAddress address, int timeout) throws java.io.IOException;
   protected abstract void create(boolean stream) throws java.io.IOException;
   protected java.io.FileDescriptor getFileDescriptor();
   protected InetAddress getInetAddress();
   protected abstract java.io.InputStream getInputStream() throws java.io.IOException;
   protected int getLocalPort();
   protected abstract java.io.OutputStream getOutputStream() throws java.io.IOException;
```

```
   protected int getPort();
   protected abstract void listen(int backlog) throws java.io.IOException;
1.4 protected abstract void sendUrgentData(int data) throws java.io.IOException;
1.3 protected void shutdownInput() throws java.io.IOException;
1.3 protected void shutdownOutput() throws java.io.IOException;
1.4 protected boolean supportsUrgentData();                                    // константа
// Защищенные поля экземпляра
   protected InetAddress address;
   protected java.io.FileDescriptor fd;
   protected int localport;
   protected int port;
}
```

Передается методам: Socket.Socket(), SocketImpl.accept()

Возвращается методами: SocketImplFactory.createSocketImpl()

SocketImplFactory Java 1.0

java.net

В этом интерфейсе определен метод, который должен создавать объекты SocketImpl. Можно регистрировать объекты SocketImplFactory, чтобы создавать экземпляры SocketImpl для классов Socket и ServerSocket. В обычных приложениях никогда не приходится применять или реализовывать этот интерфейс.

```
public interface SocketImplFactory {
// Открытые методы экземпляра
   public abstract SocketImpl createSocketImpl();
}
```

Передается методам: ServerSocket.setSocketFactory(), Socket.setSocketImplFactory()

SocketOptions Java 1.2

java.net

В этом интерфейсе определены константы, которыми представлены значения низкоуровневых параметров сокета в стиле BSD Unix, и методы, которые запрашивают и устанавливают значения этих параметров. В Java 1.2 этот интерфейс реализован классами SocketImpl и DatagramSocketImpl. В любой пользовательской реализации сокета должна быть рабочая реализация методов getOption() и setOption(). Пользовательский сокет может поддерживать и другие параметры, отличные от тех, которые определены в этом интерфейсе. Данный интерфейс применяется только в том случае, если нужно реализовать свой тип сокета. В остальных случаях для определения параметров сокета соответствующего типа можно использовать методы, определенные в классах ServerSocket, DatagramSocket и MulticastSocket.

```
public interface SocketOptions {
// Открытые константы
   public static final int IP_MULTICAST_IF;                                   // =16
1.4 public static final int IP_MULTICAST_IF2;                                 // =31
1.4 public static final int IP_MULTICAST_LOOP;                                // =18
1.4 public static final int IP_TOS;                                           // =3
   public static final int SO_BINDADDR;                                      // =15
1.4 public static final int SO_BROADCAST;                                     // =32
1.3 public static final int SO_KEEPALIVE;                                     // =8
```

```
   public static final int SO_LINGER;                                    // =128
1.4 public static final int SO_OOBINLINE;                                // =4099
   public static final int SO_RCVBUF;                                    // =4098
   public static final int SO_REUSEADDR;                                 // =4
   public static final int SO_SNDBUF;                                    // =4097
   public static final int SO_TIMEOUT;                                   // =4102
   public static final int TCP_NODELAY;                                  // =1
// Открытые методы экземпляра
   public abstract Object getOption(int optID) throws SocketException;
   public abstract void setOption(int optID, Object value) throws SocketException;
}
```

Реализации: DatagramSocketImpl, SocketImpl

SocketPermission

Java 1.2

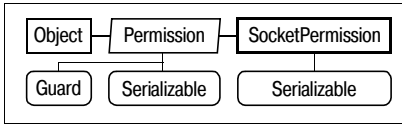
java.net ***сериализуемый***

Этот класс является потомком java.security.Permission. Он отвечает за все сетевые операции, выполняемые сокетом. Подобно остальным правам, SocketPermission состоит из имени целевого объекта и списка действий, которые можно выполнять с этим объектом. Целевым объектом является хост. Кроме того, можно указать порт(ы). Имя целевого объекта состоит из имени хоста и (необязательно) имени порта. Части имени целевого объекта должны быть разделены двоеточием. Хост может быть именем домена DNS, или числовым IP-адресом, или строкой «localhost». В имени домена слева может стоять групповой символ «*». Может быть указан один номер порта или диапазон номеров портов в форме n1-n2. Если n1 не указан, он считается равным 0, а если пропущен n2, он считается равным 65535. Если номер порта не указан, право относится ко всем портам указанного хоста. Вот несколько примеров имен целевых объектов права SocketPermission:

```
java.sun.com:80
*.sun.com:1024-2000
*:1024-
localhost:-1023
```

Кроме имени целевого объекта, в каждом праве SocketPermission должен присутствовать список действий, разделенных запятой, которые можно производить с указанными хостами и портами. Допустимые имена действий: «connect», «accept», «listen» и «resolve». «connect» представляет разрешение на соединение с указанным объектом права. «accept» разрешает принимать запрос на соединение от объекта права. «listen» представляет право на ожидание запросов соединения на указанных портах. Это действие допустимо только для локальных портов («localhost»). Наконец, действие «resolve» разрешает использовать службы имен доменов DNS для перевода имен домена в IP-адреса. При указании любого другого действия подразумевается и это действие.

Системные администраторы, которые занимаются настройкой политик безопасности, должны знать этот класс и уметь оценивать риск предоставления прав. Данный класс могут применять системные программисты, пишущие низкоуровневые библиотеки для сетевых коммуникаций или применяющие сетевые функции ОС. Разработчики приложений должны избегать прямого использования этого класса.



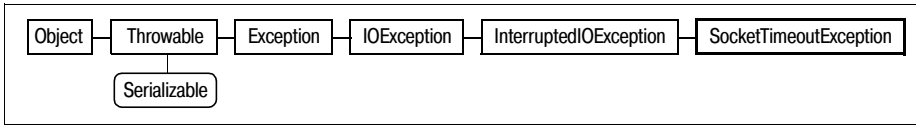

```
public final class SocketPermission extends java.security.Permission implements Serializable {
// Открытые конструкторы
   public SocketPermission(String host, String action);
// Открытые методы, замещающие Permission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
   public boolean implies(java.security.Permission p);
   public java.security.PermissionCollection newPermissionCollection();
}
```

SocketTimeoutException

Java 1.4

java.net ***сериализуемое, проверяемое***

Это исключение возникает в сокете, если во время операции чтения или приема время ожидания истекло. См. также описание метода `setSoTimeout()` класса `Socket`.



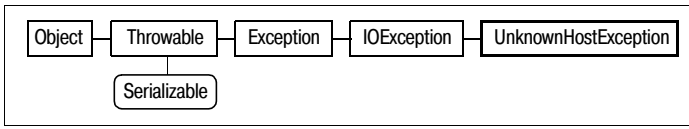

```
public class SocketTimeoutException extends java.io.InterruptedIOException {
// Открытые конструкторы
   public SocketTimeoutException();
   public SocketTimeoutException(String msg);
}
```

UnknownHostException

Java 1.0

java.net ***сериализуемое, проверяемое***

Это исключение сообщает о том, что указанное имя хоста не может быть переведено в адрес.



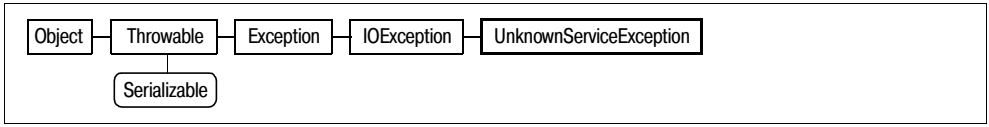

```
public class UnknownHostException extends java.io.IOException {
// Открытые конструкторы
   public UnknownHostException();
   public UnknownHostException(String host);
}
```

Генерируется методами: `InetAddress.{getAllByName(), getByAddress(), getByName(), getLocalHost()}, Socket.Socket(), javax.net.SocketFactory.createSocket(), javax.net.ssl.SSLSocket.SSLSocket()`

UnknownServiceException Java 1.0

java.net ***сериализуемое, проверяемое***

Это исключение возникает при попытке использовать неподдерживаемую сетевую службу.




```
public class UnknownServiceException extends java.io.IOException {
// Открытые конструкторы
   public UnknownServiceException();
   public UnknownServiceException(String msg);
}
```

URI Java 1.4

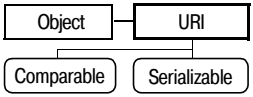
java.net ***сериализуемый, сравнимый***

`URI` – это неизменяемый класс, который представляет универсальный идентификатор ресурса (`URI`). `URI` – это обобщенный вариант унифицированного указателя информационного ресурса (`URL`), применяемый во Всемирной паутине (World Wide Web). Класс `URI` поддерживает синтаксический анализ и обработку текста строк `URI`, но не имеет сетевой функциональности, которая есть у класса `URL`. Преимущества класса `URI` перед `URL` заключаются в том, что первый предоставляет более универсальные возможности для анализа и обработки строк `URL`; он также может представлять относительные `URI`, не содержащие схему (протокол); кроме того, класс `URI` может обрабатывать `URI`, содержащие неподдерживаемые и даже неизвестные схемы.

Получить объект `URI` можно с помощью одного из конструкторов, которые могут анализировать `URI` из строки или принимать отдельные компоненты `URI`. Эти конструкторы могут вызвать перехватываемое исключение `URISyntaxException`. При использовании `URI`, полученных программно (а не введенных пользователем), можно применять метод `create()`, не вызывающий перехватываемых исключений.

После создания объекта `URI` можно использовать его методы `get` для запроса различных частей URI. Методы `getRaw()` аналогичны методам `get`, за тем исключением, что они не декодируют шестнадцатеричные управляющие последовательности вида `%xx`, содержащиеся в URI. Метод `normalize()` возвращает новый объект `URI`, в котором удалены ненужные последовательности «.» и «..» из компонента, соответствующего пути. Метод `resolve()` интерпретирует свой аргумент (может быть строкой или объектом `URI`) относительно `URI` данного объекта и возвращает результат. `relativize()` выполняет обратную операцию. Он возвращает новый объект `URI`, представляющий тот же ресурс, что и аргумент, но относительно `URI` данного объекта. И наконец, метод `toURL()` преобразует `URI`, представляющий абсолютную ссылку, в эквивалентный `URL`. Поскольку класс `URI` представляет больше возможностей для обработки текста `URL`,

его можно задействовать, например, для интерпретации относительных URL, а затем преобразовывать полученные объекты URI в URL, которые можно использовать в сетевых задачах.




```
public final class URI implements Comparable,
      Serializable {
// Открытые конструкторы
   public URI(String str) throws URISyntaxException;
   public URI(String scheme, String ssp, String fragment) throws URISyntaxException;
   public URI(String scheme, String host, String path, String fragment) throws URISyntaxException;
   public URI(String scheme, String authority, String path, String query, String fragment)
                                                                    throws URISyntaxException;
   public URI(String scheme, String userInfo, String host, int port, String path, String query,
              String fragment) throws URISyntaxException;
// Открытые методы класса
   public static URI create(String str);
// Методы доступа к свойствам (по именам свойств)
   public boolean isAbsolute();
   public String getAuthority();
   public String getFragment();
   public String getHost();
   public boolean isOpaque();
   public String getPath();
   public int getPort();
   public String getQuery();
   public String getRawAuthority();
   public String getRawFragment();
   public String getRawPath();
   public String getRawQuery();
   public String getRawSchemeSpecificPart();
   public String getRawUserInfo();
   public String getScheme();
   public String getSchemeSpecificPart();
   public String getUserInfo();
// Открытые методы экземпляра
   public URI normalize();
   public URI parseServerAuthority() throws URISyntaxException;
   public URI relativize(URI uri);
   public URI resolve(URI uri);
   public URI resolve(String str);
   public String toASCIIString();
   public URL toURL() throws MalformedURLException;
// Методы, реализующие Comparable
   public int compareTo(Object ob);
// Открытые методы, замещающие Object
   public boolean equals(Object ob);
   public int hashCode();
   public String toString();
}
```

Передается методам: java.io.File.File(), URI.{relativize(), resolve()}, javax.print.attribute.URISyntax.URISyntax(), javax.print.attribute.standard.Destination.Destination(), javax.print.attribute.standard.PrinterMoreInfo.PrinterMoreInfo(), javax.print.attribute.standard.PrinterMoreInfoManufacturer.PrinterMoreInfoManufacturer(), javax.print.attribute.standard.PrinterURI.PrinterURI()

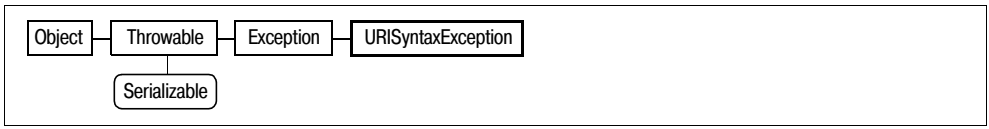
Возвращается методами: java.io.File.toURI(), URI.{create(), normalize(), parseServerAuthority(), relativize(), resolve()}, javax.print.URIException.getUnsupportedURI(), javax.print.attribute.URISyntax.getURI()

URISyntaxException Java 1.4

java.net ***сериализуемое, проверяемое***

Назначение этого класса – сообщать о том, что строка не может быть интерпретирована как URI. Метод getInput() возвращает эту строку. getReason() возвращает сообщение об ошибке. Метод getIndex() возвращает позицию, на которой найдена синтаксическая ошибка, если об этом есть сведения. getMessage() возвращает строку, удобную для восприятия пользователем, которая содержит информацию, предоставляемую вышеперечисленными методами.

Данное исключение является перехватываемым и может генерироваться всеми конструкторами URI(). Если нужно обработать полученную программным путем ссылку URI, в которой не должно содержаться синтаксических ошибок, и необходимо избежать перехватываемых исключений, применяйте не конструктор URI() с одним аргументом, а метод-фабрику URI.create().




```
public class URISyntaxException extends Exception {
// Открытые конструкторы
   public URISyntaxException(String input, String reason);
   public URISyntaxException(String input, String reason, int index);
// Открытые методы экземпляра
   public int getIndex();
   public String getInput();
   public String getReason();
// Открытые методы, замещающие Throwable
   public String getMessage();
}
```

Генерируется методами: URI.{parseServerAuthority(), URI()}

URL Java 1.0

java.net ***сериализуемый***

Этот класс представляет унифицированный указатель информационного ресурса (URL) и имеет возможность загрузки данных, на которые ссылается URL. URL можно указать в одной строке или отдельно указать протокол, хост, порт и файл. Относительную ссылку URL можно указать в виде String и объекта URL, с которым связана данная ссылка. Методы getFile(), getHost(), getProtocol() и родственные методы воз-

вращают различные компоненты URL, определяемые объектом URL. Метод sameFile() определяет, ссылаются ли объект URL в аргументе и данный URL на один и тот же файл. Метод getDefaultPort() возвращает номер порта, используемый по умолчанию протоколом объекта URL; он может отличаться от значения, возвращаемого методом getPort(). Чтобы получить объект URLConnection, с помощью которого можно загрузить объект ссылки URL, применяйте метод openConnection(). Для удобства в классе URL определены методы, которые сами создают объект URLConnection и вызывают его методы. Метод getContent() загружает данные, на которые ссылается URL, анализирует их и, если находит соответствующий объект ContentHandler, сохраняет эти данные в подходящем объекте Java (например, в виде строки или изображения). В Java 1.3 и последующих версиях можно передать массив объектов Class, соответствующих типам объектов, в которых могут быть возвращены данные из этого метода. Если нужно самостоятельно анализировать данные, на которые ссылается URL, используйте метод openStream(), возвращающий поток InputStream, из которого можно прочитать эти данные.


```
public final class URL implements Serializable {
// Открытые конструкторы
   public URL(String spec) throws MalformedURLException;
   public URL(URL context, String spec) throws MalformedURLException;
   public URL(String protocol, String host, String file) throws MalformedURLException;
1.2 public URL(URL context, String spec, URLStreamHandler handler) throws MalformedURLException;
   public URL(String protocol, String host, int port, String file) throws MalformedURLException;
1.2 public URL(String protocol, String host, int port, String file, URLStreamHandler handler)
                                                                    throws MalformedURLException;
// Открытые методы класса
   public static void setURLStreamHandlerFactory(URLStreamHandlerFactory fac);
// Методы доступа к свойствам (по именам свойств)
1.3 public String getAuthority();
   public final Object getContent() throws java.io.IOException;
1.3 public final Object getContent(Class[] classes) throws java.io.IOException;
1.4 public int getDefaultPort();
   public String getFile();
   public String getHost();
1.3 public String getPath();
   public int getPort();
   public String getProtocol();
1.3 public String getQuery();
   public String getRef();
1.3 public String getUserInfo();
// Открытые методы экземпляра
   public URLConnection openConnection() throws java.io.IOException;
   public final java.io.InputStream openStream() throws java.io.IOException;
   public boolean sameFile(URL other);
   public String toExternalForm();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();                                                    // синхронизирован
   public String toString();
// Защищенные методы экземпляра
   protected void set(String protocol, String host, int port, String file, String ref);
```

```
1.3 protected void set(String protocol, String host, int port, String authority, String userInfo,
                       String path, String query, String ref);
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

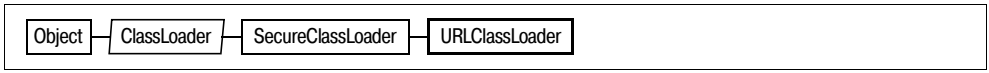
Экземпляры: `URLConnection.url`

URLClassLoader Java 1.2

java.net

Этот класс является потомком `ClassLoader` и предназначен для загрузки ненадежных Java-программ, на которые ссылаются произвольные `URL`, представляющие собой адрес каталога или файла JAR. Чтобы загрузить указанный класс с помощью объекта `URLClassLoader`, применяйте его метод `loadClass()`. Классы, загруженные этим объектом, получают все права, которые объект `java.security.Policy` предоставляет объекту `java.security.CodeSource`, связанному с ними, а также дополнительные права, позволяющие загрузчику классов читать любые файлы ресурсов этих классов. Если класс загружен из локального каталога (`file: URL`), то классу предоставляется право на чтение всех его файлов и подкаталогов. Если класс загружен из локального файла JAR, ему разрешается читать этот файл JAR. Если класс загружен из `URL`, ссылающегося на ресурс удаленного хоста, такому классу разрешается соединяться с этим хостом или принимать от него соединение. Однако такому классу это право не дается в случае, если у объекта, который создал `URLClassLoader`, нет этого права.

Для получения объекта `URLClassLoader` используйте один из конструкторов `URLClassLoader` или статических методов `newInstance()`. Если для этой цели применялся метод `newInstance()`, то метод `loadClass()` созданного объекта `URLClassLoader` проводит дополнительную проверку на наличие у объекта, вызвавшего этот метод, права на доступ к указанному пакету.




```
public class URLClassLoader extends java.security.SecureClassLoader {
// Открытые конструкторы
   public URLClassLoader(URL[] urls);
   public URLClassLoader(URL[] urls, ClassLoader parent);
   public URLClassLoader(URL[] urls, ClassLoader parent,
        URLStreamHandlerFactory factory);
// Открытые методы класса
   public static URLClassLoader newInstance(URL[] urls);
   public static URLClassLoader newInstance(URL[] urls, ClassLoader parent);
// Открытые методы экземпляра
   public URL[] getURLs();
// Защищенные методы, замещающие SecureClassLoader
   protected java.security.PermissionCollection getPermissions (java.security.CodeSource codesource);
// Открытые методы, замещающие ClassLoader
   public URL findResource(String name);
   public java.util.Enumeration findResources(String name) throws java.io.IOException;
// Защищенные методы, замещающие ClassLoader
   protected Class findClass(String name) throws ClassNotFoundException;
// Защищенные методы экземпляра
```

```
    protected void addURL(URL url);
    protected Package definePackage(String name, java.util.jar.Manifest man, URL url)
                                                                    throws IllegalArgumentException;
}
```

Возвращается методами: `URLClassLoader.newInstance()`

URLConnection

Java 1.0

java.net

Этот абстрактный класс определяет сетевое подключение к объекту, на который ссылается указанный `URL`. Метод `URL.openConnection()` возвращает экземпляр этого класса. Если нужно получить дополнительный контроль над загрузкой данных, не ограничиваясь методами, предоставленными классом `URL`, используйте объект `URLConnection`. Метод `connect()` фактически устанавливает сетевое соединение. Некоторые методы этого класса нужно вызывать до того, как создано соединение, а остальные методы могут работать только при уже установленном соединении. Последние сами вызывают метод `connect()`, если подключение еще не установлено, поэтому никогда не требуется явно вызывать этот метод. Методы `getContent()` выполняют те же действия, что и одноименные методы класса `URL`; они загружают данные, на которые ссылается URL, и определяют, к какому типу объекта Java эти данные относятся (например, строка или изображение). В Java 1.3 и последующих версиях можно передать массив объектов `Class`, соответствующих типам объектов, в которых могут быть возвращены данные из метода `getContent()`. Если нужно самостоятельно анализировать данные, на которое ссылается URL (вместо метода `getContent()`), то можно задействовать `getInputStream()` (или `getOutputStream()`, если протокол URL поддерживает запись). Этот метод возвращает поток, с помощью которого можно читать (записывать) данные ресурса, на который ссылается URL.

Перед установлением соединения можно задать поля запроса (например, заголовки запроса HTTP), уточняющие запрос URL. Используйте метод `setRequestProperty()`, чтобы установить новое значение для указанного заголовка. В Java 1.4 и последующих версиях можно задействовать метод `addRequestProperty()`, который добавляет в существующий заголовок новый элемент, отделяемый запятой. В Java 1.4 также добавлен метод `getRequestProperties()`, возвращающий текущий набор свойств запроса в виде неизменяемого объекта `Map`. В этом объекте имена заголовков запроса поставлены в соответствие объектам `List`, содержащим значения этих заголовков в виде строк.

После создания соединения можно вызывать методы для получения информации о заголовках ответа URL. Методы `getContentLength()`, `getContentType()`, `getContentEncoding()`, `getExpiration()`, `getDate()` и `getLastModified()` возвращают соответствующую информацию об объекте, на который ссылается URL, если такая информация доступна (например, из полей заголовков HTTP). Метод `getHeaderField()` возвращает поле заголовка HTTP с указанным именем или номером. Методы `getHeaderFieldInt()` и `getHeaderFieldDate()` возвращают значение указанного поля заголовка в виде числа или даты. В Java 1.4 и последующих версиях метод `getHeaderFields()` возвращает неизменяемый объект `Map`, в котором имена заголовков ответа поставлены в соответствие с неизменяемым объектом `List`, содержащим значения этих заголовков в виде строк.

Поведением объекта `URLConnection` можно управлять с помощью нескольких параметров. Их значения устанавливаются различными методами `set()` и запрашиваются соответствующими методами `get()`. Эти параметры нужно установить до вызова метода `connect()`. С помощью методов `setDoInput()` и `setDoOutput()` можно указать, как будет применяться `URLConnection` – для ввода, вывода или для ввода и вывода одновременно

(по умолчанию только для ввода). Метод `setAllowUserInteraction()` определяет, разрешено ли взаимодействие с пользователем (например, ввод пользователем пароля) во время передачи данных (по умолчанию этот параметр равен `false`). Метод `setDefaultAllowUserInteraction()` – это метод класса, позволяющий изменять значение по умолчанию для этого параметра. Метод `setUseCaches()` определяет, можно ли использовать кэшируемый вариант URL. Чтобы заставить перезагрузиться URL, установите этот параметр в `false`. Метод `setDefaultUseCaches()` устанавливает значение по умолчанию для параметра, который определяется предыдущим методом. Метод `setIfModifiedSince()` позволяет установить дату последней модификации URL. Если объект, на который ссылается URL, был модифицирован до этой даты, он не будет загружен (если эту дату можно определить).

```
public abstract class URLConnection {
// Защищенные конструкторы
   protected URLConnection(URL url);
// Открытые методы класса
   public static boolean getDefaultAllowUserInteraction();
1.1 public static FileNameMap getFileNameMap();                              // синхронизирован
   public static String guessContentTypeFromName(String fname);
   public static String guessContentTypeFromStream(java.io.InputStream is) throws java.io.IOException;
   public static void setContentHandlerFactory(ContentHandlerFactory fac);   // синхронизирован
   public static void setDefaultAllowUserInteraction(boolean defaultallowuserinteraction);
1.1 public static void setFileNameMap(FileNameMap map);
// Методы доступа к свойствам (по именам свойств)
   public boolean getAllowUserInteraction();
   public void setAllowUserInteraction(boolean allowuserinteraction);
   public Object getContent() throws java.io.IOException;
1.3 public Object getContent(Class[] classes) throws java.io.IOException;
   public String getContentEncoding();
   public int getContentLength();
   public String getContentType();
   public long getDate();
   public boolean getDefaultUseCaches();
   public void setDefaultUseCaches(boolean defaultusecaches);
   public boolean getDoInput();
   public void setDoInput(boolean doinput);
   public boolean getDoOutput();
   public void setDoOutput(boolean dooutput);
   public long getExpiration();
1.4 public java.util.Map getHeaderFields();
   public long getIfModifiedSince();
   public void setIfModifiedSince(long ifmodifiedsince);
   public java.io.InputStream getInputStream() throws java.io.IOException;
   public long getLastModified();
   public java.io.OutputStream getOutputStream() throws java.io.IOException;
1.2 public java.security.Permission getPermission() throws java.io.IOException;
1.4 public java.util.Map getRequestProperties();
   public URL getURL();
   public boolean getUseCaches();
   public void setUseCaches(boolean usecaches);
// Открытые методы экземпляра
1.4 public void addRequestProperty(String key, String value);
   public abstract void connect() throws java.io.IOException;
   public String getHeaderField(String name);                                // константа
   public String getHeaderField(int n);                                     // константа
   public long getHeaderFieldDate(String name, long Default);
```

```
   public int getHeaderFieldInt(String name, int Default);
   public String getHeaderFieldKey(int n);                                    // константа
   public String getRequestProperty(String key);
   public void setRequestProperty(String key, String value);
// Открытые методы, замещающие Object
   public String toString();
// Защищенные поля экземпляра
   protected boolean allowUserInteraction;
   protected boolean connected;
   protected boolean doInput;
   protected boolean doOutput;
   protected long ifModifiedSince;
   protected URL url;
   protected boolean useCaches;
// Устаревшие открытые методы
# public static String getDefaultRequestProperty(String key);               // константа
# public static void setDefaultRequestProperty(String key, String value);   // пустой
}
```

Подклассы: HttpURLConnection, JarURLConnection

Передается методам: java.net.ContentHandler.getContent()

Возвращается методами: URL.openConnection(), URLStreamHandler.openConnection()

Экземпляры: JarURLConnection.jarFileURLConnection

URLDecoder

Java 1.2

java.net

В этом классе определен статический метод decode(), который выполняет обратную методу URLEncoder.encode() операцию: декодирует 8-битовый текст в формате MIME вида «x-www-form-urlencoded». Это стандартная кодировка, применяемая веб-броузерами для передачи скриптам CGI и другим серверным программам содержимого форм.

```
public class URLDecoder {
// Открытые конструкторы
   public URLDecoder();
// Открытые методы класса
1.4 public static String decode(String s, String enc) throws  java.io.UnsupportedEncodingException;
// Устаревшие открытые методы
# public static String decode(String s);
}
```

URLEncoder

Java 1.0

java.net

В этом классе определен единственный статический метод, преобразующий строку в формат, принятый для записи URL. При этом пробелы заменяются на «+», а символы, не являющиеся буквами или цифрами, кроме подчеркивания, представляются в виде знака «%» и следующего за ним двузначного шестнадцатеричного числа. Обратите внимание, что такой способ кодировки работает только с 8-битовыми символами. Этот метод стандартизирует строку указанного URL: теперь в ней применяются только символы из подмножества набора ASCII, которые могут быть корректно обработаны в любой системе.

```
public class URLEncoder {
// Конструктор отсутствует
// Открытые методы класса
1.4 public static String encode(String s, String enc) throws java.io.UnsupportedEncodingException;
// Устаревшие открытые методы
# public static String encode(String s);
}
```

URLStreamHandler

Java 1.0

java.net

Этот абстрактный класс определяет метод openConnection(), который создает объект URLConnection для данного URL. Можно определять потомки этого класса для различных типов протоколов URL. Экземпляр URLStreamHandler создается объектом URLStreamHandlerFactory. В обычных приложениях этот класс никогда не применяется и не наследуется.

```
public abstract class URLStreamHandler {
// Открытые конструкторы
   public URLStreamHandler();
// Защищенные методы экземпляра
1.3 protected boolean equals(URL u1, URL u2);
1.3 protected int getDefaultPort();                // константа
1.3 protected InetAddress getHostAddress(URL u);  // синхронизирован
1.3 protected int hashCode(URL u);
1.3 protected boolean hostsEqual(URL u1, URL u2);
   protected abstract URLConnection openConnection(URL u) throws java.io.IOException;
   protected void parseURL(URL u, String spec, int start, int limit);
1.3 protected boolean sameFile(URL u1, URL u2);
1.3 protected void setURL(URL u, String protocol, String host, int port, String authority,
                          String userInfo, String path, String query, String ref);
   protected String toExternalForm(URL u);
// Устаревшие защищенные методы
# protected void setURL(URL u, String protocol, String host, int port, String file, String ref);
}
```

Передается методам: URL.URL()

Возвращается методами: URLStreamHandlerFactory.createURLStreamHandler()

URLStreamHandlerFactory

Java 1.0

java.net

В этом интерфейсе определен метод, создающий объект URLStreamHandler для указанного протокола. В обычных приложениях этот интерфейс никогда не применяется и не реализуется.

```
public interface URLStreamHandlerFactory {
// Открытые методы экземпляра
   public abstract URLStreamHandler createURLStreamHandler(String protocol);
}
```

Передается методам: URL.setURLStreamHandlerFactory(), URLClassLoader.URLClassLoader()

Глава 14

java.nio и подпакеты

В этой главе описывается новый API ввода/вывода, реализованный в пакете `java.nio` и его подпакетах. Будут рассмотрены следующие пакеты:

`java.nio`

Определяет общий класс `Buffer` и потомки этого класса, которые предназначены для работы с конкретными типами. Из них самый примечательный – класс `ByteBuffer`, который часто применяется для ввода/вывода в пакете `java.nio.channels`.

`java.nio.channels`

В этом пакете определен абстрактный класс `Channel`, служащий для организации высокоэффективного ввода/вывода и каналов файлового и сетевого ввода/вывода. Класс `Selector` предназначен для реализации неблокируемого ввода/вывода.

`java.nio.channels.spi`

Интерфейс поставщика сервисов для каналов и селекторов.

`java.nio.charset`

Здесь определены классы для преобразования последовательностей символов в байтовые значения и классы для обратного преобразования. При этом используются правила, определяемые указанным набором символов.

`java.nio.charset.spi`

Интерфейс поставщика услуг для реализаций наборов символов.

Пакет java.nio

Java 1.4

Этот пакет содержит определения буферных классов, которые являются фундаментальными классами для API `java.nio`. Для того чтобы получить общее представление о буферах, обратитесь к описанию класса `Buffer`, а в описании класса `ByteBuffer` (это самый важный из буферных классов) вы найдете подробную документацию по байтовым буферам. Другие специфичные для конкретных типов буферные классы являются близкими аналогами класса `ByteBuffer` и описаны в терминах этого класса. Описания классов, которые выполняют операции ввода/вывода для буферов, содержатся в пакете `java.nio.channels`.

Классы

```
public abstract class Buffer;
   └ public abstract class ByteBuffer extends Buffer implements Comparable;
     └ public abstract class MappedByteBuffer extends ByteBuffer;
   └ public abstract class CharBuffer extends Buffer implements CharSequence, Comparable;
   └ public abstract class DoubleBuffer extends Buffer implements Comparable;
   └ public abstract class FloatBuffer extends Buffer implements Comparable;
   └ public abstract class IntBuffer extends Buffer implements Comparable;
   └ public abstract class LongBuffer extends Buffer implements Comparable;
   └ public abstract class ShortBuffer extends Buffer implements Comparable;
public final class ByteOrder;
```

Исключения

```
public class BufferOverflowException extends RuntimeException;
public class BufferUnderflowException extends RuntimeException;
public class InvalidMarkException extends IllegalStateException;
public class ReadOnlyBufferException extends UnsupportedOperationException;
```

Buffer Java 1.4

java.nio

Этот абстрактный класс является родительским классом для всех классов буферов в API пакета `java.nio`. Буфер – это линейная (конечная) последовательность значений простых типов. В пакете `java.nio` определены потомки класса `Buffer` для всех типов Java, кроме булевого. В самом классе `Buffer` определены общие для всех буферов методы, независимые от конкретного типа. Класс `Buffer` и его потомки предназначены для использования только одним потоком выполнения. В них нет кода для синхронизации потоков, поэтому применять буфер с несколькими потоками небезопасно.

Назначение буфера – хранить данные. В классах буферов должны быть определены методы для чтения и записи этих данных. Поскольку все потомки класса `Buffer` хранят данные различных простых типов, методы `get()` и `put()` для чтения и записи данных должны быть по-разному определены в каждом классе. Эти методы подробно описаны в классе `ByteBuffer` (самом главном среди буферов); во всех остальных потомках класса `Buffer` определены такие же методы, отличающиеся только типами данных, с которыми они работают.

С каждым буфером связано четыре числовых значения:

Емкость (Capacity)

Емкость – это максимальный размер буфера; в нем можно хранить элементы, количество которых соответствует емкости. Емкость указывается при создании буфера и не может после этого изменяться. Ее можно узнать методом `capacity()`.

Граница (Limit)

Граница буфера определяет его текущий размер или индекс первого элемента, не относящегося к буферу. Нельзя читать или записывать данные за этой границей. При записи данных в буфер граница обычно соответствует емкости. При чтении данных из буфера уровень границы может быть меньше величины емкости, и в этом случае граница определяет количество полезных данных, хранящихся в буфере. Есть два метода `limit()`: первый для чтения границы буфера, а второй – для ее определения.

Текущая позиция (Position)

Текущая позиция – это индекс элемента в буфере, с которого начинается чтение или запись данных. Этот параметр применяют и изменяют методы `get()` и `put()` класса `ByteBuffer` и других потомков класса `Buffer`. Есть два метода `position()`: один запрашивает текущую позицию, а другой ее устанавливает. Текущая позиция буфера всегда больше или равна нулю и меньше или равна границе буфера. Метод `remaining()` возвращает количество элементов буфера, находящихся между текущей позицией и границей. Метод `hasRemaining()` возвращает `true`, если это число больше нуля.

Метка (Mark)

Метка буфера – это временная позиция. Метод `mark()` устанавливает метку на текущей позиции. Метод `reset()` устанавливает текущую позицию в позицию метки.

В `Buffer` определены важные методы, выполняющие основные буферные операции:

`clear()`

В действительности этот метод не очищает содержимое буфера, а устанавливает текущую позицию в нуль, далее устанавливает границу, равную емкости буфера, и сбрасывает все метки. После этого буфер готов к записи новых данных.

`flip()`

Этот метод устанавливает границу, равную текущей позиции, а затем устанавливает позицию в нуль и сбрасывает все метки. После записи данных в буфер этим методом можно «переворачивать» буфер, готовя его к чтению.

`rewind()`

Этот метод устанавливает позицию в нуль и сбрасывает все метки. Он не изменяет значение границы буфера. Этот метод можно использовать при повторном чтении данных из буфера с начальной позиции.

Объекты класса `Buffer` предназначены только для чтения; при любой попытке сохранить данные в буфере вызывается исключение `ReadOnlyBufferException`. Метод `isReadOnly()` возвращает `true`, если буфер предназначен только для чтения.

```
public abstract class Buffer {
// Конструктор отсутствует
// Открытые методы экземпляра
   public final int capacity();
   public final Buffer clear();
   public final Buffer flip();
   public final boolean hasRemaining();
   public abstract boolean isReadOnly();
   public final int limit();
   public final Buffer limit(int newLimit);
   public final Buffer mark();
   public final int position();
   public final Buffer position(int newPosition);
   public final int remaining();
   public final Buffer reset();
   public final Buffer rewind();
}
```

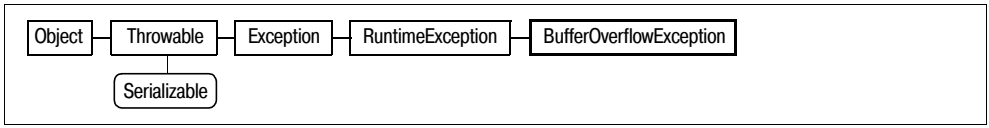
Подклассы: `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, `ShortBuffer`

Возвращается методами: `Buffer.{clear(), flip(), limit(), mark(), position(), reset(), rewind()}`

BufferOverflowException Java 1.4

java.nio ***сериализуемое, непроверяемое***

Это исключение генерируется, если метод `put()` буфера не может завершиться из-за того, что количество элементов, которые нужно записать, превышает количество элементов между текущей позицией буфера и его границей.

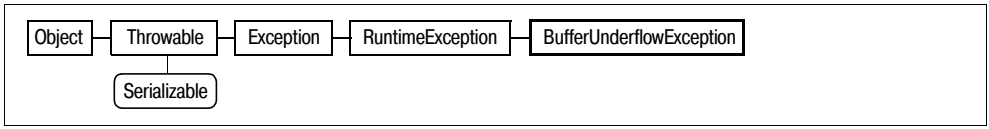



```
public class BufferOverflowException extends RuntimeException {
// Открытые конструкторы
   public BufferOverflowException();
}
```

BufferUnderflowException Java 1.4

java.nio ***сериализуемое, непроверяемое***

Это исключение возникает, если метод `get()` буфера не может завершиться из-за того, что количество элементов, которые нужно прочитать, превышает количество элементов между текущей позицией и границей буфера.




```
public class BufferUnderflowException extends RuntimeException {
// Открытые конструкторы
   public BufferUnderflowException();
}
```

ByteBuffer Java 1.4

java.nio ***сравнимый***

Класс `ByteBuffer` хранит последовательность байтовых значений и применяется в операциях ввода/вывода. `ByteBuffer` — это абстрактный класс: нельзя создать его экземпляр, вызвав конструктор. Вместо этого нужно использовать методы `allocate()`, `allocateDirect()` или `wrap()`.

`allocate()` возвращает объект `ByteBuffer` с указанной емкостью. Текущая позиция нового буфера равна нулю, а его граница равна емкости. Метод `allocateDirect()` похож на `allocate()` за исключением того, что он пытается разместить буфер в памяти таким образом, чтобы его могла непосредственно использовать операционная система. При низкоуровневом вводе/выводе такой буфер существенно эффективнее, чем обычный, но он требует значительно больших затрат ресурсов при размещении данных.

Если в памяти уже размещен массив байтовых значений, то с помощью метода `wrap()` можно создать объект `ByteBuffer`, использующий этот массив для хранения данных.

В методе wrap() с одним аргументом можно указать только массив; емкость буфера и его граница устанавливаются равными длине массива, а текущая позиция равна нулю. В другом варианте метода wrap() можно кроме массива указать смещение, с которого начинается определенная часть массива, и длину этой части. Емкость созданного объекта ByteBuffer будет равна длине всего массива, но текущая позиция будет равна указанному смещению. Граница массива будет установлена на позиции, которая соответствует сумме длины части массива и смещения.

После создания объекта ByteBuffer можно использовать различные методы get() и put() для чтения и записи данных. Некоторые из этих методов предназначены для записи и чтения отдельных байтовых значений, а другие – для работы с массивами. Методы, оперирующие отдельными байтовыми значениями, имеют две разновидности. Первые («относительные») производят чтение или запись на текущей позиции, после чего увеличивают ее на 1. Второй вариант этих методов («абсолютный») принимает дополнительный аргумент, определяющий элемент буфера, который надо прочитать или записать, и не изменяют текущую позицию метода. Два варианта метода get() читают последовательность байтов (начиная с текущей позиции и соответствующим образом увеличивая ее) в указанный массив или часть массива. Эти методы вызывают исключение BufferUnderflowException, если в буфере осталось недостаточно байтов. Два варианта методов put() («относительные») копируют байты указанного массива или части массива в буфер, начиная с текущей позиции и при этом инкрементируя ее. Они вызывают исключение BufferOverflowException, если в буфере недостаточно места для этих байтов. Еще одна форма метода put() записывает данные из указанного объекта ByteBuffer в рабочий буфер, увеличивая при этом текущую позицию в обоих буферах.

Кроме методов get() и put(), в классе ByteBuffer определены другие операции, изменяющие содержимое буфера. Метод compact() отбрасывает все байты до текущей позиции и копирует в начало буфера все данные, находящиеся между текущей позицией и границей. Далее позиция устанавливается на новую границу, а граница устанавливается равной емкости буфера. Этот метод компактно размещает данные в буфере, отбрасывая элементы, которые уже были прочитаны, и готовит буфер к дозаписи новых данных.

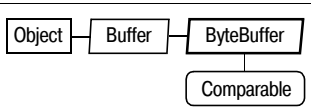
Все потомки класса Buffer, такие как CharBuffer, IntBuffer, FloatBuffer, имеют методы, аналогичные рассмотренным get() и put(), но оперирующие с другими типами данных. ByteBuffer выделяется из этих классов, так как в нем есть дополнительные методы для чтения и записи значений других простых типов в/из байтового буфера. Эти методы имеют различные названия, начинающиеся на get и put, например getInt() и putChar(). Такие методы есть для всех простых типов, кроме byte и boolean. Все эти методы читают или записывают отдельные значения. Подобно методам get() и put(), они имеют «относительные» и «абсолютные» варианты. «Относительные» методы начинают выполнять соответствующую операцию над байтом текущей позиции буфера и увеличивают позицию на соответствующее значение (два байта для char, четыре для int, восемь для double и т. д.). «Абсолютные» методы принимают в качестве аргумента индекс (байтовый индекс, который не умножается на размер значения примитивного типа) и не изменяют текущую позицию буфера. Значения простых типов, размер которых больше одного байта, могут быть записаны в буфер в обратном порядке (самый старший байт будет записываться первым) или в прямом порядке. В методах get() и put() порядок байтов указывается в виде объекта ByteOrder. Порядок байтов в объекте ByteBuffer может быть запрошен или установлен с помощью двух видов метода order(). По умолчанию порядок байтов во всех вновь созданных объектах ByteBuffer имеет значение ByteOrder.BIG_ENDIAN.

Только в классе ByteBuffer() есть методы set(), позволяющие представлять байтовый буфер как буфер для других простых типов. asCharBuffer(), asIntBuffer() и другие подобные методы возвращают буфер представления, в котором байты ByteBuffer от текущей позиции и до границы будут представлены как последовательность символов, целых чисел или значений других простых типов. Возвращаемые буферы имеют значения текущей позиции, границы и меток, не зависящие от соответствующих величин в исходном буфере. Начальная текущая позиция в новом буфере равна нулю, а его граница и емкость равны количеству байт между текущей позицией и границей исходного буфера, разделенному на размер в байтах соответствующего типа (2 – для char и short, 4 – для int и float, 8 – для long и double). Обратите внимание, что возвращенный буфер представления содержит данные, находящиеся между текущей позицией и границей байтового буфера. Последующие изменения размера исходного буфера не влияют на размер буфера представления, но изменения значений байтов в исходном буфере влекут за собой изменения соответствующих значений в буфере представления. В буфере представления применяется порядок байтов, который был в момент его создания в исходном буфере; последующие изменения порядка байтов исходного буфера не влекут за собой изменения в буфере представления. Если исходный байтовый буфер является прямым буфером, то возвращается также прямой буфер; это важно, так как ByteBuffer является единственным классом буфера, имеющим метод allocateDirect().

В классе ByteBuffer определено несколько дополнительных методов, которые, подобно get() и put(), имеют свои аналоги в других буферных классах. Метод duplicate() возвращает новый буфер, который имеет то же содержимое, что и исходный. Два буфера имеют независимые текущие позиции, границы и метки, но дублирующий буфер создается с такими же значениями этих параметров, как в исходном буфере. Дублирующий буфер является прямым, если исходный буфер прямой, и предназначен только для чтения, если исходный буфер предназначен только для чтения. Оба буфера имеют общее содержимое, а изменения содержимого одного из буферов видны из другого буфера. Метод asReadOnlyBuffer() выполняет те же действия, что и duplicate(), за исключением того, что возвращает буфер только для чтения, у которого все методы put() и другие подобные методы всегда будут вызывать исключение ReadOnlyBufferException. slice() похож на метод duplicate(), но возвращаемый им буфер представляет данные из исходного буфера, находящиеся между текущей позицией и границей. Возвращаемый буфер имеет текущую позицию, равную нулю, а его граница и емкость равны количеству оставшихся элементов в исходном буфере. Метка в новом буфере не определена. Метод isDirect() возвращает true, если буфер прямой, и false в противном случае. Если данный буфер имеет в своей основе массив (например, если он был создан методом allocate() или wrap()), то hasArray() возвращает true; array() возвращает этот массив, а arrayOffset() – смещение первого элемента буфера в этом массиве. Если hasArray() возвращает false, то методы array() и arrayOffset() будут вызывать исключение UnsupportedOperationException или ReadOnlyBufferException.

Наконец, в классе ByteBuffer и в остальных потомках класса Buffer замещены стандартные методы класса Object. Метод equals() сравнивает элементы двух буферов, находящиеся между текущей позицией и границей, и возвращает true только в том случае, если в обоих буферах количество таких элементов одинаково, а их значения совпадают. Обратите внимание, что при сравнении не учитываются элементы до текущей позиции буфера. Метод hashCode() реализован таким же образом, как метод equals(): хеш-код вычисляется только на основе элементов между текущей позицией и границей буфера. Это означает, что хеш-код меняется, если меняется содержимое буфера или текущая позиция. То есть экземпляры класса ByteBuffer и других потомков

класса Buffer не всегда подходят в качестве ключей для хеш-таблицы или объекта java.util.Map. Метод toString() возвращает сводную строку для данного буфера, но точное содержимое этой строки не определено. ByteBuffer и все остальные потомки класса Buffer реализуют интерфейс Comparable и определяют метод compareTo(), который производит поэлементное сравнение буферов между текущей позицией и границей.




```
public abstract class ByteBuffer extends Buffer implements Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static ByteBuffer allocate(int capacity);
   public static ByteBuffer allocateDirect(int capacity);
   public static ByteBuffer wrap(byte[] array);
   public static ByteBuffer wrap(byte[] array, int offset, int length);
// Методы доступа к свойствам (по имени свойства)
   public abstract char getChar();
   public abstract char getChar(int index);
   public abstract boolean isDirect();
   public abstract double getDouble();
   public abstract double getDouble(int index);
   public abstract float getFloat();
   public abstract float getFloat(int index);
   public abstract int getInt();
   public abstract int getInt(int index);
   public abstract long getLong();
   public abstract long getLong(int index);
   public abstract short getShort();
   public abstract short getShort(int index);
// Открытые методы экземпляра
   public final byte[] array();
   public final int arrayOffset();
   public abstract CharBuffer asCharBuffer();
   public abstract DoubleBuffer asDoubleBuffer();
   public abstract FloatBuffer asFloatBuffer();
   public abstract IntBuffer asIntBuffer();
   public abstract LongBuffer asLongBuffer();
   public abstract ByteBuffer asReadOnlyBuffer();
   public abstract ShortBuffer asShortBuffer();
   public abstract ByteBuffer compact();
   public abstract ByteBuffer duplicate();
   public abstract byte get();
   public abstract byte get(int index);
   public ByteBuffer get(byte[] dst);
   public ByteBuffer get(byte[] dst, int offset, int length);
   public final boolean hasArray();
   public final ByteOrder order();
   public final ByteBuffer order(ByteOrder bo);
   public ByteBuffer put(ByteBuffer src);
   public abstract ByteBuffer put(byte b);
   public final ByteBuffer put(byte[] src);
   public abstract ByteBuffer put(int index, byte b);
```

```
   public ByteBuffer put(byte[] src, int offset, int length);
   public abstract ByteBuffer putChar(char value);
   public abstract ByteBuffer putChar(int index, char value);
   public abstract ByteBuffer putDouble(double value);
   public abstract ByteBuffer putDouble(int index, double value);
   public abstract ByteBuffer putFloat(float value);
   public abstract ByteBuffer putFloat(int index, float value);
   public abstract ByteBuffer putInt(int value);
   public abstract ByteBuffer putInt(int index, int value);
   public abstract ByteBuffer putLong(long value);
   public abstract ByteBuffer putLong(int index, long value);
   public abstract ByteBuffer putShort(short value);
   public abstract ByteBuffer putShort(int index, short value);
   public abstract ByteBuffer slice();
// Методы, реализующие Comparable
   public int compareTo(Object ob);
// Открытые методы, замещающие Object
   public boolean equals(Object ob);
   public int hashCode();
   public String toString();
}
```

Подклассы: MappedByteBuffer

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

ByteOrder Java 1.4

java.nio

Этот класс является перечислением порядков байтов, используемых классом ByteBuffer. Данное перечисление обеспечивает безопасность типов. В двух константных полях определены два допустимых значения для порядков байтов. BIG_ENDIAN соответствует обратному порядку, в котором первым записывается старший байт. LITTLE_ENDIAN соответствует прямому порядку: младший байт записывается первым. Статический метод nativeOrder() возвращает одну из этих констант, которая соответствует порядку, используемому операционной системой и аппаратным обеспечением. Последний метод, toString(), возвращает строку «BIG_ENDIAN» (обратный порядок) или «LITTLE_ENDIAN» (прямой).

```
public final class ByteOrder {
// Конструктор отсутствует
// Открытые константы
   public static final ByteOrder BIG_ENDIAN;
   public static final ByteOrder LITTLE_ENDIAN;
// Открытые методы класса
   public static ByteOrder nativeOrder();
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методам: ByteBuffer.order(), javax.imageio.stream.ImageInputStream.setByteOrder(), javax.imageio.stream.ImageInputStreamImpl.setByteOrder()

Возвращается методами: ByteBuffer.order(), ByteOrder.nativeOrder(), CharBuffer.order(), DoubleBuffer.order(), FloatBuffer.order(), IntBuffer.order(),

LongBuffer.order(), ShortBuffer.order(), javax.imageio.stream.ImageInputStream.getByteOrder(), javax.imageio.stream.ImageInputStreamImpl.getByteOrder()

Экземпляры: ByteOrder.{BIG_ENDIAN, LITTLE_ENDIAN}, javax.imageio.stream.ImageInputStreamImpl.byteOrder

CharBuffer

Java 1.4

java.nio ***сравнимый***

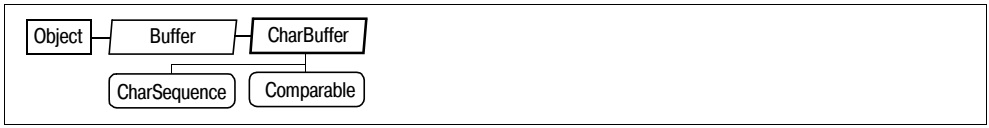
Класс CharBuffer содержит последовательность символов Unicode. Он предназначен для использования в операциях ввода/вывода. Большая часть методов этого класса является прямым подобием методов ByteBuffer за исключением того, что они принимают аргументы и возвращают значения типа char и char[] вместо byte и byte[]. Подробнее эти методы представлены в описании ByteBuffer.

Кроме того, этот класс реализует интерфейс java.lang.CharSequence, поэтому его можно применять в операциях с регулярными выражениями, которые определены в классе java.util.regex. Также CharBuffer можно задействовать там, где ожидается CharSequence.

Обратите внимание на то, CharBuffer – это абстрактный класс, который не имеет конструктора. Получить объект CharBuffer можно тремя способами:

- Вызвать статический метод allocate(). Необходимо заметить, что в этом классе нет метода allocateDirect(), который присутствует в классе ByteBuffer.
- Вызвать статический метод wrap(), чтобы создать объект CharBuffer, использующий содержимое указанного массива символов или объекта CharSequence. Обратите внимание, что при этом объект CharBuffer создается только для чтения.
- Вызвать метод asCharBuffer() объекта ByteBuffer, чтобы получить представление байтового буфера в виде объекта CharBuffer. Если данный ByteBuffer является прямым буфером, то полученный CharBuffer тоже будет прямым.

Заметьте, что этот класс содержит последовательность 16-битных символов Unicode и не может содержать текст в другой кодировке. Классы пакета java.nio.charset можно задействовать для преобразования буфера CharBuffer, содержащего символы Unicode, в объект ByteBuffer, а также для обратного преобразования.




```
public abstract class CharBuffer extends Buffer implements CharSequence, Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static CharBuffer allocate(int capacity);
   public static CharBuffer wrap(CharSequence csq);
   public static CharBuffer wrap(char[] array);
   public static CharBuffer wrap(char[] array, int offset, int length);
   public static CharBuffer wrap(CharSequence csq, int start, int end);
// Открытые методы экземпляра
   public final char[] array();
   public final int arrayOffset();
   public abstract CharBuffer asReadOnlyBuffer();
```

```
    public abstract CharBuffer compact();
    public abstract CharBuffer duplicate();
    public abstract char get();
    public abstract char get(int index);
    public CharBuffer get(char[] dst);
    public CharBuffer get(char[] dst, int offset, int length);
    public final boolean hasArray();
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public final CharBuffer put(String src);
    public abstract CharBuffer put(char c);
    public CharBuffer put(CharBuffer src);
    public final CharBuffer put(char[] src);
    public abstract CharBuffer put(int index, char c);
    public CharBuffer put(char[] src, int offset, int length);
    public CharBuffer put(String src, int start, int end);
    public abstract CharBuffer slice();
// Методы, реализующие CharSequence
    public final char charAt(int index);
    public final int length();
    public abstract CharSequence subSequence(int start, int end);
    public String toString();
// Методы, реализующие Comparable
    public int compareTo(Object ob);
// Открытые методы, замещающие Object
    public boolean equals(Object ob);
    public int hashCode();
}
```

Передается методам: `CharBuffer.put()`, `java.nio.charset.Charset.encode()`, `java.nio.charset.CharsetDecoder.{decode(), decodeLoop(), flush(), implFlush()}`, `java.nio.charset.CharsetEncoder.{encode(), encodeLoop()}`

Возвращается методами: Методов слишком много, чтобы их перечислить.

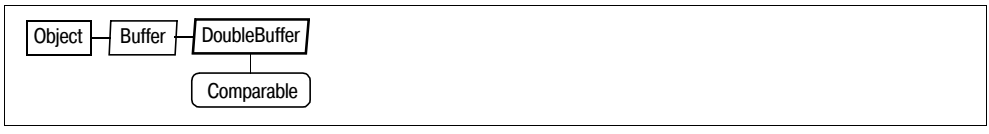
DoubleBuffer

Java 1.4

java.nio ***сравнимый***

Объект `DoubleBuffer` содержит последовательность значений типа `double` и используется в операциях ввода/вывода. Большая часть методов этого класса эквивалентна методам класса `ByteBuffer`, за исключением того, что методы `DoubleBuffer` используют `double` и `double[]` вместо `byte` и `byte[]` в качестве типов аргументов и возвращаемых значений. Этих методы представлены в описании `ByteBuffer`.

Класс `DoubleBuffer` – это абстрактный класс, который не имеет конструктора. Создать экземпляр этого класса можно с помощью статических методов `allocate()` или `wrap()`, которые аналогичны одноименным методам класса `ByteBuffer`. Представление объекта `ByteBuffer` можно создать в виде `DoubleBuffer`, вызвав метод `asDoubleBuffer()` данного объекта.



```
public abstract class DoubleBuffer extends Buffer implements Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static DoubleBuffer allocate(int capacity);
   public static DoubleBuffer wrap(double[] array);
   public static DoubleBuffer wrap(double[] array, int offset, int length);
// Открытые методы экземпляра
   public final double[] array();
   public final int arrayOffset();
   public abstract DoubleBuffer asReadOnlyBuffer();
   public abstract DoubleBuffer compact();
   public abstract DoubleBuffer duplicate();
   public abstract double get();
   public abstract double get(int index);
   public DoubleBuffer get(double[] dst);
   public DoubleBuffer get(double[] dst, int offset, int length);
   public final boolean hasArray();
   public abstract boolean isDirect();
   public abstract ByteOrder order();
   public DoubleBuffer put(DoubleBuffer src);
   public abstract DoubleBuffer put(double d);
   public final DoubleBuffer put(double[] src);
   public abstract DoubleBuffer put(int index, double d);
   public DoubleBuffer put(double[] src, int offset, int length);
   public abstract DoubleBuffer slice();
// Методы, реализующие Comparable
   public int compareTo(Object ob);
// Открытые методы, замещающие Object
   public boolean equals(Object ob);
   public int hashCode();
   public String toString();
}
```

Передается методам: DoubleBuffer.put()

Возвращается методами: ByteBuffer.asDoubleBuffer(), DoubleBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap()}

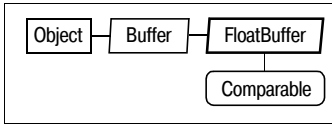
FloatBuffer

Java 1.4

java.nio ***сравнимый***

FloatBuffer содержит последовательность значений float и используется в операциях ввода/вывода. Большая часть методов этого класса аналогична методам класса ByteBuffer за исключением того, что методы FloatBuffer используют float и float[] вместо byte и byte[] в качестве типов аргументов и возвращаемых значений. Более подробно эти методы представлены в описании ByteBuffer.

Класс FloatBuffer является абстрактным и не имеет конструктора. Создать его экземпляр можно с помощью статических методов allocate() или wrap(), которые аналогичны одноименным методам класса ByteBuffer. Представление объекта ByteBuffer можно создать в виде FloatBuffer, вызвав метод asFloatBuffer() этого объекта.




```
public abstract class FloatBuffer extends Buffer implements Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static FloatBuffer allocate(int capacity);
   public static FloatBuffer wrap(float[] array);
   public static FloatBuffer wrap(float[] array, int offset, int length);
// Открытые методы экземпляра
   public final float[] array();
   public final int arrayOffset();
   public abstract FloatBuffer asReadOnlyBuffer();
   public abstract FloatBuffer compact();
   public abstract FloatBuffer duplicate();
   public abstract float get();
   public abstract float get(int index);
   public FloatBuffer get(float[] dst);
   public FloatBuffer get(float[] dst, int offset, int length);
   public final boolean hasArray();
   public abstract boolean isDirect();
   public abstract ByteOrder order();
   public FloatBuffer put(FloatBuffer src);
   public abstract FloatBuffer put(float f);
   public final FloatBuffer put(float[] src);
   public abstract FloatBuffer put(int index, float f);
   public FloatBuffer put(float[] src, int offset, int length);
   public abstract FloatBuffer slice();
// Методы, реализующие Comparable
   public int compareTo(Object ob);
// Открытые методы, замещающие Object
   public boolean equals(Object ob);
   public int hashCode();
   public String toString();
}
```

Передается методам: `FloatBuffer.put()`

Возвращается методами: `ByteBuffer.asFloatBuffer()`, `FloatBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap()}`

IntBuffer

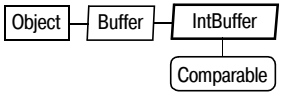
Java 1.4

java.nio ***сравнимый***

Объект `IntBuffer` содержит последовательность значений типа `int` и применяется в операциях ввода/вывода. Большая часть методов этого класса аналогична методам класса `ByteBuffer` за исключением того, что методы `IntBuffer` используют `int` и `int[]` вместо `byte` и `byte[]` в качестве типов аргументов и возвращаемых значений. Более подробно эти методы представлены в описании `ByteBuffer`.

Класс `IntBuffer` является абстрактным и не имеет конструктора. Создать его экземпляр можно с помощью статических методов `allocate()` или `wrap()`, которые аналогич-

ны одноименным методам класса ByteBuffer. Представление объекта ByteBuffer можно создать в виде IntBuffer, вызвав метод asIntBuffer() этого объекта.




```
public abstract class IntBuffer extends Buffer implements Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static IntBuffer allocate(int capacity);
   public static IntBuffer wrap(int[] array);
   public static IntBuffer wrap(int[] array, int offset, int length);
// Открытые методы экземпляра
   public final int[] array();
   public final int arrayOffset();
   public abstract IntBuffer asReadOnlyBuffer();
   public abstract IntBuffer compact();
   public abstract IntBuffer duplicate();
   public abstract int get();
   public abstract int get(int index);
   public IntBuffer get(int[] dst);
   public IntBuffer get(int[] dst, int offset, int length);
   public final boolean hasArray();
   public abstract boolean isDirect();
   public abstract ByteOrder order();
   public IntBuffer put(IntBuffer src);
   public abstract IntBuffer put(int i);
   public final IntBuffer put(int[] src);
   public abstract IntBuffer put(int index, int i);
   public IntBuffer put(int[] src, int offset, int length);
   public abstract IntBuffer slice();
// Методы, реализующие Comparable
   public int compareTo(Object ob);
// Открытые методы, замещающие Object
   public boolean equals(Object ob);
   public int hashCode();
   public String toString();
}
```

Передается методам: IntBuffer.put()

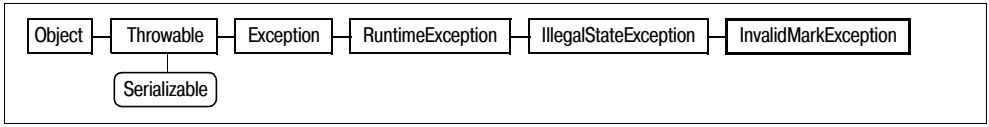
Возвращается методами: ByteBuffer.asIntBuffer(), IntBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap()}

InvalidMarkException

Java 1.4

java.nio ***сериализуемое, непроверяемое***

Это исключение возникает, если метод reset() не может установить новое значение текущей позиции буфера из-за того, что не определена метка.




```
public class InvalidMarkException extends IllegalStateException {
// Открытые конструкторы
   public InvalidMarkException();
}
```

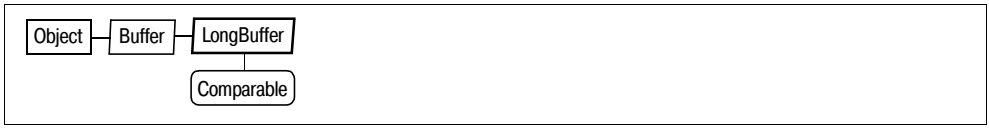
LongBuffer

Java 1.4

java.nio ***сравнимый***

Объект LongBuffer содержит последовательность значений типа long и применяется в операциях ввода/вывода. Большая часть методов этого класса аналогична методам класса ByteBuffer за исключением того, что методы LongBuffer используют long и long[] вместо byte и byte[] в качестве типов аргументов и возвращаемых значений. Более подробно эти методы представлены в описании ByteBuffer.

Класс LongBuffer является абстрактным и не имеет конструктора. Создать его экземпляр можно с помощью статических методов allocate() или wrap(), которые аналогичны одноименным методам класса ByteBuffer. Представление объекта ByteBuffer можно создать в виде LongBuffer, вызвав метод asLongBuffer() этого объекта.




```
public abstract class LongBuffer extends Buffer
       implements Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static LongBuffer allocate(int capacity);
   public static LongBuffer wrap(long[] array);
   public static LongBuffer wrap(long[] array, int offset, int length);
// Открытые методы экземпляра
   public final long[] array();
   public final int arrayOffset();
   public abstract LongBuffer asReadOnlyBuffer();
   public abstract LongBuffer compact();
   public abstract LongBuffer duplicate();
   public abstract long get();
   public abstract long get(int index);
   public LongBuffer get(long[] dst);
   public LongBuffer get(long[] dst, int offset, int length);
   public final boolean hasArray();
   public abstract boolean isDirect();
   public abstract ByteOrder order();
   public LongBuffer put(LongBuffer src);
   public abstract LongBuffer put(long l);
   public final LongBuffer put(long[] src);
   public abstract LongBuffer put(int index, long l);
```

```
    public LongBuffer put(long[] src, int offset, int length);
    public abstract LongBuffer slice();
// Методы, реализующие Comparable
    public int compareTo(Object ob);
// Открытые методы, замещающие Object
    public boolean equals(Object ob);
    public int hashCode();
    public String toString();
}
```

Передается методам: `LongBuffer.put()`

Возвращается методами: `ByteBuffer.asLongBuffer()`, `LongBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap()}`

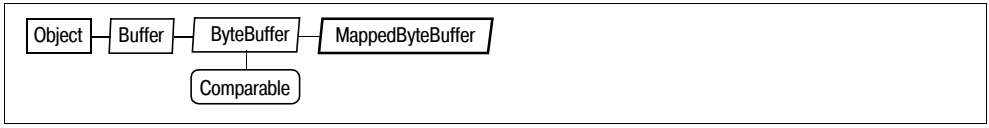
MappedByteBuffer

Java 1.4

java.nio ***сравнимый***

Этот класс является буфером типа `ByteBuffer` и представляет отображение в память части файла. Создать объект `MappedByteBuffer` можно с помощью метода `map()` класса `java.nio.channels.FileChannel`. Все объекты класса `MappedByteBuffer` являются прямыми буферами. Метод `isLoaded()` возвращает информацию о том, где сейчас находится содержимое буфера: в первичной памяти (`true`) или на диске (`false`). Если возвращается `true`, то операции с буфером, возможно, будут протекать быстрее. Метод `load()` запрашивает, но не требует загрузки содержимого буфера в первичную память. Выполнение этой операции не гарантируется. Если буфер находится в режиме чтения и записи, то метод `force()` записывает все изменения содержимого буфера в данный файл. Если файл находится на локальном устройстве, то гарантируется, что его содержимое будет обновлено до возвращения из метода `force()`. Для сетевых файлов такие гарантии не даются.

Обратите внимание, что файл, связанный с буфером `MappedByteBuffer`, может иметь коллективный доступ, то есть содержимое такого буфера может изменяться асинхронно, если содержимое файла изменено другим потоком или процессом (асинхронные изменения данного файла могут не влиять на буфер; это зависит от платформы). Более того, если другой поток или процесс усекает файл, некоторые или даже все элементы буфера больше не будут соответствовать содержимому файла. В этом случае попытка прочитать или записать недоступный элемент буфера вызовет сразу же или спустя некоторое время исключение, определяемое реализацией Java.



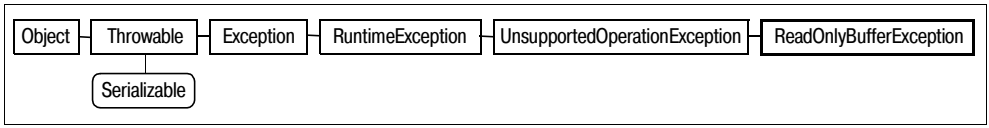

```
public abstract class MappedByteBuffer extends ByteBuffer {
// Конструктор отсутствует
// Открытые методы экземпляра
    public final MappedByteBuffer force();
    public final boolean isLoaded();
    public final MappedByteBuffer load();
}
```

Возвращается методами: `MappedByteBuffer.{force(), load()}`, `java.nio.channels.FileChannel.map()`

ReadOnlyBufferException Java 1.4

java.nio ***сериализуемое, непроверяемое***

Это исключение сообщает о том, что буфер создан только для чтения, а его методам `put()` и `compact()` не разрешено изменять содержимое буфера.



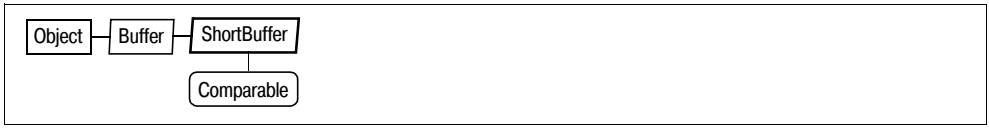

```
public class ReadOnlyBufferException extends UnsupportedOperationException {
// Открытые конструкторы
   public ReadOnlyBufferException();
}
```

ShortBuffer Java 1.4

java.nio ***сравнимый***

Объект `ShortBuffer` содержит последовательность значений типа `short` и применяется в операциях ввода/вывода. Большая часть методов этого класса аналогична методам класса `ByteBuffer` за исключением того, что методы `ShortBuffer` используют `short` и `short[]` вместо `byte` и `byte[]` в качестве типов аргументов и возвращаемых значений. Более подробно эти методы представлены в описании `ByteBuffer`.

Класс `ShortBuffer` является абстрактным и не имеет конструктора. Создать его экземпляр можно с помощью статических методов `allocate()` или `wrap()`, которые аналогичны одноименным методам класса `ByteBuffer`. Представление объекта `ByteBuffer` можно создать в виде `ShortBuffer`, вызвав метод `asShortBuffer()` этого объекта.




```
public abstract class ShortBuffer extends Buffer implements Comparable {
// Конструктор отсутствует
// Открытые методы класса
   public static ShortBuffer allocate(int capacity);
   public static ShortBuffer wrap(short[] array);
   public static ShortBuffer wrap(short[] array, int offset, int length);
// Открытые методы экземпляра
   public final short[] array();
   public final int arrayOffset();
   public abstract ShortBuffer asReadOnlyBuffer();
   public abstract ShortBuffer compact();
   public abstract ShortBuffer duplicate();
   public abstract short get();
   public abstract short get(int index);
   public ShortBuffer get(short[] dst);
   public ShortBuffer get(short[] dst, int offset, int length);
   public final boolean hasArray();
```

```
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public ShortBuffer put(ShortBuffer src);
    public abstract ShortBuffer put(short s);
    public final ShortBuffer put(short[] src);
    public abstract ShortBuffer put(int index, short s);
    public ShortBuffer put(short[] src, int offset, int length);
    public abstract ShortBuffer slice();
// Методы, реализующие Comparable
    public int compareTo(Object ob);
// Открытые методы, замещающие Object
    public boolean equals(Object ob);
    public int hashCode();
    public String toString();
}
```

Передается методам: ShortBuffer.put()

Возвращается методами: ByteBuffer.asShortBuffer(), ShortBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap()}

Пакет java.nio.channels

Java 1.4

Это центральный пакет нового API ввода/вывода. Channel – это канал связи для передачи байтов в объект java.nio.ByteBuffer или из него. Каналы служат для той же цели, что и потоки InputStream и OutputStream пакета java.io, но никак не зависят от классов этого пакета и имеют некоторые важные особенности, которых нет в API пакета java.io. Класс Channels определяет методы, связывающие API java.io и API java.nio.channels. Они возвращают каналы, основанные на потоках, или потоки, созданные на основе каналов.

Интерфейс Channel содержит метод, проверяющий, закрыт ли канал, и метод для закрытия канала. Остальные интерфейсы этого пакета расширяют Channel, дополняя его методами read() и write(), предназначенными для записи байтов из канала в один или несколько байтовых буферов и записи в канал байтов из буфера(ов).

Класс FileChannel определяет методы для чтения и записи файлов, основанные на каналах, и предоставляет дополнительные возможности для работы с файлами, которых нет в пакете java.io, например захват файла (locking) и отображение файла в память. Классы SocketChannel, ServerSocketChannel и DatagramChannel представляют каналы для связи по сети. В классе Pipe определены два внутренних класса, использующие абстракцию каналов для взаимодействия потоков.

Сетевые и межпотоковые каналы наследуют класс SelectableChannel и могут работать в неблокируемом режиме, когда методы read() и write() возвращаются сразу же, даже если канал не готов для чтения или записи. Неблокируемый ввод/вывод и работа в сети, неосуществимые с помощью абстракции потоков из пакетов java.io и java.net, являются самым главным нововведением API java.nio. Selector является ключевым классом в неблокируемом вводе/выводе. Он необходим для эффективного использования неблокируемых каналов и позволяет программе, производящей операции ввода/вывода, регистрироваться одновременно на нескольких каналах. Метод select() этого класса блокируется до тех пор, пока какой-нибудь канал не будет готов для ввода/вывода. Эта технология незаменима при написании масштабируемых высокоэффективных сетевых служб. Подробнее об этом рассказано в описании классов Selector и SelectionKey.

Наконец, этот пакет предоставляет средства для многоуровневой обработки ошибок в виде многочисленных классов исключений. Некоторые из них могут вызываться только одним методом в `java.nio` API.

Интерфейсы

```
public interface ByteChannel extends ReadableByteChannel, WritableByteChannel;
public interface Channel;
public interface GatheringByteChannel extends WritableByteChannel;
public interface InterruptibleChannel extends Channel;
public interface ReadableByteChannel extends Channel;
public interface ScatteringByteChannel extends ReadableByteChannel;
public interface WritableByteChannel extends Channel;
```

Классы

```
public final class Channels;
public abstract class DatagramChannel extends java.nio.channels.spi.AbstractSelectableChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel;
public abstract class FileChannel extends java.nio.channels.spi.AbstractInterruptibleChannel
implements ByteChannel, GatheringByteChannel, ScatteringByteChannel;
public static class FileChannel.MapMode;
public abstract class FileLock;
public abstract class Pipe;
public abstract static class Pipe.SinkChannel extends java.nio.channels.spi.AbstractSelectableChannel
    implements GatheringByteChannel,WritableByteChannel;
public abstract static class Pipe.SourceChannel
    extends java.nio.channels.spi.AbstractSelectableChannel
    implements ReadableByteChannel, ScatteringByteChannel;
public abstract class SelectableChannel extends java.nio.channels.spi.AbstractInterruptibleChannel
    implements Channel;
public abstract class SelectionKey;
public abstract class Selector;
public abstract class ServerSocketChannel extends java.nio.channels.spi.AbstractSelectableChannel;
public abstract class SocketChannel extends java.nio.channels.spi.AbstractSelectableChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel;
```

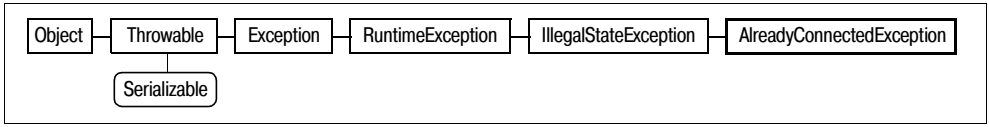
Исключения

```
public class AlreadyConnectedException extends IllegalStateException;
public class CancelledKeyException extends IllegalStateException;
public class ClosedChannelException extends java.io.IOException;
   └ public class AsynchronousCloseException extends ClosedChannelException;
      └ public class ClosedByInterruptException extends AsynchronousCloseException;
public class ClosedSelectorException extends IllegalStateException;
public class ConnectionPendingException extends IllegalStateException;
public class FileLockInterruptionException extends java.io.IOException;
public class IllegalBlockingModeException extends IllegalStateException;
public class IllegalSelectorException extends IllegalArgumentException;
public class NoConnectionPendingException extends IllegalStateException;
public class NonReadableChannelException extends IllegalStateException;
public class NonWritableChannelException extends IllegalStateException;
public class NotYetBoundException extends IllegalStateException;
public class NotYetConnectedException extends IllegalStateException;
public class OverlappingFileLockException extends IllegalStateException;
public class UnresolvedAddressException extends IllegalArgumentException;
public class UnsupportedAddressTypeException extends IllegalArgumentException;
```

AlreadyConnectedException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение вызывается методом `connect()` класса `SocketChannel`, если соединение уже создано.

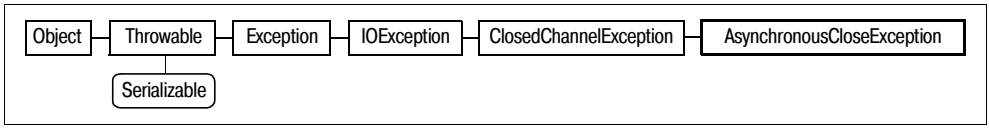



```
public class AlreadyConnectedException extends IllegalStateException {
// Открытые конструкторы
   public AlreadyConnectedException();
}
```

AsynchronousCloseException Java 1.4

java.nio.channels ***сериализуемое, проверяемое***

Это исключение возникает при аварийном завершении операции блокируемого ввода/вывода из-за того, что другой поток асинхронно закрыл канал. См. также `ClosedByInterruptException`.




```
public class AsynchronousCloseException extends ClosedChannelException {
// Открытые конструкторы
   public AsynchronousCloseException();
}
```

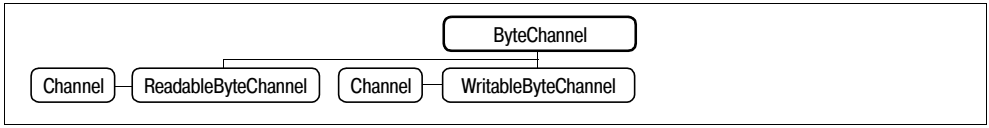
Подклассы: `ClosedByInterruptException`

Генерируется методами: `java.nio.channels.spi.AbstractInterruptibleChannel.end()`

ByteChannel Java 1.4

java.nio.channels

Этот интерфейс расширяет `ReadableByteChannel` и `WritableByteChannel`, но не определяет собственных методов. Он служит для удобства объединения этих двух интерфейсов.



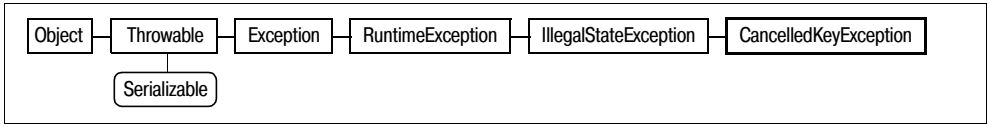

```
public interface ByteChannel extends ReadableByteChannel, WritableByteChannel {
}
```

Реализации: `DatagramChannel`, `FileChannel`, `SocketChannel`

CancelledKeyException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение возникает при попытке повторно вызвать метод `cancel()` объекта `SelectionKey`.




```
public class CancelledKeyException extends IllegalStateException {
// Открытые конструкторы
   public CancelledKeyException();
}
```

Channel Java 1.4

java.nio.channels

Этот интерфейс определяет канал связи для ввода и вывода. `Channel` является высокоуровневым общим интерфейсом и расширяется более специализированными интерфейсами, такими как `ReadableByteChannel` и `WritableByteChannel`. В интерфейсе `Channel` определено только два метода: `isOpen()` определяет, открыт ли канал, а `close()` закрывает канал. Каналы открываются при их создании. Канал закрывается навсегда; после этого он не может применяться для ввода/вывода.

Многие реализации каналов могут прерываться и асинхронно закрываться. Чтобы сообщить об этом, они реализуют интерфейс `InterruptibleChannel`. См. также `InterruptibleChannel`.

```
public interface Channel {
// Открытые методы экземпляра
   public abstract void close() throws java.io.IOException;
   public abstract boolean isOpen();
}
```

Реализации: `InterruptibleChannel`, `ReadableByteChannel`, `SelectableChannel`, `WritableByteChannel`, `java.nio.channels.spi.AbstractInterruptibleChannel`

Channels Java 1.4

java.nio.channels

Этот класс определяет статические методы, связывающие классы байтовых и символьных потоков пакета `java.io` с классами каналов пакета `java.nio.channels`. Класс `Channels` не предназначен для создания экземпляров. В этом классе есть только статические методы. Эти методы создают каналы байтов на основе байтовых потоков `java.io` и байтовые потоки на основе каналов. Обратите внимание, что объект канала, возвращаемый методами `newChannel()`, может не реализовывать `InterruptibleChannel`. Следовательно, такой канал не может быть прерван или асинхронно закрыт в отличие от других каналов этого пакета. В каналах также определены методы для создания потоков символов на основе байтового канала и кодировки символов (`java.io.Re-`

ader и java.io.Writer). Кодировку можно указать через имя набора символов или в виде объекта CharsetDecoder или CharsetEncoder. См. также java.nio.charset.

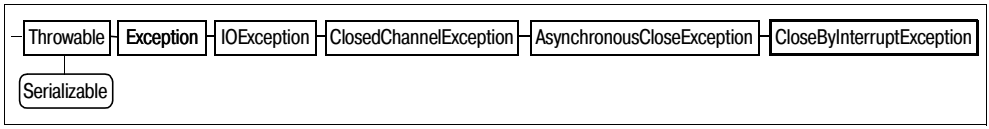
```
public final class Channels {
// Конструктор отсутствует
// Открытые методы класса
   public static ReadableByteChannel newChannel(java.io.InputStream in);
   public static WritableByteChannel newChannel(java.io.OutputStream out);
   public static java.io.InputStream newInputStream(ReadableByteChannel ch);
   public static java.io.OutputStream newOutputStream(WritableByteChannel ch);
   public static java.io.Reader newReader(ReadableByteChannel ch, String csName);
   public static java.io.Reader newReader(ReadableByteChannel ch,
                                          java.nio.charset.CharsetDecoder dec, int minBufferCap);
   public static java.io.Writer newWriter(WritableByteChannel ch, String csName);
   public static java.io.Writer newWriter(WritableByteChannel ch,
                                          java.nio.charset.CharsetEncoder enc, int minBufferCap);
}
```

ClosedByInterruptException

Java 1.4

java.nio.channels ***сериализуемое, проверяемое***

Этот тип исключения вызывается потоком, заблокированным при операции ввода/вывода через канал, когда другой поток вызывал метод interrupt() данного канала. Это исключение является потомком AsynchronousCloseException. Побочным эффектом прерывания потока будет закрытие канала.



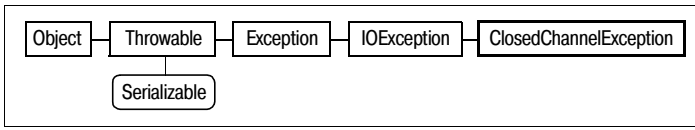

```
public class ClosedByInterruptException extends AsynchronousCloseException {
// Открытые конструкторы
   public ClosedByInterruptException();
}
```

ClosedChannelException

Java 1.4

java.nio.channels ***сериализуемое, проверяемое***

Это исключение возникает при попытке произвести ввод/вывод через канал, который был закрыт методом close() или определенной операцией ввода/вывода. (Например, у канала SocketChannel могут отдельно закрываться части, отвечающие за чтение и запись.) Каналы могут быть закрыты асинхронно, а потоки, заблокированные в ожидании завершения операции ввода/вывода, в этом случае вызывают один из потомков данного исключения. См. также AsynchronousCloseException и ClosedByInterruptException.



```
public class ClosedChannelException extends java.io.IOException {
// Открытые конструкторы
   public ClosedChannelException();
}
```

Подклассы: AsynchronousCloseException

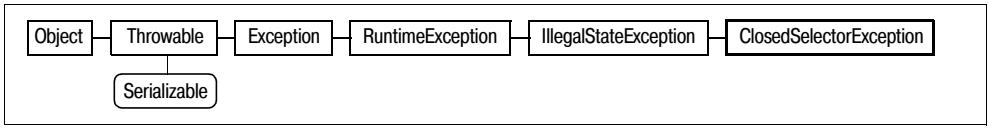
Генерируется методами: SelectableChannel.register(), java.nio.channels.spi.AbstractSelectableChannel.register()

ClosedSelectorException

Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение возникает при попытке повторно вызвать метод close() объекта Selector.



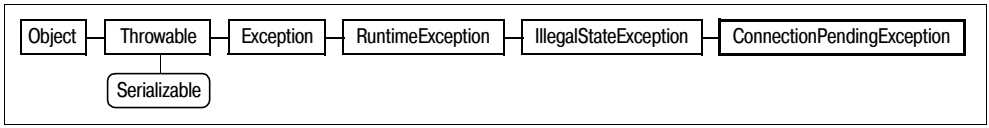

```
public class ClosedSelectorException extends IllegalStateException {
// Открытые конструкторы
   public ClosedSelectorException();
}
```

ConnectionPendingException

Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Данное исключение возникает при вызове метода connect() объекта SocketChannel, если для канала, представленного этим объектом, уже устанавливается соединение. См. также SocketChannel.isConnectionPending().




```
public class ConnectionPendingException extends IllegalStateException {
// Открытые конструкторы
   public ConnectionPendingException();
}
```

DatagramChannel

Java 1.4

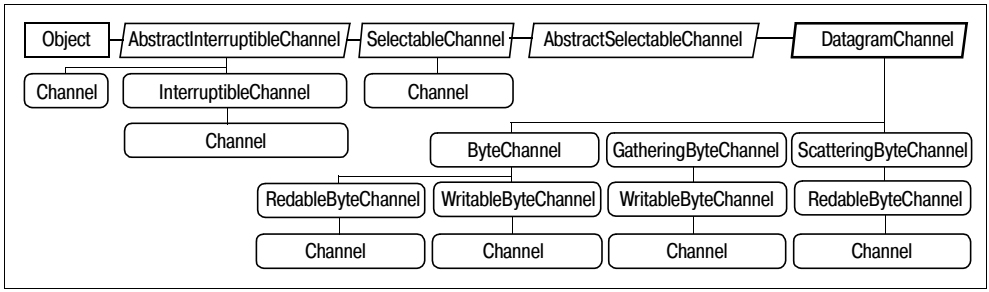
java.nio.channels

Данный класс реализует канал связи, основанный на сетевых дейтаграммах. Экземпляр этого класса можно создать с помощью статического метода open(). Метод socket(), возвращающий объект java.net.DatagramSocket, на котором основан канал, позволяет установить низкоуровневые параметры сокета.

Метод `send()` посылает адресату, порт и хост которого указаны в объекте `java.net.SocketAddress`, оставшиеся байты из указанного буфера `ByteBuffer` в форме дейтаграммы. Метод `receive()` выполняет противоположную операцию: при получении дейтаграммы он сохраняет ее содержимое в указанном буфере, отбрасывая лишние байты, и возвращает объект `SocketAddress`, определяющий отправителя дейтаграммы (или `null`, если канал был в неблокируемом режиме и не ожидалось получение дейтаграммы).

Как правило, методы `send()` и `receive()` при каждом вызове проверяют, имеет ли приложение разрешение на связь с удаленным хостом. Если приложение будет применять класс `DatagramChannel` для обмена дейтаграммами с одним удаленным хостом и портом, используйте метод `connect()` для подключения к адресу, указанному в виде объекта `SocketAddress`. В этом случае метод `connect()` выполняет необходимую проверку прав только один раз и позволяет в дальнейшем осуществлять связь без задержек. Как только соединение установлено, можно задействовать стандартные методы `read()` и `write()`, определенные в интерфейсах `ReadableByteChannel`, `WritableByteChannel`, `GatheringByteChannel` и `ScatteringByteChannel`. Подобно методу `receive()`, метод `read()` без уведомления отбрасывает полученные байты, которые не помещаются в указанный буфер `ByteBuffer`. Методы `read()` и `write()` вызывают исключение `NotYetConnectedException`, если не был вызван метод `connect()`.

`DatagramChannel` наследует класс `SelectableChannel`; метод `validOps()` предоставляет возможность выбора между операциями чтения и записи. Объекты `DatagramChannel` обеспечивают потоковую безопасность. Операции чтения и записи могут производиться одновременно; этот класс гарантирует, что в данный момент времени только один поток может читать из канала, и только один – записывать в него.




```
public abstract class DatagramChannel extends java.nio.channels.spi.AbstractSelectableChannel
      implements ByteChannel, GatheringByteChannel, ScatteringByteChannel {
// Защищенные конструкторы
   protected DatagramChannel(java.nio.channels.spi.SelectorProvider provider);
// Открытые методы класса
   public static DatagramChannel open() throws java.io.IOException;
// Открытые методы экземпляра
   public abstract DatagramChannel connect(java.net.SocketAddress remote) throws java.io.IOException;
   public abstract DatagramChannel disconnect() throws java.io.IOException;
   public abstract boolean isConnected();
   public abstract java.net.SocketAddress receive(java.nio.ByteBuffer dst) throws java.io.IOException;
   public abstract int send(java.nio.ByteBuffer src, java.net.SocketAddress target)
                                                    throws java.io.IOException;
   public abstract java.net.DatagramSocket socket();
// Методы, реализующие GatheringByteChannel
   public final long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;
```

```
    public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length)
                                                    throws java.io.IOException;
// Методы, реализующие ReadableByteChannel
    public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;
// Методы, реализующие ScatteringByteChannel
    public final long read(java.nio.ByteBuffer[] dsts) throws java.io.IOException;
    public abstract long read(java.nio.ByteBuffer[] dsts, int offset, int length)
                                                    throws java.io.IOException;
// Методы, реализующие WritableByteChannel
    public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;
// Открытые методы, замещающие SelectableChannel
    public final int validOps();                                          // константа
}
```

Возвращается методами: java.net.DatagramSocket.getChannel(), DatagramChannel.{connect(),disconnect(), open()}, java.nio.channels.spi.SelectorProvider.openDatagramChannel()

FileChannel Java 1.4

java.nio.channels

Этот класс реализует канал связи, предназначенный для чтения и записи файлов. В нем представлены стандартные методы read() и write() интерфейсов ReadableByteChannel, WritableByteChannel, GatheringByteChannel и ScatteringByteChannel. Кроме того, класс FileChannel предоставляет методы для произвольного доступа к файлам, передачи данных между файлом и другим каналом, блокировки файла, отображения его в памяти, запроса и изменения размера файла, записи изменений из буфера на диск. (Эти важные функции подробно описаны ниже.) Следует заметить, что в отличие от сетевых операций файловые операции не блокируют выполнение на длительное время, поэтому класс FileChannel не наследует SelectableChannel (это единственный класс канала, который не наследует SelectableChannel) и не может применяться с объектами Selector.

Класс FileChannel не имеет открытого конструктора и статических методов-фабрик. Чтобы получить объект FileChannel, нужно сначала создать объект FileInputStream, FileOutputStream или RandomAccessFile (см. описание пакета java.io), а затем вызвать метод getChannel() этого объекта. Если применяется поток FileInputStream, то в новом канале будет разрешено только чтение, а если используется FileOutputStream, то в канале будет разрешена только запись. Если канал FileChannel создается объектом RandomAccessFile, то ему будет разрешено либо только чтение, либо чтение и запись – в зависимости от аргумента mode конструктора RandomAccessFile.

Объект FileChannel имеет текущую позицию, или файловый указатель. Текущую позицию можно установить или запросить с помощью двух методов, имеющих одинаковое имя position(). Текущая позиция канала FileChannel и потока RandomAccessFile, из которого получен этот канал, всегда совпадают; изменение текущей позиции канала отразится на изменении текущей позиции потока, и наоборот. Начальный указатель позиции в FileChannel – это текущая позиция потока RandomAccessFile в момент вызова метода getChannel(). Если FileChannel создан из потока FileOutputStream, открытого в режиме дозаписи, то любые данные, записываемые в канал, всегда записываются в конец файла, а указатель текущей позиции также ставится в конец файла.

После создания объекта FileChannel можно задействовать стандартные методы read() и write(), определенные в различных интерфейсах каналов. Кроме обновления текущей позиции буфера при чтении и записи в него данных, эти методы также обновля-

ют указатель текущей позиции файла, по которому производится чтение или запись этих данных. Методы `read()` возвращают количество прочитанных байтов или –1, если в файле больше не осталось непрочитанных байтов. Методы `write()` увеличивают размер файла, если они производят запись в конец файла. В классе `FileChannel` также определены методы `read()` и `write()`, которые выполняют соответствующую операцию независимо от текущей позиции файла и принимают позицию в качестве аргумента; эти методы читают или записывают байты, начиная с указанной позиции. Несмотря на то что текущая позиция буфера `ByteBuffer` меняется, эти методы не изменяют значение указателя на текущую позицию в файле. Если указанная позиция находится за пределами файла, метод `read()` не производит чтения и возвращает –1, а метод `write()` увеличивает размер файла. При этом значения байтов между предыдущим концом файла и указанной позицией остаются неопределенными.

Обычно данные считываются из канала `FileChannel` и сразу же записываются в другой канал (например, `SocketChannel` в случае веб-сервера). Или же данные, полученные из другого канала, сразу же записываются в `FileChannel` (например, в FTP-клиенте). `FileChannel` предоставляет для этого два высокоэффективных метода `transferTo()` и `transferFrom()`, которые не используют временный буфер `ByteBuffer`. Метод `transferTo()` читает из данного канала `FileChannel` требуемое количество байт, начиная с указанной позиции, и записывает их в указанный канал. Он не меняет текущую позицию канала `FileChannel` и возвращает количество фактически переданных байтов. Метод `transferFrom()` выполняет обратное действие: он читает указанное количество байт, начиная с заданной позиции указанного канала, и записывает их в канал `FileChannel` в указанную позицию. При этом он не меняет текущей позиции данного канала `FileChannel` и возвращает количество фактически переданных байт. Оба метода меняют текущую позицию указанного канала, если он является `FileChannel`.

Метод `size()` возвращает размер данного файла (в байтах). Метод `truncate()` уменьшает размер файла до указанного значения, отсекая все содержимое файла, находящееся за новой границей. Если указанный размер больше или равен текущему, то файл не меняется. Если текущая позиция файла больше, чем новый размер файла, то она устанавливается на новую границу.

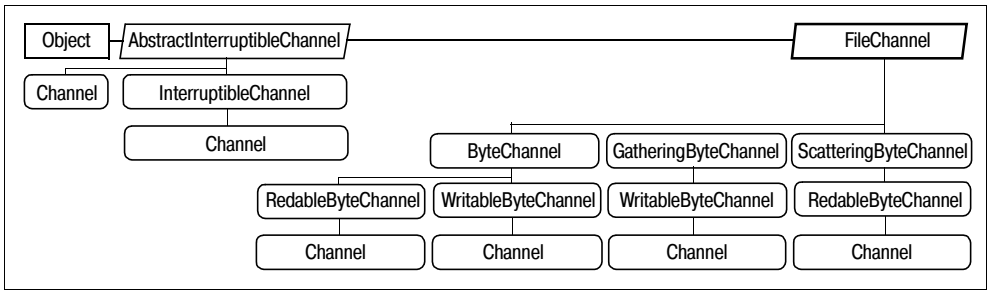
Метод `force()` применяется для принудительной записи всех изменений из буфера на устройство, хранящее файл. Если файл находится на локальном устройстве (то есть не в сети), то метод `force()` гарантирует, что все изменения, проведенные с момента открытия канала или с момента предыдущего вызова `force()`, будут записаны на диск. Аргумент этого метода определяет (в порядке рекомендации), нужно ли вместе с изменениями записывать в файл метаданные (например, время последнего изменения). Если он равен `true`, система запишет в файл обновленное содержимое и метаданные. Если `false`, система может не записать метаданные. Заметьте, что метод `force()` требуется вызывать, если данные выводятся в файл непосредственно через канал `FileChannel`. Изменения в файле, выполненные посредством объекта `MappedByteBuffer`, возвращенного методом `map()` (описан ниже), записываются без вызова метода `force()` объекта `MappedByteBuffer`.

В классе `FileChannel` определено два блокируемых метода `lock()` и два неблокируемых метода `tryLock()` для захвата файла или части файла с целью предотвращения одновременного доступа к этому файлу другой программы. (Эти методы не подходят для запрета одновременного доступа к файлу двух потоков внутри ВМ Java.) Варианты этих методов без аргументов предпринимают попытку произвести захват всего файла для монопольного использования. Одноименные методы с тремя аргументами предпринимают попытку захватить указанную часть файла. Кроме того, они могут производить захват без монополизации. (Такой захват не разрешает другим процессам

производить блокировку для монопольного использования, но не запрещает захват без монополизации. Как правило, захват для совместного использования осуществляется при чтении файла, который не должен одновременно изменяться другими процессами, а монопольный захват производится перед записью в файл, чтобы не допустить одновременное чтение файла другими процессами.) Метод `tryLock()` возвращает объект `FileLock` или `null`, если файл уже был захвачен и возник конфликт блокировок. Метод `lock()` в этом случае блокируется; он никогда не возвращает `null`. (Блокировки подробно представлены в описании `FileLock`.) Механизм захвата файлов объекта `FileChannel` использует средства, реализованные операционной системой. Некоторые ОС осуществляют принудительную блокировку файлов: если какой-то процесс захватил файл, операционная система перекрывает другим процессам доступ к этому файлу. В других ОС параллельно выполняющимся процессам не разрешается осуществлять конфликтующий захват; в этом случае для успешной блокировки файла требуется взаимодействие между такими процессами. Некоторые ОС не поддерживают захват без монополизации: даже если запрашивается такой захват, производится монопольная блокировка.

Метод `map()` возвращает объект `MappedByteBuffer`, представляющий указанный участок файла. Чтение содержимого файла может производиться непосредственно из буфера, а данные, помещенные в буфер, будут записаны в файл (если отображение файла в буфер создано в режиме чтения/записи). Отображение, представленное объектом `MappedByteBuffer`, остается в силе, пока этот объект не освободится сборщиком мусора; буфер продолжает функционировать, даже если породивший его канал `FileChannel` закрыт. Отображение файлов в память может производиться в трех различных режимах, определяющих, могут ли в буфер записываться данные и что при этом должно происходить. Этих режимы представлены в описании `FileChannel.MapMode`.

Метод `map()` опирается на систему управления памятью, предоставляемую операционной системой. Поэтому некоторые детали реализации могут отличаться на разных платформах. В частности, не определено, должны ли изменения в файле, произведенные после вызова `map()`, отражаться на содержимом буфера `MappedByteBuffer`. Как правило, использовать отображения файла эффективнее, чем выполнять операции с файлом без отображения, но это ощутимо, если размер файла достаточно большой.




```
public abstract class FileChannel extends java.nio.channels.spi.AbstractInterruptibleChannel
      implements ByteChannel, GatheringByteChannel, ScatteringByteChannel {
// Защищенные конструкторы
   protected FileChannel();
// Внутренние классы
   public static class MapMode;
// Открытые методы экземпляра
   public abstract void force(boolean metaData) throws java.io.IOException;
```

```
    public final FileLock lock() throws java.io.IOException;
    public abstract FileLock lock(long position, long size, boolean shared)
                                        throws java.io.IOException;
    public abstract java.nio.MappedByteBuffer map(FileChannel.MapMode mode,
              long position, long size)  throws java.io.IOException;
    public abstract long position() throws java.io.IOException;
    public abstract FileChannel position(long newPosition) throws java.io.IOException;
    public abstract int read(java.nio.ByteBuffer dst, long position) throws java.io.IOException;
    public abstract long size() throws java.io.IOException;
    public abstract long transferFrom(ReadableByteChannel src, long position, long count)
                                        throws java.io.IOException;
    public abstract long transferTo(long position, long count, WritableByteChannel target)
                                        throws java.io.IOException;
    public abstract FileChannel truncate(long size) throws java.io.IOException;
    public final FileLock tryLock() throws java.io.IOException;
    public abstract FileLock tryLock(long position, long size, boolean shared)
                                        throws java.io.IOException;
    public abstract int write(java.nio.ByteBuffer src, long position) throws java.io.IOException;
// Методы, реализующие GatheringByteChannel
    public final long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;
    public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length)
                                                    throws java.io.IOException;
// Методы, реализующие ReadableByteChannel
    public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;
// Методы, реализующие ScatteringByteChannel
    public final long read(java.nio.ByteBuffer[] dsts) throws java.io.IOException;
    public abstract long read(java.nio.ByteBuffer[] dsts, int offset, int length)
                                                    throws java.io.IOException;
// Методы, реализующие WritableByteChannel
    public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;
}
```

Передается методам: `FileLock.FileLock()`

Возвращается методами: `java.io.FileInputStream.getChannel()`, `java.io.FileOutputStream.getChannel()`, `java.io.RandomAccessFile.getChannel()`, `FileChannel.{position(), truncate()}`, `FileLock.channel()`

FileChannel.MapMode

Java 1.4

java.nio.channels

Этот класс определяет три константы, равные допустимым значениям аргумента `mode` метода `map()` класса `FileChannel`:

`READ_ONLY`

Отображение файла в память создается только для чтения. Содержимое буфера `MappedByteBuffer`, возвращенного методом `map()`, можно читать, но нельзя изменять.

`READ_WRITE`

Создается двунаправленное отображение файла в память: содержимое возвращенного буфера можно изменять, и все изменения будут в дальнейшем записаны в файл. Объект `FileChannel` должен быть создан потоком `java.io.RandomAccessFile`, открытым с доступом для чтения/записи.

`PRIVATE`

Буфер можно изменять, но изменения не будут записаны в файл. Этот режим также называется «copy-on-write».

```
public static class FileChannel.MapMode {
// Конструктор отсутствует
// Открытые константы
   public static final FileChannel.MapMode PRIVATE;
   public static final FileChannel.MapMode READ_ONLY;
   public static final FileChannel.MapMode READ_WRITE;
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методам: FileChannel.map()

Экземпляры: FileChannel.MapMode.{PRIVATE, READ_ONLY, READ_WRITE}

FileLock

Java 1.4

javf.nio.channels

Объект FileLock возвращается методами lock() и tryLock() класса FileChannel и представляет захват файла или части файла. (Эти методы представлены в описании FileChannel.) Когда блокировка файла больше не нужна, ее нужно снять методом release(). Блокировка также будет снята при закрытии канала или при завершении работы виртуальной машины. Метод isValid() возвращает true, если блокировка еще не снята, и false в противном случае.

Методы channel(), position(), size() и isShared() возвращают базовую информацию о блокировке: канал FileChannel, данные о захваченной части файла, режим захвата (монопольный или для совместного использования). Если захвачен весь файл, метод size() возвращает значение (Long.MAX_VALUE), превышающее фактический размер файла. Если операционная система не поддерживает захват для совместного использования, то метод isShared() может возвратить false, даже если был запрошен захват без монополизации. Метод overlaps() удобен для проверки принадлежности указанного участка файла захваченному участку; в случае принадлежности он возвращает true.

```
public abstract class FileLock {
// Защищенные конструкторы
   protected FileLock(FileChannel channel, long position, long size, boolean shared);
// Открытые методы экземпляра
   public final FileChannel channel();
   public final boolean isShared();
   public abstract boolean isValid();
   public final boolean overlaps(long position, long size);
   public final long position();
   public abstract void release() throws java.io.IOException;
   public final long size();
// Открытые методы, замещающие Object
   public final String toString();
}
```

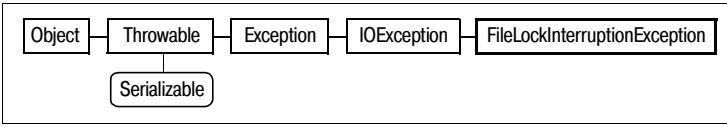
Возвращается методами: FileChannel.{lock(), tryLock()}

FileLockInterruptionException

Java 1.4

java.nio.channels ***сериализуемое, проверяемое***

Это исключение возникает при вызове метода interrupt() заблокированного потока, который ожидает захват файла. См. также FileChannel.lock().



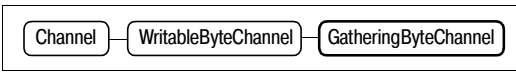

```
public class FileLockInterruptionException extends java.io.IOException {
// Открытые конструкторы
   public FileLockInterruptionException();
}
```

GatheringByteChannel

Java 1.4

java.nio.channels

Этот интерфейс расширяет `WritableByteChannel` и добавляет два дополнительных метода `write()`, которые могут собрать данные из одного или нескольких буферов и вывести их в канал. Этим методам передается массив объектов `ByteBuffer` и (необязательно) смещение и длина, определяющие часть буфера, которая будет использоваться. Метод `write()` предпринимает попытку записать в канал все оставшиеся данные из всех указанных буферов (в том порядке, в котором они стоят в массиве). Этот метод возвращает количество фактически записанных байтов. Исключения и потоковая безопасность этих методов рассмотрены в описании `WritableByteChannel`.




```
public interface GatheringByteChannel extends WritableByteChannel {
// Открытые методы экземпляра
   public abstract long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;
   public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length)
                                                          throws java.io.IOException;
}
```

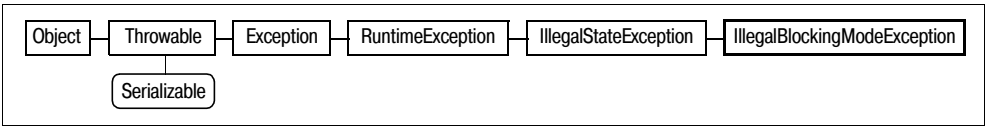
Реализации: `DatagramChannel`, `FileChannel`, `Pipe.SinkChannel`, `SocketChannel`

IllegalBlockingModeException

Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение сообщает о попытке использовать канал с неверным режимом блокировки. Исключение этого типа генерируется методом `SelectableChannel.register()`, если канал находится в неблокируемом режиме.

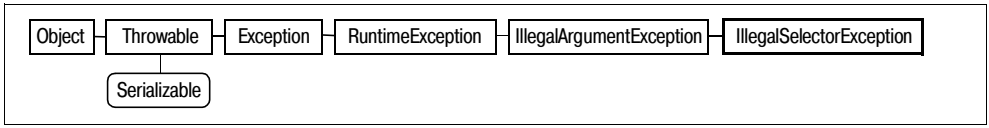



```
public class IllegalBlockingModeException extends IllegalStateException {
// Открытые конструкторы
   public IllegalBlockingModeException();
}
```

IllegalSelectorException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение возникает при попытке объекта `Selector` зарегистрировать канал `SelectableChannel`, если канал и селектор не были созданы одним и тем же объектом `java.nio.channels.spi.SelectorProvider`.




```
public class IllegalSelectorException extends IllegalArgumentException {
// Открытые конструкторы
   public IllegalSelectorException();
}
```

InterruptibleChannel Java 1.4

java.nio.channels

Каналы, реализующие этот интерфейс-маркер, обладают двумя важными свойствами, применяемыми в многопоточных программах: их можно асинхронно закрывать и прерывать. Когда вызывается метод `close()` класса `InterruptibleChannel`, все остальные потоки, блокированные при ожидании завершения операции ввода/вывода через этот канал, прекратят ожидание и вызовут исключение `AsynchronousCloseException`. Кроме того, если поток заблокирован при ожидании завершения операции ввода/вывода через канал `InterruptibleChannel`, то другой поток может вызвать метод `interrupt()` данного потока. Этот метод устанавливает состояние прерывания в блокированном потоке; после этого поток прекращает ожидание и получает исключение `ClosedByInterruptException` (потомок исключения `AsynchronousCloseException`). Канал, заблокировавший поток, закрывается (побочный эффект прерывания потока). Возможность прерывания потоков заслуживает особого внимания, поскольку прерывание было ненадежным в API старого пакета `java.io`.

Все конкретные реализации каналов, входящие в состав данного пакета, реализуют интерфейс `InterruptibleChannel`. Но необходимо обратить внимание на то, что такие методы, как `Channels.newChannel()`, могут возвращать непрерываемые каналы. Чтобы проверить, реализует ли канал неизвестного типа этот интерфейс, применяйте оператор `instanceof`.

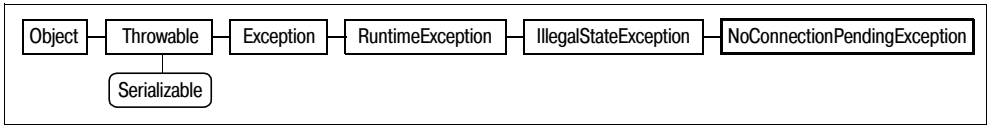

```
public interface InterruptibleChannel extends Channel {
// Открытые методы экземпляра
   public abstract void close() throws java.io.IOException;
}
```

Реализации: `java.nio.channels.spi.AbstractInterruptibleChannel`

NoConnectionPendingException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение сообщает о том, что метод `SocketChannel.finishConnect()` используется без предварительного вызова метода `SocketChannel.connect()`.

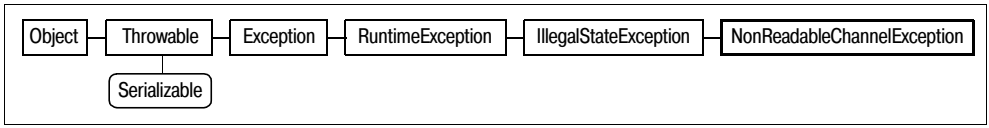



```
public class NoConnectionPendingException extends IllegalStateException {
// Открытые конструкторы
   public NoConnectionPendingException();
}
```

NonReadableChannelException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение возникает, если вызван метод `read()` канала читаемого типа, который не был открыт для чтения (например, канал `FileChannel`, полученный из объекта `FileOutputStream`).

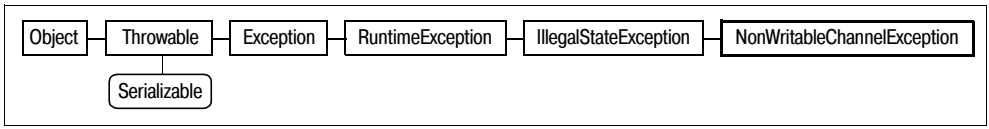



```
public class NonReadableChannelException extends IllegalStateException {
// Открытые конструкторы
   public NonReadableChannelException();
}
```

NonWritableChannelException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение сообщает о вызове метода `write()` канала записываемого типа, который не был открыт для записи (например, канал `FileChannel`, полученный из объекта `FileInputStream`).

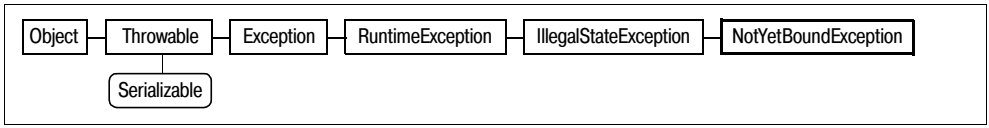



```
public class NonWritableChannelException extends IllegalStateException {
// Открытые конструкторы
   public NonWritableChannelException();
}
```

NotYetBoundException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение возникает, если метод `ServerSocketChannel.accept()` вызван до того, как серверный сокет был связан с локальным портом. Вызовите метод `socket().bind()` объекта `ServerSocketChannel`, чтобы связать соответствующий сокет `java.net.ServerSocket` с локальным адресом.

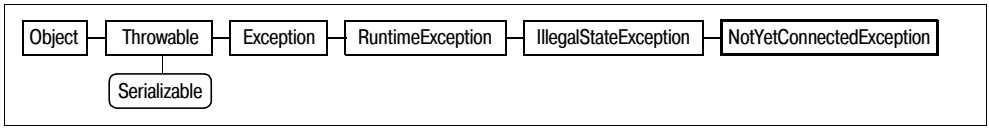



```
public class NotYetBoundException extends IllegalStateException {
// Открытые конструкторы
   public NotYetBoundException();
}
```

NotYetConnectedException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение возникает при вызове методов `read()` или `write()` канала `SocketChannel`, который еще не соединился с удаленным хостом. См. также `SocketChannel.connect()`.

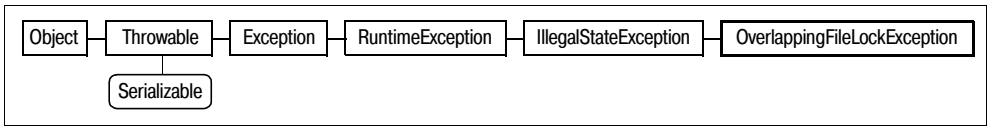



```
public class NotYetConnectedException extends IllegalStateException {
// Открытые конструкторы
   public NotYetConnectedException();
}
```

OverlappingFileLockException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение вызывается методами `lock()` и `tryLock()` канала `FileChannel`, если запрошенный для блокировки участок файла покрывает участок, уже захваченный другим потоком внутри ВМ или участок, уже запрошенный для блокировки. Механизм захвата файлов канала `FileChannel` разработан таким образом, что он запрещает одновременный доступ к захваченному участку двух процессов. Два потока внутри ВМ не должны запрашивать блокировку пересекающихся участков одного файла. Любая попытка вызовет данное исключение.



```
public class OverlappingFileLockException extends IllegalStateException {
// Открытые конструкторы
   public OverlappingFileLockException();
}
```

Pipe Java 1.4

java.nio.channels

Абстракция межпотокового канала (pipe) позволяет осуществить одностороннюю передачу данных из одного потока в другой. Межпотоковый канал имеет читающую и записывающую стороны, которые представлены объектами, реализующими интерфейсы ReadableByteChannel и WritableByteChannel. Создать новый межпотоковый канал можно статическим методом Pipe.open(). Метод sink() возвращает объект Pipe.SinkChannel, который представляет записывающую сторону канала, а source() возвращает объект Pipe.SourceChannel читающей стороны.

Программисту, знакомому с каналами системы Unix, могут быть непонятны названия и возвращаемые значения методов sink() и source(). Канал Unix – это механизм межпроцессного взаимодействия, связанный с двумя процессами: источником (source) данных и приемником, или стоком (sink), этих данных. Зная эту концептуальную модель межпроцессного канала, можно предположить, что источник получает канал (channel) для записи методом source(), а приемник читает из канала, полученного методом sink().

Однако класс Pipe не использует модель канала Unix. Его можно применять для связи между двумя потоками, однако стороны канала не связаны с этими потоками; источников и приемников может быть несколько. Следовательно, в Pipe API источником и приемником байтов служит сам межпотоковый канал: данные читаются из источника и записываются в приемник.

```
public abstract class Pipe {
// Защищенные конструкторы
   protected Pipe();
// Внутренние классы
   public abstract static class SinkChannel
        extends java.nio.channels.spi.AbstractSelectableChannel
        implements GatheringByteChannel, WritableByteChannel;
   public abstract static class SourceChannel
        extends java.nio.channels.spi.AbstractSelectableChannel
        implements ReadableByteChannel, ScatteringByteChannel;
// Открытые методы класса
   public static Pipe open() throws java.io.IOException;
// Открытые методы экземпляра
   public abstract Pipe.SinkChannel sink();
   public abstract Pipe.SourceChannel source();
}
```

Возвращается методами: Pipe.open(), java.nio.channels.spi.SelectorProvider.openPipe()

Pipe.SinkChannel Java 1.4

java.nio.channels

Этот открытый внутренний класс представляет записывающую сторону межпотокового канала. Байты, записанные в объект Pipe.SinkChannel, становятся доступными

в соответствующем объекте Pipe.SourceChannel данного межпотокового канала. Чтобы получить объект Pipe.SinkChannel, нужно сначала создать объект Pipe методом Pipe.open(), а затем вызвать метод sink() этого объекта. См. также класс Pipe.

Класс Pipe.SinkChannel реализует интерфейсы WritableByteChannel и GatheringByteChannel и определяет методы write() этих интерфейсов. Этот класс наследует SelectableChannel, поэтому его можно применять с объектом Selector. Он замещает абстрактный метод validOps() класса SelectableChannel, поэтому данный метод возвращает SelectionKey.OP_WRITE. В этом классе не определены собственные методы.

```
public abstract static class Pipe.SinkChannel extends java.nio.channels.spi.AbstractSelectableChannel
        implements GatheringByteChannel,WritableByteChannel {
// Защищенные конструкторы
   protected SinkChannel(java.nio.channels.spi.SelectorProvider provider);
// Открытые методы, замещающие SelectableChannel
   public final int validOps();  // константа
}
```

Возвращается методами: Pipe.sink()

Pipe.SourceChannel

Java 1.4

java.nio.channels

Этот открытый внутренний класс представляет читающую сторону межпотокового канала. Данные, записанные в соответствующий записывающий конец данного канала (см. Pipe.SinkChannel), доступны для чтения через этот объект. Чтобы получить объект Pipe.SourceChannel, нужно сначала создать объект Pipe методом Pipe.open(), а затем вызвать метод source() этого объекта. См. также класс Pipe.

Класс Pipe.SourceChannel реализует интерфейсы ReadableByteChannel и ScatteringByteChannel и определяет методы read() этих интерфейсов. Этот класс наследует SelectableChannel, поэтому его можно применять с объектом Selector. В этом классе замещен абстрактный метод validOps(), чтобы возвращать SelectionKey.OP_READ. В классе Pipe.SourceChannel нет собственных методов.

```
public abstract static class Pipe.SourceChannel extends java.nio.channels.spi.AbstractSelectableChannel
       implements ReadableByteChannel, ScatteringByteChannel {
// Защищенные конструкторы
   protected SourceChannel(java.nio.channels.spi.SelectorProvider provider);
// Открытые методы, замещающие SelectableChannel
   public final int validOps();  // константа
}
```

Возвращается методами: Pipe.source()

ReadableByteChannel

Java 1.4

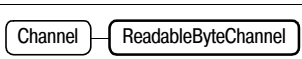
java.nio.channels

Этот интерфейс является потомком Channel и определяет единственный метод read(), читающий байты из канала и записывающий их в указанный буфер ByteBuffer, обновляя при этом текущую позицию буфера. Метод read() пытается прочитать такое количество байт, которое заполнило бы буфер до конца (см. Buffer.remaining()), но в действительности может прочитать меньшее количество. Например, в неблокируемом канале метод read() выполнит возврат немедленно, даже если для чтения не доступен ни один байт. Метод read() возвращает количество фактически прочитанных

байт (оно может равняться нулю в случае неблокируемого канала) или –1, если в канале больше не доступен ни один байт (например, если достигнут конец файла или закрылся сокет).

В объявлении метода `read()` указано, что он может вызывать исключение `IOException`. Точнее, он может вызывать исключение `ClosedChannelException`, если канал закрыт. Если канал закрыт асинхронно или блокированный поток прерван, метод `read()` может завершиться, вызвав исключение `AsynchronousCloseException` или `ClosedByInterruptException`. Этот метод может сгенерировать непроверяемое исключение `NonReadableChannelException`, если он был вызван в закрытом канале или канале, разрешающем чтение.

Реализации данного интерфейса должны обеспечивать потоковую безопасность: только один поток в данный момент времени может осуществлять чтение через канал. Если в данный момент проходит операция чтения, то любой вызов метода `read()` блокирует поток до тех пор, пока операция не завершится. Некоторые каналы могут одновременно осуществлять чтение и запись, но ни один канал не разрешает одновременно осуществлять две операции чтения.




```
public interface ReadableByteChannel extends Channel {
// Открытые методы экземпляра
   public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;
}
```

Реализации: `ByteChannel`, `Pipe.SourceChannel`, `ScatteringByteChannel`

Передается методам: `Channels.{newInputStream(), newReader()}`, `FileChannel.transferFrom()`

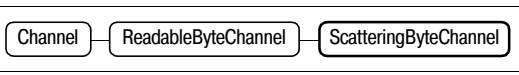
Возвращается методами: `Channels.newChannel()`

ScatteringByteChannel

Java 1.4

java.nio.channels

Этот интерфейс расширяет `ReadableByteChannel` и добавляет в него два метода `read()`, которые читают байты из канала и раскладывают (scatter) их по массиву (или части массива) буферов. Эти методы принимают массив объектов `ByteBuffer` в качестве аргумента и могут также принимать смещение и длину, определяющие часть массива. Метод `read()` пытается прочитать данные, заполняя ими оставшееся место в каждом буфере (в том порядке, в котором буферы стоят в массиве). На самом деле процесс размещения данных в буферах более стройный и линейный, чем следует из его английского названия (scattering – разбрасывание). Этот метод возвращает количество фактически прочитанных байтов, которое может отличаться от суммы оставшихся байтов в буферах. Исключения, вызываемые методами `read()`, и потоковая безопасность представлены в описании `ReadableByteChannel`.




```
public interface ScatteringByteChannel extends ReadableByteChannel {
// Открытые методы экземпляра
   public abstract long read(java.nio.ByteBuffer[] dsts) throws java.io.IOException;
```

```
    public abstract long read(java.nio.ByteBuffer[] dsts, int offset, int length)
                                                    throws java.io.IOException;
}
```

Реализации: DatagramChannel, FileChannel, Pipe.SourceChannel, SocketChannel

SelectableChannel

Java 1.4

java.nio.channels

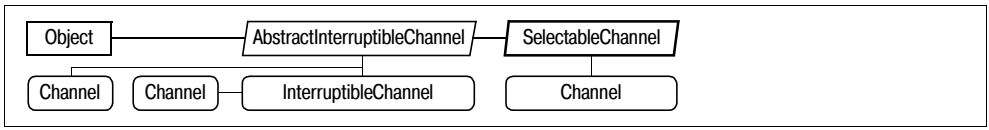
Этот абстрактный класс содержит API для каналов, которые могут использоваться с объектом Selector. Эти API позволяют потоку блокироваться, ожидая освобождения одного из группы каналов. Кроме FileChannel, все классы каналов в пакете java.nio.channels являются потомками SelectableChannel.

Выбираемый канал (типа SelectableChannel) может быть зарегистрирован объектом Selector только в том случае, если он неблокируемый, поэтому класс SelectableChannel определяет метод configureBlocking(). Чтобы перевести поток в неблокируемый режим, нужно в качестве аргумента передать этому методу значение false. Если передать этому методу true, то методы read() и write() будут блокировать поток. Чтобы узнать текущий режим выбираемого канала, нужно вызвать метод isBlocking().

Создать объект SelectableChannel можно с помощью объекта Selector, вызвав метод register() канала (не селектора). Есть два вида этого метода; оба принимают в качестве аргументов Selector и битовую маску, определяющую операции, допустимые с этим каналом. (В разделе «SelectionKey» описаны флаги, которые можно передавать через этот параметр, производя над ними операцию побитового ИЛИ.) Оба метода возвращают объект SelectionKey, который представляет регистрацию канала в селекторе. Один из вариантов метода register() принимает в качестве аргумента произвольный объект, который присоединяется к SelectionKey и позволяет связать с ним произвольные данные. Метод validOps() возвращает битовую маску, определяющую набор операций, который может выполнять произвольный канал. Битовая маска, передаваемая методу register(), может содержать только те флаги, которые установлены в битовой маске, возвращаемой методом validOps().

Обратите внимание, что в классе SelectableChannel не определен свой метод deregister(). Поэтому, чтобы удалить канал из набора каналов, контролируемых объектом Selector, нужно вызвать метод cancel() объекта SelectionKey, возвращенного методом register().

Чтобы узнать, зарегистрирован ли объект SelectableChannel в каком-нибудь селекторе, нужно вызвать метод isRegistered(). (Заметьте, что один канал может быть зарегистрирован несколькими объектами Selector.) Если объект SelectionKey, возвращенный методом register(), не был обработан в программе, можно в дальнейшем получить его методом keyFor(). Дополнительная информация об уплотнении (multiplexing) каналов представлена в описании Selector и SelectionKey.




```
public abstract class SelectableChannel extends
java.nio.channels.spi.AbstractInterruptibleChannel implements Channel {
// Защищенные конструкторы
```

```
    protected SelectableChannel();
// Открытые методы экземпляра
    public abstract Object blockingLock();
    public abstract SelectableChannel configureBlocking(boolean block) throws java.io.IOException;
    public abstract boolean isBlocking();
    public abstract boolean isRegistered();
    public abstract SelectionKey keyFor(Selector sel);
    public abstract java.nio.channels.spi.SelectorProvider provider();
    public final SelectionKey register(Selector sel, int ops) throws ClosedChannelException;
    public abstract SelectionKey register(Selector sel, int ops, Object att)
                                                          throws ClosedChannelException;
    public abstract int validOps();
}
```

Подклассы: java.nio.channels.spi.AbstractSelectableChannel

Возвращается методами: SelectableChannel.configureBlocking(), SelectionKey.channel(), java.nio.channels.spi.AbstractSelectableChannel.configureBlocking()

SelectionKey Java 1.4

java.nio.channels

Класс SelectionKey представляет регистрацию (ключ) канала SelectableChannel в объекте Selector и определяет выбранный канал и операции, к которым готов этот канал. После вызова метода select() селектора метод selectedKeys() возвращает множество (объект Set) объектов SelectionKey, определяя канал или каналы, которые готовы для чтения, записи или другой операции.

Для создания объекта SelectionKey нужно передать объект Selector методу register() канала SelectableChannel. Методы channel() и selector() полученного объекта SelectionKey возвращают объекты SelectableChannel и Selector, связанные с данным ключом.

Чтобы отменить регистрацию канала в селекторе, нужно вызвать метод cancel() объекта SelectionKey. Метод isValid() определяет, является ли SelectionKey действительным ключом – он возвращает true в том случае, если до этого момента не был вызван метод cancel() и не были закрыты канал и селектор.

Основная задача объекта SelectionKey – хранить множество операций канала, в которых «заинтересован» селектор и которые он контролирует, и множество операций, которые селектор определил как готовые для выполнения на данном канале. Оба множества представлены в виде битовых масок (а не в виде объектов java.util.Set), сформированных с помощью операции побитового ИЛИ из констант OP_, которые определены в данном классе. Вот список этих констант:

OP_READ

Во множестве операций, в которых «заинтересован» селектор, этот флаг определяет операцию чтения. Во множестве готовых для выполнения операций он определяет, что канал содержит данные для чтения, достигнут конец потока, канал был удаленно закрыт или произошла ошибка.

OP_WRITE

Во множестве операций, в которых «заинтересован» селектор, этот флаг определяет операцию чтения. Во множестве готовых для выполнения операций этот бит определяет, что канал готов для записи данных, канал был закрыт или произошла ошибка.

OP_CONNECT

Во множестве операций, в которых «заинтересован» селектор, этот флаг определяет операции соединения сокета. Во множестве готовых для выполнения операций он определяет, что канал сокета готов для создания соединения или произошла ошибка.

OP_ACCEPT

Во множестве операций, в которых «заинтересован» селектор, этот флаг определяет, операцию принятия соединения серверным сокетом. Во множестве готовых для выполнения операций он определяет, что серверный сокет готов к принятию соединения или произошла ошибка.

Вариант без аргументов метода interestOps() позволяет запросить множество операций, в которых «заинтересован» селектор. Начальным значением этого множества может быть значение, переданное в качестве аргумента методу register() канала. Его можно менять с помощью метода interestOps() с одним аргументом. (Обратите внимание, что для установки и запроса значения этого множества применяются одноименные методы.) Текущее состояние множества готовых для выполнения операций можно узнать с помощью метода readyOps(). Для проверки отдельных флагов битовой маски можно также использовать методы isReadable(), isWritable(), isConnectable() и isAcceptable(). Явно установить состояние множества готовых для выполнения операций невозможно; оно обновляется при каждом вызове метода select(). Отметим, что необходимо удалять объект SelectionKey из множества Set, возвращаемого методом Selector.selectedKeys(), чтобы битовая маска множества операций, готовых для выполнения, была очищена в начале новой операции выбора. Если SelectionKey не был удален из множества выбранных ключей, то объект Selector полагает, что множество операций, готовых для выполнения, еще не было обработано, и не очищает его флаги.

Чтобы связать произвольный объект с ключом SelectionKey, применяйте метод attach(), а чтобы запросить этот объект – метод attachment(). Возможность связать данные с ключом полезна в том случае, если селектор применяется с несколькими каналами: можно предоставить контекст, необходимый для обработки выбранного ключа SelectionKey.

```
public abstract class SelectionKey {
// Защищенные конструкторы
   protected SelectionKey();
// Открытые константы
   public static final int OP_ACCEPT;                                    // =16
   public static final int OP_CONNECT;                                   // =8
   public static final int OP_READ;                                      // =1
   public static final int OP_WRITE;                                     // =4
// Методы доступа к свойствам (по имени свойства)
   public final boolean isAcceptable();
   public final boolean isConnectable();
   public final boolean isReadable();
   public abstract boolean isValid();
   public final boolean isWritable();
// Открытые методы экземпляра
   public final Object attach(Object ob);
   public final Object attachment();
   public abstract void cancel();
   public abstract SelectableChannel channel();
   public abstract int interestOps();
```

```
    public abstract SelectionKey interestOps(int ops);
    public abstract int readyOps();
    public abstract Selector selector();
}
```

Подклассы: java.nio.channels.spi.AbstractSelectionKey

Возвращается методами: SelectableChannel.{keyFor(), register()}, SelectionKey.interestOps(), java.nio.channels.spi.AbstractSelectableChannel.{keyFor(), register()}, java.nio.channels.spi.AbstractSelector.register()

Selector Java 1.4

java.nio.channels

Selector — это объект, осуществляющий текущий контроль за несколькими неблокируемыми каналами SelectableChannel и выбирающий канал или каналы (после блокировки), которые готовы к осуществлению операций ввода/вывода. Создать новый объект Selector можно с помощью статического метода open(). Затем нужно зарегистрировать каналы, которые будет контролировать объект; для этого нужно передать объект Selector методу register() канала (register() определен в абстрактном классе SelectableChannel). Кроме селектора этому методу нужно передать битовую маску, определяющую, какие операции ввода/вывода на этом канале (чтение, запись, создание или принятие соединения) будет контролировать Selector. Метод register() возвращает объект SelectionKey. (В классе SelectionKey также определены константы, применяемые для формирования битовой маски операций ввода/вывода.) Обратите внимание, что при регистрации объекта SelectableChannel он должен быть в неблокируемом режиме; этот режим можно включить с помощью метода configureBlocking() класса SelectableChannel.

После того как каналы зарегистрированы в объекте Selector, нужно вызвать метод select(), блокирующий выполнение до тех пор, пока один или несколько каналов не будут готовы для ввода/вывода. Один вариант метода select() принимает в качестве аргумента значение времени ожидания в миллисекундах. Если по прошествии этого времени ни один канал не готов для ввода/вывода, то этот метод выполняет возврат. Кроме того, методы select() выполняют возврат, если один из каналов закрыт или в каком-нибудь канале произошла ошибка либо был вызван метод wakeup() объекта Selector или метод interrupt() заблокированного потока. Метод selectNow() аналогичен методу select(): он не блокирует поток, а просто опрашивает каждый канал и определяет, какой канал готов для ввода/вывода. Возвращаемое значение методов selectNow() и select() — это количество каналов, готовых для ввода/вывода. Оно может равняться нулю.

Методы select() и selectNow() возвращают только количество каналов, готовых для ввода/вывода; они не возвращают сами каналы. Чтобы получить каналы, нужно вызвать метод selectedKeys(), возвращающий объект java.util.Set, который содержит объекты SelectionKey. Как правило, после вызова методов select() и selectedKeys() используется объект java.util.Iterator для полученного множества Set. С помощью этого объекта ключи SelectionKey, представляющие каналы, которые готовы для ввода/вывода, обрабатываются в цикле. Для определения готовности канала применяйте метод channel() объекта SelectionKey, а затем вызовите метод readyOps(), isReadable(), isWritable() или какой-нибудь другой подобный метод, чтобы определить операцию ввода/вывода, которая подготовлена к выполнению на данном канале. Ключ SelectionKey присутствует во множестве, возвращаемом методом selectedKeys(), пока он не будет явно удален, поэтому после выполнения операции ввода/вывода через канал

данного ключа SelectionKey ключ нужно удалить из этого множества. Для этого применяется метод remove() объекта Iterator, связанного с данным объектом Set.

Методы selectedKeys() класса Selector дополняются методом keys(), который также возвращает множество Set объектов SelectionKey. Оно представляет собой полный список каналов, контролируемых объектом Selector. Это множество нельзя изменить, кроме как вызвав метод cancel() объекта SelectionKey, отменяющий этот ключ. Отмененные ключи удаляются из множества, возвращаемого методом keys(), при следующем вызове метода select() или selectNow().

Метод wakeup() разблокирует поток, заблокированный при вызове метода select(), и сразу же выполняет возврат. Если в момент вызова метода wakeup() ни один поток не заблокирован методом select(), то при следующем вызове метод select() или selectNow() немедленно выполняет возврат.

Когда объект Selector больше не нужен, его следует закрыть методом close(). Если при этом метод select() заблокировал какой-нибудь поток, этот метод немедленно завершится, как при вызове wakeup(). После вызова метода close() нельзя вызывать никаких других методов данного объекта Selector. Метод isOpen() возвращает true, пока селектор открыт, и false, если он уже закрыт.

Класс Selector обеспечивает потоковую безопасность. Но необходимо заметить, что объект Set, возвращаемый методом selectedKeys(), ее не обеспечивает; в данный момент времени этот объект можно использовать только в одном потоке.

```
public abstract class Selector {
// Защищенные конструкторы
   protected Selector();
// Открытые методы класса
   public static Selector open() throws java.io.IOException;
// Открытые методы экземпляра
   public abstract void close() throws java.io.IOException;
   public abstract boolean isOpen();
   public abstract java.util.Set keys();
   public abstract java.nio.channels.spi.SelectorProvider provider();
   public abstract int select() throws java.io.IOException;
   public abstract int select(long timeout) throws java.io.IOException;
   public abstract java.util.Set selectedKeys();
   public abstract int selectNow() throws java.io.IOException;
   public abstract Selector wakeup();
}
```

Подклассы: java.nio.channels.spi.AbstractSelector

Передается методам: SelectableChannel.{keyFor(), register()}, java.nio.channels.spi.AbstractSelectableChannel.{keyFor(), register()}

Возвращается методами: SelectionKey.selector(), Selector.{open(), wakeup()}

ServerSocketChannel Java 1.4

java.nio.channels

Этот класс из пакета java.nio является подобием класса java.net.ServerSocket. Он представляет канал с возможностью выбора, который можно задействовать в серверах для принятия соединений от клиентов. В отличие от других классов каналов из этого пакета, данный класс не используется для чтения или записи байтов; он не реализует ни один из интерфейсов ByteChannel и предназначен только для принятия и

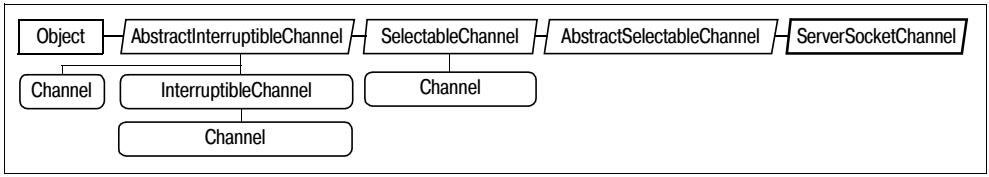
установления соединений с клиентами, а не для общения с этими клиентами. Класс ServerSocketChannel имеет два отличия от класса java.net.ServerSocket. Во-первых, ServerSocketChannel можно перевести в неблокируемое состояние и использовать с объектом Selector. Во-вторых, метод accept() класса ServerSocketChannel возвращает объект SocketChannel, а не Socket, поэтому связь с клиентом, от которого было принято соединение, можно осуществлять средствами пакета java.nio.

Создать объект ServerSocketChannel можно с помощью статического метода open(). Затем нужно вызвать метод socket(), чтобы получить соответствующий объект ServerSocket, после чего можно использовать метод bind() этого объекта для связи серверного сокета с указанным портом на локальном хосте. Кроме того, можно вызвать любой другой метод класса ServerSocket для установки параметров сокета.

Чтобы принять новое соединение через данный канал ServerSocketChannel, нужно просто вызвать метод accept(). Если канал находится в блокируемом режиме, этот метод заблокирует выполнение до тех пор, пока не будет установлено соединение с клиентом, и возвратит объект SocketChannel, связанный с этим клиентом. В неблокируемом режиме (см. описание метода configureBlocking(), который унаследован в данном классе) метод accept() возвращает канал SocketChannel, если клиент в данный момент ожидает соединения; в противном случае метод немедленно возвращает null. Чтобы можно было получить сообщение о том, что клиент ожидает связи, задействуйте унаследованный метод register() для регистрации неблокируемого канала ServerSocketChannel в объекте Selector, указав операцию принятия соединения (константа SelectionKey.OP_ACCEPT). См. также Selector и SelectionKey.

Обратите внимание, что объект SocketChannel, возвращенный методом accept(), всегда находится в неблокируемом режиме, независимо от режима ServerSocketChannel.

Класс ServerSocketChannel обеспечивает потоковую безопасность; в данный момент времени только один поток может вызвать метод accept(). Когда объект ServerSocketChannel больше не нужен, его следует закрыть методом close().




```
public abstract class ServerSocketChannel extends java.nio.channels.spi.AbstractSelectableChannel {
// Защищенные конструкторы
   protected ServerSocketChannel(java.nio.channels.spi.SelectorProvider provider);
// Открытые методы класса
   public static ServerSocketChannel open() throws java.io.IOException;
// Открытые методы экземпляра
   public abstract SocketChannel accept() throws java.io.IOException;
   public abstract java.net.ServerSocket socket();
// Открытые методы, замещающие SelectableChannel
   public final int validOps();
}
```

Возвращается методами: java.net.ServerSocket.getChannel(), ServerSocketChannel.open(), java.nio.channels.spi.SelectorProvider.openServerSocketChannel()

SocketChannel Java 1.4

java.nio.channels

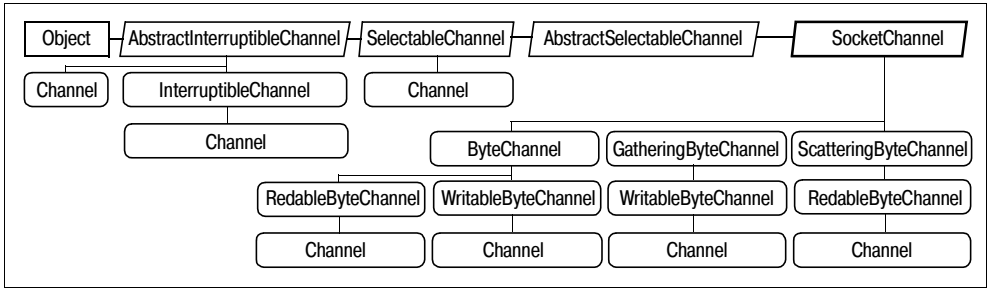
Этот класс представляет канал для связи посредством сокета (java.net.Socket). Он реализует интерфейсы ReadableByteChannel и WritableByteChannel, GatheringByteChannel и ScatteringByteChannel. Этот класс наследует SelectableChannel и может быть использован с объектом Selector.

Создать новый канал SocketChannel можно с помощью одного из статических методов open(). Вариант этого метода без аргументов создает новый канал SocketChannel, но не устанавливает соединение с удаленным хостом. Другой метод open() открывает новый канал и устанавливает соединение с адресом, указанным в объекте java.net.SocketAddress. Если при создании сокета не было установлено соединение, можно явно создать соединение с помощью метода connect(). Создание канала и соединения по отдельности необходимо для установления неблокируемого соединения. Для этого нужно сначала перевести канал в неблокируемый режим с помощью унаследованного метода configureBlocking(). Затем нужно вызвать метод connect(); он завершится сразу же, не ожидая, пока установится соединение. Затем можно зарегистрировать канал в объекте Selector, задав операцию SelectionKey.OP_CONNECT. Когда пришло сообщение о готовности канала к соединению (см. Selector и SelectionKey), нужно вызвать неблокируемый метод finishConnect() для завершения установления соединения. Метод isConnected() возвращает true, если соединение установлено, и false в противном случае. Метод isConnectionPending() возвращает true, если connect() был вызван в блокируемом режиме и до сих пор не завершился или если он был вызван в неблокируемом режиме, но до сих пор не был вызван метод finishConnect().

После того как был открыт канал SocketChannel и установлено соединение, через этот канал можно читать и записывать байтовые значения с помощью нескольких методов read() и write(). Класс SocketChannel обеспечивает потоковую безопасность: операции чтения и записи могут проходить одновременно, но не разрешается выполнение нескольких операций чтения или нескольких операций записи в один момент времени. Если канал SocketChannel переведен в неблокируемый режим, его можно зарегистрировать в селекторе, используя константы OP_READ и OP_WRITE класса SelectionKey, чтобы объект Selector сообщал о готовности канала к чтению или записи.

Метод socket() возвращает объект java.net.Socket, связанный с каналом SocketChannel. Этот объект можно применять для настройки параметров сокета, для связывания его с указанным локальным адресом, для закрытия сокета или выключения его входа или выхода. (См. java.net.Socket.) Хотя все объекты SocketChannel имеют связанные объекты Socket, обратное утверждение не верно: нельзя получить канал SocketChannel из объекта Socket, если этот объект не был создан методом SocketChannel.open().

Когда работа с каналом SocketChannel закончена, его нужно закрыть методом close(). Можно также отдельно закрыть вход и выход канала с помощью методов socket().shutdownInput() и socket().shutdownOutput(). После того как вход был закрыт, все операции чтения (включая те, которые были заблокированы ранее) возвратят –1, сообщая о достижении конца потока. После закрытия выхода все последующие операции записи в канал будут генерировать исключение ClosedChannelException, а текущие заблокированные операции записи завершатся с исключением AsynchronousCloseException.




```
public abstract class SocketChannel extends java.nio.channels.spi.AbstractSelectableChannel
      implements ByteChannel, GatheringByteChannel, ScatteringByteChannel {
// Защищенные конструкторы
   protected SocketChannel(java.nio.channels.spi.SelectorProvider provider);
// Открытые методы класса
   public static SocketChannel open() throws java.io.IOException;
   public static SocketChannel open(java.net.SocketAddress remote) throws java.io.IOException;
// Открытые методы экземпляра
   public abstract boolean connect(java.net.SocketAddress remote) throws java.io.IOException;
   public abstract boolean finishConnect() throws java.io.IOException;
   public abstract boolean isConnected();
   public abstract boolean isConnectionPending();
   public abstract java.net.Socket socket();
// Методы, реализующие GatheringByteChannel
   public final long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;
   public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length)
                                                          throws java.io.IOException;
// Методы, реализующие ReadableByteChannel
   public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;
// Методы, реализующие ScatteringByteChannel
   public final long read(java.nio.ByteBuffer[] dsts) throws java.io.IOException;
   public abstract long read(java.nio.ByteBuffer[] dsts, int offset, int length)
                                                       throws java.io.IOException;
// Методы, реализующие WritableByteChannel
   public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;
// Открытые методы, замещающие SelectableChannel
   public final int validOps();
}
```

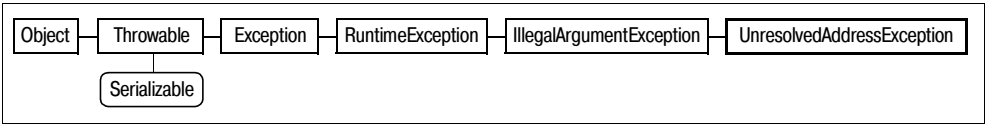
Возвращается методами: `java.net.Socket.getChannel()`, `ServerSocketChannel.accept()`, `SocketChannel.open()`, `java.nio.channels.spi.SelectorProvider.openSocketChannel()`

UnresolvedAddressException

Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Данное исключение сообщает об использовании адреса `java.net.SocketAddress`, который невозможно определить – например, если объект `java.net.InetSocketAddress` содержит неизвестное имя хоста.

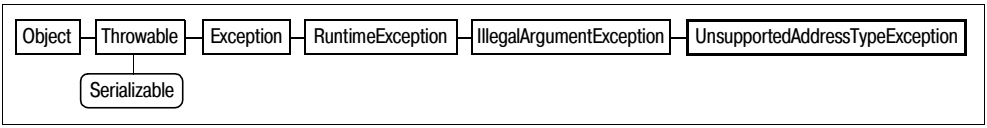



```
public class UnresolvedAddressException extends IllegalArgumentException {
// Открытые конструкторы
   public UnresolvedAddressException();
}
```

UnsupportedAddressTypeException Java 1.4

java.nio.channels ***сериализуемое, непроверяемое***

Это исключение сообщает об использовании потомка `java.net.SocketAddress`, который неизвестен или не поддерживается в текущей реализации. Для обеспечения устойчивости можно применять адрес типа `java.net.InetSocketAddress`, который поддерживается везде.




```
public class UnsupportedAddressTypeException extends IllegalArgumentException {
// Открытые конструкторы
   public UnsupportedAddressTypeException();
}
```

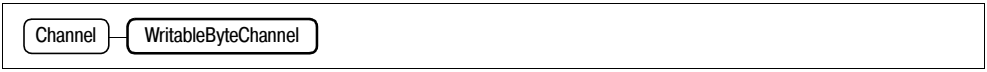
WritableByteChannel Java 1.4

java.nio.channels

Этот интерфейс, наследующий `Channel`, содержит метод `write()`, который записывает байты из указанного буфера в канал, изменяя при этом текущую позицию буфера. Он записывает все оставшиеся байты в буфер (см. `Buffer.remaining()`), но в некоторых случаях это невозможно (например, если канал неблокируемый), поэтому метод `write()` возвращает количество байтов, которые он фактически записал в канал.

В объявлении метода `write()` указано, что он вызывает исключение `IOException`. Говоря более точно, он может вызвать исключение `ClosedChannelException`, если канал закрыт. Если канал был закрыт асинхронно или заблокированный поток был прерван, метод `write()` может завершиться с исключением `AsynchronousCloseException` или `ClosedByInterruptException`. Кроме того, метод `write()` может вызвать непроверяемое исключение `NonWritableChannelException`, если соответствующий канал не разрешает чтение (например, `FileChannel`).

От реализаций интерфейса `WritableByteChannel` требуется обеспечение потоковой безопасности: в данный момент времени только один поток может производить операцию записи в канал. Если в данный момент проходит операция записи, то любой вызов метода `write()` блокирует выполнение до тех пор, пока операция не закончится. В одних реализациях каналов разрешается одновременное чтение и запись, а в других – нет.




```
public interface WritableByteChannel extends Channel {
// Открытые методы экземпляра
   public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;
}
```

Реализации: ByteChannel, GatheringByteChannel, Pipe.SinkChannel

Передается методами: Channels.{newOutputStream(), newWriter()}, FileChannel.transferTo()

Возвращается методами: Channels.newChannel()

Пакет java.nio.channels.spi

Java 1.4

В этом пакете определены четыре класса, используемые при реализации каналов и селекторов пакета java.nio.channels. Здесь также определен класс SelectorProvider, позволяющий заменять реализации каналов и селекторов по умолчанию на пользовательские реализации. При создании приложений этот пакет не требуется, за исключением редких случаев, когда нужно явно установить реализацию SelectorProvider с помощью метода SelectorProvider.provider().

Классы

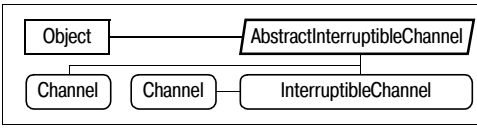
```
public abstract class AbstractInterruptibleChannel implements java.nio.channels.Channel,
            java.nio.channels.InterruptibleChannel;
public abstract class AbstractSelectableChannel extends java.nio.channels.SelectableChannel;
public abstract class AbstractSelectionKey extends java.nio.channels.SelectionKey;
public abstract class AbstractSelector extends java.nio.channels.Selector;
public abstract class SelectorProvider;
```

AbstractInterruptibleChannel

Java 1.4

java.nio.channels.spi

Этот класс удобен при создании новых классов каналов. При разработке обычных приложений этот класс никогда не требуется применять или наследовать.




```
public abstract class AbstractInterruptibleChannel implements java.nio.channels.Channel,
      java.nio.channels.InterruptibleChannel {
// Защищенные конструкторы
   protected AbstractInterruptibleChannel();
// Методы, реализующие Channel
   public final void close() throws java.io.IOException;
   public final boolean isOpen();
// Защищенные методы экземпляра
   protected final void begin();
   protected final void end(boolean completed) throws java.nio.channels.AsynchronousCloseException;
```

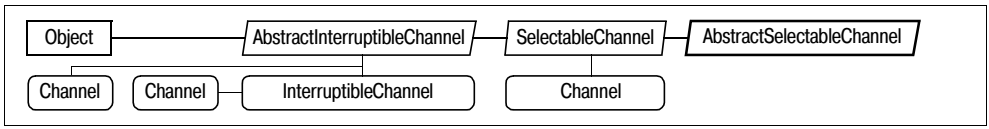
```
   protected abstract void implCloseChannel() throws java.io.IOException;
}
```

Подклассы: java.nio.channels.FileChannel, java.nio.channels.SelectableChannel

AbstractSelectableChannel Java 1.4

java.nio.channels.spi

Этот класс удобен при реализации новых классов выбираемых каналов. Здесь определены общие методы класса SelectableChannel в виде защищенных методов, имена которых начинаются с «impl». При разработке приложений этот класс никогда не требуется применять или наследовать.




```
public abstract class AbstractSelectableChannel extends java.nio.channels.SelectableChannel {
// Защищенные конструкторы
   protected AbstractSelectableChannel(SelectorProvider provider);
// Открытые методы, замещающие SelectableChannel
   public final Object blockingLock();
   public final java.nio.channels.SelectableChannel configureBlocking(boolean block)
        throws java.io.IOException;
   public final boolean isBlocking();
   public final boolean isRegistered();
   public final java.nio.channels.SelectionKey keyFor(java.nio.channels.Selector sel);
   public final SelectorProvider provider();
   public final java.nio.channels.SelectionKey register(java.nio.channels.Selector sel, int ops, Object att)
        throws java.nio.channels.ClosedChannelException;
// Защищенные методы, замещающие AbstractInterruptibleChannel
   protected final void implCloseChannel() throws java.io.IOException;
// Защищенные методы экземпляра
   protected abstract void implCloseSelectableChannel() throws java.io.IOException;
   protected abstract void implConfigureBlocking(boolean block)
        throws java.io.IOException;
}
```

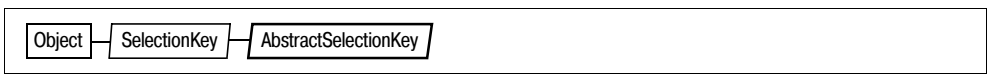
Подклассы: java.nio.channels.DatagramChannel, java.nio.channels.Pipe.SinkChannel, java.nio.channels.Pipe.SourceChannel, java.nio.channels.ServerSocketChannel, java.nio.channels.SocketChannel

Передается методам: AbstractSelector.register()

AbstractSelectionKey Java 1.4

java.nio.channels.spi

Этот класс удобен при реализации новых классов SelectionKey. В приложениях этот класс никогда не требуется использовать или наследовать.



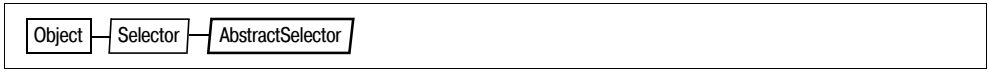
```
public abstract class AbstractSelectionKey extends java.nio.channels.SelectionKey {
// Защищенные конструкторы
   protected AbstractSelectionKey();
// Открытые методы, замещающие SelectionKey
   public final void cancel();
   public final boolean isValid();
}
```

Передается методам: `AbstractSelector.deregister()`

AbstractSelector Java 1.4

java.nio.channels.spi

Этот класс удобен для создания новых классов `Selector`. В приложениях никогда не требуется использовать или наследовать этот класс.




```
public abstract class AbstractSelector extends java.nio.channels.Selector {
// Защищенные конструкторы
   protected AbstractSelector(SelectorProvider provider);
// Открытые методы, замещающие Selector
   public final void close() throws java.io.IOException;
   public final boolean isOpen();
   public final SelectorProvider provider();
// Защищенные методы экземпляра
   protected final void begin();
   protected final java.util.Set cancelledKeys();
   protected final void deregister(AbstractSelectionKey key);
   protected final void end();
   protected abstract void implCloseSelector() throws java.io.IOException;
   protected abstract java.nio.channels.SelectionKey register(AbstractSelectableChannel ch,
                                                              int ops, Object att);
}
```

Возвращается методами: `SelectorProvider.openSelector()`

SelectorProvider Java 1.4

java.nio.channels.spi

Этот класс является главным поставщиком сервисов для каналов и селекторов из пакета `java.nio.channels`. Потомки этого класса реализуют методы-фабрики, которые создают и открывают каналы сокетов, серверных сокетов, дейтаграмм, межпотоковые каналы и объекты `Selector`. В каждой ВМ Java есть один исходный объект `SelectorProvider`; его можно получить с помощью статического метода `SelectorProvider.provider()`.

Можно указать пользовательскую реализацию класса `SelectorProvider`, присвоив имя этого класса системному свойству `java.nio.channels.spi.SelectorProvider`. Кроме того, имя этого класса можно задать в файле `META-INF/services/java.nio.channels.spi.SelectorProvider`, входящем в файл JAR данного приложения. Метод `provider()` сначала ищет системное свойство, а затем производит поиск в архиве JAR. Если он ничего не находит, то создает экземпляр исходного класса `SelectorProvider`.

В приложениях необязательно применять только исходный класс SelectorProvider. Можно создать объекты других классов SelectorProvider и вызывать их методы open() для создания каналов.

```
public abstract class SelectorProvider {
// Защищенные конструкторы
   protected SelectorProvider();
// Открытые методы класса
   public static SelectorProvider provider();
// Открытые методы экземпляра
   public abstract java.nio.channels.DatagramChannel openDatagramChannel()
        throws java.io.IOException;
   public abstract java.nio.channels.Pipe openPipe() throws java.io.IOException;
   public abstract AbstractSelector openSelector() throws java.io.IOException;
   public abstract java.nio.channels.ServerSocketChannel openServerSocketChannel()
        throws java.io.IOException;
   public abstract java.nio.channels.SocketChannel openSocketChannel() throws java.io.IOException;
}
```

Передается методам: java.nio.channels.DatagramChannel.DatagramChannel(), java.nio.channels.Pipe.SinkChannel.SinkChannel(), java.nio.channels.Pipe.SourceChannel.SourceChannel(), java.nio.channels.ServerSocketChannel.ServerSocketChannel(), java.nio.channels.SocketChannel.SocketChannel(), AbstractSelectableChannel.AbstractSelectableChannel(), AbstractSelector.AbstractSelector()

Возвращается методами: java.nio.channels.SelectableChannel.provider(), java.nio.channels.Selector.provider(), AbstractSelectableChannel.provider(), AbstractSelector.provider(), SelectorProvider.provider()

Пакет java.nio.charset

Java 1.4

В этом пакете содержатся классы, представляющие наборы символов, или кодировки. Здесь также определены методы для преобразования символов в байтовые значения и для обратного преобразования. Ключевым классом этого пакета является Charset. С помощью статического метода forName() можно получить объект Charset для указанной кодировки по ее имени. В классе Charset для удобства определены методы encode() и decode(), но для полного контроля над процессом кодирования и декодирования нужно получить объекты CharsetEncoder и CharsetDecoder из объекта Charset.

В платформе Java 1.1 появилась возможность кодирования и декодирования символов. Для этой цели определено несколько классов и методов. Некоторые из них применяют основной региональный набор символов системы, другие – набор символов, указанный в аргументе метода или конструктора. (Например, конструкторы String(), java.io.InputStreamReader() и java.io.OutputStreamWriter().) В Java 1.4 в пакете java.nio.charset определены открытые методы для кодирования и декодирования символов, с которыми можно явно работать в приложениях. Тем не менее в большинстве обычных приложений они не требуются. Можно ограничиться системным набором символов по умолчанию и указывать имена других наборов при необходимости. Даже в приложениях, применяющих пакет java.nio.channels, можно не использовать непосредственное кодирование и декодирование, а просто передавать имя набора символов методам newReader() и newWriter() класса java.nio.channels.Channels.

Классы

```
public abstract class Charset implements Comparable;
public abstract class CharsetDecoder;
public abstract class CharsetEncoder;
public class CoderResult;
public class CodingErrorAction;
```

Исключения

```
public class CharacterCodingException extends java.io.IOException;
   └ public class MalformedInputException extends CharacterCodingException;
   └ public class UnmappableCharacterException extends CharacterCodingException;
public class IllegalCharsetNameException extends IllegalArgumentException;
public class UnsupportedCharsetException extends IllegalArgumentException;
```

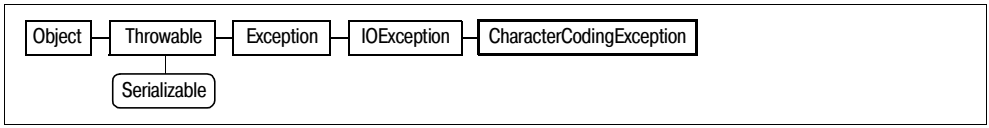
Ошибки

```
public class CoderMalfunctionError extends Error;
```

CharacterCodingException Java 1.4

java.nio.charset ***сериализуемое, проверяемое***

Назначение этого класса – сообщать об ошибках при кодировании и декодировании символов. Этот общий класс является родительским классом для более конкретных типов исключений. Обратите внимание на то, что версии методов `CharsetEncoder.encode()` и `CharsetDecoder.decode()` с одним аргументом могут вызывать исключение этого типа, а одноименные методы с тремя аргументами сообщают об ошибке при кодировании в возвращаемом значении типа `CoderResult`. Методы `encode()` и `decode()` класса `Charset` не вызывают этого исключения, поскольку они заменяют входные данные с неправильным форматом, а также байты или символы, для которых не найдены соответствующие значения. См. `CodingErrorAction`.




```
public class CharacterCodingException extends java.io.IOException {
// Открытые конструкторы
   public CharacterCodingException();
}
```

Подклассы: `MalformedInputException`, `UnmappableCharacterException`

Генерируется методами: `CharsetDecoder.decode()`, `CharsetEncoder.encode()`, `CoderResult.throwException()`

Charset Java 1.4

java.nio.charset ***сравнимый***

Класс `Charset` представляет набор символов, или кодировку. Каждый объект `Charset` имеет каноническое имя, возвращаемое методом `name()`, и набор псевдонимов (aliases), возвращаемый методом `aliases()`. Производить поиск соответствующего объекта `Charset` можно по имени или по псевдониму с помощью статического метода

Charset.forName(), который может вызывать исключение UnsupportedCharsetException, если указанный набор символов не установлен в системе. Проверить, поддерживается ли набор символов, заданный именем или псевдонимом, можно с помощью статического метода isSupported(). Получить все установленные наборы символов можно с помощью метода availableCharsets(), который возвращает отсортированную таблицу соответствия объектов Charset и канонических имен. Обратите внимание, что имена наборов символов не чувствительны к регистру. При передаче имени кодировки в методы isSupported() и forName() вы можете записывать его как заглавными, так и строчными буквами. Также необходимо заметить, что в Java есть несколько классов и методов, в которых набор символов указывается по имени, а не через объект Charset. (Например, java.io.InputStreamReader, java.io.OutputStreamWriter, String.getBytes() и java.nio.channels.Channels.newWriter().) При работе с такими классами и методами объект Charset не нужен.

Во всех реализациях Java должны поддерживаться по крайней мере шесть наборов символов:

Каноническое имя	Описание
US-ASCII	ASCII, 7 бит.
ISO-8859-1	Расширенный 8-битный набор ASCII, включающий символы, которые применяются в большинстве западноевропейских языков. Также известен под именем ISO-LATIN-1.
UTF-8	8-битовая кодировка символов Unicode, совместимая с US-ASCII.
UTF-16BE	16-битная кодировка символов Unicode, использующая обратный порядок байтов.
UTF-16LE	16-битная кодировка символов Unicode, использующая прямой порядок байтов.
UTF-16	16-битная кодировка символов Unicode, порядок байтов которой определяется символом «метка порядка байтов» («byte order mark»). Порядок считается обратным, если эта метка отсутствует. Символы кодируются в обратном порядке, а затем ставится соответствующая метка порядка байтов.

После того как получен объект Charset с помощью метода forName() или availableCharsets(), можно использовать метод encode() для преобразования текста из String или CharBuffer в буфер ByteBuffer или метод decode() для преобразования байтов из буфера ByteBuffer в символы объекта CharBuffer. Эти удобные методы предназначены для создания нового объекта CharsetEncoder или CharsetDecoder. Они указывают на то, что входные данные с неправильным форматом, а также байты или символы, для которых не найдено соответствие, должны заменяться определенными строками или байтами. Затем они вызывают метод encode() или decode() кодировщика или декодировщика соответственно. Чтобы получить полный контроль над процессом кодирования, можно создать свой объект CharsetEncoder или CharsetDecoder с помощью метода newEncoder() или newDecoder(). См. также CharsetDecoder.

Вместо прямого использования объекта Charset, CharsetEncoder или CharsetDecoder можно передать кодировщик или декодировщик статическим методам класса java.nio.channels.Channels. В этом случае будет получен объект java.io.Reader или java.io.Writer, который можно применять для чтения символов из канала или записи в него символов.

Обратите внимание, что не все объекты Charset поддерживают кодирование. Некоторые объекты могут определить исходную кодировку символов при декодировании, но не могут кодировать символы. Чтобы определить, может ли данный объект Charset кодировать символы, применяйте метод canEncode().

В классе Charset также имеются другие методы (собственные, реализованные или замещенные). Метод displayName() возвращает локализованное имя набора символов или его каноническое имя, если нет его локализации. Метод toString() возвращает текстовое представление набора символов, зависящее от реализации. Метод equals() сравнивает два набора символов по их каноническим именам. Класс Charset реализует интерфейс Comparable; его метод compareTo() упорядочивает наборы символов по их каноническим именам. contains() возвращает true, если указанный набор символов «содержится» в данном наборе, то есть каждый символ, представленный в указанном наборе, может быть представлен в данном наборе. Обратите внимание, что эти представления не обязательно должны быть одинаковыми. Метод isRegistered() возвращает true, если данный набор зарегистрирован в реестре наборов символов IANA (см. *http://www.iana.org/assignments/character-sets*).


```
public abstract class Charset implements Comparable {
// Защищенные конструкторы
   protected Charset(String canonicalName, String[] aliases);
// Открытые методы класса
   public static java.util.SortedMap availableCharsets();
   public static Charset forName(String charsetName);
   public static boolean isSupported(String charsetName);
// Открытые методы экземпляра
   public final java.util.Set aliases();
   public boolean canEncode();                                    // константа
   public abstract boolean contains(Charset cs);
   public final java.nio.CharBuffer decode(java.nio.ByteBuffer bb);
   public String displayName();
   public String displayName(java.util.Locale locale);
   public final java.nio.ByteBuffer encode(java.nio.CharBuffer cb);
   public final java.nio.ByteBuffer encode(String str);
   public final boolean isRegistered();
   public final String name();
   public abstract CharsetDecoder newDecoder();
   public abstract CharsetEncoder newEncoder();
// Методы, реализующие Comparable
   public final int compareTo(Object ob);
// Открытые методы, замещающие Object
   public final boolean equals(Object ob);
   public final int hashCode();
   public final String toString();
}
```

Передается методам: java.io.InputStreamReader.InputStreamReader(), java.io.OutputStreamWriter.OutputStreamWriter(), Charset.contains(), CharsetDecoder.CharsetDecoder(), CharsetEncoder.CharsetEncoder()

Возвращается методами: Charset.forName(), CharsetDecoder.{charset(), detectedCharset()}, CharsetEncoder.charset(), java.nio.charset.spi.CharsetProvider.charsetForName()

CharsetDecoder Java 1.4

java.nio.charset

Класс CharsetDecoder является «механизмом декодирования», конвертирующим последовательность байтов в последовательность символов на основе кодировки набора символов. Объект CharsetDecoder можно получить из объекта Charset, представляющего набор символов, которые нужно декодировать. Если последовательность байтов, которую нужно декодировать, полностью находится в буфере ByteBuffer, его можно передать методу decode() с одним аргументом. Этот метод позволяет удобно декодировать байтовые значения и записывать полученные символы в новый буфер CharBuffer, при необходимости восстанавливая или сбрасывая буфер декодировщика. Этот метод вызывает исключение, если есть проблемы с декодируемыми данными.

Как правило, применяется метод decode() с тремя аргументами, а процесс декодирования проходит в три этапа:

1. Если объект CharsetDecoder применяется не в первый раз, нужно вызвать метод reset().
2. После этого вызывается метод decode() необходимое количество раз. Третий аргумент должен быть true, если этот метод вызывается в последний раз. Первый аргумент метода decode() – объект ByteBuffer, а второй – объект CharBuffer, в который будут записаны полученные символы. Возвращаемое значение данного метода – это объект CoderResult, описывающий состояние текущей операции декодирования (возможные возвращаемые значения описаны ниже). Как правило, метод decode() выполняет возврат после того, как он декодировал все байты из входного буфера. В этом случае нужно заполнить входной буфер байтами, которые далее должны быть декодированы, и прочитать символы из выходного буфера, вызвав его метод compact(), чтобы освободить место для дальнейших символов. Если возникает непредвиденная проблема в CharsetDecoder, то метод decode() вызывает исключение CoderMalfunctionError.
3. Затем нужно передать выходной буфер CharBuffer методу flush(), чтобы вывести все оставшиеся символы.

Метод decode() возвращает объект CoderResult, определяющий состояние операции декодирования. Если возвращаемое значение равно CoderResult.UNDERFLOW, это означает, что метод decode() выполнил возврат, поскольку все байты из входного буфера были прочитаны и требуются следующие входные данные. Если возвращаемое значение равно CoderResult.OVERFLOW, то метод decode() выполнил возврат из-за того, что буфер CharBuffer заполнился до конца, и в него больше не могут быть помещены символы. В остальных случаях возвращается объект CoderResult, метод isError() которого возвращает true. Существуют два основных типа ошибок, возникающих при декодировании. Метод isMalformed() возвращает true, если входные байтовые значения не допустимы для данного набора символов. Это байты из участка длиной length(), начинающегося с текущей позиции входного буфера. Или же, если метод isUnmappable() возвращает true, входной буфер содержит символы, не представленные в Unicode. Соответствующий участок входного буфера начинается с текущей позиции и имеет длину length().

По умолчанию объект CharsetDecoder всегда сообщает о недопустимых входных данных и о неверных символах, возвращая объект CoderResult. Это поведение можно изменить, передав методам onMalformedInput() и onUnmappableCharacter() объект CodingErrorAction. (Действие для этих типов ошибок, определенное в данный момент, можно узнать с помощью методов malformedInputAction() и unmappableCharacterAction().) Класс

CodingErrorAction содержит три константы, представляющие возможные действия. По умолчанию это REPORT. Действие IGNORE указывает объекту CharsetDecoder на необходимость игнорировать (пропускать) входные данные с неверным форматом и неправильные символы. Действие REPLACE указывает на то, что нужно заменять определенной строкой входные данные с неправильным форматом и символы, для которых не найдено соответствующее представление. Эту строку можно задать методом replaceWith() и получить методом replacement().

Методы averageCharsPerByte() и maxCharsPerByte() возвращают соответственно среднее и максимальное количество символов, произведенных декодировщиком, в расчете на 1 байт. Эти значения помогают выбрать размер буфера CharBuffer, который будет применяться для декодирования.

Класс CharsetDecoder не обеспечивает потоковую безопасность; в данный момент времени только один поток может использовать экземпляр этого класса.

CharsetDecoder – абстрактный класс. При определении новых наборов символов возникает необходимость создать потомки этого класса и определить абстрактный метод decodeLoop(), вызываемый методом decode().

```
public abstract class CharsetDecoder {
// Защищенные конструкторы
   protected CharsetDecoder(Charset cs, float averageCharsPerByte, float maxCharsPerByte);
// Открытые методы экземпляра
   public final float averageCharsPerByte();
   public final Charset charset();
   public final java.nio.CharBuffer decode(java.nio.ByteBuffer in) throws CharacterCodingException;
   public final CoderResult decode(java.nio.ByteBuffer in, java.nio.CharBuffer out,
                                                            boolean endOfInput);
   public Charset detectedCharset();
   public final CoderResult flush(java.nio.CharBuffer out);
   public boolean isAutoDetecting();  // константа
   public boolean isCharsetDetected();
   public CodingErrorAction malformedInputAction();
   public final float maxCharsPerByte();
   public final CharsetDecoder onMalformedInput(CodingErrorAction newAction);
   public final CharsetDecoder onUnmappableCharacter(CodingErrorAction newAction);
   public final String replacement();
   public final CharsetDecoder replaceWith(String newReplacement);
   public final CharsetDecoder reset();
   public CodingErrorAction unmappableCharacterAction();
// Защищенные методы экземпляра
   protected abstract CoderResult decodeLoop(java.nio.ByteBuffer in, java.nio.CharBuffer out);
   protected CoderResult implFlush(java.nio.CharBuffer out);
   protected void implOnMalformedInput(CodingErrorAction newAction);                  // пустой
   protected void implOnUnmappableCharacter(CodingErrorAction newAction);             // пустой
   protected void implReplaceWith(String newReplacement);                             // пустой
   protected void implReset();                                                        // пустой
}
```

Передается методам: java.io.InputStreamReader.InputStreamReader(), java.nio.channels.Channels.newReader()

Возвращается методами: Charset.newDecoder(), CharsetDecoder.{onMalformedInput(), onUnmappableCharacter(), replaceWith(), reset()}

CharsetEncoder Java 1.4

java.nio.charset

Класс CharsetEncoder представляет «механизм кодирования», который преобразует последовательность символов в последовательность байтов на основе кодировки набора символов. Получить объект CharsetEncoder можно с помощью метода newEncoder() объекта Charset, представляющего требуемую кодировку.

Класс CharsetEncoder работает так же, как CharsetDecoder, только в обратном направлении. Метод encode() кодирует символы, которые считывает из буфера CharBuffer, и сохраняет полученные байты в буфере ByteBuffer. Подробно этот процесс рассмотрен в описании класса CharsetDecoder.

```
public abstract class CharsetEncoder {
// Защищенные конструкторы
   protected CharsetEncoder(Charset cs, float averageBytesPerChar,float maxBytesPerChar);
   protected CharsetEncoder(Charset cs, float averageBytesPerChar,float maxBytesPerChar,
                                        byte[] replacement);
// Открытые методы экземпляра
   public final float averageBytesPerChar();
   public boolean canEncode(CharSequence cs);
   public boolean canEncode(char c);
   public final Charset charset();
   public final java.nio.ByteBuffer encode(java.nio.CharBuffer in)
        throws CharacterCodingException;
   public final CoderResult encode(java.nio.CharBuffer in, java.nio.ByteBuffer out,
        boolean endOfInput);
   public final CoderResult flush(java.nio.ByteBuffer out);
   public boolean isLegalReplacement(byte[] repl);
   public CodingErrorAction malformedInputAction();
   public final float maxBytesPerChar();
   public final CharsetEncoder onMalformedInput(CodingErrorAction newAction);
   public final CharsetEncoder onUnmappableCharacter(CodingErrorAction newAction);
   public final byte[] replacement();
   public final CharsetEncoder replaceWith(byte[] newReplacement);
   public final CharsetEncoder reset();
   public CodingErrorAction unmappableCharacterAction();
// Защищенные методы экземпляра
   protected abstract CoderResult encodeLoop(java.nio.CharBuffer in,
        java.nio.ByteBuffer out);
   protected CoderResult implFlush(java.nio.ByteBuffer out);
   protected void implOnMalformedInput(CodingErrorAction newAction);                  // пустой
   protected void implOnUnmappableCharacter(CodingErrorAction newAction);             // пустой
   protected void implReplaceWith(byte[] newReplacement);                             // пустой
   protected void implReset();                                                        // пустой
}
```

Передается методам: java.io.OutputStreamWriter.OutputStreamWriter(), java.nio.channels.Channels.newWriter()

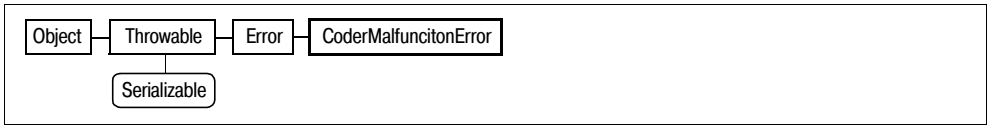
Возвращается методами: Charset.newEncoder(), CharsetEncoder.{onMalformedInput(), onUnmappableCharacter(), replaceWith(), reset()}

CoderMalfunctionError

Java 1.4

java.nio.charset ***сериализуемый, ошибка***

Этот класс сообщает о сбое при работе объекта `CharsetEncoder` или `CharsetDecoder`, возникающем, как правило, из-за неизвестной и непоправимой ошибки. Ошибки такого типа возникают в методах `encode()` и `decode()`, когда защищенные методы `encodeLoop()` и `decodeLoop()`, на которых основаны первые два метода, вызывают непредвиденное исключение.




```
public class CoderMalfunctionError extends Error {
// Открытые конструкторы
   public CoderMalfunctionError(Exception cause);
}
```

CoderResult

Java 1.4

java.nio.charset

Объект `CoderResult` определяет результат вызова метода `CharsetDecoder.decode()` или `CharsetEncoder.encode()`. Возможны четыре причины, по которым эти методы выполняют возврат:

- Если все байты были декодированы или все символы закодированы, а входной буфер пуст, то возвращаемое значение будет константным объектом `CoderResult.UNDERFLOW`. Это значение сообщает о том, что кодирование завершено, поскольку больше нет входных данных. Метод `isUnderflow()` полученного объекта будет возвращать `true`, а `isError()` — `false`. Это нормальное возвращаемое значение.
- Если еще есть входные данные, но в выходном буфере больше нет места для кодированных данных, то возвращаемое значение будет константным объектом `CoderResult.OVERFLOW`. Метод `isOverflow()` этого объекта возвратит `true`, а `isError()` — `false`. Это нормальное возвращаемое значение.
- Если входные данные находятся в неверном формате и содержат символы или байтовые значения, недопустимые в данном наборе символов, а объект `CharsetEncoder` или `CharsetDecoder` не определяет, что такие входные данные должны игнорироваться или заменяться, то возвращаемым значением будет объект `CoderResult`, методы `isError()` и `isMalformed()` которого возвращают `true`. Текущая позиция входного буфера устанавливается на первом символе или байте с неправильным форматом, а метод `length()` возвращаемого объекта определяет количество таких символов или байтов.
- Возвращаемым значением будет объект `CoderResult`, методы `isError()` и `isUnmappable()` которого возвращают `true`, если входные данные имеют правильный формат, но содержат символы или байты, которые не могут быть закодированы или декодированы с использованием указанного набора символов, а объект `CharsetEncoder` или `CharsetDecoder` не определяет, что такие данные должны быть игнорированы или заменены. Текущая позиция входного буфера устанавливается на первом символе или байте, для которого не найдено представление в данном набо-

ре, а метод length() возвращенного объекта определяет количество таких символов или байтов.

```
public class CoderResult {
// Конструктор отсутствует
// Открытые константы
   public static final CoderResult OVERFLOW;
   public static final CoderResult UNDERFLOW;
// Открытые методы класса
   public static CoderResult malformedForLength(int length);
   public static CoderResult unmappableForLength(int length);
// Методы доступа к свойствам (по имени свойства)
   public boolean isError();
   public boolean isMalformed();
   public boolean isOverflow();
   public boolean isUnderflow();
   public boolean isUnmappable();
// Открытые методы экземпляра
   public int length();
   public void throwException() throws CharacterCodingException;
// Открытые методы, замещающие Object
   public String toString();
}
```

Возвращается методами: CharsetDecoder.{decode(), decodeLoop(), flush(), implFlush()}, CharsetEncoder.{encode(), encodeLoop(), flush(), implFlush()}, CoderResult.{malformedForLength(), unmappableForLength()}

Экземпляры: CoderResult.{OVERFLOW, UNDERFLOW}

CodingErrorAction Java 1.4

java.nio.charset

Этот класс является перечислением, обеспечивающим типовую безопасность. В этом перечислении содержатся три константы, являющиеся допустимыми аргументами для методов onMalformedInput() и onUnmappableCharacter() классов CharsetDecoder и CharsetEncoder. Эти константы определяют, как должны обрабатываться ошибки, связанные с неверным входным форматом и неверными символами. Здесь определены следующие константы:

CodingErrorAction.REPORT

Указывает на то, что об ошибке нужно сообщить. В этом случае методы decode() и encode() с тремя аргументами возвращают объект CoderResult, а одноименные методы с одним аргументом вызывают исключение MalformedInputException или UnmappableCharacterException. Это действие по умолчанию для ошибок обоих типов.

CodingErrorAction.IGNORE

Определяет, что входные данные с неверным форматом или неверные символы должны пропускаться и об этих ошибках не нужно сообщать.

CodingErrorAction.REPLACE

Определяет, что входные данные с неверным форматом и неверные символы должны пропускаться и вместо них на выходе должна вставляться замещающая строка или последовательность байтов.

Дополнительная информация представлена в описании CharsetDecoder.

```
public class CodingErrorAction {
// Конструктор отсутствует
// Открытые константы
   public static final CodingErrorAction IGNORE;
   public static final CodingErrorAction REPLACE;
   public static final CodingErrorAction REPORT;
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методами: CharsetDecoder.{implOnMalformedInput(), implOnUnmappableCharacter(), onMalformedInput(), onUnmappableCharacter()}, CharsetEncoder.{implOnMalformedInput(), implOnUnmappableCharacter(), onMalformedInput(), onUnmappableCharacter()}

Возвращается методами: CharsetDecoder.{malformedInputAction(), unmappableCharacterAction()}, CharsetEncoder.{malformedInputAction(), unmappableCharacterAction()}

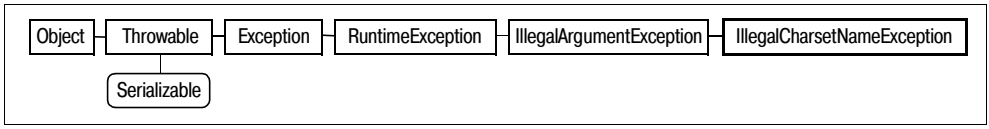
Экземпляры: CodingErrorAction.{IGNORE, REPLACE, REPORT}

IllegalCharsetNameException

Java 1.4

java.nio.charset ***сериализуемое, непроверяемое***

Это исключение сообщает о недопустимом имени набора символов (например, имени, передаваемом в качестве аргумента методу Charset.forName() или Charset.isSupported()). Имя набора символов может содержать только символы A–Z (в верхнем и нижнем регистре), цифры 0–9, дефисы, символы подчеркивания, двоеточия и точки. Это имя должно начинаться с буквы или цифры, но не со знака пунктуации.



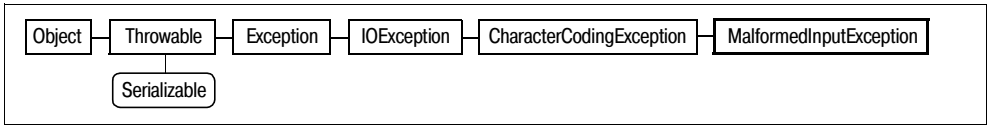

```
public class IllegalCharsetNameException extends IllegalArgumentException {
// Открытые конструкторы
   public IllegalCharsetNameException(String charsetName);
// Открытые методы экземпляра
   public String getCharsetName();
}
```

MalformedInputException

Java 1.4

java.nio.charset ***сериализуемое, проверяемое***

Это исключение сообщает о том, что входные данные метода CharsetDecoder.decode() или CharsetEncoder.encode() имеют неверный формат.



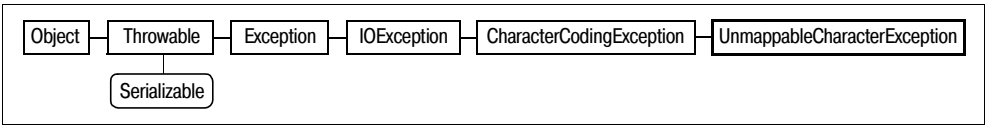

```
public class MalformedInputException extends CharacterCodingException {
// Открытые конструкторы
```

```
    public MalformedInputException(int inputLength);
// Открытые методы экземпляра
    public int getInputLength();
// Открытые методы, замещающие Throwable
    public String getMessage();
}
```

UnmappableCharacterException Java 1.4

java.nio.charset ***сериализуемое, проверяемое***

Это исключение сообщает о том, что входные данные метода `CharsetDecoder.decode()` или `CharsetEncoder.encode()` содержат символ или последовательность байтов, которым не найдено соответствие в указанном наборе символов.

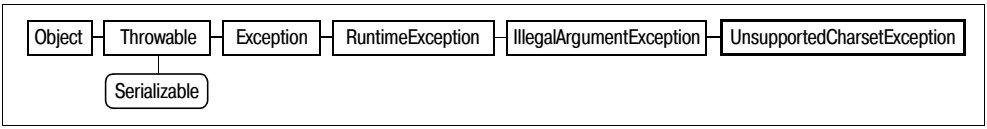



```
public class UnmappableCharacterException extends CharacterCodingException {
// Открытые конструкторы
    public UnmappableCharacterException(int inputLength);
// Открытые методы экземпляра
    public int getInputLength();
// Открытые методы, замещающие Throwable
    public String getMessage();
}
```

UnsupportedCharsetException Java 1.4

java.nio.charset ***сериализуемое, непроверяемое***

Это исключение сообщает о том, что запрошенный набор символов не поддерживается текущей платформой. Оно вызывается методом `Charset.forName()`, когда для указанного имени набора символов невозможно получить объект `Charset`. См. также `Charset.isSupported()`.




```
public class UnsupportedCharsetException extends IllegalArgumentException {
// Открытые конструкторы
    public UnsupportedCharsetException(String charsetName);
// Открытые методы экземпляра
    public String getCharsetName();
}
```

Пакет java.nio.charset.spi Java 1.4

Этот пакет содержит класс-«поставщик» («provider»), который предназначен для использования системными программистами, разрабатывающими новые реализации класса Charset. Он применяется в тех случаях, когда нужно, чтобы эти реализации были доступны в системе. Прикладным программистам никогда не требуются этот пакет и содержащийся в нем класс.

Классы

```
public abstract class CharsetProvider;
```

CharsetProvider Java 1.4

java.nio.charset.spi

Системные программисты, разрабатывающие новые наборы символов (реализации класса Charset), должны реализовать данный класс, чтобы эти наборы были доступны в системе. Метод charsetForName() должен возвращать объект Charset, соответствующий указанному имени. Метод charsets() должен возвращать объект java.util.Iterator, с помощью которого можно будет обрабатывать в цикле множество объектов Charset, определенных в поставщике.

Класс CharsetProvider и связанные с ним реализации Charset должны быть упакованы в файл JAR. Чтобы этот файл был доступен в системе, он должен находиться в каталоге расширений *jre/lib/ext/* (или в другом месте нахождения расширений). Данный архив JAR должен содержать файл *META-INF/services/java.nio.charset.spi.CharsetProvider*, в котором находится имя класса, реализующего CharsetProvider.

```
public abstract class CharsetProvider {
// Защищенные конструкторы
   protected CharsetProvider();
// Открытые методы экземпляра
   public abstract java.nio.charset.Charset charsetForName(String charsetName);
   public abstract java.util.Iterator charsets();
}
```

Глава 15

java.security и подпакеты

В этой главе описан пакет `java.security` и его подпакеты. Вот их описание:

`java.security`

Это крупный пакет, содержащий большую часть инфраструктуры безопасности Java, в том числе группу классов, позволяющих контролировать доступ с помощью ограничений (policies) и прав (permissions), и группу классов, предоставляющих такие возможности аутентификации, как цифровые подписи.

`java.security.acl`

Этот пакет реализует инфраструктуру списков контроля доступа (Access Control Lists). Он не используется механизмом безопасности Java и был замещен классами пакета `java.security`.

`java.security.cert`

Этот пакет определяет классы и интерфейсы для работы с сертификатами открытых ключей, списками отзыва сертификатов (CRL), а также (в Java 1.4 и последующих версиях) для работы с цепочками сертификатов (или путями сертификатов). Данный пакет определяет общие классы, которые должны работать с любым типом сертификатов, и подклассы для сертификатов X.509 и CRL.

`java.security.interfaces`

Этот пакет определяет интерфейсы для ключей шифрования, использующих специальные алгоритмы. Приложения, предназначенные для поддержки этих алгоритмов, должны реализовать эти интерфейсы.

`java.security.spec`

Этот пакет содержит классы, которые определяют прозрачное, переносимое представление объектов, использующих специальные алгоритмы, например ключи шифрования. Экземпляры этих классов могут применяться с любым механизмом обеспечения безопасности.

Пакет java.security

Java 1.1

Пакет `java.security` содержит классы и интерфейсы, которые реализуют архитектуру безопасности Java. Эти классы можно разделить на две основных категории. Первая категория – это классы, реализующие контроль доступа и предотвращающие выпол-

нение потенциально опасных операций ненадежным кодом. Вторая категория – это классы аутентификации, реализующие профили сообщений и цифровые подписи и умеющие аутентифицировать классы Java и другие объекты.

Центральный класс контроля доступа – это класс AccessController; он применяет установленный объект Policy для определения наличия у данного класса прав доступа к указанным системным ресурсам. Классы Permissions и ProtectionDomain также являются важными частями архитектуры управления доступом Java.

Ключевые классы для аутентификации – это классы MessageDigest и Signature; они вычисляют и проверяют криптографические профили сообщений и цифровые подписи. Эти классы применяют технологии шифрования с открытым ключом и опираются на интерфейсы PublicKey и PrivateKey. Кроме того, они опираются на инфраструктуру других классов. Например, класс SecureRandom применяется для получения криптостойких псевдослучайных чисел, класс KeyPairGenerator – для генерации пар открытых и закрытых ключей, а класс KeyStore – для управления коллекциями ключей и сертификатов. (Этот пакет определяет интерфейс Certificate, но он больше не используется; см. пакет java.security.cert, в котором есть более предпочтительный класс Certificate.)

Класс CodeSource объединяет классы аутентификации с классами контроля доступа. Он представляет исходный код Java-класса как URL и устанавливает объект java.security.cert.Certificate, содержащий цифровые подписи этого кода. Классы AccessController и Policy обращаются к объекту CodeSource класса при принятии решения о предоставлении доступа.

Все возможности этого пакета, касающиеся шифрования и аутентификации, зависят от поставщика (provider). Другими словами, они реализованы в виде модулей поставщика безопасности и могут легко встраиваться в среду Java 1.2 и последующих версий. Поэтому в дополнение к определению API безопасности этот пакет определяет интерфейс поставщика услуг (service provider interface, SPI). Различные классы, имена которых заканчиваются на «Spi», являются частью SPI. Реализации поставщиков должны быть подклассами этих Spi-классов, но у приложения никогда не возникает необходимости их использовать. Каждый поставщик средств безопасности представлен классом Provider, а класс Security позволяет динамически устанавливать новых поставщиков.

Пакет java.security содержит несколько полезных служебных классов. Например, классы DigestInputStream и DigestOutputStream облегчают вычисление профилей сообщений. Класс GuardedObject предоставляет изменяемый контроль доступа для отдельных объектов. Класс SignedObject защищает целостность произвольного объекта Java, связывая с ним цифровую подпись, что позволяет легко выявить вмешательство в объект. Несмотря на то что пакет java.security содержит криптографические классы для аутентификации, он не содержит классов для шифрования и дешифрования. Такая функциональность является частью расширения Java Cryptography Extension, или JCE, определенного в пакете javax.crypto и его подпакетах. JCE является частью базовой платформы Java 1.4 и последующих версий. Оно доступно в виде стандартного расширения для Java 1.2 и Java 1.3.

Интерфейсы

```
public interface Certificate;
public interface DomainCombiner;
public interface Guard;
public interface Key extends Serializable;
public interface Principal;
```

```
public interface PrivateKey extends Key;
public interface PrivilegedAction;
public interface PrivilegedExceptionAction;
public interface PublicKey extends Key;
```

Коллекции

```
public abstract class Provider extends java.util.Properties;
```

Другие классы

```
public final class AccessControlContext;
public final class AccessController;
public class AlgorithmParameterGenerator;
public abstract class AlgorithmParameterGeneratorSpi;
public class AlgorithmParameters;
public abstract class AlgorithmParametersSpi;
public class CodeSource implements Serializable;
public class DigestInputStream extends java.io.FilterInputStream;
public class DigestOutputStream extends java.io.FilterOutputStream;
public class GuardedObject implements Serializable;
public abstract class Identity implements Principal, Serializable;
   └ public abstract class IdentityScope extends Identity;
   └ public abstract class Signer extends Identity;
public class KeyFactory;
public abstract class KeyFactorySpi;
public final class KeyPair implements Serializable;
public abstract class KeyPairGeneratorSpi;
   └ public abstract class KeyPairGenerator extends KeyPairGeneratorSpi;
public class KeyStore;
public abstract class KeyStoreSpi;
public abstract class MessageDigestSpi;
   └ public abstract class MessageDigest extends MessageDigestSpi;
public abstract class Permission implements Guard, Serializable;
   └ public final class AllPermission extends Permission;
   └ public abstract class BasicPermission extends Permission implements Serializable;
      └ public final class SecurityPermission extends BasicPermission;
   └ public final class UnresolvedPermission extends Permission implements Serializable;
public abstract class PermissionCollection implements Serializable;
   └ public final class Permissions extends PermissionCollection implements Serializable;
public abstract class Policy;
public class ProtectionDomain;
public class SecureClassLoader extends ClassLoader;
public class SecureRandom extends java.util.Random;
public abstract class SecureRandomSpi implements Serializable;
public final class Security;
public abstract class SignatureSpi;
   └ public abstract class Signature extends SignatureSpi;
public final class SignedObject implements Serializable;
```

Исключения

```
public class AccessControlException extends SecurityException;
public class GeneralSecurityException extends Exception;
   └ public class DigestException extends GeneralSecurityException;
   └ public class InvalidAlgorithmParameterException extends GeneralSecurityException;
   └ public class KeyException extends GeneralSecurityException;
      └ public class InvalidKeyException extends KeyException;
      └ public class KeyManagementException extends KeyException;
```

```
    └ public class KeyStoreException extends GeneralSecurityException;
    └ public class NoSuchAlgorithmException extends GeneralSecurityException;
    └ public class NoSuchProviderException extends GeneralSecurityException;
    └ public class SignatureException extends GeneralSecurityException;
    └ public class UnrecoverableKeyException extends GeneralSecurityException;
public class InvalidParameterException extends IllegalArgumentException;
public class PrivilegedActionException extends Exception;
public class ProviderException extends RuntimeException;
```

AccessControlContext

Java 1.2

java.security

Этот класс инкапсулирует состояние стека вызовов. Метод `checkPermission()` может принимать решения о предоставлении доступа на основе сохраненного состояния стека вызовов. Проверка возможности доступа обычно выполняется методом `AccessController.checkPermission()`, который проверяет, есть ли у стека вызовов необходимые права. Впрочем, иногда необходимо принять решение о предоставлении доступа на основе предыдущего состояния стека. Вызовите `AccessController.getContext()` для создания объекта `AccessControlContext` на основе конкретного состояния стека. В Java 1.3 этот класс имеет конструктор, который принимает произвольный контекст в форме массива объектов `ProtectionDomain` и связывает объект `DomainCombiner` с существующим `AccessControlContext`. Этот класс применяется только в коде на системном уровне; в приложениях он используется редко.

```
public final class AccessControlContext {
// Открытые методы
   public AccessControlContext(ProtectionDomain[] context);
1.3 public AccessControlContext(AccessControlContext acc, DomainCombiner combiner);
// Открытые методы экземпляра
   public void checkPermission(java.security.Permission perm) throws AccessControlException;
1.3 public DomainCombiner getDomainCombiner();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

Передается методам: `AccessControlContext.AccessControlContext()`, `AccessController.doPrivileged()`, `javax.security.auth.Subject.{doAsPrivileged(), getSubject()}`

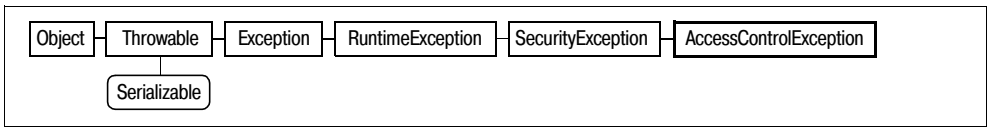
Возвращается методами: `AccessController.getContext()`

AccessControlException

Java 1.2

java.security ***сериализуемое, непроверяемое***

Вызывается `AccessController` для того, чтобы просигнализировать об отказе в доступе. Метод `getPermission()` возвращает объект `Permission` (если такой существует), который был вовлечен в отклоненный запрос доступа.



```
public class AccessControlException extends SecurityException {
// Открытые методы
   public AccessControlException(String s);
   public AccessControlException(String s, java.security.Permission p);
// Открытые методы экземпляра
   public java.security.Permission getPermission();
}
```

Генерируется методами: AccessControlContext.checkPermission(), AccessController.checkPermission()

AccessController

Java 1.2

java.security

В Java 1.2 статические методы этого класса реализуют механизм контроля доступа по умолчанию. Метод checkPermission() проходит по стеку вызовов текущего потока и проверяет, есть ли у классов в стеке вызовов необходимые права. Если права есть, метод checkPermission() возвращает управление и выполнение продолжается. Если прав нет, метод checkPermission() вызывает исключение AccessControlException. В Java 1.2 метод checkPermission() класса java.lang.SecurityManager по умолчанию вызывает метод AccessController.checkPermission(). Код системного уровня, которому нужно выполнить проверку, должен вызвать метод класса SecurityManager вместо прямого вызова AccessController. Пока вы не пишете системный код, который должен контролировать доступ к системным ресурсам, вам никогда не придется использовать этот класс или метод SecurityManager.checkPermission().

Различные методы doPrivileged() выполняют блоки привилегированного кода, инкапсулированного в объектах PrivilegedAction и PrivilegedExceptionAction. Когда метод checkPermission() проходит по стеку вызовов потока, он останавливается по достижении привилегированного блока, который был выполнен с помощью doPrivileged(). Это означает, что привилегированный код может выполняться с полным набором прав, даже если он был вызван ненадежным кодом или кодом с более низким уровнем прав. Более подробная информация представлена в описании PrivilegedAction.

Метод getContext() возвращает объект AccessControlContext, который представляет собой текущий контекст безопасности вызывающего объекта. Такой контекст можно сохранить и при будущих вызовах (возможно, они будут выполняться из другого потока). Версия doPrivileged() с двумя аргументами применяется для принудительной проверки прав доступа с использованием того же AccessControlContext.

```
public final class AccessController {
// Конструктор отсутствует
// Открытые методы класса
   public static void checkPermission(java.security.Permission perm)
       throws AccessControlException;
   public static Object doPrivileged(PrivilegedExceptionAction action)
       throws PrivilegedActionException;                          // зависит от платформы
   public static Object doPrivileged(PrivilegedAction action);    // зависит от платформы
   public static Object doPrivileged(PrivilegedExceptionAction action,
       AccessControlContext context) throws PrivilegedActionException;  // зависит от платформы
   public static Object doPrivileged(PrivilegedAction action, AccessControlContext context);
                                                                  // зависит от платформы
   public static AccessControlContext getContext();
}
```

AlgorithmParameterGenerator Java 1.2

java.security

Этот класс определяет общий API для генерации параметров алгоритма шифрования. Обычно это Signature или javax.crypto.Cipher. Создайте объект AlgorithmParameterGenerator, вызвав один из статических методов-фабрик getInstance(). При этом указывается имя алгоритма и (необязательно) имя или объект Provider необходимого поставщика. Поставщик «SUN» по умолчанию поддерживает алгоритм «DSA». Поставщик «SunJCE», поставляемый с JCE, поддерживает «DiffieHellman». Как только вы получили генератор, проинициализируйте его, вызвав метод init() и указав размер алгоритмо-независимого параметра (в битах) или алгоритмо-зависимый объект AlgorithmParameterSpec. В качестве источника случайных чисел при вызове init() можно также указать SecureRandom. После создания и инициализации генератора AlgorithmParameterGenerator вызовите generateParameters() для генерации объекта AlgorithmParameters.

```
public class AlgorithmParameterGenerator {
// Защищенные конструкторы
   protected AlgorithmParameterGenerator(AlgorithmParameterGeneratorSpi paramGenSpi,
                                         Provider provider, String algorithm);
// Открытые методы класса
   public static AlgorithmParameterGenerator getInstance(String algorithm)
                                         throws NoSuchAlgorithmException;
1.4 public static AlgorithmParameterGenerator getInstance(String algorithm, Provider provider)
                                         throws NoSuchAlgorithmException;
   public static AlgorithmParameterGenerator getInstance(String algorithm, String provider)
                                 throws NoSuchAlgorithmException, NoSuchProviderException;
// Открытые методы экземпляра
   public final AlgorithmParameters generateParameters();
   public final String getAlgorithm();
   public final Provider getProvider();
   public final void init(java.security.spec.AlgorithmParameterSpec genParamSpec)
                       throws InvalidAlgorithmParameterException;
   public final void init(int size);
   public final void init(java.security.spec.AlgorithmParameterSpec genParamSpec,
                       eRandom random) throws InvalidAlgorithmParameterException;
   public final void init(int size, SecureRandom random);
}
```

Возвращается методами: AlgorithmParameterGenerator.getInstance()

AlgorithmParameterGeneratorSpi Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для генерации параметров алгоритма. Для каждого поддерживаемого алгоритма поставщик средств безопасности должен реализовать конкретный подкласс этого класса. Приложения никогда не используют и никогда не создают подклассов данного класса.

```
public abstract class AlgorithmParameterGeneratorSpi {
// Открытые методы
   public AlgorithmParameterGeneratorSpi();
// Защищенные методы экземляра
   protected abstract AlgorithmParameters engineGenerateParameters();
```

```
   protected abstract void engineInit(java.security.spec.AlgorithmParameterSpec genParamSpec,
                                      SecureRandom random) throws InvalidAlgorithmParameterException;
   protected abstract void engineInit(int size, SecureRandom random);
}
```

Передается методам: AlgorithmParameterGenerator.AlgorithmParameterGenerator()

AlgorithmParameters Java 1.2

java.security

Этот класс является общим, непрозрачным представлением параметров, используемых некоторыми алгоритмами шифрования. Вы можете создать экземпляр этого класса с помощью одного из статических методов `getInstance()` **фабрики, указав желаемый алгоритм и (необязательно) желаемого поставщика. Поставщик по умолчанию «SUN» поддерживает алгоритм «DSA». Поставщик «SunJCE», поставляемый с JCE, поддерживает «DES», «DESede», «PBE», «Blowfish» и «DiffieHellman». Как только вы получили объект** `AlgorithmParameters`**, проинициализируйте его, передав методу** `init()` **необходимый объект** `java.security.spec.AlgorithmParameterSpec`**, который зависит от алгоритма, или закодированные значения параметра в байтовом массиве. Кроме того, объект** `AlgorithmParameters` **можно создать с помощью** `AlgorithmParameterGenerator`**. Метод** `getEncoded()` **возвращает проинициализированные параметры алгоритма в байтовом массиве, используя кодирование по умолчанию, зависящее от алгоритма, либо заданный формат кодирования.**

```
public class AlgorithmParameters {
// Защищенные конструкторы
   protected AlgorithmParameters(AlgorithmParametersSpi paramSpi, Provider provider,
                                                                  String algorithm);
// Открытые методы класса
   public static AlgorithmParameters getInstance(String algorithm)
                                throws NoSuchAlgorithmException;
   public static AlgorithmParameters getInstance(String algorithm, String provider)
                                throws NoSuchAlgorithmException, NoSuchProviderException;
1.4 public static AlgorithmParameters getInstance(String algorithm, Provider provider)
                                throws NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final String getAlgorithm();
   public final byte[] getEncoded() throws java.io.IOException;
   public final byte[] getEncoded(String format) throws java.io.IOException;
   public final java.security.spec.AlgorithmParameterSpec getParameterSpec(Class paramSpec)
                                throws java.security.spec.InvalidParameterSpecException;
   public final Provider getProvider();
   public final void init(java.security.spec.AlgorithmParameterSpec paramSpec)
                                throws java.security.spec.InvalidParameterSpecException;
   public final void init(byte[] params) throws java.io.IOException;
   public final void init(byte[] params, String format) throws java.io.IOException;
// Открытые методы замещающие Object
   public final String toString();
}
```

Передается методам: javax.crypto.Cipher.init(), javax.crypto.CipherSpi.engineInit(), javax.crypto.EncryptedPrivateKeyInfo.EncryptedPrivateKeyInfo(), javax.crypto.ExemptionMechanism.init(), javax.crypto.ExemptionMechanismSpi.engineInit()

Возвращается методами: AlgorithmParameterGenerator.generateParameters(), AlgorithmParameterGeneratorSpi.engineGenerateParameters(), AlgorithmParameters.getInstance(),

Signature.getParameters(), SignatureSpi.engineGetParameters(),
javax.crypto.Cipher.getParameters(), javax.crypto.CipherSpi.engineGetParameters(),
javax.crypto.EncryptedPrivateKeyInfo.getAlgParameters()

AlgorithmParametersSpi

Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для AlgorithmParameters. Поставщик средств безопасности должен реализовать конкретный подкласс этого класса для каждого поддерживаемого алгоритма шифрования. Приложения никогда не используют и никогда не создают подклассов этого класса.

```
public abstract class AlgorithmParametersSpi {
// Открытые методы
   public AlgorithmParametersSpi();
// Защищенные методы экземпляра
   protected abstract byte[] engineGetEncoded() throws java.io.IOException;
   protected abstract byte[] engineGetEncoded(String format) throws java.io.IOException;
   protected abstract java.security.spec.AlgorithmParameterSpec engineGetParameterSpec(Class
                        paramSpec) throws java.security.spec.InvalidParameterSpecException;
   protected abstract void engineInit(java.security.spec.AlgorithmParameterSpec paramSpec)
                                    throws java.security.spec.InvalidParameterSpecException;
   protected abstract void engineInit(byte[] params) throws java.io.IOException;
   protected abstract void engineInit(byte[] params, String format) throws java.io.IOException;
   protected abstract String engineToString();
}
```

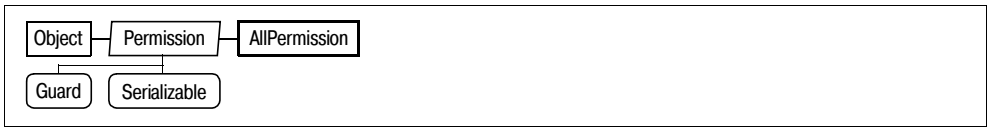
Передается методам: AlgorithmParameters.AlgorithmParameters()

AllPermission

Java 1.2

java.security ***сериализуемый***

Этот класс является подклассом класса Permission, у которого метод implies() всегда возвращает true. Это означает, что код, которому были даны права AllPermission, получает все возможные привилегии. Этот класс существует для удобного предоставления всех прав абсолютно надежному коду. Применять его нужно осторожно. Обычно приложения не работают с объектом Permission напрямую.




```
public final class AllPermission extends java.security.Permission {
// Открытые методы
   public AllPermission();
   public AllPermission(String name, String actions);
// Открытые методы, замещающие Permission
   public boolean equals(Object obj);
   public String getActions();                              // по умолчанию:"<all actions>"
   public int hashCode();                                   // константа
   public boolean implies(java.security.Permission p);      // константа
   public PermissionCollection newPermissionCollection();
}
```

BasicPermission

Java 1.2

java.security ***сериализуемый***

Этот класс `Permission` является абстрактным родительским классом для многих простых типов прав. `BasicPermission` обычно применяют в качестве родительского класса для реализации поименованных прав. У таких прав есть имя, или целевой объект, но они не поддерживают действий. Метод `implies()` класса `BasicPermission` определяет возможность работы с простыми групповыми символами (wildcard). Целевой объект «*» определяет права для любого объекта. Целевой объект «x.*» определяет права для любого объекта, имя которого начинается с «x.». Обычно приложения не работают с объектом `Permission` напрямую.


```
public abstract class BasicPermission extends java.security.Permission implements Serializable {
// Открытые методы
   public BasicPermission(String name);
   public BasicPermission(String name, String actions);
// Открытые методы, замещающие Permission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
   public boolean implies(java.security.Permission p);
   public PermissionCollection newPermissionCollection();
}
```

Подклассы: `java.awt.AWTPermission`, `java.io.SerializablePermission`, `RuntimePermission`, `java.lang.reflect.ReflectPermission`, `java.net.NetPermission`, `SecurityPermission`, `java.sql.SQLPermission`, `java.util.PropertyPermission`, `java.util.logging.LoggingPermission`, `javax.net.ssl.SSLPermission`, `javax.security.auth.AuthPermission`, `javax.security.auth.kerberos.DelegationPermission`, `javax.sound.sampled.AudioPermission`

Certificate

Java 1.1; устарел в Java 1.2

java.security

Этот интерфейс применялся в Java 1.1 для представления сертификата подлинности. Начиная с Java 1.2 он больше не используется. Ему на смену пришел пакет `java.security.cert`. См. также `java.security.cert.Certificate`.

```
public interface Certificate {
// Открытые методы экземпляра
   public abstract void decode(java.io.InputStream stream) throws KeyException, java.io.IOException;
   public abstract void encode(java.io.OutputStream stream) throws KeyException,java.io.IOException;
    public abstract String getFormat();
    public abstract java.security.Principal getGuarantor();
    public abstract java.security.Principal getPrincipal();
    public abstract PublicKey getPublicKey();
    public abstract String toString(boolean detailed);
}
```

Передается методам: Identity.{addCertificate(), removeCertificate()}

Возвращается методами: Identity.certificates()

CodeSource

Java 1.2

java.security ***сериализуемый***

Этот класс представляет собой исходный код Java-класса, определенный адресом URL, с которого класс был загружен, и набором цифровых подписей, связанных с классом. Объект CodeSource создается с указанием объекта java.net.URL и массива объектов java.security.cert.Certificate. Только приложениям, создающим собственные объекты ClassLoader, может потребоваться использовать или создавать подклассы этого класса.

Когда CodeSource представляет определенную часть Java-кода, он включает в себя полный адрес и набор сертификатов, используемых для подписи кода. Если объект CodeSource определяет ProtectionDomain, то адрес может содержать групповые символы, а массив сертификатов является минимально необходимым набором подписей. Метод implies() такого объекта проверяет, действительно ли Java-класс был получен с указанного адреса и имеет ли он необходимый набор подписей.


```
public class CodeSource implements Serializable {
// Открытые методы
   public CodeSource(java.net.URL url, java.security.cert.Certificate[] certs);
// Открытые методы экземпляра
   public final java.security.cert.Certificate[] getCertificates();
   public final java.net.URL getLocation();
   public boolean implies(CodeSource codesource);
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: java.net.URLClassLoader.getPermissions(), CodeSource.implies(), java.security.Policy.getPermissions(), ProtectionDomain.ProtectionDomain(), SecureClassLoader.{defineClass(), getPermissions()}, javax.security.auth.Policy.getPermissions()

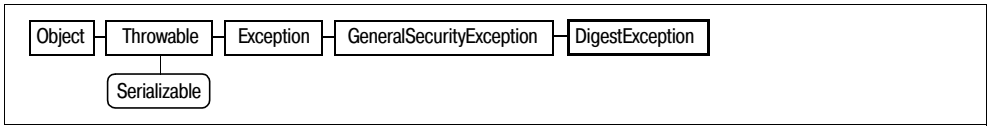
Возвращается методами: ProtectionDomain.getCodeSource()

DigestException

Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о проблемах при создании профиля сообщения.



```
public class DigestException extends GeneralSecurityException {
// Открытые методы
   public DigestException();
   public DigestException(String msg);
}
```

Генерируется методами: `MessageDigest.digest()`, `MessageDigestSpi.engineDigest()`

DigestInputStream Java 1.1

java.security

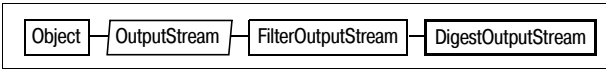
Этот класс представляет собой входной поток байтов и ассоциированный объект `MessageDigest`. Байты, читаемые одним из методов `read()`, автоматически передаются методу `update()` объекта `MessageDigest`. После завершения чтения байтов можно вызвать метод `digest()` объекта `MessageDigest` для получения профиля сообщения. Если же вы хотите вычислить профиль только для некоторых байтов, прочитанных из потока, применяйте метод `on()` для включения и выключения функции вычисления профиля. По умолчанию эта функция включена; вызывайте `on(false)` для ее выключения. См. также `DigestOutputStream` и `MessageDigest`.


```
public class DigestInputStream extends java.io.FilterInputStream {
// Открытые методы
   public DigestInputStream(java.io.InputStream stream, MessageDigest digest);
// Открытые методы экземпляра
   public MessageDigest getMessageDigest();
   public void on(boolean on);
   public void setMessageDigest(MessageDigest digest);
// Открытые методы, замещающие FilterInputStream
   public int read() throws java.io.IOException;
   public int read(byte[] b, int off, int len) throws java.io.IOException;
// Открытые методы, замещающие Object
   public String toString();
// Защищенные поля экземпляра
   protected MessageDigest digest;
}
```

DigestOutputStream Java 1.1

java.security

Этот класс представляет выходной поток байтов и ассоциированный объект `MessageDigest`. Байты, записываемые одним из методов `write()`, автоматически передаются методу `update()` объекта `MessageDigest`. После завершения записи байтов можно вызвать метод `digest()` объекта `MessageDigest` для получения профиля сообщения. Если вы хотите вычислить профиль только для некоторых байтов, записанных в поток, применяйте метод `on()` для включения и выключения функции вычисления профиля. По умолчанию эта функция включена; вызывайте `on(false)` для ее выключения. См. также `DigestOutputStream` и `MessageDigest`.




```
public class DigestOutputStream extends java.io.FilterOutputStream {
// Открытые методы
   public DigestOutputStream(java.io.OutputStream stream, MessageDigest digest);
// Открытые методы экземпляра
   public MessageDigest getMessageDigest();
   public void on(boolean on);
   public void setMessageDigest(MessageDigest digest);
// Открытые методы, замещающие FilterOutputStream
   public void write(int b) throws java.io.IOException;
   public void write(byte[] b, int off, int len) throws java.io.IOException;
// Открытые методы, замещающие Object
   public String toString();
// Защищенные поля экземпляра
   protected MessageDigest digest;
}
```

DomainCombiner

Java 1.3

java.security

Этот интерфейс определяет единственный метод `combine()`, который объединяет два массива объектов `ProtectionDomain` в один равнозначный (и, возможно, оптимизированный) массив. `DomainCombiner` можно связать с существующим объектом `AccessControlContext`, вызвав конструктор `AccessControlContext()` с двумя аргументами. Затем, когда метод `checkPermission()` объекта `AccessControlContext` вызван или объект `AccessControlContext` передан методу `doPrivileged()` класса `AccessController`, указанный объект `DomainCombiner` объединяет области защиты текущего кадра стека с областями защиты, инкапсулированными в `AccessControlContext`. Этот класс применяется только на системном уровне; обычные приложения используют его редко.

```
public interface DomainCombiner {
// Открытые методы экземпляра
   public abstract ProtectionDomain[] combine(ProtectionDomain[] currentDomains,
                                              ProtectionDomain[] assignedDomains);
}
```

Реализации: `javax.security.auth.SubjectDomainCombiner`

Передается методам: `AccessControlContext.AccessControlContext()`

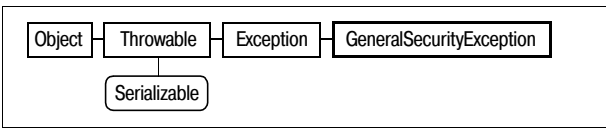
Возвращается методами: `AccessControlContext.getDomainCombiner()`

GeneralSecurityException

Java 1.2

java.security ***сериализуемое, проверяемое***

Этот класс является родительским классом большинства исключений, определенных в пакете `java.security`.



```
public class GeneralSecurityException extends Exception {
// Открытые методы
   public GeneralSecurityException();
   public GeneralSecurityException(String msg);
}
```

Подклассы: Классов слишком много, чтобы их перечислить.

Guard

Java 1.2

java.security

Этот интерфейс ограничивает доступ к объекту. Методу `checkGuard()` передается объект, к которому запрашивается доступ. Если доступ должен быть предоставлен, метод `checkGuard()` просто возвращает управление. Если в доступе отказано, метод `checkGuard()` вызывает исключение `java.lang.SecurityException`. В основном объект `Guard` применяется классом `GuardedObject`. Обратите внимание, что интерфейс `Guard` реализован всеми объектами `Permission`.

```
public interface Guard {
// Открытые методы экземпляра
   public abstract void checkGuard(Object object) throws SecurityException;
}
```

Реализации: `java.security.Permission`

Передается методам: `GuardedObject.GuardedObject()`

GuardedObject

Java 1.2

java.security ***сериализуемый***

Этот класс применяет объект `Guard` для защиты от неавторизованного доступа к произвольным инкапсулированным объектам. Создайте `GuardedObject`, указав объект и `Guard` для него. Метод `getObject()` вызывает метод `checkGuard()` объекта `Guard` для определения возможности доступа. Если доступ предоставлен, метод `getObject()` возвращает инкапсулированный объект. В противном случае он вызывает исключение `java.lang.SecurityException`.


```
public class GuardedObject implements Serializable {
// Открытые методы
   public GuardedObject(Object object, Guard guard);
// Открытые методы экземпляра
   public Object getObject() throws SecurityException;
}
```

Identity

Java 1.1; устарел в Java 1.2

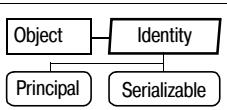
java.security ***сериализуемый***

Это устаревший класс. В Java 1.1 он реализовывал сущность, или принципал (объект `Principal`) с ассоциированным объектом `PublicKey`. В Java 1.1 открытый ключ для

именованного объекта может быть получен из системного хранилища ключей с помощью следующей строки:

```
IdentityScope.getSystemScope().getIdentity(name).getPublicKey()
```

Начиная с Java 1.2 класс Identity и родственные классы IdentityScope и Signer больше не используются. Вместо них применяются классы KeyStore и java.security.cert.Certificate.




```
public abstract class Identity implements java.security.Principal, Serializable {
// Открытые методы
   public Identity(String name);
   public Identity(String name, IdentityScope scope) throws KeyManagementException;
// Защищенные конструкторы
   protected Identity();
// Методы доступа к свойствам (по имени свойства)
   public String getInfo();
   public void setInfo(String info);
   public final String getName();                                    // Реализует:Principal
   public PublicKey getPublicKey();
   public void setPublicKey(PublicKey key) throws KeyManagementException;
   public final IdentityScope getScope();
// Открытые методы экземпляра
   public void addCertificate(java.security.Certificate certificate) throws KeyManagementException;
   public java.security.Certificate[] certificates();
   public void removeCertificate(java.security.Certificate certificate)
                                    throws KeyManagementException;
   public String toString(boolean detailed);
// Методы, реализующие Principal
   public final boolean equals(Object identity);
   public final String getName();
   public int hashCode();
   public String toString();
// Защищенные методы экземляра
   protected boolean identityEquals(Identity identity);
}
```

Подклассы: IdentityScope, Signer

Передается методам: Identity.identityEquals(), IdentityScope.{addIdentity(), removeIdentity()}

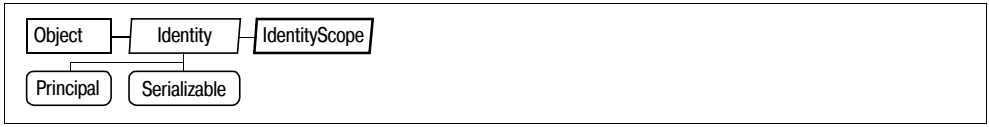
Возвращается методами: IdentityScope.getIdentity()

IdentityScope

Java 1.1; устарел в Java 1.2

java.security ***сериализуемый***

Это устаревший класс. В Java 1.1 он применялся для представления группы объектов Identity и Signer с ассоциированными объектами PublicKey и PrivateKey. В Java 1.2 он был заменен классом KeyStore.




```
public abstract class IdentityScope extends Identity {
// Открытые методы
   public IdentityScope(String name);
   public IdentityScope(String name, IdentityScope scope) throws KeyManagementException;
// Защищенные конструкторы
   protected IdentityScope();
// Открытые методы класса
   public static IdentityScope getSystemScope();
// Защищенные методы класса
   protected static void setSystemScope(IdentityScope scope);
// Открытые методы экземпляра
   public abstract void addIdentity(Identity identity) throws KeyManagementException;
   public abstract Identity getIdentity(String name);
   public Identity getIdentity(java.security.Principal principal);
   public abstract Identity getIdentity(PublicKey key);
   public abstract java.util.Enumeration identities();
   public abstract void removeIdentity(Identity identity) throws KeyManagementException;
   public abstract int size();
// Открытые методы, замещающие Identity
   public String toString();
}
```

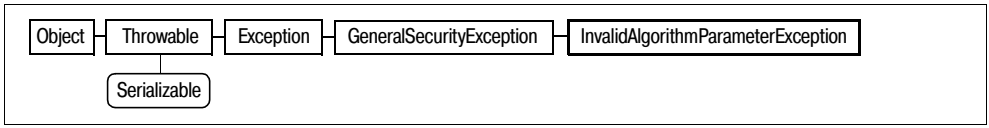
Передается методам: `Identity.Identity()`, `IdentityScope.{IdentityScope(), setSystemScope()}`, `Signer.Signer()`

Возвращается методами: `Identity.getScope()`, `IdentityScope.getSystemScope()`

InvalidAlgorithmParameterException Java 1.2

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что один или несколько параметров алгоритма, обычно передаваемых через объект `java.security.spec.AlgorithmParameterSpec`, неверны.



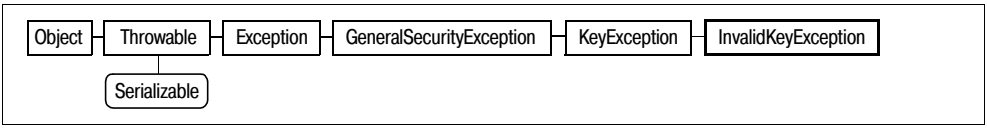

```
public class InvalidAlgorithmParameterException extends GeneralSecurityException {
// Открытые методы
   public InvalidAlgorithmParameterException();
   public InvalidAlgorithmParameterException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

InvalidKeyException Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что Key неверный.



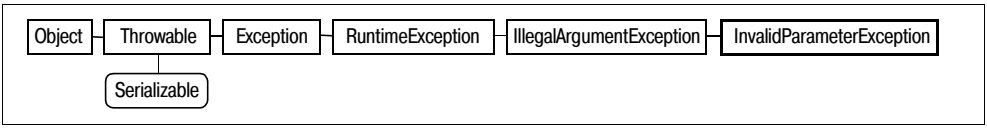

```
public class InvalidKeyException extends KeyException {
// Открытые методы
   public InvalidKeyException();
   public InvalidKeyException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

InvalidParameterException Java 1.1

java.security ***сериализуемое, непроверяемое***

Этот подкласс класса java.lang.IllegalArgumentException сигнализирует о том, что параметр, переданный в один из методов системы безопасности, неверен. Этот тип исключения применяется редко.




```
public class InvalidParameterException extends IllegalArgumentException {
// Открытые методы
   public InvalidParameterException();
   public InvalidParameterException(String msg);
}
```

Генерируется методами: Signature.{getParameter(), setParameter()}, SignatureSpi.{engineGetParameter(), engineSetParameter()}, Signer.setKeyPair(), java.security.interfaces.DSAKeyPairGenerator.initialize()

Key Java 1.1

java.security ***сериализуемый***

Этот интерфейс определяет высокоуровневые характеристики всех ключей шифрования. Метод getAlgorithm() возвращает имя алгоритма шифрования (например, RSA), применяемого с ключом. Метод getFormat() возвращает имя внешней кодировки (например, X.509), применяемое с ключом. Метод getEncoded() возвращает ключ в виде массива байтов, закодированный в формате, возвращаемом методом getFormat().

```
public interface Key extends Serializable {
// Открытые константы
1.2 public static final long serialVersionUID;                    // =6603384152749567654
// Открытые методы экземпляра
   public abstract String getAlgorithm();
   public abstract byte[] getEncoded();
   public abstract String getFormat();
}
```

Реализации: PrivateKey, PublicKey, javax.crypto.SecretKey

Передается методами: Методов слишком много, чтобы их перечислить.

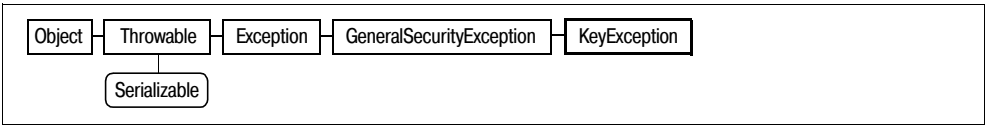
Возвращается методами: KeyFactory.translateKey(), KeyFactorySpi.engineTranslateKey(), KeyStore.getKey(), KeyStoreSpi.engineGetKey(), javax.crypto.Cipher.unwrap(), javax.crypto.CipherSpi.engineUnwrap(), javax.crypto.KeyAgreement.doPhase(), javax.crypto.KeyAgreementSpi.engineDoPhase()

KeyException

Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках с ключом. См. также подклассы InvalidKeyException и KeyManagementException.




```
public class KeyException extends GeneralSecurityException {
// Открытые методы
   public KeyException();
   public KeyException(String msg);
}
```

Подклассы: InvalidKeyException, KeyManagementException

Генерируется методами: java.security.Certificate.{decode(), encode()}, Signer.setKeyPair()

KeyFactory

Java 1.2

java.security

Этот класс выполняет преобразование асимметричных ключей между двумя представлениями, используемыми в API безопасности Java. java.security.Key – это непрозрачное, алгоритмо-независимое представление ключа, широко применяемого в данном API. java.security.spec.KeySpec – это интерфейс-маркер, реализованный прозрачным, алгоритмо-зависимым представлением ключей. KeyFactory применяется вместе с открытыми и закрытыми ключами; при работе с симметричными или секретными ключами обратитесь к описанию javax.crypto.SecretKeyFactory.

Для конвертации Key в KeySpec или обратного преобразования создайте KeyFactory, вызвав один из статических методов фабрики getInstance() с указанием имени алгоритма (например, DSA или RSA) и (необязательно) имени или объекта Provider желаемого поставщика. Затем с помощью generatePublic() или generatePrivate() создайте

объект PublicKey или PrivateKey из соответствующего KeySpec. Или задействуйте getKeySpec() для получения KeySpec данного ключа. Поскольку конкретный алгоритм шифрования может использовать несколько реализаций KeySpec, необходимо указать требуемый класс KeySpec.

Если вам не нужно передавать ключи между приложениями и/или системой, вы можете задействовать KeyStore для хранения и получения ключей и сертификатов, избегая KeySpec и KeyFactory.

```
public class KeyFactory {
// Защищенные конструкторы
   protected KeyFactory(KeyFactorySpi keyFacSpi, Provider provider, String algorithm);
// Открытые методы класса
   public static KeyFactory getInstance(String algorithm) throws NoSuchAlgorithmException;
   public static KeyFactory getInstance(String algorithm, String provider)
          throws NoSuchAlgorithmException, NoSuchProviderException;
1.4 public static KeyFactory getInstance(String algorithm, Provider provider)
         throws NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final PrivateKey generatePrivate(java.security.spec.KeySpec keySpec)
         throws java.security.spec.InvalidKeySpecException;
   public final PublicKey generatePublic(java.security.spec.KeySpec keySpec)
         throws java.security.spec.InvalidKeySpecException;
   public final String getAlgorithm();
   public final java.security.spec.KeySpec getKeySpec(Key key, Class keySpec)
         throws java.security.spec.InvalidKeySpecException;
   public final Provider getProvider();
   public final Key translateKey(Key key) throws InvalidKeyException;
}
```

Возвращается методами: KeyFactory.getInstance()

KeyFactorySpi Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для KeyFactory. Поставщик услуг должен реализовать конкретный подкласс этого класса для каждого поддерживаемого алгоритма шифрования. Приложения никогда не используют и никогда не создают подклассов этого класса.

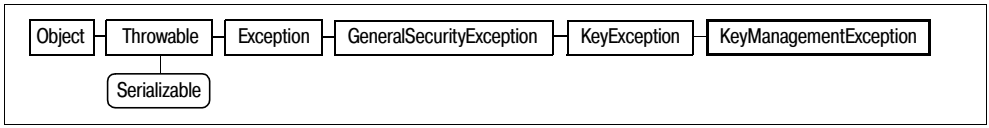
```
public abstract class KeyFactorySpi {
// Открытые методы
   public KeyFactorySpi();
// Защищенные методы экземпляра
   protected abstract PrivateKey engineGeneratePrivate(java.security.spec.KeySpec keySpec)
        throws java.security.spec.InvalidKeySpecException;
   protected abstract PublicKey engineGeneratePublic(java.security.spec.KeySpec keySpec)
        throws java.security.spec.InvalidKeySpecException;
   protected abstract java.security.spec.KeySpec engineGetKeySpec(Key key, Class keySpec)
        throws java.security.spec.InvalidKeySpecException;
   protected abstract Key engineTranslateKey(Key key) throws InvalidKeyException;
}
```

Передается методам: KeyFactory.KeyFactory()

KeyManagementException Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует об ошибке при выполнении операции с ключом. В Java 1.2 это исключение вызывается только устаревшими методами.




```
public class KeyManagementException extends KeyException {
// Открытые методы
   public KeyManagementException();
   public KeyManagementException(String msg);
}
```

Генерируется методами:
`Identity.{addCertificate(), Identity(), removeCertificate(), setPublicKey()}, IdentityScope.{addIdentity(), IdentityScope(), removeIdentity()}, Signer.Signer(), javax.net.ssl.SSLContext.init(), javax.net.ssl.SSLContextSpi.engineInit()`

KeyPair Java 1.1

java.security ***сериализуемый***

Этот класс является простым контейнером для объектов `PublicKey` и `PrivateKey`. Так как `KeyPair` содержит незащищенный закрытый ключ, он должен применяться с такой же осторожностью, с какой используется объект `PrivateKey`.


```
public final class KeyPair implements Serializable {
// Открытые методы
   public KeyPair(PublicKey publicKey, PrivateKey privateKey);
// Открытые методы экземпляра
   public PrivateKey getPrivate();
   public PublicKey getPublic();
}
```

Передается методам: `Signer.setKeyPair()`

Возвращается методами: `KeyPairGenerator.{generateKeyPair(), genKeyPair()}, KeyPairGeneratorSpi.generateKeyPair()`

KeyPairGenerator Java 1.1

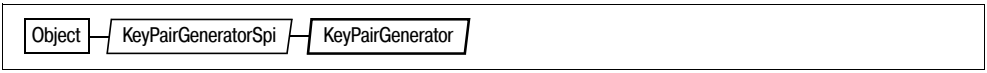
java.security

Этот класс генерирует пары открытых/закрытых ключей для указанных алгоритмов шифрования. Для создания `KeyPairGenerator` вызовите один из статических методов `getInstance()`, указав имя алгоритма и (необязательно) имя или объект `Provider`. Поставщик по умолчанию «SUN», поставляемый с Java 1.2, поддерживает только алго-

ритм «DSA». Поставщик «SunJCE» из Java Cryptography Extension (JCE) поддерживает алгоритм «DiffieHellman».

Как только вы создали KeyPairGenerator, проинициализируйте его, вызвав метод initialize(). Вы можете выполнить алгоритмо-независимую инициализацию, просто указав необходимый размер ключа в битах. Другой вариант – предоставить соответствующий объект AlgorithmParameterSpec алгоритму генерации ключей. В обоих случаях можно предоставить собственный генератор случайных чисел в виде объекта SecureRandom. После создания и инициализации KeyPairGenerator вызовите genKeyPair() для создания объекта KeyPair. Помните, что KeyPair содержит PrivateKey, который должен храниться в тайне.

По историческим причинам KeyPairGenerator расширяет KeyPairGeneratorSpi. Приложения не должны использовать методов, унаследованных от этого класса.




```
public abstract class KeyPairGenerator extends KeyPairGeneratorSpi {
// Защищенные конструкторы
   protected KeyPairGenerator(String algorithm);
// Открытые методы класса
   public static KeyPairGenerator getInstance(String algorithm) throws NoSuchAlgorithmException;
1.4 public static KeyPairGenerator getInstance(String algorithm, Provider provider)
        throws NoSuchAlgorithmException;
   public static KeyPairGenerator getInstance(String algorithm, String provider)
        throws NoSuchAlgorithmException, NoSuchProviderException;
// Открытые методы экземпляра
1.2 public final KeyPair genKeyPair();
   public String getAlgorithm();
1.2 public final Provider getProvider();
1.2 public void initialize(java.security.spec.AlgorithmParameterSpec params)
        throws InvalidAlgorithmParameterException;
   public void initialize(int keysize);
// Открытые методы, замещающие KeyPairGeneratorSpi
   public KeyPair generateKeyPair();  // константа
1.2 public void initialize(java.security.spec.AlgorithmParameterSpec params, SecureRandom random)
        throws InvalidAlgorithmParameterException;  // пустой
   public void initialize(int keysize, SecureRandom random);  // пустой
}
```

Возвращается методами: KeyPairGenerator.getInstance()

KeyPairGeneratorSpi Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для KeyPairGenerator. Поставщик услуг должен реализовать конкретный подкласс этого класса для каждого алгоритма, для которого он может генерировать ключи. Приложения никогда не используют и никогда не создают подклассов этого класса.

```
public abstract class KeyPairGeneratorSpi {
// Открытые методы
   public KeyPairGeneratorSpi();
// Открытые методы экземпляра
   public abstract KeyPair generateKeyPair();
```

```
   public void initialize(java.security.spec.AlgorithmParameterSpec params, SecureRandom random)
        throws InvalidAlgorithmParameterException;
   public abstract void initialize(int keysize, SecureRandom random);
}
```

Подклассы: KeyPairGenerator

KeyStore Java 1.2

java.security

Этот класс ставит в соответствие имена, или псевдонимы, объектам Key и java.security.cert.Certificate. Получите объект KeyStore, вызвав один из статических методов getInstance() и указав тип необходимого хранилища ключей и (необязательно) необходимого поставщика. Для указания типа «Java Key Store», определенного в Java, применяйте «JKS». Из-за ограничения по экспорту в Соединенных Штатах это хранилище поддерживает только слабое шифрование закрытых ключей. Если у вас установлено расширение Java Cryptography Extension, используйте тип «JCEKS» и поставщика «SunJCE» для получения реализации KeyStore, которая предлагает более сильное шифрование ключей на основе пароля. После создания KeyStore используйте метод load(), чтобы прочитать его содержимое из потока, указав необязательный пароль, который проверяет целостность потока данных. Хранилища ключей обычно считываются из файлов с именем .keystore, расположенных в домашнем каталоге пользователя.

Объект KeyStore может содержать открытый и закрытый ключи. Открытый ключ представлен объектом Certificate. Для поиска сертификата открытого ключа по имени применяется getCertificate(), а для добавления нового сертификата открытого ключа в хранилище – setCertificateEntry(). Закрытый ключ в хранилище содержит объект Key, защищенный паролем, и массив объектов Certificate, которые представляют собой цепочку сертификатов открытого ключа, соответствующего закрытому ключу. Используйте getKey() и getCertificateChain() для поиска ключа и цепочки сертификатов. При создании нового закрытого ключа применяется setKeyEntry(). Вы должны предоставить пароль для чтения или записи закрытого ключа в хранилище; этим паролем шифруются данные ключа, причем у каждого закрытого ключа должен быть собственный пароль. При использовании JCE можно сохранять объекты javax.crypto.SecretKey в хранилище KeyStore. Секретные ключи хранятся так же, как и закрытые ключи, за исключением того, что у них нет цепочки сертификатов.

Для удаления объекта из хранилища KeyStore применяется метод deleteEntry(). Если вы изменяете содержимое хранилища, используйте метод store() для сохранения хранилища в потоке. Можно указать пароль, который используется для проверки целостности данных. Этот пароль не применяется для шифрования хранилища.

```
public class KeyStore {
// Защищенные конструкторы
   protected KeyStore(KeyStoreSpi keyStoreSpi, Provider provider, String type);
// Открытые методы класса
   public static final String getDefaultType();
   public static KeyStore getInstance(String type) throws KeyStoreException;
   public static KeyStore getInstance(String type, String provider)
        throws KeyStoreException, NoSuchProviderException;
1.4 public static KeyStore getInstance(String type, Provider provider)
        throws KeyStoreException;
// Открытые методы экземпляра
   public final java.util.Enumeration aliases() throws KeyStoreException;
```

```
   public final boolean containsAlias(String alias) throws KeyStoreException;
   public final void deleteEntry(String alias) throws KeyStoreException;
   public final java.security.cert.Certificate getCertificate(String alias)
        throws KeyStoreException;
   public final String getCertificateAlias(java.security.cert.Certificate cert)
        throws KeyStoreException;
   public final java.security.cert.Certificate[] getCertificateChain(String alias)
        throws KeyStoreException;
   public final java.util.Date getCreationDate(String alias) throws KeyStoreException;
   public final Key getKey(String alias, char[] password)
        throws KeyStoreException, NoSuchAlgorithmException, UnrecoverableKeyException;
   public final Provider getProvider();
   public final String getType();
   public final boolean isCertificateEntry(String alias) throws KeyStoreException;
   public final boolean isKeyEntry(String alias) throws KeyStoreException;
   public final void load(java.io.InputStream stream, char[] password)
        throws java.io.IOException, NoSuchAlgorithmException,
      java.security.cert.CertificateException;
   public final void setCertificateEntry(String alias, java.security.cert.Certificate cert)
        throws KeyStoreException;
  public final void setKeyEntry(String alias, byte[] key, java.security.cert.Certificate[] chain)
        throws KeyStoreException;
   public final void setKeyEntry(String alias, Key key, char[] password,
        java.security.cert.Certificate[] chain) throws KeyStoreException;
   public final int size() throws KeyStoreException;
   public final void store(java.io.OutputStream stream, char[] password)
        throws KeyStoreException, java.io.IOException,
        NoSuchAlgorithmException, java.security.cert.CertificateException;
}
```

Передается методам: java.security.cert.PKIXBuilderParameters.PKIXBuilderParameters(), java.security.cert.PKIXParameters.PKIXParameters(), javax.net.ssl.KeyManagerFactory.init(), javax.net.ssl.KeyManagerFactorySpi.engineInit(), javax.net.ssl.TrustManagerFactory.init(), javax.net.ssl.TrustManagerFactorySpi.engineInit()

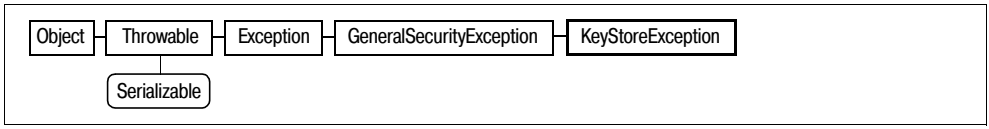
Возвращается методами: KeyStore.getInstance()

KeyStoreException

Java 1.2

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках с KeyStore.




```
public class KeyStoreException extends GeneralSecurityException {
// Открытые методы
   public KeyStoreException();
   public KeyStoreException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

KeyStoreSpi Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для KeyStore. Поставщик средств безопасности должен реализовать конкретный подкласс этого класса для каждого типа KeyStore, который он поддерживает. Приложения никогда не используют и никогда не создают подклассов этого класса.

```
public abstract class KeyStoreSpi {
// Открытые методы
   public KeyStoreSpi();
// Открытые методы экземпляра
   public abstract java.util.Enumeration engineAliases();
   public abstract boolean engineContainsAlias(String alias);
   public abstract void engineDeleteEntry(String alias) throws KeyStoreException;
   public abstract java.security.cert.Certificate engineGetCertificate(String alias);
   public abstract String engineGetCertificateAlias(java.security.cert.Certificate cert);
   public abstract java.security.cert.Certificate[] engineGetCertificateChain(String alias);
   public abstract java.util.Date engineGetCreationDate(String alias);
   public abstract Key engineGetKey(String alias, char[] password)
        throws NoSuchAlgorithmException, UnrecoverableKeyException;
   public abstract boolean engineIsCertificateEntry(String alias);
   public abstract boolean engineIsKeyEntry(String alias);
   public abstract void engineLoad(java.io.InputStream stream, char[] password)
        throws java.io.IOException, NoSuchAlgorithmException,
        java.security.cert.CertificateException;
   public abstract void engineSetCertificateEntry(String alias,
        java.security.cert.Certificate cert) throws KeyStoreException;
   public abstract void engineSetKeyEntry(String alias, byte[] key,
        java.security.cert.Certificate[] chain) throws KeyStoreException;
   public abstract void engineSetKeyEntry(String alias, Key key, char[] password,
        java.security.cert.Certificate[] chain) throws KeyStoreException;
   public abstract int engineSize();
   public abstract void engineStore(java.io.OutputStream stream, char[] password)
       throws java.io.IOException, NoSuchAlgorithmException, java.security.cert.CertificateException;
}
```

Передается методам: KeyStore.KeyStore()

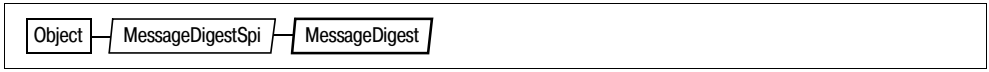
MessageDigest Java 1.1

java.security

Этот класс вычисляет профиль сообщения (также известный как криптографическая контрольная сумма) для произвольной последовательности байтов. Получите объект MessageDigest, вызвав один из статических методов фабрики getInstance() и указав необходимый алгоритм (например, SHA или MD5) и (необязательно) необходимого поставщика. Затем укажите данные, для которых нужно вычислить профиль, вызвав любой из методов update() один или несколько раз. И наконец, вызовите метод digest(), который вычисляет профиль сообщения и возвращает его как массив байтов. Если у вас есть только один массив байтов, для которого нужно вычислить профиль, вы можете сразу передать его методу digest() и пропустить вызов update(). Когда вы вызываете метод digest(), объект MessageDigest() сбрасывается. После этого он готов к вычислению нового профиля. Кроме того, можно сбросить MessageDigest без вычисления профиля, вызвав метод reset(). Для вычисления профиля части сообщения без

сбрасывания MessageDigest нужно создать копию MessageDigest и вызвать метод digest() копии. Обратите внимание, что копии можно сделать не для всех реализаций, поэтому метод clone() может вызвать исключение.

Класс MessageDigest часто применяется вместе с классами DigestInputStream и DigestOutputStream, которые автоматизируют вызовы метода update().




```
public abstract class MessageDigest extends MessageDigestSpi {
// Защищенные конструкторы
   protected MessageDigest(String algorithm);
// Открытые методы класса
   public static MessageDigest getInstance(String algorithm) throws NoSuchAlgorithmException;
1.4 public static MessageDigest getInstance(String algorithm, Provider provider)
        throws NoSuchAlgorithmException;
   public static MessageDigest getInstance(String algorithm, String provider)
        throws NoSuchAlgorithmException, NoSuchProviderException;
   public static boolean isEqual(byte[] digesta, byte[] digestb);
// Открытые методы экземпляра
   public byte[] digest();
   public byte[] digest(byte[] input);
1.2 public int digest(byte[] buf, int offset, int len) throws DigestException;
   public final String getAlgorithm();
1.2 public final int getDigestLength();
1.2 public final Provider getProvider();
   public void reset();
   public void update(byte[] input);
   public void update(byte input);
   public void update(byte[] input, int offset, int len);
// Открытые методы, замещающие MessageDigestSpi
   public Object clone() throws CloneNotSupportedException;
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методам: DigestInputStream.{DigestInputStream(), setMessageDigest()}, DigestOutputStream.{DigestOutputStream(), setMessageDigest()}

Возвращается методами: DigestInputStream.getMessageDigest(), DigestOutputStream.getMessageDigest(), MessageDigest.getInstance()

Экземпляры: DigestInputStream.digest, DigestOutputStream.digest

MessageDigestSpi Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для MessageDigest. Поставщик средств безопасности должен реализовать конкретный подкласс этого класса для каждого алгоритма вычисления профиля, который он поддерживает. Приложения никогда не используют и никогда не создают подклассов этого класса.

```
public abstract class MessageDigestSpi {
// Открытые методы
   public MessageDigestSpi();
```

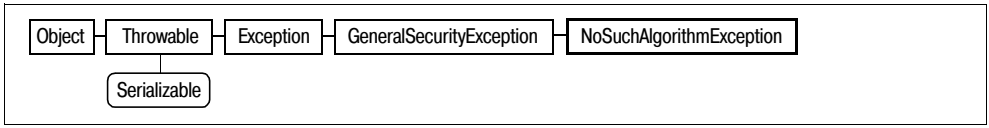
```
// Открытые методы, замещающие Object
   public Object clone() throws CloneNotSupportedException;
// Защищенные методы экземляра
   protected abstract byte[] engineDigest();
   protected int engineDigest(byte[] buf, int offset, int len) throws DigestException;
   protected int engineGetDigestLength();  // константа
   protected abstract void engineReset();
   protected abstract void engineUpdate(byte input);
   protected abstract void engineUpdate(byte[] input, int offset, int len);
}
```

Подклассы: MessageDigest

NoSuchAlgorithmException Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о недоступности запрошенного алгоритма шифрования. Оно вызывается методами фабрики `getInstance()` в пакете `java.security`.



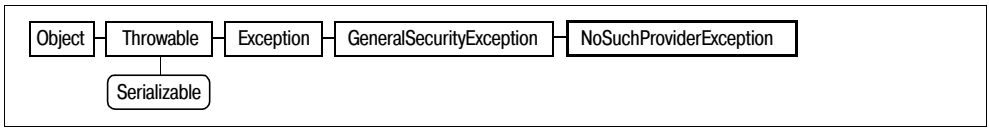

```
public class NoSuchAlgorithmException extends GeneralSecurityException {
// Открытые методы
   public NoSuchAlgorithmException();
   public NoSuchAlgorithmException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

NoSuchProviderException Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о недоступности запрошенного поставщика услуг шифрования. Оно вызывается методами фабрики `getInstance()` в пакете `java.security`.




```
public class NoSuchProviderException extends GeneralSecurityException {
// Открытые методы
   public NoSuchProviderException();
   public NoSuchProviderException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

Permission

Java 1.2

java.security ***сериализуемый***

Этот абстрактный класс представляет собой системный ресурс, например файл в файловой системе, или системную функциональность, например возможность установить сетевое соединение. Конкретные подклассы `Permission`, такие как `java.io.FilePermission` и `java.net.SocketPermission`, представляют конкретные типы ресурсов. Объекты `Permission` применяются системным кодом, который запрашивает доступ к ресурсу. Кроме того, они применяются объектами `Policy`, которые разрешают доступ к ресурсу. Метод `AccessController.checkPermission()` рассматривает исходный Java-код, выполняемый в данный момент, и определяет набор прав, предоставленных этому коду текущим объектом `Policy`. После этого данный метод проверяет, включен ли указанный объект `Permission` в этот набор. С появлением Java 1.2 этот механизм контроля доступа стал фундаментальным.

У каждого права есть имя, иногда называемое целевым объектом и (необязательно) список действий, разделенных запятыми. Например, имя `FilePermission` – это имя файла или каталога, для которых это право установлено. Действия, связанные с этим правом, могут быть следующими: «read», «write» или «read, write». Интерпретация имени и списка действий полностью ложится на реализацию `Permission`. Многие права поддерживают использование групповых символов; например, у `FilePermission` может быть имя «/tmp/*», которое представляет собой доступ к любому файлу в каталоге */tmp*. Объект `Permission` должен быть неизменным, поэтому его реализации никогда не должны определять методов `setName()` и `setActions()`.

Один из самых важных методов, определенных в `Permission` – это `implies()`. Данный метод должен вернуть `true`, если объект `Permission` включен в другой объект `Permission`. Например, если приложение запрашивает право `FilePermission` с именем «/tmp/test» и действие «read», а текущие настройки безопасности включают в себя объект `FilePermission` с именем «/tmp/*» и действия «read, write», то запрос удовлетворяется, поскольку запрошенное право входит в уже разрешенное право.

Как правило, только коду системного уровня нужно напрямую работать с классом `Permission` и его подклассами. Системные администраторы, конфигурирующие систему безопасности, должны знать различные подклассы `Permission`. Приложения, которые хотят расширить механизм контроля доступа Java для предоставления настраиваемого контроля доступа к своим ресурсам, должны создать подкласс класса `Permission` для определения собственных типов прав.


```
public abstract class Permission implements Guard, Serializable {
// Открытые методы
   public Permission(String name);
// Открытые методы экземпляра
   public abstract String getActions();
   public final String getName();
   public abstract boolean implies(java.security.Permission permission);
   public PermissionCollection newPermissionCollection();                    // константа
// Методы, реализующие Guard
   public void checkGuard(Object object) throws SecurityException;
```

```
// Открытые методы, замещающие Object
   public abstract boolean equals(Object obj);
   public abstract int hashCode();
   public String toString();
}
```

Подклассы: java.io.FilePermission, java.net.SocketPermission, AllPermission, BasicPermission, UnresolvedPermission, javax.security.auth.PrivateCredentialPermission, javax.security.auth.kerberos.ServicePermission

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.net.HttpURLConnection.getPermission(), java.net.URLConnection.getPermission(), AccessControlException.getPermission()

PermissionCollection

Java 1.2

java.security ***сериализуемый***

Этот класс применяется классом Permission для хранения коллекции объектов Permission одного и того же типа. Подобно классу Permission, класс PermissionCollection определяет метод implies(), который может определить, входит ли запрошенное право Permission в коллекцию. Некоторым типам Permission может потребоваться специальный тип PermissionCollection, чтобы правильно реализовать метод implies(). В этом случае подкласс должен замещать метод newPermissionCollection(), чтобы он возвращал Permission соответствующего типа. PermissionCollection применяется системным кодом, который управляет настройками безопасности. Приложения используют его редко.


```
public abstract class PermissionCollection implements Serializable {
// Открытые методы
   public PermissionCollection();
// Открытые методы экземпляра
   public abstract void add(java.security.Permission permission);
   public abstract java.util.Enumeration elements();
   public abstract boolean implies(java.security.Permission permission);
   public boolean isReadOnly();
   public void setReadOnly();
// Открытые методы, замещающие Object
   public String toString();
}
```

Подклассы: Permissions

Передается методам: ProtectionDomain.ProtectionDomain()

Возвращается методами: Методов слишком много, чтобы их перечислить.

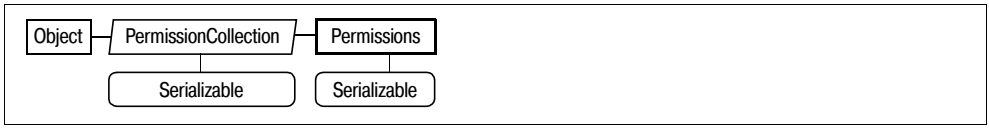
Permissions

Java 1.2

java.security ***сериализуемый***

Этот класс хранит произвольную коллекцию объектов Permission. При добавлении объектов Permission с помощью метода add() они группируются по внутренним набо-

рам объектов PermissionCollection, каждый из которых содержит объекты Permission одного типа. Применяйте метод elements(), чтобы получить перечисление (Enumeration) объектов Permission в коллекции. Права применяются системным кодом, который управляет настройками безопасности. Приложения используют его редко.




```
public final class Permissions extends PermissionCollection implements Serializable {
// Открытые методы
   public Permissions();
// Открытые методы, замещающие PermissionCollection
   public void add(java.security.Permission permission);
   public java.util.Enumeration elements();
   public boolean implies(java.security.Permission permission);
}
```

Policy Java 1.2

java.security

Этот класс представляет собой настройки безопасности, которые определяют права, назначаемые коду на основе исходного кода и подписавшей стороны. Кроме того, в Java 1.4 и последующих версиях эти права зависят от пользователя, который запускает код. В данный момент времени существует только один объект Policy. Получите объект Policy, вызвав метод getPolicy(). Код, имеющий соответствующие права, может указать новый объект Policy, вызвав метод setPolicy(). Метод refresh() является запросом к объекту Policy на обновление его состояния (например, повторное чтение конфигурационного файла). В основном класс Policy применяется на системном уровне. Приложениям не нужно использовать этот класс, если они не реализуют собственный механизм контроля доступа.

До Java 1.4 этот класс устанавливал связь между объектами CodeSource и PermissionCollection. Метод getPermissions() является центральным методом класса Policy; он оценивает Policy для переданного ему объекта CodeSource и возвращает соответствующий объект PermissionCollection, представляющий собой статический набор прав, доступных коду из этого источника.

В Java 1.4 объект ProtectionDomain можно использовать для инкапсуляции CodeSource и набора пользователей, которые запускают код. В этой версии появился новый метод getPermissions(), который возвращает PermissionsCollection, соответствующий указанному ProtectionDomain. Кроме того, появился новый метод implies(), который динамически опрашивает Policy, чтобы определить, есть ли указанная привилегия у данного ProtectionDomain.

```
public abstract class Policy {
// Открытые методы
   public Policy();
// Открытые методы класса
   public static java.security.Policy getPolicy();
   public static void setPolicy(java.security.Policy policy);
// Открытые методы экземпляра
   public abstract PermissionCollection getPermissions(CodeSource codesource);
```

```
1.4 public PermissionCollection getPermissions(ProtectionDomain domain);
1.4 public boolean implies(ProtectionDomain domain, java.security.Permission permission);
   public abstract void refresh();
}
```

Передается методам: java.security.Policy.setPolicy()

Возвращается методами: java.security.Policy.getPolicy()

Principal

Java 1.1

java.security

Этот интерфейс представляет любую сущность, которая может служить принципалом (principal) в криптографических транзакциях любого рода. Например, класс Principal может представлять человека, компьютер или организацию.

```
public interface Principal {
// Открытые методы экземпляра
   public abstract boolean equals(Object another);
   public abstract String getName();
   public abstract int hashCode();
   public abstract String toString();
}
```

Реализации: Identity, java.security.acl.Group, javax.security.auth.kerberos.KerberosPrincipal, javax.security.auth.x500.X500Principal

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.security.Certificate.{getGuarantor(), getPrincipal()}, ProtectionDomain.getPrincipals(), java.security.acl.AclEntry.getPrincipal(), java.security.cert.X509Certificate.{getIssuerDN(), getSubjectDN()}, java.security.cert.X509CRL.getIssuerDN(), javax.security.cert.X509Certificate.{getIssuerDN(), getSubjectDN()}

PrivateKey

Java 1.1

java.security ***сериализуемый***

Этот интерфейс представляет закрытый ключ шифрования. Он расширяет интерфейс Key, но не добавляет в него новых методов. Интерфейс существует для четкого различения открытого и закрытого ключей. См. также PublicKey.


```
public interface PrivateKey extends Key {
// Открытые константы
1.2 public static final long serialVersionUID;  // =6034044314589513430
}
```

Реализации: java.security.interfaces.DSAPrivateKey, java.security.interfaces.RSAPrivateKey, javax.crypto.interfaces.DHPrivateKey

Передается методам: KeyPair.KeyPair(), Signature.initSign(), SignatureSpi.engineInitSign(), SignedObject.SignedObject(), javax.security.auth.x500.X500PrivateCredential.X500PrivateCredential()

Возвращается методами: KeyFactory.generatePrivate(), KeyFactorySpi.engineGeneratePrivate(), KeyPair.getPrivate(), Signer.getPrivateKey(), javax.net.ssl.X509KeyManager.getPrivateKey(), javax.security.auth.x500.X500PrivateCredential.getPrivateKey()

PrivilegedAction

Java 1.2

java.security

Этот интерфейс определяет блок кода (метод run()), который должен быть выполнен как привилегированный код методом AccessController.doPrivileged(). Когда привилегированный код выполняется таким образом, AccessController проверяет только права вызывающего кода, а не права всего стека вызовов. Этот код обычно является абсолютно безопасным системным кодом, имеющим полный набор прав. Следовательно, привилегированный код выполняется с полным набором прав, даже если системный код вызывается небезопасным кодом, не имеющим вообще никаких прав.

Обычно привилегированный код необходим только при написании безопасной системной библиотеки (например, пакета расширений Java), которая должна читать локальные файлы или выполнять другие запрещенные действия, даже если она вызвана небезопасным кодом. Например, класс, который должен вызвать System.loadLibrary() для загрузки родных методов, должен выполнить вызов loadLibrary() внутри метода run() объекта PrivilegedAction. Если ваш привилегированный код может вызвать проверяемое исключение, то реализуйте его в методе run() объекта PrivilegedExceptionAction.

Будьте очень осторожны при реализации этого интерфейса. Для минимизации вероятности появления дыр в безопасности тело метода run() должно быть по возможности коротким.

```
public interface PrivilegedAction {
// Открытые методы экземпляра
   public abstract Object run();
}
```

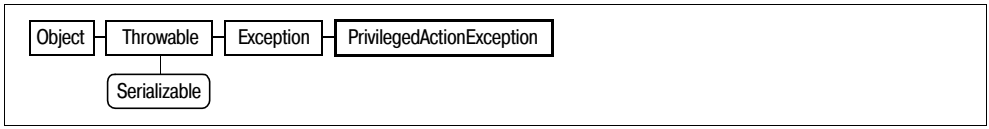
Передается методам: AccessController.doPrivileged(), javax.security.auth.Subject.{doAs(), doAsPrivileged()}

PrivilegedActionException

Java 1.2

java.security ***сериализуемое, проверяемое***

Это исключение является оберткой вокруг обычного объекта Exception, вызванного PrivilegedExceptionAction, который выполняется методом AccessController.doPrivileged(). Применяйте метод getException() для получения объекта Exception. В Java 1.4 и последующих версиях можно использовать более универсальный метод getCause().




```
public class PrivilegedActionException extends Exception {
// Открытые методы
   public PrivilegedActionException(Exception exception);
// Открытые методы экземпляра
```

```
   public Exception getException();
// Открытые методы, замещающие Throwable
1.4 public Throwable getCause();
1.3 public String toString();
}
```

Генерируется методами: AccessController.doPrivileged(), javax.security.auth.Subject.{doAs(), doAsPrivileged()}

PrivilegedExceptionAction Java 1.2

java.security

Этот интерфейс подобен PrivilegedAction за исключением того, что его метод run() может вызвать исключение. См. также PrivilegedAction.

```
public interface PrivilegedExceptionAction {
// Открытые методы экземпляра
   public abstract Object run() throws Exception;
}
```

Передается методам: AccessController.doPrivileged(), javax.security.auth.Subject.{doAs(), doAsPrivileged()}

ProtectionDomain Java 1.2

java.security

Этот класс представляет собой «область защиты»: набор прав, связанных с кодом на основе исходного кода, и (необязательно) идентификаторов пользователей, которые выполняют этот код. Используйте метод getProtectionDomain() объекта Class для получения ProtectionDomain, частью которого он является.

В версиях Java, предшествующих 1.4, ProtectionDomain просто связывает CodeSource с правами PermissionCollection, предоставленными Policy коду. Набор прав является статическим, а метод implies() проверяет, входит ли Permission в права, предоставленные этому ProtectionDomain.

В Java 1.4 и последующих версиях ProtectionDomain может быть создан с помощью конструктора с четырьмя аргументами, который связывает PermissionCollection с ClassLoader и массивом объектов Principal в дополнение к CodeSource. В этом случае ProtectionDomain представляет собой права, назначенные коду, который загружен из данного источника с помощью указанного загрузчика и выполняется от имени одного или нескольких указанных принципалов. Когда ProtectionDomain проинициализирован через четырехаргументный конструктор, то PermissionCollection не является статическим, а метод implies() вызывает метод implies() текущего объекта Policy перед проверкой указанной коллекции прав. Такой механизм позволяет обновлять настройки безопасности (например, добавлять новые права указанному пользователю) без перезапуска таких долго работающих программ, как серверы.

```
public class ProtectionDomain {
// Открытые методы
   public ProtectionDomain(CodeSource codesource, PermissionCollection permissions);
1.4 public ProtectionDomain(CodeSource codesource, PermissionCollection permissions,
                            ClassLoader classloader, java.security.Principal[] principals);
// Открытые методы экземпляра
1.4 public final ClassLoader getClassLoader();
```

```
    public final CodeSource getCodeSource();
    public final PermissionCollection getPermissions();
1.4 public final java.security.Principal[] getPrincipals();
    public boolean implies(java.security.Permission permission);
// Открытые методы, замещающие Object
    public String toString();
}
```

Передается методам: ClassLoader.defineClass(), AccessControlContext.AccessControlContext(), DomainCombiner.combine(), java.security.Policy.{getPermissions(), implies()}, javax.security.auth.SubjectDomainCombiner.combine()

Возвращается методами: Class.getProtectionDomain(), DomainCombiner.combine(), javax.security.auth.SubjectDomainCombiner.combine()

Provider Java 1.1

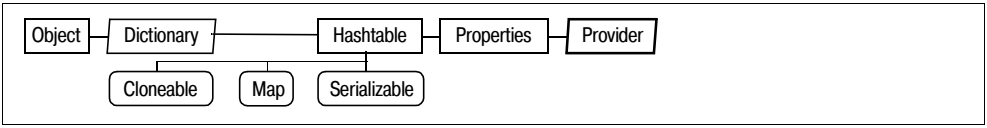
java.security ***клонируемый, сериализуемый, коллекция***

Этот класс представляет поставщика средств безопасности. Он указывает имена классов для реализации одного или нескольких алгоритмов для вычисления профилей сообщений, цифровых подписей, генерации ключей, перекодирования ключей, управления ключами, генерации криптостойких случайных чисел, преобразования сертификатов и управления параметрами алгоритмов. Методы getName(), getVersion() и getInfo() возвращают информацию о поставщике. Поставщик Provider наследуется от класса Properties и поддерживает пары «имя-значение» свойства. Эти пары определяют возможности реализации класса Provider. Имя каждого свойства имеет такую форму:

```
service_type.algorithm_name
```

Значение соответствующего свойства – это имя класса, который реализует алгоритм. Например, Provider определяет имена свойств «Signature.DSA», «MessageDigest.MD5» и «KeyStore.JKS». Значения данных свойств – это имена классов, реализующих SignatureSpi, MessageDigestSpi и KeyStoreSpi. Другие свойства, определенные классом Provider, применяются для определения псевдонимов имен алгоритмов. Например, свойство Alg.Alias.MessageDigest.SHA1 может иметь значение «SHA». Это означает, что имя алгоритма «SHA1» является псевдонимом имени «SHA».

В Java установка поставщиков средств безопасности зависит от реализации. В реализации Sun файл ${java.home}/lib/security/java.security определяет имена классов всех установленных реализаций класса Provider. Приложение может установить свои собственные реализации класса Provider с помощью методов addProvider() и insertProviderAt() класса Security. Большинству приложений не нужно напрямую использовать класс Provider. Некоторые приложения могут явно указывать имя желаемого Provider при вызове статического метода фабрики getInstance(). Собственного поставщика нужно устанавливать только приложениям, предъявляющим собственные требования к шифрованию.



```
public abstract class Provider extends java.util.Properties {
// Защищенные конструкторы
   protected Provider(String name, double version, String info);
// Открытые методы экземпляра
   public String getInfo();
   public String getName();
   public double getVersion();
// Открытые методы, замещающие Properties
1.2 public void load(java.io.InputStream inStream) throws java.io.IOException; // синхронизирован
// Открытые методы, замещающие Hashtable
1.2 public void clear();                                                      // синхронизирован
1.2 public java.util.Set entrySet();                                          // синхронизирован
1.2 public java.util.Set keySet();
1.2 public Object put(Object key, Object value);                              // синхронизирован
1.2 public void putAll(java.util.Map t);                                      // синхронизирован
1.2 public Object remove(Object key);                                         // синхронизирован
   public String toString();
1.2 public java.util.Collection values();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

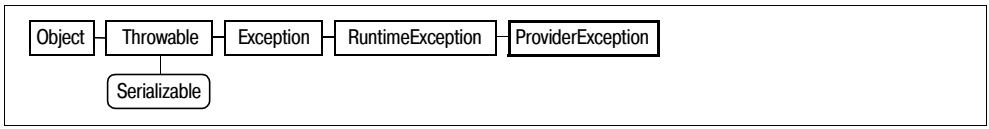
Возвращается методами: Методов слишком много, чтобы их перечислить.

ProviderException

Java 1.1

java.security ***сериализуемое, непроверяемое***

Это исключение сигнализирует о вызове исключения внутри поставщика услуг шифрования. Обратите внимание на то, что `ProviderException` расширяет `RuntimeException` и, следовательно, является непроверяемым исключением, которое может быть вызвано любым методом без объявления его в секции `throws`.




```
public class ProviderException extends RuntimeException {
// Открытые методы
   public ProviderException();
   public ProviderException(String s);
}
```

PublicKey

Java 1.1

java.security ***сериализуемый***

Этот интерфейс представляет собой открытый ключ шифрования. Он расширяет интерфейс `Key`, но не добавляет новых методов. Интерфейс существует для четкого различения открытого и закрытого ключей. См. также `PrivateKey`.

```
public interface PublicKey extends Key {
// Открытые константы
1.2 public static final long serialVersionUID;  // =7187392471159151072
}
```

Реализации: java.security.interfaces.DSAPublicKey, java.security.interfaces.RSAPublicKey, javax.crypto.interfaces.DHPublicKey

Передается методам: **Методов слишком много, чтобы их перечислить.**

Возвращается методами: java.security.Certificate.getPublicKey(), Identity.getPublicKey(), KeyFactory.generatePublic(), KeyFactorySpi.engineGeneratePublic(), KeyPair.getPublic(), java.security.cert.Certificate.getPublicKey(), java.security.cert.PKIXCertPathValidatorResult.getPublicKey(), java.security.cert.TrustAnchor.getCAPublicKey(), java.security.cert.X509CertSelector.getSubjectPublicKey(), javax.security.cert.Certificate.getPublicKey()

SecureClassLoader Java 1.2

java.security

Этот класс добавляет два защищенных метода к методам, которые уже определены классом ClassLoader**. Методу** defineClass() **передается объект** CodeSource**, который представляет собой исходный код загружаемого класса. Он вызывает метод** getPermissions()**, чтобы получить** PermissionCollection **для этого** CodeSource**. Затем он использует** CodeSource **и** PermissionCollection **для создания** ProtectionDomain**, который передается методу** defineClass() **родительского класса.**

Реализация по умолчанию метода getPermissions() **использует** Policy **по умолчанию, чтобы определить соответствующий набор прав для данного исходного кода. Ценность** SecureClassLoader **состоит в том, что подклассы могут использовать его метод** defineClass() **для загрузки классов без необходимости напрямую работать с классами** ProtectionDomain **и** Policy**. Подкласс может определить собственные настройки безопасности, подменив метод** getPermissions()**. В Java 1.2 и последующих версиях любое приложение, реализующее собственный загрузчик классов, должно расширять класс** SecureClassLoader **вместо прямого наследования класса** ClassLoader**. Впрочем, большинство приложений могут использовать класс** java.net.URLClassLoader **и не должны создавать подклассов.**


```
public class SecureClassLoader extends ClassLoader {
// Защищенные конструкторы
   protected SecureClassLoader();
   protected SecureClassLoader(ClassLoader parent);
// Защищенные методы экземляра
   protected final Class defineClass(String name, byte[] b, int off, int len,CodeSource cs);
   protected PermissionCollection getPermissions(CodeSource codesource);
}
```

Подклассы: java.net.URLClassLoader

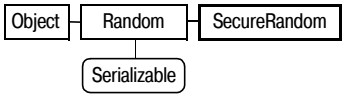
SecureRandom

Java 1.2

java.security *сериализуемый*

Этот класс генерирует криптостойкие псевдослучайные байты. Хотя класс SecureRandom определяет открытые конструкторы, предпочтительной техникой получения объекта SecureRandom является вызов одного из статических методов фабрики getInstance() с указанием требуемого алгоритма генерации псевдослучайных чисел и (необязательно) желаемого поставщика этого алгоритма. В реализации Java от Sun поставляется алгоритм с названием «SHA1PRNG» в поставщике «SUN».

После получения объекта SecureRandom вызовите метод nextBytes() для заполнения массива псевдослучайными числами. Кроме того, можно вызвать любой из методов, определенных подклассом Random для получения случайных чисел. При первом вызове одного из этих методов метод SecureRandom() использует метод generateSeed() для генерации начального значения псевдослучайной последовательности. Если у вас есть источник случайных чисел или псевдослучайные байты высокого качества, вы можете указать собственное начальное значение, вызвав метод setSeed(). Повторные вызовы setSeed() уточняют существующее начальное значение, а не заменяют его. Кроме того, можно вызвать метод generateSeed() для генерации начальных значений, которые будут применяться с другими генераторами псевдослучайных чисел. Метод generateSeed() может использовать алгоритм, отличный от алгоритма nextBytes(), и выдавать хорошие случайные числа за счет увеличения времени вычисления.




```
public class SecureRandom extends java.util.Random {
// Открытые методы
   public SecureRandom();
   public SecureRandom(byte[] seed);
// Защищенные конструкторы
1.2 protected SecureRandom(SecureRandomSpi secureRandomSpi, Provider provider);
// Открытые методы класса
1.2 public static SecureRandom getInstance(String algorithm)
        throws NoSuchAlgorithmException;
1.2 public static SecureRandom getInstance(String algorithm, String provider)
        throws NoSuchAlgorithmException, NoSuchProviderException;
1.4 public static SecureRandom getInstance(String algorithm, Provider provider)
        throws NoSuchAlgorithmException;
   public static byte[] getSeed(int numBytes);
// Открытые методы экземпляра
1.2 public byte[] generateSeed(int numBytes);
1.2 public final Provider getProvider();                          // по умолчанию:Sun
   public void setSeed(byte[] seed);                              // синхронизирован
// Открытые методы, замещающие Random
   public void nextBytes(byte[] bytes);                           // синхронизирован
   public void setSeed(long seed);
// Защищенные методы, замещающие Random
   protected final int next(int numBits);
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

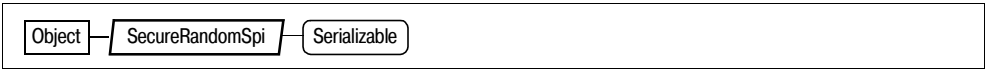
Возвращается методами: SecureRandom.getInstance()

Экземпляры: SignatureSpi.appRandom

SecureRandomSpi Java 1.2

java.security ***сериализуемый***

Этот абстрактный класс определяет интерфейс поставщика услуг для SecureRandom. Поставщик услуг должен реализовать конкретный подкласс этого класса для каждого поддерживаемого алгоритма генерации псевдослучайных чисел. Приложениям нет необходимости использовать этот класс.




```
public abstract class SecureRandomSpi implements Serializable {
// Открытые методы
   public SecureRandomSpi();
// Защищенные методы экземпляра
   protected abstract byte[] engineGenerateSeed(int numBytes);
   protected abstract void engineNextBytes(byte[] bytes);
   protected abstract void engineSetSeed(byte[] seed);
}
```

Передается методам: SecureRandom.SecureRandom()

Security Java 1.1

java.security

Этот класс определяет статические методы для управления списком установленных поставщиков средств безопасности и для чтения и установки значений различных свойств, используемых системой безопасности Java. По существу, это интерфейс к файлу свойств java.home/lib/security/java.security, который включен в реализацию Java от Sun. Применяйте методы getProperty() и setProperty() для получения и установки значений свойств безопасности, чьи значения по умолчанию хранятся в этом файле.

Одна из важных особенностей файла свойств java.security заключается в том, что он определяет набор поставщиков средств безопасности и предпочитаемый порядок, в котором они должны использоваться. Метод getProviders() возвращает массив объектов Provider, упорядоченный в том порядке, в котором они указаны в файле. В Java 1.3 и последующих версиях существуют варианты этого метода, которые возвращают только поставщиков, реализующих алгоритмы, указанные в объекте String или Map. Кроме того, объект Provider можно найти по имени с помощью метода getProvider(). Обратите внимание, что имя поставщика – это строка, возвращаемая методом getName() класса Provider, а не имя класса Provider.

Вы можете изменить набор поставщиков, устанавливаемых по умолчанию из файла java.security. Используйте метод addProvider() для добавления нового объекта Provider в конец списка. При этом новый объект будет иметь более низкий приоритет по сравнению с другими поставщиками. Используйте метод insertProviderAt() для вставки поставщика в указанную позицию списка. Обратите внимание, что позиции предпочтительности поставщика начинаются с 1. Задав позицию 1, вы тем самым указываете, что поставщик является самым предпочтительным. И наконец, применяйте метод removeProvider() для удаления поставщика с заданным именем.

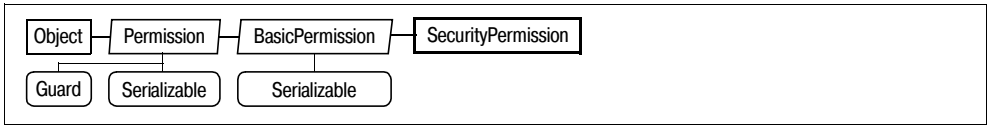
В Java 1.4 и последующих версиях метод getAlgorithms() возвращает объект Set, который включает в себя имена всех поддерживаемых алгоритмов (от всех установленных поставщиков) для указанной службы. Имя службы определяет запрашиваемую категорию службы безопасности. Это значение не чувствительно к регистру. Оно имеет то же имя, что и один из ключевых служебных классов из этого пакета или пакета, связанного с безопасностью, например «Signature», «MessageDigest» и «KeyStore» (из этого пакета) или «Cipher» (из пакета javax.crypto).

```
public final class Security {
// Конструктор отсутствует
// Открытые методы класса
   public static int addProvider(Provider provider);
1.4 public static java.util.Set getAlgorithms(String serviceName);
   public static String getProperty(String key);
   public static Provider getProvider(String name);                    // синхронизирован
   public static Provider[] getProviders();                            // синхронизирован
1.3 public static Provider[] getProviders(java.util.Map filter);
1.3 public static Provider[] getProviders(String filter);
   public static int insertProviderAt(Provider provider, int position); // синхронизирован
   public static void removeProvider(String name);                     // синхронизирован
   public static void setProperty(String key, String datum);
// Устаревшие открытые методы
# public static String getAlgorithmProperty(String algName, String propName);
}
```

SecurityPermission Java 1.2

java.security ***сериализуемый***

Этот класс является подклассом Permission, который представляет доступ к различным методам объектов Policy, Security, Provider, Signer и Identity. Объекты SecurityPermission определяются только по имени; они не используют список действий. Важные имена SecurityPermission – это «getPolicy» и «setPolicy», которые предоставляют возможность запрашивать и устанавливать системные настройки безопасности путем вызова методов Policy.getPolicy() и Policy.setPolicy(). Обычно приложениям не нужно использовать этот класс.




```
public final class SecurityPermission extends BasicPermission {
// Открытые методы
   public SecurityPermission(String name);
   public SecurityPermission(String name, String actions);
}
```

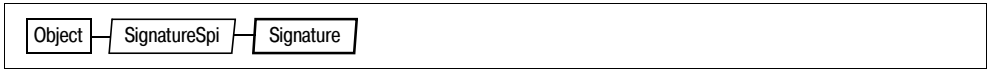
Signature Java 1.1

java.security

Этот класс вычисляет и проверяет цифровые подписи. Получите объект Signature, вызвав один из статических методов фабрики getInstance() и указав требуемый алго-

ритм цифровой подписи и (необязательно) желаемого поставщика этого алгоритма. *Цифровая подпись* – это, по существу, профиль сообщения, зашифрованный одним из алгоритмов шифрования с открытым ключом. Поэтому при указании алгоритма цифровой подписи нужно указать алгоритм профиля и алгоритм шифрования. Единственный алгоритм, поддерживаемый поставщиком по умолчанию «SUN», – это «SHA1withDSA».

После получения объекта `Signature` нужно проинициализировать его перед созданием или проверкой цифровых подписей. Для инициализации создания цифровой подписи вызовите метод `initSign()` и укажите закрытый ключ, который будет использоваться для создания подписи. Для инициализации проверки цифровой подписи вызовите метод `initVerify()` и передайте ему открытый ключ автора подписи. После инициализации объекта `Signature` вызовите метод `update()` один или несколько раз для указания байтов, которые нужно подписать или проверить. Наконец, для создания цифровой подписи вызовите метод `sign()`, передав ему массив, в котором будет храниться подпись. Или передайте байты цифровой подписи в метод `verify()`, который вернет `true`, если подпись верна, и `false` в противном случае. После вызова `sign()` или `verify()` объект `Signature` сбрасывается в начальное состояние и может применяться для создания или проверки другой подписи.




```
public abstract class Signature extends SignatureSpi {
// Защищенные конструкторы
   protected Signature(String algorithm);
// Защищенные константы
   protected static final int SIGN;                                          // =2
   protected static final int UNINITIALIZED;                                 // =0
   protected static final int VERIFY;                                        // =3
// Открытые методы класса
   public static Signature getInstance(String algorithm) throws NoSuchAlgorithmException;
1.4 public static Signature getInstance(String algorithm, Provider provider)
        throws NoSuchAlgorithmException;
   public static Signature getInstance(String algorithm, String provider)
        throws NoSuchAlgorithmException, NoSuchProviderException;
// Открытые методы экземпляра
   public final String getAlgorithm();
1.4 public final AlgorithmParameters getParameters();
1.2 public final Provider getProvider();
   public final void initSign(PrivateKey privateKey) throws InvalidKeyException;
1.2 public final void initSign(PrivateKey privateKey, SecureRandom random)
        throws InvalidKeyException;
1.3 public final void initVerify(java.security.cert.Certificate certificate)
        throws InvalidKeyException;
   public final void initVerify(PublicKey publicKey) throws InvalidKeyException;
1.2 public final void setParameter(java.security.spec.AlgorithmParameterSpec params)
        throws InvalidAlgorithmParameterException;
   public final byte[] sign() throws SignatureException;
1.2 public final int sign(byte[] outbuf, int offset, int len) throws SignatureException;
   public final void update(byte[] data) throws SignatureException;
   public final void update(byte b) throws SignatureException;
   public final void update(byte[] data, int off, int len) throws SignatureException;
   public final boolean verify(byte[] signature) throws SignatureException;
1.4 public final boolean verify(byte[] signature, int offset, int length)
```

```
        throws SignatureException;
// Открытые методы, замещающие SignatureSpi
    public Object clone() throws CloneNotSupportedException;
// Открытые методы, замещающие Object
    public String toString();
// Защищенные поля экземпляра
    protected int state;
// Устаревшие открытые методы
# public final Object getParameter(String param) throws InvalidParameterException;
# public final void setParameter(String param, Object value) throws InvalidParameterException;
}
```

Передается методам: `SignedObject.{SignedObject(), verify()}`

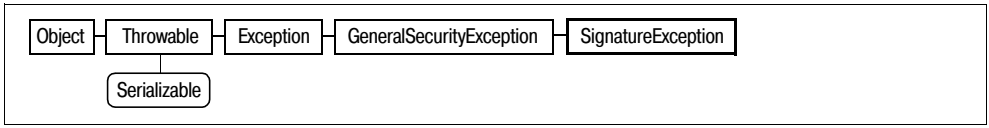
Возвращается методами: `Signature.getInstance()`

SignatureException

Java 1.1

java.security ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках при создании или проверке цифровой подписи.




```
public class SignatureException extends GeneralSecurityException {
// Открытые методы
    public SignatureException();
    public SignatureException(String msg);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

SignatureSpi

Java 1.2

java.security

Этот абстрактный класс определяет интерфейс поставщика услуг для `Signature`. Поставщик услуг должен реализовать конкретный подкласс для каждого поддерживаемого алгоритма цифровой подписи. Приложениям никогда не понадобится использовать или создавать подкласс этого класса.

```
public abstract class SignatureSpi {
// Открытые методы
    public SignatureSpi();
// Открытые методы, замещающие Object
    public Object clone() throws CloneNotSupportedException;
// Защищенные методы экземпляра
1.4 protected AlgorithmParameters engineGetParameters();
    protected abstract void engineInitSign(PrivateKey privateKey) throws InvalidKeyException;
    protected void engineInitSign(PrivateKey privateKey, SecureRandom random)
        throws InvalidKeyException;
```

```
    protected abstract void engineInitVerify(PublicKey publicKey) throws InvalidKeyException;
    protected void engineSetParameter(java.security.spec.AlgorithmParameterSpec params)
        throws InvalidAlgorithmParameterException;
    protected abstract byte[] engineSign() throws SignatureException;
    protected int engineSign(byte[] outbuf, int offset, int len) throws SignatureException;
    protected abstract void engineUpdate(byte b) throws SignatureException;
    protected abstract void engineUpdate(byte[] b, int off, int len) throws SignatureException;
    protected abstract boolean engineVerify(byte[] sigBytes) throws SignatureException;
1.4 protected boolean engineVerify(byte[] sigBytes, int offset, int length) throws SignatureException;
// Защищенные поля экземпляра
    protected SecureRandom appRandom;
// Устаревшие защищенные методы
# protected abstract Object engineGetParameter(String param) throws InvalidParameterException;
# protected abstract void engineSetParameter(String param, Object value)
        throws InvalidParameterException;
}
```

Подклассы: Signature

SignedObject

Java 1.2

java.security ***сериализуемый***

Этот класс применяет цифровую подпись к любому Java-объекту, который поддерживает сохранение и восстановление. Создайте SignedObject, указав объект, который должен быть подписан, PrivateKey, используемый для подписи, и объект Signature для создания подписи. Конструктор SignedObject() сохраняет объект в массиве байтов и создает цифровую подпись для этих байтов.

После создания объекта SignedObject его обычно сериализуют для хранения или передачи другому потоку или процессу Java. Как только объект SignedObject воссоздан, целостность объекта, который он содержит, может быть проверена вызовом метода verify() и передачей ему объекта PublicKey автора и объекта Signature, выполняющего проверку. Вне зависимости от того, была ли выполнена проверка и насколько она была успешной, можно вызвать метод getObject() для восстановления упакованного объекта.

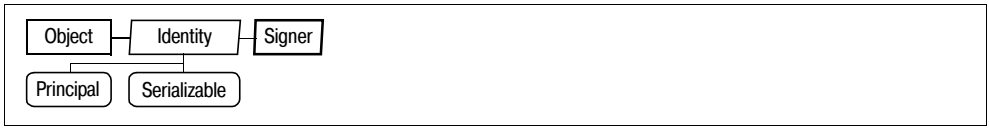

```
public final class SignedObject implements Serializable {
// Открытые методы
    public SignedObject(Serializable object, PrivateKey signingKey, Signature signingEngine)
        throws java.io.IOException, InvalidKeyException,  SignatureException;
// Открытые методы экземпляра
    public String getAlgorithm();
    public Object getObject() throws java.io.IOException, ClassNotFoundException;
    public byte[] getSignature();
    public boolean verify(PublicKey verificationKey, Signature verificationEngine)
        throws InvalidKeyException, SignatureException;
}
```

Signer

Java 1.1; устарел в Java 1.2

java.security ***сериализуемый***

Этот устаревший класс применялся в Java 1.1 для представления сущности, или принципала (объект Principal) с ассоциированным объектом PrivateKey, который позволяет создавать цифровые подписи. Начиная с Java 1.2 этот класс и родственные ему классы Identity и IdentityScope были заменены на KeyStore и java.security.cert.Certificate. См. также Identity.



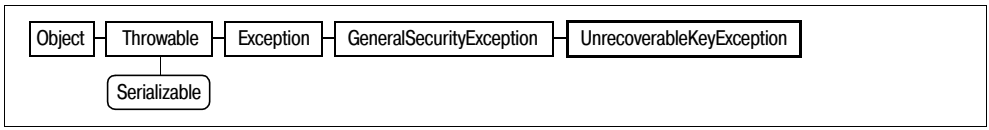

```
public abstract class Signer extends Identity {
// Открытые методы
   public Signer(String name);
   public Signer(String name, IdentityScope scope) throws KeyManagementException;
// Защищенные конструкторы
   protected Signer();
// Открытые методы экземпляра
   public PrivateKey getPrivateKey();
   public final void setKeyPair(KeyPair pair) throws InvalidParameterException, KeyException;
// Открытые методы, замещающие Identity
   public String toString();
}
```

UnrecoverableKeyException

Java 1.2

java.security ***сериализуемое, проверяемое***

Это исключение вызывается, если объект Key не может быть получен из хранилища KeyStore. Зачастую это происходит при использовании неверного пароля.



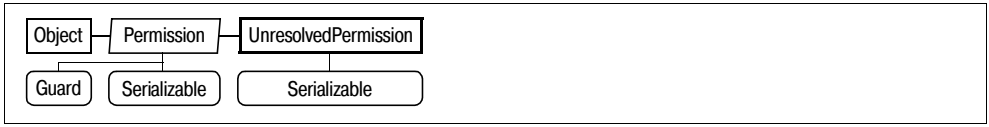

```
public class UnrecoverableKeyException extends GeneralSecurityException {
// Открытые методы
   public UnrecoverableKeyException();
   public UnrecoverableKeyException(String msg);
}
```

Генерируется методами: KeyStore.getKey(), KeyStoreSpi.engineGetKey(), javax.net.ssl.KeyManagerFactory.init(), javax.net.ssl.KeyManagerFactorySpi.engineInit()

UnresolvedPermission Java 1.2

java.security ***сериализуемый***

Этот класс предназначен для внутреннего использования. Он предоставляет механизм для задержанного назначения прав. Объект UnresolvedPermission содержит текстовое представление объекта Permission, который может быть позже задействован для создания настоящего объекта Permission. Приложениям никогда не понадобится использовать этот класс.




```
public final class UnresolvedPermission extends java.security.Permission implements Serializable {
// Открытые методы
   public UnresolvedPermission(String type, String name, String actions,
        java.security.cert.Certificate[] certs);
// Открытые методы, замещающие Permission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
   public boolean implies(java.security.Permission p);  // константа
   public PermissionCollection newPermissionCollection();
   public String toString();
}
```

Пакет java.security.acl Java 1.1

Пакет java.security.acl определяет, но не реализует, неполную структуру для работы со списками контроля доступа (access control list, ACL). Этот пакет был добавлен в Java 1.1, но замещен в Java 1.2 механизмами контроля доступа из пакета java.security. Обратите внимание на классы Permission и Policy из этого пакета. Применять данный пакет не рекомендуется.

Интерфейсы

```
public interface Acl extends Owner;
public interface AclEntry extends Cloneable;
public interface Group extends java.security.Principal;
public interface Owner;
public interface Permission;
```

Исключения

```
public class AclNotFoundException extends Exception;
public class LastOwnerException extends Exception;
public class NotOwnerException extends Exception;
```

Acl Java 1.1

java.security.acl

Этот интерфейс представляет собой *список контроля доступа*, или ACL. ACL – это список объектов AclEntry; большинство методов класса управляют этим списком. Исключение составляет метод checkPermission(), который проверяет, предоставляет ли данный ACL указанные права java.security.acl.Permission заданному принципалу java.security.Principal. Обратите внимание, что Acl расширяет Owner. Методы интерфейса Owner поддерживают список владельцев ACL. Изменять ACL могут только владельцы.


```
public interface Acl extends Owner {
// Открытые методы экземпляра
   public abstract boolean addEntry(java.security.Principal caller, AclEntry entry)
        throws NotOwnerException;
   public abstract boolean checkPermission(java.security.Principal principal,
                                           java.security.acl.Permission permission);
   public abstract java.util.Enumeration entries();
   public abstract String getName();
   public abstract java.util.Enumeration getPermissions(java.security.Principal user);
   public abstract boolean removeEntry(java.security.Principal caller, AclEntry entry)
        throws NotOwnerException;
   public abstract void setName(java.security.Principal caller, String name)
        throws NotOwnerException;
   public abstract String toString();
}
```

AclEntry Java 1.1

java.security.acl ***клонируемый***

Этот интерфейс определяет один элемент списка ACL. Каждый AclEntry представляет собой набор объектов java.security.acl.Permission, которые разрешены или запрещены для данного объекта java.security.Principal. По умолчанию AclEntry представляет права, данные принципалу. Вызовите метод setNegativePermissions(), если вы хотите, чтобы AclEntry представлял набор непредоставленных прав.


```
public interface AclEntry extends Cloneable {
// Открытые методы экземпляра
   public abstract boolean addPermission(java.security.acl.Permission permission);
   public abstract boolean checkPermission(java.security.acl.Permission permission);
   public abstract Object clone();
   public abstract java.security.Principal getPrincipal();
   public abstract boolean isNegative();
   public abstract java.util.Enumeration permissions();
   public abstract boolean removePermission(java.security.acl.Permission permission);
```

```
    public abstract void setNegativePermissions();
    public abstract boolean setPrincipal(java.security.Principal user);
    public abstract String toString();
}
```

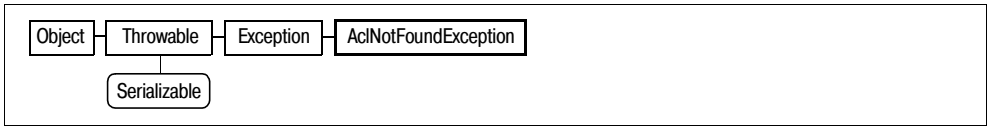
Передается методам: `Acl.{addEntry(), removeEntry()}`

AclNotFoundException

Java 1.1

java.security.acl ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что указанный `Acl` не может быть найден. Обратите внимание, что ни один из интерфейсов в пакете `java.security.acl` не вызывает этого исключения; оно предназначено для использования в реализации `Acl`.




```
public class AclNotFoundException extends Exception {
// Открытые методы
    public AclNotFoundException();
}
```

Group

Java 1.1

java.security.acl

Этот интерфейс представляет собой набор, или группу, объектов `java.security.Principal`. Методы этого интерфейса служат для управления членами группы. Заметьте, что `Group` расширяет интерфейс `Principal`, следовательно, вы можете использовать объект `Group` везде, где вы можете задействовать объект `Principal`.

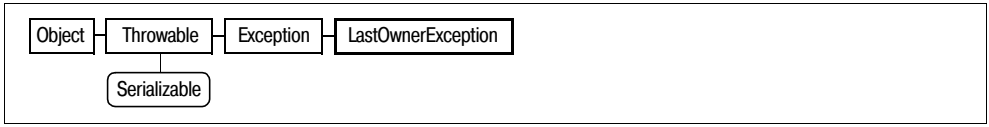

```
public interface Group extends java.security.Principal {
// Открытые методы экземпляра
    public abstract boolean addMember(java.security.Principal user);
    public abstract boolean isMember(java.security.Principal member);
    public abstract java.util.Enumeration members();
    public abstract boolean removeMember(java.security.Principal user);
}
```

LastOwnerException

Java 1.1

java.security.acl ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что у `Acl` или `Owner` остался единственный владелец, который не может быть удален.




```
public class LastOwnerException extends Exception {
// Открытые методы
   public LastOwnerException();
}
```

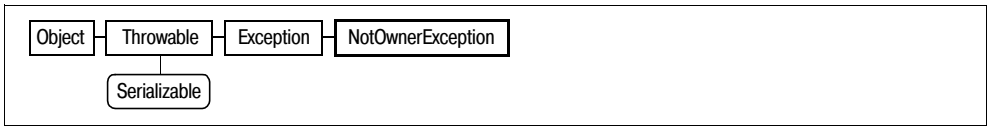
Генерируется методами: `Owner.deleteOwner()`

NotOwnerException

Java 1.1

java.security.acl ***сериализуемое, проверяемое***

Это исключение генерируется различными методами объектов `Acl` и `Owner`, когда они вызываются объектом `Principal`, который не является владельцем.




```
public class NotOwnerException extends Exception {
// Открытые методы
   public NotOwnerException();
}
```

Генерируется методами: `Acl.{addEntry(), removeEntry(), setName()}`, `Owner.{addOwner(), deleteOwner()}`

Owner

Java 1.1

java.security.acl

Этот интерфейс представляет собой владельца или владельцев списка ACL. Интерфейс определяет методы для управления списком владельцев и проверки их наличия в этом списке.

```
public interface Owner {
// Открытые методы экземпляра
   public abstract boolean addOwner(java.security.Principal caller, java.security.Principal owner)
         throws NotOwnerException;
   public abstract boolean deleteOwner(java.security.Principal caller, java.security.Principal owner)
         throws NotOwnerException, LastOwnerException;
   public abstract boolean isOwner(java.security.Principal owner);
}
```

Реализации: `Acl`

Permission Java 1.1

java.security.acl

Этот интерфейс представляет права. Значение прав полностью зависит от реализации. Не путайте этот интерфейс с новым классом `java.security.Permission`. Также обратите внимание, что этот интерфейс не имеет метода `implies()`, как класс `java.security.Permission`. Поэтому он менее универсален.

```
public interface Permission {
// Открытые методы экземпляра
   public abstract boolean equals(Object another);
   public abstract String toString();
}
```

Передается методам: `Acl.checkPermission()`, `AclEntry.{addPermission(), checkPermission(), removePermission()}`

Пакет java.security.cert Java 1.2

Пакет `java.security.cert` содержит классы для работы с сертификатами идентичности, цепочками сертификатов (также известными как пути сертификатов) и списками аннулированных сертификатов (certificate revocation list, CRL). Он определяет основные классы `Certificate` и `CRL`, а также классы `X509Certificate` и `X509CRL`, которые обеспечивают полную поддержку сертификатов и списков X.509. Класс `CertPath` реализует цепочку сертификатов, а класс `CertPathValidator` предоставляет возможность проверить эту цепочку. Класс `CertificateFactory` служит в качестве парсера сертификатов, позволяя конвертировать поток байтов (или байтов в кодировке base64) в объект `Certificate`, `CertPath` или `CRL`. В дополнение к алгоритмо-независимому API класса `CertificateFactory` этот пакет также определяет низкоуровневые, алгоритмо-зависимые классы для работы с цепочками сертификатов с использованием стандарта PKIX.

Данный пакет заменил устаревший интерфейс `java.security.Certificate` и устаревший пакет `javax.security.cert`, применявшиеся в ранних версиях JAAS API перед добавлением в Java пакета `javax.security.auth` и его подпакетов.

Интерфейсы

```
public interface CertPathBuilderResult extends Cloneable;
public interface CertPathParameters extends Cloneable;
public interface CertPathValidatorResult extends Cloneable;
public interface CertSelector extends Cloneable;
public interface CertStoreParameters extends Cloneable;
public interface CRLSelector extends Cloneable;
public interface PolicyNode;
public interface X509Extension;
```

Классы

```
public abstract class Certificate implements Serializable;
   └ public abstract class X509Certificate extends Certificate implements X509Extension;
public class CertificateFactory;
public abstract class CertificateFactorySpi;
public abstract class CertPath implements Serializable;
public class CertPathBuilder;
public abstract class CertPathBuilderSpi;
```

```
public class CertPathValidator;
public abstract class CertPathValidatorSpi;
public class CertStore;
public abstract class CertStoreSpi;
public class CollectionCertStoreParameters implements CertStoreParameters;
public abstract class CRL;
   └ public abstract class X509CRL extends CRL implements X509Extension;
public class LDAPCertStoreParameters implements CertStoreParameters;
public abstract class PKIXCertPathChecker implements Cloneable;
public class PKIXCertPathValidatorResult implements CertPathValidatorResult;
   └ public class PKIXCertPathBuilderResult extends PKIXCertPathValidatorResult
                                        implements CertPathBuilderResult;
public class PKIXParameters implements CertPathParameters;
   └ public class PKIXBuilderParameters extends PKIXParameters;
public final class PolicyQualifierInfo;
public class TrustAnchor;
public class X509CertSelector implements CertSelector;
public abstract class X509CRLEntry implements X509Extension;
public class X509CRLSelector implements CRLSelector;
```

Защищенные внутренние классы

```
protected static class Certificate.CertificateRep implements Serializable;
protected static class CertPath.CertPathRep implements Serializable;
```

Исключения

```
public class CertificateException extends java.security.GeneralSecurityException;
   └ public class CertificateEncodingException extends CertificateException;
   └ public class CertificateExpiredException extends CertificateException;
   └ public class CertificateNotYetValidException extends CertificateException;
   └ public class CertificateParsingException extends CertificateException;
public class CertPathBuilderException extends java.security.GeneralSecurityException;
public class CertPathValidatorException extends java.security.GeneralSecurityException;
public class CertStoreException extends java.security.GeneralSecurityException;
public class CRLException extends java.security.GeneralSecurityException;
```

Certificate Java 1.2

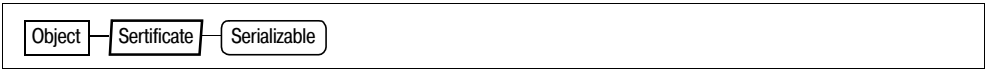
java.security.cert ***сериализуемый***

Этот абстрактный класс представляет собой сертификат открытого ключа (или идентичности). *Сертификат* – это объект, содержащий имя сущности и открытый ключ для данной сущности. Сертификаты выдаются и снабжаются цифровой подписью третьей стороной, считающейся надежной. Как правило, в этой роли выступает *издатель сертификатов* (certificate authority, CA). Выдачей и подписью сертификата издатель подтверждает, что, основываясь на его исследовании, личность или организация, указанная в сертификате, в действительности та, за кого себя выдает, а открытый ключ в сертификате действительно принадлежит этой личности (организации). Иногда сторона, подписавшая сертификат, не является надежным издателем сертификатов, а сам сертификат сопровождается сертификатом издателя, который может быть подписан другим ненадежным промежуточным издателем, предоставившим собственный сертификат. Цепочки таких сертификатов известны как пути сертификатов. Дополнительная информация представлена в описании `CertPath`.

Используйте `CertificateFactory` для преобразования потока байтов в объект `Certificate`; метод `getEncoded()` выполняет обратное преобразование. Метод `verify()` позволяет

проверить цифровую подпись издателя, выдавшего сертификат. Если подпись не может быть проверена, то сертификат не следует считать надежным. Вызовите метод `getPublicKey()`, чтобы получить объект `java.security.PublicKey` сертификата. Обратите внимание, что этот класс не определяет метода для получения объекта `Principal`, ассоциированного с объектом `PublicKey`. Данная функциональность зависит от типа сертификата. См., например, `X509Certificate.getSubjectDN()`.

Не путайте этот класс с интерфейсом `java.security.Certificate`, определенным в Java 1.1 и признанным устаревшим в Java 1.2.




```
public abstract class Certificate implements Serializable {
// Защищенные конструкторы
   protected Certificate(String type);
// Внутренние классы
1.3 protected static class CertificateRep implements Serializable;
// Открытые методы экземпляра
   public abstract byte[] getEncoded() throws java.security.cert.CertificateEncodingException;
   public abstract java.security.PublicKey getPublicKey();
   public final String getType();
   public abstract void verify(java.security.PublicKey key)
      throws java.security.cert.CertificateException,java.security.NoSuchAlgorithmException,
      java.security.InvalidKeyException, java.security.NoSuchProviderException,
      java.security.SignatureException;
   public abstract void verify(java.security.PublicKey key, String sigProvider)
      throws java.security.cert.CertificateException,java.security.NoSuchAlgorithmException,
      java.security.InvalidKeyException, java.security.NoSuchProviderException,
      java.security.SignatureException;
// Открытые методы, замещающие Object
   public boolean equals(Object other);
   public int hashCode();
   public abstract String toString();
// Защищенные методы экземляра
1.3 protected Object writeReplace() throws java.io.ObjectStreamException;
}
```

Подклассы: `java.security.cert.X509Certificate`

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: `java.net.JarURLConnection.getCertificates()`,
`java.security.CodeSource.getCertificates()`,
`java.security.KeyStore.{getCertificate(), getCertificateChain()}`,
`java.security.KeyStoreSpi.{engineGetCertificate(), engineGetCertificateChain()}`,
`CertificateFactory.generateCertificate()`,
`CertificateFactorySpi.engineGenerateCertificate()`,
`java.util.jar.JarEntry.getCertificates()`,
`javax.net.ssl.HandshakeCompletedEvent.{getLocalCertificates(), getPeerCertificates()}`,
`javax.net.ssl.HttpsURLConnection.{getLocalCertificates(), getServerCertificates()}`,
`javax.net.ssl.SSLSession.{getLocalCertificates(), getPeerCertificates()}`

Certificate.CertificateRep Java 1.3

java.security.cert ***сериализуемый***

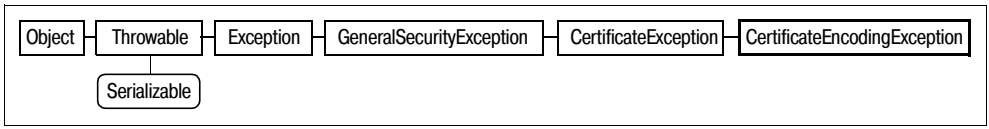
Этот защищенный внутренний класс обеспечивает альтернативное представление сертификата, которое может быть использовано для сохранения какой-нибудь простой реализации Certificate с помощью метода writeReplace(). Приложениям этот класс обычно не нужен.

```
protected static class Certificate.CertificateRep implements Serializable {
// Защищенные конструкторы
   protected CertificateRep(String type, byte[] data);
// Защищенные методы экземляра
   protected Object readResolve() throws java.io.ObjectStreamException;
}
```

CertificateEncodingException Java 1.2

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует об ошибке при шифровании сертификата.



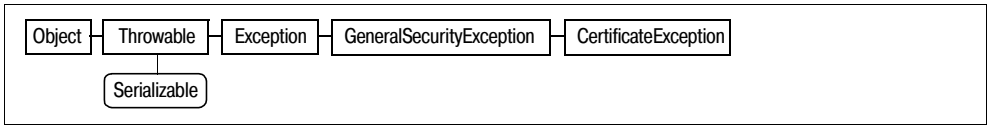

```
public class CertificateEncodingException extends java.security.cert.CertificateException {
// Открытые методы
   public CertificateEncodingException();
   public CertificateEncodingException(String message);
}
```

Генерируется методами: java.security.cert.Certificate.getEncoded(), CertPath.getEncoded(), java.security.cert.X509Certificate.getTBSCertificate()

CertificateException Java 1.2

java.security.cert ***сериализуемое, проверяемое***

Этот класс является родительским классом для нескольких более специфичных типов исключений, которые могут вызываться при работе с сертификатами.




```
public class CertificateException extends java.security.GeneralSecurityException {
// Открытые методы
   public CertificateException();
   public CertificateException(String msg);
}
```

Подклассы: java.security.cert.CertificateEncodingException, java.security.cert.CertificateExpiredException, java.security.cert.CertificateNotYetValidException, java.security.cert.CertificateParsingException

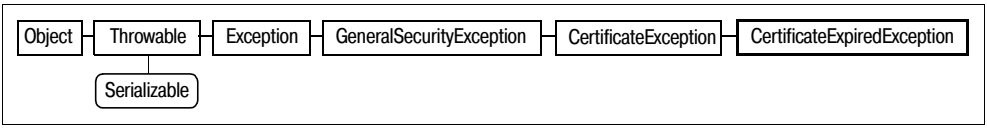
Генерируется методами: Методов слишком много, чтобы их перечислить.

CertificateExpiredException

Java 1.2

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что срок действия сертификата уже истек или истечет по достижении указанной даты.




```
public class CertificateExpiredException extends java.security.cert.CertificateException {
// Открытые методы
   public CertificateExpiredException();
   public CertificateExpiredException(String message);
}
```

Генерируется методами: java.security.cert.X509Certificate.checkValidity()

CertificateFactory

Java 1.2

java.security.cert

Этот класс определяет методы для получения сертификатов, цепочек сертификатов (путей сертификатов) и списков аннулированных сертификатов из потока байтов. Для получения CertificateFactory вызовите один из статических методов фабрики getInstance(), указав тип сертификата или CRL для разбора, а при необходимости – требуемого поставщика услуг сборки. Поставщик по умолчанию «SUN» определяет единственный тип сертификата «X.509», поэтому обычно CertificateFactory можно получить с помощью следующего кода:

```
CertificateFactory certFactory = CertificateFactory.getInstance("X.509");
```

Как только вы получили CertificateFactory для требуемого типа сертификата, вызовите метод generateCertificate(), чтобы собрать объект Certificate из указанного потока байтов или вызвать метод generateCertificates(), чтобы собрать группу несвязанных сертификатов (то есть сертификатов, которые не формируют цепочку сертификатов) из потока и вернуть ее как коллекцию Collection объектов Certificate. Подобным образом вызовите метод generateCRL(), чтобы собрать единичный объект CRL из потока байтов, или метод generateCRLs(), чтобы собрать коллекцию Collection объектов CRL из потока. Эти методы CertificateFactory читают указанный поток до конца. Если поток поддерживает методы mark() и reset(), то CertificateFactory возвращает указатель потока в позицию, в которой закончилось чтение последнего сертификата или CRL. Если вы указали тип сертификата «X.509», то объекты Certificate и CRL, возвращаемые CertificateFactory, могут быть безопасно приведены к типам X509Certificate и X509CRL. Фабрика сертификатов для сертификатов X.509 может собрать сертификат, закодированный в двоичной или шестнадцатеричной форме. Если сертифи-

кат представлен в шестнадцатеричной форме, то он должен начинаться со строки «-----BEGIN CERTIFICATE-----» и заканчиваться строкой «-----END CERTIFICATE-----».

Методы generateCertPath() возвращают объект CertPath, представляющий цепочку сертификатов. Эти методы могут создать объект CertPath из списка List объектов Certificate или прочитав сертификаты из потока. Укажите кодировку цепочки сертификатов, передав имя стандарта кодирования в метод generateCertPath(). Поставщик по умолчанию «SUN» поддерживает кодировки «PKCS7» и «PkiPath». Метод getCertPathEncodings() возвращает список Iterator кодировок, поддерживаемых поставщиком. Первая кодировка, возвращенная из списка, является кодировкой, используемой по умолчанию.

```
public class CertificateFactory {
// Защищенные конструкторы
   protected CertificateFactory(CertificateFactorySpi certFacSpi,
        java.security.Provider provider, String type);
// Открытые методы класса
   public static final CertificateFactory getInstance(String type)
        throws java.security.cert.CertificateException;
1.4 public static final CertificateFactory getInstance(String type, java.security.Provider
        provider) throws java.security.cert.CertificateException;
   public static final CertificateFactory getInstance(String type, String provider)
        throws java.security.cert.CertificateException, java.security.NoSuchProviderException;
// Открытые методы экземпляра
   public final java.security.cert.Certificate generateCertificate(java.io.InputStream inStream)
        throws java.security.cert.CertificateException;
   public final java.util.Collection generateCertificates(java.io.InputStream inStream)
        throws java.security.cert.CertificateException;
1.4 public final CertPath generateCertPath(java.util.List certificates)
        throws java.security.cert.CertificateException;
1.4 public final CertPath generateCertPath(java.io.InputStream inStream)
        throws java.security.cert.CertificateException;
1.4 public final CertPath generateCertPath(java.io.InputStream inStream, String encoding)
        throws java.security.cert.CertificateException;
   public final CRL generateCRL(java.io.InputStream inStream) throws CRLException;
   public final java.util.Collection generateCRLs(java.io.InputStream inStream)
        throws CRLException;
1.4 public final java.util.Iterator getCertPathEncodings();
   public final java.security.Provider getProvider();
   public final String getType();
}
```

Возвращается методами: CertificateFactory.getInstance()

CertificateFactorySpi Java 1.2

java.security.cert

Этот абстрактный класс определяет интерфейс поставщика услуг для класса CertificateFactory. Поставщик услуг должен реализовать этот класс для каждого типа сертификатов, который он желает поддерживать. Приложениям никогда не нужно использовать или создавать подклассы этого класса.

```
public abstract class CertificateFactorySpi {
// Открытые методы
   public CertificateFactorySpi();
// Открытые методы экземпляра
```

```
   public abstract java.security.cert.Certificate engineGenerateCertificate(java.io.InputStream inStream)
        throws java.security.cert.CertificateException;
   public abstract java.util.Collection engineGenerateCertificates(java.io.InputStream inStream)
        throws java.security.cert.CertificateException;
1.4 public CertPath engineGenerateCertPath(java.util.List certificates)
        throws java.security.cert.CertificateException;
1.4 public CertPath engineGenerateCertPath(java.io.InputStream inStream)
        throws java.security.cert.CertificateException;
1.4 public CertPath engineGenerateCertPath(java.io.InputStream inStream, String encoding)
        throws java.security.cert.CertificateException;
   public abstract CRL engineGenerateCRL(java.io.InputStream inStream) throws CRLException;
   public abstract java.util.Collection engineGenerateCRLs(java.io.InputStream inStream)
        throws CRLException;
1.4 public java.util.Iterator engineGetCertPathEncodings();
}
```

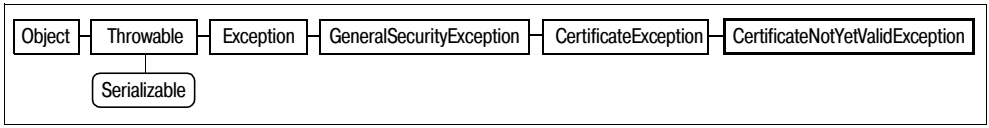
Передается методами: `CertificateFactory.CertificateFactory()`

CertificateNotYetValidException

Java 1.2

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что сертификат еще не действителен или не будет действителен до указанной даты.




```
public class CertificateNotYetValidException extends java.security.cert.CertificateException {
// Открытые методы
   public CertificateNotYetValidException();
   public CertificateNotYetValidException(String message);
}
```

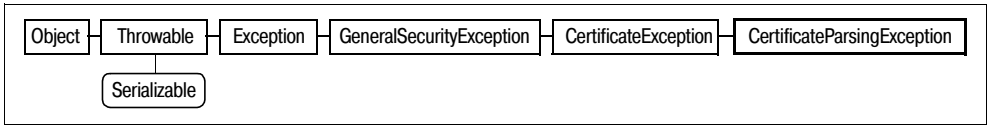
Генерируется методами: `java.security.cert.X509Certificate.checkValidity()`

CertificateParsingException

Java 1.2

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует об ошибке при сборке сертификата.




```
public class CertificateParsingException extends java.security.cert.CertificateException {
// Открытые методы
   public CertificateParsingException();
   public CertificateParsingException(String message);
}
```

Генерируется методами: java.security.cert.X509Certificate.{getExtendedKeyUsage(), getIssuerAlternativeNames(), getSubjectAlternativeNames()}

CertPath

Java 1.4

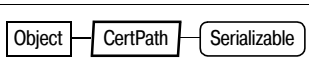
java.security.cert ***сериализуемый***

CertPath – это неизменная последовательность или цепочка сертификатов, которая устанавливает «путь сертификата» от неизвестной «конечной сущности» к известному и безопасному издателю сертификатов или «якорю доверия». Используйте CertPathValidator для проверки достоверности цепочки сертификатов и открытого ключа, представленного в сертификате конечной стороны.

Метод getType() возвращает тип сертификатов в CertPath. Для цепочек сертификатов X.509 (единственный тип, поддерживаемый поставщиком «SUN») этот метод возвращает «X.509». Метод getCertificates() возвращает объект java.util.List, который содержит объекты Certificate, входящие в цепочку. Для цепочек X.509 этот список содержит объекты X509Certificate. Кроме того, для путей сертификатов X.509 список List, возвращаемый методом getCertificates(), начинается с сертификата конечной стороны, а заканчивается сертификатом, подписанным первоначальным издателем. Издатель любого сертификата, за исключением последнего, должен быть субъектом следующего сертификата в списке. Если конечная сторона представляет сертификат, который подписан непосредственно первоначальным издателем, то список List содержит только данный сертификат. Обратите внимание, что список сертификатов не содержит сертификат первоначального издателя. Открытые ключи безопасного издателя должны быть заранее известны системе. В реализации JDK от Sun сертификаты открытых ключей надежных издателей хранятся в файле jre/lib/security/cacerts.

Объекты CertPath могут быть созданы с помощью CertificateFactory или посредством объекта CertPathBuilder (на нижнем уровне). CertificateFactory может собрать или разобрать объект CertPath из двоичного потока. Методы getEncoded() выполняют обратное преобразование и кодируют CertPath в массив байтов. Первое возвращаемое имя кодировки – это кодировка по умолчанию, но вы можете выбрать любую поддерживаемую кодировку, используя версию метода getEncoded() с одним параметром. Поставщик по умолчанию «SUN» поддерживает кодировки с именами «PKCS7» и «PkiPath».

Объекты CertPath неизменны, подобно объекту List, возвращаемому методом getCertificates(), и объектам Certificate, содержащимся в списке. Более того, все методы объекта CertPath являются потокобезопасными.




```
public abstract class CertPath implements Serializable {
// Защищенные конструкторы
   protected CertPath(String type);
// Внутренние классы
   protected static class CertPathRep implements Serializable;
// Открытые методы экземпляра
   public abstract java.util.List getCertificates();
   public abstract byte[] getEncoded() throws java.security.cert.CertificateEncodingException;
   public abstract byte[] getEncoded(String encoding) throws java.security.cert.CertificateEncodingException;
   public abstract java.util.Iterator getEncodings();
   public String getType();
```

```
// Открытые методы, замещающие Object
   public boolean equals(Object other);
   public int hashCode();
   public String toString();
// Защищенные методы экземляра
   protected Object writeReplace() throws java.io.ObjectStreamException;
}
```

Передается методам: CertPathValidator.validate(), CertPathValidatorException.CertPathValidatorException(), CertPathValidatorSpi.engineValidate(), PKIXCertPathBuilderResult.PKIXCertPathBuilderResult()

Возвращается методами: CertificateFactory.generateCertPath(), CertificateFactorySpi.engineGenerateCertPath(), CertPathBuilderResult.getCertPath(), CertPathValidatorException.getCertPath(), PKIXCertPathBuilderResult.getCertPath()

CertPath.CertPathRep

Java 1.4

java.security.cert ***сериализуемый***

Этот защищенный внутренний класс определяет представление CertPath для сохранения и последующего восстановления. Это представление не зависит от реализации. Приложениям никогда не нужно использовать этот класс.

```
protected static class CertPath.CertPathRep implements Serializable {
// Защищенные конструкторы
   protected CertPathRep(String type, byte[] data);
// Защищенные методы экземляра
   protected Object readResolve() throws java.io.ObjectStreamException;
}
```

CertPathBuilder

Java 1.4

java.security.cert

CertPathBuilder пытается построить путь от указанного сертификата к сертификату первоначального издателя. В отличие от метода CertificateFactory.generateCertPath(), который может применяться сервером для разбора цепочки сертификатов, предоставленной ему клиентом, этот класс используется для создания новой цепочки сертификатов. Он может применяться клиентом, которому нужно отправить цепочку сертификатов на сервер. API CertPathBuilder зависит от поставщика и не зависит от алгоритма, однако использование других алгоритмов, отличных от стандартов «PKIX» (которые работают с цепочками сертификатов X.509), требует соответствующей внешней реализации CertPathParameters и CertPathBuilderResult.

Получите объект CertPathBuilder, вызвав один из статических методов фабрики getInstance(). При этом укажите требуемый алгоритм и (необязательно) желаемого поставщика. Алгоритм «PKIX» – это единственный алгоритм, поддерживаемый поставщиком по умолчанию «SUN». Кроме того, это единственный алгоритм, для которого в пакете определены необходимые алгоритмо-зависимые классы. Получив объект CertPathBuilder, создайте объект CertPath, передав объект CertPathParameters в метод build(). Объект CertPathParameters является интерфейсом-маркером, который не определяет собственных методов, поэтому для предоставления информации, необходимой при создании CertPath, нужно использовать алгоритмо-зависимую реализацию этого интерфейса, например PKIXBuilderParameters. Метод build() возвращает объект

CertPathBuilderResult. Используйте метод getCertPath() возвращенного объекта для получения созданного объекта CertPath. Алгоритмо-зависимая реализация PKIXCertPathBuilderResult имеет дополнительные методы, которые возвращают более специфичные для алгоритма результаты.

```
public class CertPathBuilder {
// Защищенные конструкторы
   protected CertPathBuilder(CertPathBuilderSpi builderSpi, java.security.Provider provider,
                                                                    String algorithm);
// Открытые методы класса
   public static final String getDefaultType();
   public static CertPathBuilder getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static CertPathBuilder getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException,java.security.NoSuchProviderException;
   public static CertPathBuilder getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final CertPathBuilderResult build(CertPathParameters params)
        throws CertPathBuilderException,java.security.InvalidAlgorithmParameterException;
   public final String getAlgorithm();
   public final java.security.Provider getProvider();
}
```

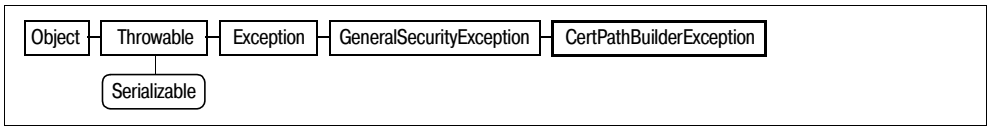
Возвращается методами: CertPathBuilder.getInstance()

CertPathBuilderException

Java 1.4

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках при создании пути сертификата с помощью CertPathBuilder.




```
public class CertPathBuilderException extends java.security.GeneralSecurityException {
// Открытые методы
   public CertPathBuilderException();
   public CertPathBuilderException(Throwable cause);
   public CertPathBuilderException(String msg);
   public CertPathBuilderException(String msg, Throwable cause);
// Открытые методы, замещающие Throwable
   public Throwable getCause();                                    // по умолчанию:null
   public String getMessage();                                     // по умолчанию:null
   public void printStackTrace();
   public void printStackTrace(java.io.PrintWriter pw);
   public void printStackTrace(java.io.PrintStream ps);
   public String toString();
}
```

Генерируется методами: CertPathBuilder.build(), CertPathBuilderSpi.engineBuild()

CertPathBuilderResult Java 1.4

java.security.cert ***клонируемый***

Объект этого типа возвращается методом build() класса CertPathBuilder. Метод getCertPath() возвращает созданный объект CertPath; этот метод никогда не вернет null. Алгоритмо-зависимая реализация PKIXCertPathBuilderResult определяет другие методы, предназначенные для возвращения дополнительной информации о созданном пути.


```
public interface CertPathBuilderResult extends Cloneable {
// Открытые методы экземпляра
   public abstract Object clone();
   public abstract CertPath getCertPath();
}
```

Реализации: PKIXCertPathBuilderResult

Возвращается методами: CertPathBuilder.build(), CertPathBuilderSpi.engineBuild()

CertPathBuilderSpi Java 1.4

java.security.cert

Этот абстрактный класс определяет интерфейс поставщика услуг для интерфейса CertPathBuilder. Поставщики услуг должны реализовать этот интерфейс, но приложениям никогда не нужно его использовать.

```
public abstract class CertPathBuilderSpi {
// Открытые методы
   public CertPathBuilderSpi();
// Открытые методы экземпляра
   public abstract CertPathBuilderResult engineBuild(CertPathParameters params)
        throws CertPathBuilderException, java.security.InvalidAlgorithmParameterException;
}
```

Передается методам: CertPathBuilder.CertPathBuilder()

CertPathParameters Java 1.4

java.security.cert ***клонируемый***

Интерфейс CertPathParameters – это интерфейс-маркер для объектов, которые хранят параметры (например, набор безопасных издателей) для проверки или создания пути сертификатов с помощью CertPathValidator и CertPathBuilder. Он не определяет собственных методов, но требует, чтобы все реализации включали в себя работающий метод clone(). При проверке или создании объекта CertPath нужно использовать алгоритмо-зависимую реализацию этого интерфейса, например PKIXParameters или PKIXBuilderParameters. Работать с этим интерфейсом напрямую редко бывает полезно.

```
public interface CertPathParameters extends Cloneable {
// Открытые методы экземпляра
   public abstract Object clone();
}
```

Реализации: PKIXParameters

Передается методам: CertPathBuilder.build(), CertPathBuilderSpi.engineBuild(), CertPathValidator.validate(), CertPathValidatorSpi.engineValidate()

CertPathValidator Java 1.4

java.security.cert

Этот класс проверяет цепочки сертификатов, устанавливая цепочку доверия от конечной стороны к безопасному издателю. Таким образом, он устанавливает истинность открытого ключа, представленного в сертификате. CertPathValidator основан на поставщике и не зависит от алгоритма. Для получения экземпляра CertPathValidator вызовите один из статических методов getInstance(), указав имя выбранного алгоритма проверки и (необязательно) поставщика. Алгоритм «PKIX» для проверки сертификатов X.509 – это единственный алгоритм, поддерживаемый поставщиком по умолчанию «SUN».

После получения объекта CertPathValidator его можно применять для проверки цепочек сертификатов, передавая объект CertPath, который нужно проверить, в метод validate() вместе с объектом CertPathParameters, определяющим надежных издателей и другие параметры для проверки. CertPathParameters – это простой интерфейс-маркер. Нужно применять его алгоритмо-зависимую реализацию, например PKIXParameters. Если проверка не прошла, то метод validate() вызовет исключение CertPathValidatorException. Оно может содержать индекс элемента в цепочке сертификатов, который не прошел проверку. Если проверка успешна, метод validate() вернет CertPathValidatorResult. Если вам интересны детали проверки (например, надежный издатель или открытый ключ конечной стороны), то вы можете привести результат к алгоритмо-зависимому подтипу, скажем, PKIXCertPathValidatorResult, и использовать его методы для получения дополнительной информации о результате.

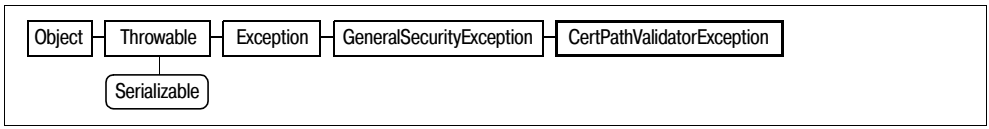
```
public class CertPathValidator {
// Защищенные конструкторы
   protected CertPathValidator(CertPathValidatorSpi validatorSpi,
             java.security.Provider provider, String algorithm);
// Открытые методы класса
   public static final String getDefaultType();
   public static CertPathValidator getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static CertPathValidator getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException,java.security.NoSuchProviderException;
   public static CertPathValidator getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final String getAlgorithm();
   public final java.security.Provider getProvider();
   public final CertPathValidatorResult validate(CertPath certPath, CertPathParameters params)
        throws CertPathValidatorException,java.security.InvalidAlgorithmParameterException;
}
```

Возвращается методами: CertPathValidator.getInstance()

CertPathValidatorException Java 1.4

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках при проверке цепочки сертификатов с помощью CertPathValidator. Метод getCertPath() возвращает проверяемый объект CertPath, а метод getIndex() возвращает индекс элемента внутри пути сертификатов, который вызвал исключение (или –1, если эта информация недоступна).




```
public class CertPathValidatorException extends java.security.GeneralSecurityException {
// Открытые методы
   public CertPathValidatorException();
   public CertPathValidatorException(Throwable cause);
   public CertPathValidatorException(String msg);
   public CertPathValidatorException(String msg, Throwable cause);
   public CertPathValidatorException(String msg, Throwable cause, CertPath certPath, int index);
// Открытые методы экземпляра
   public CertPath getCertPath();                                          // по умолчанию:null
   public int getIndex();                                                  // по умолчанию:-1
// Открытые методы, замещающие Throwable
   public Throwable getCause();                                            // по умолчанию:null
   public String getMessage();                                             // по умолчанию:null
   public void printStackTrace();
   public void printStackTrace(java.io.PrintStream ps);
   public void printStackTrace(java.io.PrintWriter pw);
   public String toString();
}
```

Генерируется методами: CertPathValidator.validate(), CertPathValidatorSpi.engineValidate(), PKIXCertPathChecker.{check(), init()}

CertPathValidatorResult Java 1.4

java.security.cert ***клонируемый***

Этот интерфейс-маркер определяет тип объекта, возвращаемого методом validate() объекта CertPathValidator, но не предоставляет никакой информации о содержимом этого объекта. Если вам нужна подробная информация о результатах проверки объекта CertPath, то значение, возвращаемое методом validate(), необходимо привести к алгоритмо-зависимой реализации этого интерфейса, например PKIXCertPathValidatorResult.


```
public interface CertPathValidatorResult extends Cloneable {
// Открытые методы экземпляра
   public abstract Object clone();
}
```

Реализации: PKIXCertPathValidatorResult

Возвращается методами: CertPathValidator.validate(), CertPathValidatorSpi.engineValidate()

CertPathValidatorSpi Java 1.4

java.security.sert

Этот абстрактный класс определяет интерфейс поставщика услуг для класса CertPathValidator. Поставщики услуг должны реализовать данный интерфейс, но приложениям никогда не нужно его использовать.

```
public abstract class CertPathValidatorSpi {
// Открытые методы
   public CertPathValidatorSpi();
// Открытые методы экземпляра
   public abstract CertPathValidatorResult engineValidate(CertPath certPath,
      CertPathParameters params) throws CertPathValidatorException,
      java.security.InvalidAlgorithmParameterException;
}
```

Передается методам: CertPathValidator.CertPathValidator()

CertSelector Java 1.4

java.security.cert ***клонируемый***

Этот интерфейс определяет API для выяснения соответствия сертификата Certificate определенным критериям. Реализации применяются для указания критериев, по которым сертификат или сертификаты должны выбираться из объекта CertStore. Метод match() должен исследовать переданный ему объект Certificate и вернуть true, если он соответствует критериям, определенным реализацией. В описании X509CertSelector представлена подробная информация о реализации, работающей с сертификатами X.509. Для получения детальной информации о похожем интерфейсе, который используется при выборе объектов CRL из CertStore, обратитесь к описанию CRLSelector.


```
public interface CertSelector extends Cloneable {
// Открытые методы экземпляра
   public abstract Object clone();
   public abstract boolean match(java.security.cert.Certificate cert);
}
```

Реализации: X509CertSelector

Передается методам: CertStore.getCertificates(), CertStoreSpi.engineGetCertificates(), PKIXBuilderParameters.PKIXBuilderParameters(), PKIXParameters.setTargetCertConstraints()

Возвращается методами: PKIXParameters.getTargetCertConstraints()

CertStore Java 1.4

java.security.cert

Объект CertStore – это хранилище объектов Certificate и CRL. Вы можете запросить у CertStore коллекцию java.util.Collection объектов Certificate или CRL, передав объект CertSelector или CRLSelector в метод getCertificates() или getCRLs(). Класс CertStore концептуально подобен классу java.security.KeyStore, но есть несколько важных различий в назначении этих классов. Класс CertStore создан для хранения небольших локальных коллекций закрытых ключей и надежных сертификатов. В отличие от него, класс CertStore может представлять большие открытые базы ненадежных сертификатов (например, в форме сервера LDAP).

Получите объект CertStore, вызвав метод getInstance() и указав имя необходимого типа CertStore и объект CertStoreParameters, который специфичен для этого типа. Кроме того, можно указать желаемого поставщика вашего объекта CertStore. Поставщик по умолчанию «SUN» определяет два типа CertStore с именами «LDAP» и «Collection», которые необходимо применять с объектами LDAPCertStoreParameters и CollectionCertStoreParameters соответственно. Тип «LDAP» получает сертификаты и списки аннулированных сертификатов с сервера LDAP, а тип «Collection» – из указанного объекта Collection.

Класс CertStore может быть непосредственно полезен приложениям, которые хотят получать сертификаты с сервера LDAP. Кроме того, он используется методами PKIXParameters.addCertStore() и PKIXParameters.setCertStores() для указания источников сертификатов, применяемых классами CertPathBuilder и CertPathValidator.

Все открытые методы CertStore являются потокобезопасными.

```
public class CertStore {
// Защищенные конструкторы
   protected CertStore(CertStoreSpi storeSpi, java.security.Provider provider, String type,
                                                                  CertStoreParameters params);
// Открытые методы класса
   public static final String getDefaultType();
   public static CertStore getInstance(String type, CertStoreParameters params)
      throws java.security.InvalidAlgorithmParameterException,
      java.security.NoSuchAlgorithmException;
   public static CertStore getInstance(String type, CertStoreParameters params, String provider)
      throws java.security.InvalidAlgorithmParameterException,
      java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
   public static CertStore getInstance(String type, CertStoreParameters params,
java.security.Provider provider)
      throws java.security.NoSuchAlgorithmException,java.security.InvalidAlgorithmParameterException;
// Открытые методы экземпляра
   public final java.util.Collection getCertificates(CertSelector selector)
      throws CertStoreException;
   public final CertStoreParameters getCertStoreParameters();
   public final java.util.Collection getCRLs(CRLSelector selector) throws CertStoreException;
   public final java.security.Provider getProvider();
   public final String getType();
}
```

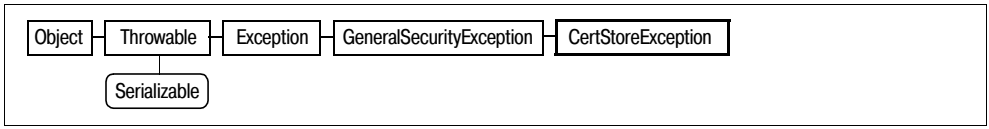
Передается методам: PKIXParameters.addCertStore()

Возвращается методами: CertStore.getInstance()

CertStoreException Java 1.4

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках при запросе сертификата или списка аннулированных сертификатов у `CertStore`.




```
public class CertStoreException extends java.security.GeneralSecurityException {
// Открытые методы
   public CertStoreException();
   public CertStoreException(Throwable cause);
   public CertStoreException(String msg);
   public CertStoreException(String msg, Throwable cause);
// Открытые методы, замещающие Throwable
   public Throwable getCause();                                    // по умолчанию:null
   public String getMessage();                                     // по умолчанию:null
   public void printStackTrace();
   public void printStackTrace(java.io.PrintWriter pw);
   public void printStackTrace(java.io.PrintStream ps);
   public String toString();
}
```

Генерируется методами: `CertStore.{getCertificates(), getCRLs()}`, `CertStoreSpi.{engineGetCertificates(), engineGetCRLs()}`

CertStoreParameters Java 1.4

java.security.cert ***клонируемый***

Этот интерфейс-маркер определяет тип (но не содержимое) объекта параметров, передаваемого в метод `CertStore.getInstance()`. Он не определяет никаких собственных методов, но требует, чтобы все реализующие его классы могли создавать копии. Применяйте одну из реализаций класса для объектов `CertStore` типа «LDAP» и «Collection».


```
public interface CertStoreParameters extends Cloneable {
// Открытые методы экземпляра
   public abstract Object clone();
}
```

Реализации: `CollectionCertStoreParameters`, `LDAPCertStoreParameters`

Передается методами: `CertStore.{CertStore(), getInstance()}`, `CertStoreSpi.CertStoreSpi()`

Возвращается методами: `CertStore.getCertStoreParameters()`

CertStoreSpi Java 1.4

java.security.cert

Этот абстрактный класс определяет интерфейс поставщика услуг для класса CertStore. Поставщики услуг должны реализовать этот интерфейс, но приложениям никогда не нужно его использовать.

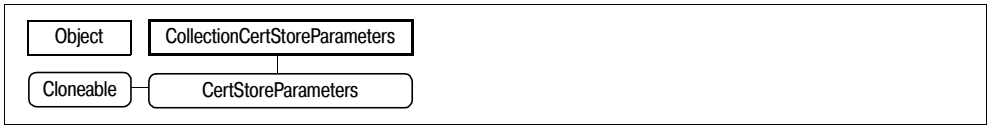
```
public abstract class CertStoreSpi {
// Открытые методы
   public CertStoreSpi(CertStoreParameters params)
          throws java.security.InvalidAlgorithmParameterException;
// Открытые методы экземпляра
   public abstract java.util.Collection engineGetCertificates(CertSelector selector)
          throws CertStoreException;
   public abstract java.util.Collection engineGetCRLs(CRLSelector selector)
          throws CertStoreException;
}
```

Передается методами: CertStore.CertStore()

CollectionCertStoreParameters Java 1.4

java.security.cert ***клонируемый***

Эта конкретная реализация CertStoreParameters применяется при создании объекта CertStore типа «Collection». Передайте в конструктор коллекцию Collection объектов Certificate и CRL.




```
public class CollectionCertStoreParameters implements CertStoreParameters {
// Открытые методы
   public CollectionCertStoreParameters();
   public CollectionCertStoreParameters(java.util.Collection collection);
// Открытые методы экземпляра
   public java.util.Collection getCollection();
// Методы, реализующие CertStoreParameters
   public Object clone();
// Открытые методы, замещающие Object
   public String toString();
}
```

CRL Java 1.2

java.security.cert

Этот абстрактный класс представляет собой *список аннулированных сертификатов* (certificate revocation list, CRL). CRL представляет собой объект, выпущенный издателем сертификатов (или другим издателем), в котором содержатся аннулированные сертификаты. Это означает, что теперь они недействительны и должны отвергаться. Для создания CRL из потока байтов применяйте CertificateFactory. Используйте ме-

тод isRevoked() для проверки наличия данного сертификата в списке CRL. Обратите внимание, что типо-зависимые подклассы CRL, например X509CRL, могут предоставлять доступ к значительно большему объему информации о списке аннулированных сертификатов.

```
public abstract class CRL {
// Защищенные конструкторы
   protected CRL(String type);
// Открытые методы экземпляра
   public final String getType();
   public abstract boolean isRevoked(java.security.cert.Certificate cert);
// Открытые методы, замещающие Object
   public abstract String toString();
}
```

Подклассы: X509CRL

Передается методам: CRLSelector.match(), X509CRLSelector.match()

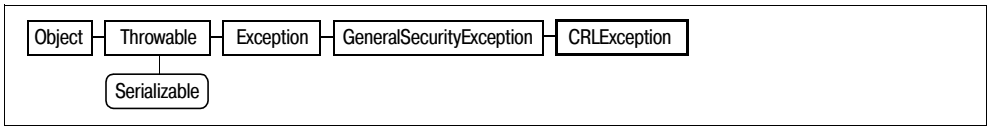
Возвращается методами: CertificateFactory.generateCRL(), CertificateFactorySpi.engineGenerateCRL()

CRLException

Java 1.2

java.security.cert ***сериализуемое, проверяемое***

Это исключение сигнализирует об ошибке при работе с CRL.




```
public class CRLException extends java.security.GeneralSecurityException {
// Открытые методы
   public CRLException();
   public CRLException(String message);
}
```

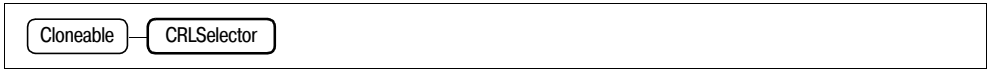
Генерируется методами: CertificateFactory.{generateCRL(), generateCRLs()}, CertificateFactorySpi.{engineGenerateCRL(), engineGenerateCRLs()}, X509CRL.{getEncoded(), getTBSCertList(), verify()}, X509CRLEntry.getEncoded()

CRLSelector

Java 1.4

java.security.cert ***клонируемый***

Этот интерфейс определяет API для выяснения соответствия объекта CRL некоторым критериям. Реализации применяются для указания критериев, по которым объекты CRL должны выбираться из CertStore. Метод match() должен исследовать полученный объект CRL и вернуть true, если он соответствует критериям, определенным реализацией. Реализация, работающая с сертификатами X.509, представлена в описании X509CRLSelector. Подобный интерфейс, применяемый при выборе объектов Certificate из CertStore, представлен в описании CertSelector.




```
public interface CRLSelector extends Cloneable {
// Открытые методы экземпляра
   public abstract Object clone();
   public abstract boolean match(CRL crl);
}
```

Реализации: X509CRLSelector

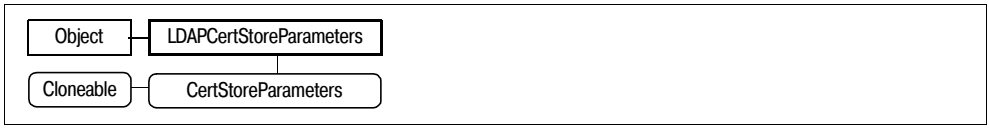
Передается методам: CertStore.getCRLs(), CertStoreSpi.engineGetCRLs()

LDAPCertStoreParameters

Java 1.4

java.security.cert ***клонируемый***

Эта конкретная реализация CertStoreParameters используется при создании объекта CertStore типа «LDAP». Она указывает имя LDAP-сервера, с которым нужно соединяться, и (необязательно) порт.



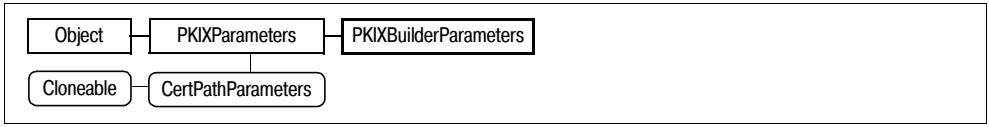

```
public class LDAPCertStoreParameters implements CertStoreParameters {
// Открытые методы
   public LDAPCertStoreParameters();
   public LDAPCertStoreParameters(String serverName);
   public LDAPCertStoreParameters(String serverName, int port);
// Открытые методы экземпляра
   public int getPort();                                    // по умолчанию:389
   public String getServerName();                           // по умолчанию:"localhost"
// Методы, реализующие CertStoreParameters
   public Object clone();
// Открытые методы, замещающие Object
   public String toString();
}
```

PKIXBuilderParameters

Java 1.4

java.security.cert ***клонируемый***

Экземпляры этого класса применяются для задания параметров метода build() объекта CertPathBuilder. Эти параметры должны включать в себя два обязательных параметра, которые передаются конструктору. Первый параметр – это источник надежных издателей, который может быть передан как набор Set объектов TrustAnchor или как объект java.security.KeyStore. Второй необходимый параметр – объект CertSelector (обычно X509CertSelector), который указывает критерии выбора сертификатов для создания пути сертификатов. В дополнение к параметрам, передаваемым конструктору, этот класс наследует несколько методов для задания других параметров и определяет метод setMaxPathLength() для указания максимальной длины создаваемой цепочки сертификатов.



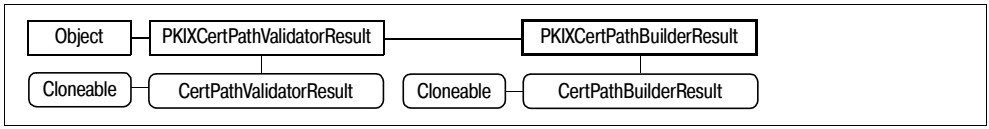

```
public class PKIXBuilderParameters extends PKIXParameters {
// Открытые методы
   public PKIXBuilderParameters(java.security.KeyStore keystore, CertSelector targetConstraints)
        throws java.security.KeyStoreException, java.security.InvalidAlgorithmParameterException;
   public PKIXBuilderParameters(java.util.Set trustAnchors, CertSelector targetConstraints)
        throws java.security.InvalidAlgorithmParameterException;
// Открытые методы экземпляра
   public int getMaxPathLength();
   public void setMaxPathLength(int maxPathLength);
// Открытые методы, замещающие PKIXParameters
   public String toString();
}
```

PKIXCertPathBuilderResult

Java 1.4

java.security.cert ***клонируемый***

Экземпляры этого класса возвращаются методом `build()` объекта `CertPathBuilder`, созданным для алгоритма «PKIX». Метод `getCertPath()` возвращает созданный объект `CertPath`, а методы, унаследованные от родительского класса, возвращают дополнительную информацию, например открытый ключ субъекта цепочки сертификата и сведения о надежном издателе, завершающем цепочку.




```
public class PKIXCertPathBuilderResult extends PKIXCertPathValidatorResult
       implements CertPathBuilderResult {
// Открытые методы
   public PKIXCertPathBuilderResult(CertPath certPath, TrustAnchor trustAnchor,
       PolicyNode policyTree, java.security.PublicKey subjectPublicKey);
// Методы, реализующие CertPathBuilderResult
   public CertPath getCertPath();
// Открытые методы, замещающие PKIXCertPathValidatorResult
   public String toString();
}
```

PKIXCertPathChecker

Java 1.4

java.security.cert ***клонируемый***

Этот абстрактный класс определяет механизм расширения для построения и проверки путей сертификатов по алгоритмам PKIX. Большинству приложений этот класс никогда не потребуется. Один или несколько объектов `PKIXCertPathChecker` можно передать в метод `setCertPathCheckers()` или `addCertPathChecker()` объекта `PKIXParameters` или `PKIXBuilderParameters`, который передается методам `build()` или `validate()` объек-

тов CertPathBuilder или CertPathValidator. Метод check() всех объектов PKIXCertPathChecker, зарегистрированный таким образом, будет вызываться для каждого сертификата, участвующего в алгоритме создания или проверки. Метод check() должен вызвать исключение CertPathValidatorException, если сертификат не прошел тест. Метод init() вызывается, чтобы сообщить контролеру (checker) о необходимости сбросить свое состояние в исходное и указать направление, в котором будут представлены сертификаты. Контролерам необязательно поддерживать прямое направление; они должны вернуть false как результат метода isForwardCheckingSupported(), если они не поддерживают это направление.


```
public abstract class PKIXCertPathChecker implements Cloneable {
// Защищенные конструкторы
   protected PKIXCertPathChecker();
// Открытые методы экземпляра
   public abstract void check(java.security.cert.Certificate cert,
      java.util.Collection unresolvedCritExts) throws CertPathValidatorException;
   public abstract java.util.Set getSupportedExtensions();
   public abstract void init(boolean forward) throws CertPathValidatorException;
   public abstract boolean isForwardCheckingSupported();
// Открытые методы, замещающие Object
   public Object clone();
}
```

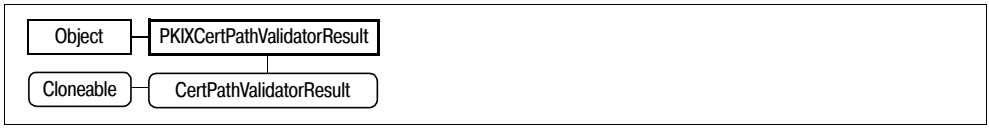
Передается методам: PKIXParameters.addCertPathChecker()

PKIXCertPathValidatorResult

Java 1.4

java.security.cert ***клонируемый***

Экземпляры этого класса возвращаются при удачной проверке методом validate() объекта CertPathValidator, созданного по алгоритму «PKIX». Метод getPublicKey() возвращает проверенный открытый ключ субъекта цепочки сертификатов. Метод getTrustAnchor() возвращает TrustAnchor, завершающий цепочку.




```
public class PKIXCertPathValidatorResult implements CertPathValidatorResult {
// Открытые методы
   public PKIXCertPathValidatorResult(TrustAnchor trustAnchor,
         PolicyNode policyTree, java.security.PublicKey subjectPublicKey);
// Открытые методы экземпляра
   public PolicyNode getPolicyTree();
   public java.security.PublicKey getPublicKey();
   public TrustAnchor getTrustAnchor();
// Методы, реализующие CertPathValidatorResult
   public Object clone();
// Открытые методы, замещающие Object
   public String toString();
}
```

Подклассы: PKIXCertPathBuilderResult

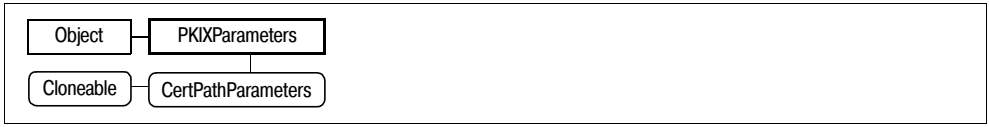
PKIXParameters

Java 1.4

java.security.cert ***клонируемый***

Эта реализация CertPathParameters определяет параметры, которые передаются методу validate() объекта PKIX CertPathValidator, а также подмножество параметров, которые передаются методу build() объекта PKIX CertPathValidator. Полное понимание этого класса требует детального обсуждения создания и проверки пути сертификатов по алгоритму PKIX. Такое обсуждение выходит за рамки данной книги. Впрочем, здесь описаны некоторые важные параметры.

При создании объекта PKIXParameters нужно указать надежных издателей. Для этого нужно передать в конструктор объект KeyStore, содержащий ключи надежных издателей. После создания объекта PKIXParameters вы можете изменить набор объектов TrustAnchor с помощью метода setTrustAnchors(). С помощью метода setCertStores() укажите набор Set объектов CertStore, по которому нужно искать сертификаты, или добавьте один объект CertStore с помощью метода addCertStore(). Если необходимо проверять достоверность сертификата не в текущий момент времени, используйте метод setDate() для указания интересующей даты.




```
public class PKIXParameters implements CertPathParameters {
// Открытые методы
   public PKIXParameters(java.security.KeyStore keystore)
        throws java.security.KeyStoreException, java.security.InvalidAlgorithmParameterException;
   public PKIXParameters(java.util.Set trustAnchors)
        throws java.security.InvalidAlgorithmParameterException;
// Методы доступа к свойствам (по имени свойства)
   public boolean isAnyPolicyInhibited();
   public void setAnyPolicyInhibited(boolean val);
   public java.util.List getCertPathCheckers();
   public void setCertPathCheckers(java.util.List checkers);
   public java.util.List getCertStores();
   public void setCertStores(java.util.List stores);
   public java.util.Date getDate();
   public void setDate(java.util.Date date);
   public boolean isExplicitPolicyRequired();
   public void setExplicitPolicyRequired(boolean val);
   public java.util.Set getInitialPolicies();
   public void setInitialPolicies(java.util.Set initialPolicies);
   public boolean isPolicyMappingInhibited();
   public void setPolicyMappingInhibited(boolean val);
   public boolean getPolicyQualifiersRejected();
   public void setPolicyQualifiersRejected(boolean qualifiersRejected);
   public boolean isRevocationEnabled();
   public void setRevocationEnabled(boolean val);
   public String getSigProvider();
   public void setSigProvider(String sigProvider);
   public CertSelector getTargetCertConstraints();
   public void setTargetCertConstraints(CertSelector selector);
```

```
   public java.util.Set getTrustAnchors();
   public void setTrustAnchors(java.util.Set trustAnchors)
        throws java.security.InvalidAlgorithmParameterException;
// Открытые методы экземпляра
   public void addCertPathChecker(PKIXCertPathChecker checker);
   public void addCertStore(CertStore store);
// Методы, реализующие CertPathParameters
   public Object clone();
// Открытые методы, замещающие Object
   public String toString();
}
```

Подклассы: PKIXBuilderParameters

PolicyNode

Java 1.4

java.security.cert

Этот класс представляет собой узел в дереве настроек безопасности, созданном алгоритмом проверки пути сертификатов PKIX. Обсуждение расширений настроек безопасности X.509 и их использования в алгоритме проверки пути сертификатов PKIX выходит за рамки данной книги.

```
public interface PolicyNode {
// Методы доступа к свойствам (по имени свойства)
   public abstract java.util.Iterator getChildren();
   public abstract boolean isCritical();
   public abstract int getDepth();
   public abstract java.util.Set getExpectedPolicies();
   public abstract PolicyNode getParent();
   public abstract java.util.Set getPolicyQualifiers();
   public abstract String getValidPolicy();
}
```

Передается методам: PKIXCertPathBuilderResult.PKIXCertPathBuilderResult(), PKIXCertPathValidatorResult.PKIXCertPathValidatorResult()

Возвращается методами: PKIXCertPathValidatorResult.getPolicyTree(), PolicyNode.getParent()

PolicyQualifierInfo

Java 1.4

java.security.cert

Этот класс является низкоуровневым представлением подробной информации о настройках безопасности из расширения сертификата X.509. Обсуждение расширений настроек безопасности X.509 и их использования в алгоритмах пути сертификатов PKIX выходит за рамки данной книги.

```
public final class PolicyQualifierInfo {
// Открытые методы
   public PolicyQualifierInfo(byte[] encoded) throws java.io.IOException;
// Открытые методы экземпляра
   public byte[] getEncoded();
   public byte[] getPolicyQualifier();
   public String getPolicyQualifierId();
// Открытые методы, замещающие Object
   public String toString();
}
```

TrustAnchor Java 1.4

java.security.cert

Класс TrustAnchor представляет собой сертификат авторитетного источника, который служит начальной точкой («якорем») цепочки сертификатов. Объект TrustAnchor включает уточненное имя (по стандарту X.500) и открытый ключ надежного издателя. Вы можете указать эти значения явно или передать объект X509Certificate в конструктор TrustAnchor(). Кроме того, обе формы конструктора TrustAnchor() позволяют указать массив байтов, содержащий двоичное представление расширения «Ограничение имени» («Name Constraints»). Описание формата и смысла такого ограничения выходят за рамки данной книги; большинство приложений могут просто указать null в качестве значения этого аргумента конструктора.

```
public class TrustAnchor {
// Открытые методы
   public TrustAnchor(java.security.cert.X509Certificate trustedCert, byte[] nameConstraints);
   public TrustAnchor(String caName, java.security.PublicKey pubKey, byte[] nameConstraints);
// Открытые методы экземпляра
   public final String getCAName();
   public final java.security.PublicKey getCAPublicKey();
   public final byte[] getNameConstraints();
   public final java.security.cert.X509Certificate getTrustedCert();
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методам: PKIXCertPathBuilderResult.PKIXCertPathBuilderResult(), PKIXCertPathValidatorResult.PKIXCertPathValidatorResult()

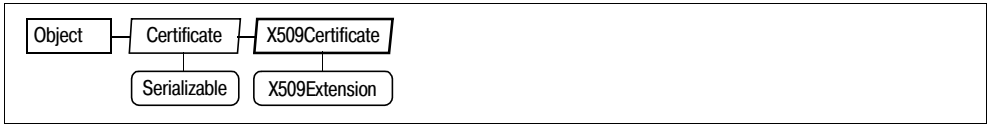
Возвращается методами: PKIXCertPathValidatorResult.getTrustAnchor()

X509Certificate Java 1.2

java.security.cert ***сериализуемый***

Этот класс представляет собой сертификат X.509. Его различные методы предоставляют полный доступ к содержимому сертификата. Полное понимание данного класса требует детальных знаний стандарта X.509, описание которого выходит за рамки данной книги. Тем не менее здесь представлены наиболее важные методы. Метод getSubjectDN() возвращает объект Principal, для которого выпущен этот сертификат, а унаследованный метод getPublicKey() возвращает объект PublicKey, который сертификат связывает с объектом Principal. Метод getIssuerDN() возвращает объект Principal, который представляет собой издателя сертификата. Если вам известен открытый ключ для этого Principal, вы можете передать его в метод verify() для проверки цифровой подписи издателя и истинности сертификата. Метод checkValidity() проверяет, не истек ли срок действия сертификата. Обратите внимание на то, что методы verify() и getPublicKey() унаследованы от класса Certificate.

Получите объект X509Certificate, создав CertificateFactory для типа сертификата «X.509» и вызвав метод generateCertificate() для сборки сертификата X.509 из потока байтов. Наконец, приведите результат метода Certificate к типу X509Certificate.




```
public abstract class X509Certificate extends java.security.cert.Certificate
                                              implements X509Extension {
// Защищенные конструкторы
   protected X509Certificate();
// Методы доступа к свойствам (по имени свойства)
   public abstract int getBasicConstraints();
1.4 public java.util.List getExtendedKeyUsage() throws java.security.cert.CertificateParsingException;
1.4 public java.util.Collection getIssuerAlternativeNames()
        throws java.security.cert.CertificateParsingException;
   public abstract java.security.Principal getIssuerDN();
   public abstract boolean[] getIssuerUniqueID();
1.4 public javax.security.auth.x500.X500Principal getIssuerX500Principal();
   public abstract boolean[] getKeyUsage();
   public abstract java.util.Date getNotAfter();
   public abstract java.util.Date getNotBefore();
   public abstract java.math.BigInteger getSerialNumber();
   public abstract String getSigAlgName();
   public abstract String getSigAlgOID();
   public abstract byte[] getSigAlgParams();
   public abstract byte[] getSignature();
1.4 public java.util.Collection getSubjectAlternativeNames()
        throws java.security.cert.CertificateParsingException;
   public abstract java.security.Principal getSubjectDN();
   public abstract boolean[] getSubjectUniqueID();
1.4 public javax.security.auth.x500.X500Principal getSubjectX500Principal();
   public abstract byte[] getTBSCertificate() throws java.security.cert.CertificateEncodingException;
   public abstract int getVersion();
// Открытые методы экземпляра
   public abstract void checkValidity() throws java.security.cert.CertificateExpiredException,
        java.security.cert.CertificateNotYetValidException;
   public abstract void checkValidity(java.util.Date date)
        throws java.security.cert.CertificateExpiredException,
        java.security.cert.CertificateNotYetValidException;
}
```

Передается методам: TrustAnchor.TrustAnchor(), X509CertSelector.setCertificate(), X509CRLSelector.setCertificateChecking(), javax.net.ssl.X509TrustManager.{checkClientTrusted(), checkServerTrusted()}, javax.security.auth.x500.X500PrivateCredential.X500PrivateCredential()

Возвращается методами: TrustAnchor.getTrustedCert(), X509CertSelector.getCertificate(), X509CRLSelector.getCertificateChecking(), javax.net.ssl.X509KeyManager.getCertificateChain(), javax.net.ssl.X509TrustManager.getAcceptedIssuers(), javax.security.auth.x500.X500PrivateCredential.getCertificate()

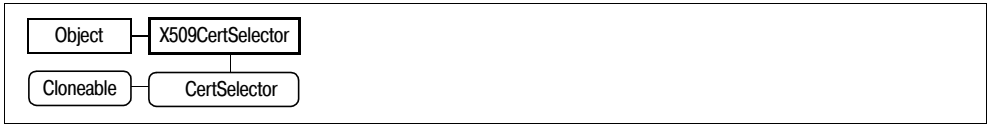
X509CertSelector Java 1.4

java.security.cert ***клонируемый***

Этот класс является реализацией CertSelector **для сертификатов X.509. Его методы позволяют указать значения различных полей и расширений сертификата. Метод**

match() вернет true только для сертификатов, в которых заданы эти поля и расширения. Полное понимание этого класса требует детальных знаний стандарта X.509, описание которого выходит за рамки данной книги. Тем не менее здесь представлены наиболее важные методы.

Если нужно проверить один конкретный сертификат, просто передайте X509Certificate в метод setCertificate(). Укажите субъект сертификата с помощью методов setSubject(), setSubjectAlternativeNames() или addSubjectAlternativeName(); укажите издателя сертификата с помощью метода setIssuer(); укажите открытый ключ сертификата с помощью метода setPublicKey(); укажите срок подлинности сертификата с помощью метода setCertificateValid() и серийный номер издателя сертификата с помощью метода setSerialNumber().




```
public class X509CertSelector implements CertSelector {
// Открытые методы
   public X509CertSelector();
// Методы доступа к свойствам (по имени свойства)
   public byte[] getAuthorityKeyIdentifier();                          // по умолчанию:null
   public void setAuthorityKeyIdentifier(byte[] authorityKeyID);
   public int getBasicConstraints();                                     // по умолчанию:-1
   public void setBasicConstraints(int minMaxPathLen);
   public java.security.cert.X509Certificate getCertificate();         // по умолчанию:null
   public void setCertificate(java.security.cert.X509Certificate cert);
   public java.util.Date getCertificateValid();                        // по умолчанию:null
   public void setCertificateValid(java.util.Date certValid);
   public java.util.Set getExtendedKeyUsage();                         // по умолчанию:null
   public void setExtendedKeyUsage(java.util.Set keyPurposeSet) throws java.io.IOException;
   public byte[] getIssuerAsBytes() throws java.io.IOException;        // по умолчанию:null
   public String getIssuerAsString();                                  // по умолчанию:null
   public boolean[] getKeyUsage();                                     // по умолчанию:null
   public void setKeyUsage(boolean[] keyUsage);
   public boolean getMatchAllSubjectAltNames();                        // по умолчанию:true
   public void setMatchAllSubjectAltNames(boolean matchAllNames);
   public byte[] getNameConstraints();                                 // по умолчанию:null
   public void setNameConstraints(byte[] bytes) throws java.io.IOException;
   public java.util.Collection getPathToNames();                       // по умолчанию:null
   public void setPathToNames(java.util.Collection names) throws java.io.IOException;
   public java.util.Set getPolicy();                                   // по умолчанию:null
   public void setPolicy(java.util.Set certPolicySet) throws java.io.IOException;
   public java.util.Date getPrivateKeyValid();                         // по умолчанию:null
   public void setPrivateKeyValid(java.util.Date privateKeyValid);
   public java.math.BigInteger getSerialNumber();                      // по умолчанию:null
   public void setSerialNumber(java.math.BigInteger serial);
   public java.util.Collection getSubjectAlternativeNames();           // по умолчанию:null
   public void setSubjectAlternativeNames(java.util.Collection names) throws java.io.IOException;
   public byte[] getSubjectAsBytes() throws java.io.IOException;       // по умолчанию:null
   public String getSubjectAsString();                                 // по умолчанию:null
   public byte[] getSubjectKeyIdentifier();                            // по умолчанию:null
   public void setSubjectKeyIdentifier(byte[] subjectKeyID);
   public java.security.PublicKey getSubjectPublicKey();               // по умолчанию:null
```

```
    public void setSubjectPublicKey(java.security.PublicKey key);
    public void setSubjectPublicKey(byte[] key) throws java.io.IOException;
    public String getSubjectPublicKeyAlgID();                          // по умолчанию:null
    public void setSubjectPublicKeyAlgID(String oid) throws java.io.IOException;
// Открытые методы экземпляра
    public void addPathToName(int type, String name) throws java.io.IOException;
    public void addPathToName(int type, byte[] name) throws java.io.IOException;
    public void addSubjectAlternativeName(int type, String name) throws java.io.IOException;
    public void addSubjectAlternativeName(int type, byte[] name) throws java.io.IOException;
    public void setIssuer(String issuerDN) throws java.io.IOException;
    public void setIssuer(byte[] issuerDN) throws java.io.IOException;
    public void setSubject(byte[] subjectDN) throws java.io.IOException;
    public void setSubject(String subjectDN) throws java.io.IOException;
// Методы, реализующие CertSelector
    public Object clone();
    public boolean match(java.security.cert.Certificate cert);
// Открытые методы замещающие Object
    public String toString();
}
```

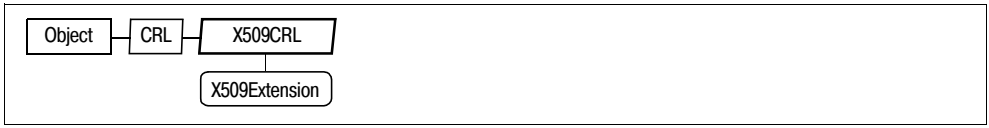
X509CRL

Java 1.2

java.security.cert

Этот класс представляет собой список аннулированных сертификатов X.509, который состоит из набора объектов X509CRLEntry. Методы этого класса предоставляют доступ ко всем сведениям о списке и требуют полного понимания стандарта X.509, описание которого выходит за рамки данной книги. Метод verify() применяется для проверки цифровой подписи списка, чтобы убедиться в том, что он пришел из указанного источника. Унаследованный метод isRevoked() применяется для выяснения того, был ли данный сертификат аннулирован. Если вам интересна дата аннулирования сертификата, то с помощью метода getRevokedCertificate() можно получить объект X509CRLEntry этого сертификата. Вызовите метод getThisUpdate() для получения даты издания этого списка. Метод getNextUpdate() применяется для выяснения того, был ли список замещен новой версией. Используйте метод getRevokedCertificates() для получения набора Set всех объектов X509CRLEntry из списка.

Получите объект X509CRL, создав CertificateFactory для типа сертификата «X.509» и вызвав метод generateCRL() для сборки списка X.509 из потока байтов. Наконец, приведите результат метода к типу X509CRL.




```
public abstract class X509CRL extends CRL implements X509Extension {
// Защищенные конструкторы
    protected X509CRL();
// Методы доступа к свойствам (по имени свойства)
    public abstract byte[] getEncoded() throws CRLException;
    public abstract java.security.Principal getIssuerDN();
1.4 public javax.security.auth.x500.X500Principal getIssuerX500Principal();
    public abstract java.util.Date getNextUpdate();
    public abstract java.util.Set getRevokedCertificates();
```

```
   public abstract String getSigAlgName();
   public abstract String getSigAlgOID();
   public abstract byte[] getSigAlgParams();
   public abstract byte[] getSignature();
   public abstract byte[] getTBSCertList() throws CRLException;
   public abstract java.util.Date getThisUpdate();
   public abstract int getVersion();
// Открытые методы экземпляра
   public abstract X509CRLEntry getRevokedCertificate(java.math.BigInteger serialNumber);
   public abstract void verify(java.security.PublicKey key) throws CRLException,
      java.security.NoSuchAlgorithmException, java.security.InvalidKeyException,
      java.security.NoSuchProviderException, java.security.SignatureException;
   public abstract void verify(java.security.PublicKey key, String sigProvider)
      throws CRLException,java.security.NoSuchAlgorithmException,
      java.security.InvalidKeyException, java.security.NoSuchProviderException,
      java.security.SignatureException;
// Открытые методы, замещающие Object
   public boolean equals(Object other);
   public int hashCode();
}
```

X509CRLEntry

Java 1.2

java.security.cert

Этот класс представляет собой единичный элемент X509CRL. Он содержит серийный номер и дату аннулирования сертификата.


```
public abstract class X509CRLEntry implements X509Extension {
// Открытые методы
   public X509CRLEntry();
// Методы доступа к свойствам (по имени свойства)
   public abstract byte[] getEncoded() throws CRLException;
   public abstract java.util.Date getRevocationDate();
   public abstract java.math.BigInteger getSerialNumber();
// Открытые методы экземпляра
   public abstract boolean hasExtensions();
// Открытые методы замещающие Object
   public boolean equals(Object other);
   public int hashCode();
   public abstract String toString();
}
```

Возвращается методами: X509CRL.getRevokedCertificate()

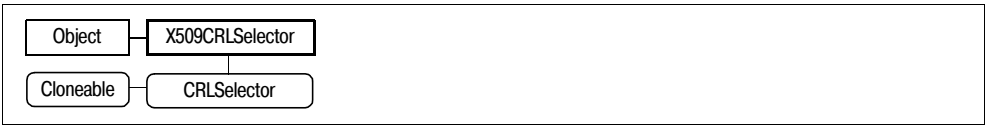
X509CRLSelector

Java 1.4

java.security.cert ***клонируемый***

Этот класс является реализацией CRLSelector для списков X.509. Его методы позволяют указать критерии, которые метод match() будет использовать при проверке объектов CRL. Метод addIssuerName() используется для указания допустимого издателя, а

метод setIssuerNames() – для указания коллекции Collection верных имен. Используйте метод setDateAndTime() для указания срока достоверности списка, а методы setMinCRLNumber() и setMaxCRLNumber() – для указания границ последовательности номеров списка. Если вы выбираете список для проверки аннулирования конкретного объекта X509Certificate, то передайте этот объект в метод setCertificateChecking(). В действительности данный метод не устанавливает возвращаемые объекты CRL, но может помочь CertStore оптимизировать поиск соответствующего списка.




```
public class X509CRLSelector implements CRLSelector {
// Открытые методы
   public X509CRLSelector();
// Методы доступа к свойствам (по имени свойства)
   public java.security.cert.X509Certificate getCertificateChecking();      // по умолчанию:null
   public void setCertificateChecking(java.security.cert.X509Certificate cert);
   public java.util.Date getDateAndTime();                                  // по умолчанию:null
   public void setDateAndTime(java.util.Date dateAndTime);
   public java.util.Collection getIssuerNames();                            // по умолчанию:null
   public void setIssuerNames(java.util.Collection names) throws java.io.IOException;
   public java.math.BigInteger getMaxCRL();                                 // по умолчанию:null
   public java.math.BigInteger getMinCRL();                                 // по умолчанию:null
// Открытые методы экземпляра
   public void addIssuerName(String name) throws java.io.IOException;
   public void addIssuerName(byte[] name) throws java.io.IOException;
   public void setMaxCRLNumber(java.math.BigInteger maxCRL);
   public void setMinCRLNumber(java.math.BigInteger minCRL);
// Методы, реализующие CRLSelector
   public Object clone();
   public boolean match(CRL crl);
// Открытые методы, замещающие Object
   public String toString();
}
```

X509Extension

Java 1.2

java.security.cert

Этот интерфейс определяет методы обработки набора расширений для сертификатов и списков X.509. Каждое расширение имеет имя, или OID (идентификатор объекта), которое идентифицирует тип расширения. Расширение может быть помечено как критическое или некритическое. Некритические расширения, чьи идентификаторы не распознаны, можно смело игнорировать. Тем не менее, если критическое расширение не распознано, объект Certificate или CRL должен быть отвергнут. Каждое расширение имеет массив байтов в качестве своего значения. Разумеется, интерпретация этих байтов зависит от идентификатора расширения. Специфичные расширения определены стандартом X.509 и родственными стандартами, а их детальное рассмотрение выходит за рамки данной книги.

```
public interface X509Extension {
// Открытые методы экземпляра
```

```
    public abstract java.util.Set getCriticalExtensionOIDs();
    public abstract byte[] getExtensionValue(String oid);
    public abstract java.util.Set getNonCriticalExtensionOIDs();
    public abstract boolean hasUnsupportedCriticalExtension();
}
```

Реализации: java.security.cert.X509Certificate, X509CRL, X509CRLEntry

Пакет java.security.interfaces Java 1.1

Как следует из имени пакета java.security.interfaces, он содержит только интерфейсы. Эти интерфейсы определяют методы, которые предоставляют алгоритмо-зависимую информацию об открытых и закрытых ключах DSA и RSA (например, значения ключей и значения параметров инициализации). Например, если вы используете алгоритм RSA и работаете с объектом java.security.PublicKey, то вы можете привести этот PublicKey к объекту RSAPublicKey и использовать специфичные для RSA методы, определенные в RSAPublicKey, для получения значения ключа.

Пакет java.security.interfaces был представлен в Java 1.1. В Java 1.2 пакет java.security.spec является более предпочтительным способом получения алгоритмо-зависимой информации о ключах и параметрах алгоритма. Тем не менее этот пакет остается полезным в Java 1.2 – он применяется для идентификации типов данных объектов PublicKey и PrivateKey.

Интерфейсы этого пакета обычно интересны только программистам, которые сами реализуют поставщика средств безопасности или алгоритм шифрования. Обычно использование этого пакета требует хорошего знания математики, лежащей в основе шифрования DSA и RSA с открытым ключом.

Интерфейсы

```
public interface DSAKey;
public interface DSAKeyPairGenerator;
public interface DSAParams;
public interface DSAPrivateKey extends DSAKey, java.security.PrivateKey;
public interface DSAPublicKey extends DSAKey, java.security.PublicKey;
public interface RSAKey;
public interface RSAMultiPrimePrivateCrtKey extends RSAPrivateKey;
public interface RSAPrivateCrtKey extends RSAPrivateKey;
public interface RSAPrivateKey extends java.security.PrivateKey, RSAKey;
public interface RSAPublicKey extends java.security.PublicKey, RSAKey;
```

DSAKey Java 1.1

java.security.interfaces

Этот интерфейс определяет метод, который должен быть реализован открытым и закрытым ключами DSA.

```
public interface DSAKey {
// Открытые методы экземпляра
    public abstract DSAParams getParams();
}
```

Реализации: DSAPrivateKey, DSAPublicKey

DSAKeyPairGenerator Java 1.1

java.security.interfaces

Этот интерфейс определяет алгоритмо-зависимые методы инициализации KeyPairGenerator для ключей DSA. Для генерации пары ключей DSA применяйте статический метод-фабрику getInstance() из java.security.KeyPairGenerator. В качестве имени алгоритма укажите «DSA». Если вы хотите выполнить DSA-зависимую инициализацию, то приведите возвращаемое значение KeyPairGenerator к DSAKeyPairGenerator и вызовите один из методов initialize(), определенных в этом интерфейсе. И наконец, сгенерируйте ключи, вызвав метод generateKeyPair() объекта KeyPairGenerator.

```
public interface DSAKeyPairGenerator {
// Открытые методы экземпляра
   public abstract void initialize(DSAParams params, java.security.SecureRandom random)
        throws java.security.InvalidParameterException;
   public abstract void initialize(int modlen, boolean genParams,
        java.security.SecureRandom random) throws java.security.InvalidParameterException;
}
```

DSAParams Java 1.1

java.security.interfaces

Данный интерфейс определяет методы для получения параметров g, p и q алгоритма DSA. Эти методы полезны, если вы хотите самостоятельно выполнять криптографические вычисления. Использование предоставленных методов требует глубокого знания математики, лежащей в основе шифрования DSA с открытым ключом.

```
public interface DSAParams {
// Открытые методы экземпляра
   public abstract java.math.BigInteger getG();
   public abstract java.math.BigInteger getP();
   public abstract java.math.BigInteger getQ();
}
```

Реализации: java.security.spec.DSAParameterSpec

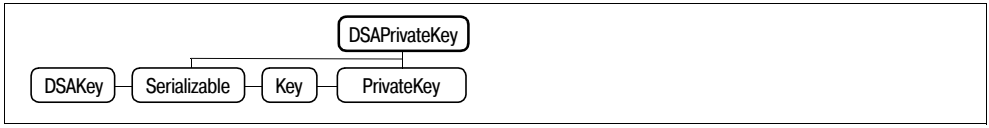
Передается методам: DSAKeyPairGenerator.initialize()

Возвращается методами: DSAKey.getParams()

DSAPrivateKey Java 1.1

java.security.interfaces ***сериализуемый***

Этот интерфейс представляет закрытый ключ DSA и обеспечивает прямой доступ к содержащемуся в нем значению ключа. Если вы работаете с закрытым ключом, который является ключом DSA, то вы можете привести PrivateKey к DSAPrivateKey.




```
public interface DSAPrivateKey extends DSAKey, java.security.PrivateKey {
// Открытые константы
```

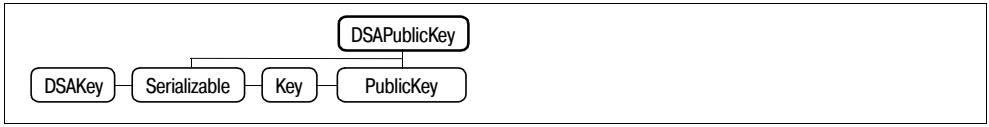
```
1.2 public static final long serialVersionUID;                    // =7776497482533790279
// Открытые методы экземпляра
   public abstract java.math.BigInteger getX();
}
```

DSAPublicKey

Java 1.1

java.security.interfaces ***сериализуемый***

Этот интерфейс представляет открытый ключ DSA и обеспечивает прямой доступ к содержащемуся в нем значению ключа. Если вы работаете с открытым ключом, который является ключом DSA, то вы можете привести PublicKey к DSAPublicKey.




```
public interface DSAPublicKey extends DSAKey, java.security.PublicKey {
// Открытые константы
1.2 public static final long serialVersionUID;                    // =1234526332779022332
// Открытые методы экземпляра
   public abstract java.math.BigInteger getY();
}
```

RSAKey

Java 1.3

java.security.interfaces

Это родительский интерфейс для RSAPublicKey и RSAPrivateKey; он определяет метод, разделяемый двумя классами. До Java 1.3 метод getModulus() был определен отдельно для RSAPublicKey и RSAPrivateKey.

```
public interface RSAKey {
// Открытые методы экземпляра
   public abstract java.math.BigInteger getModulus();
}
```

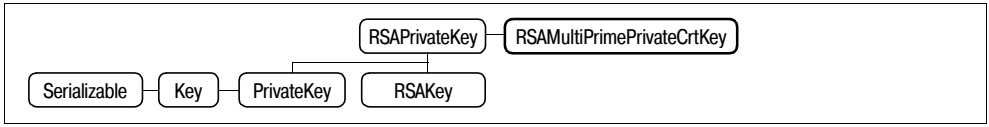
Реализации: RSAPrivateKey, RSAPublicKey

RSAMultiPrimePrivateCrtKey

Java 1.4

java.security.interfaces ***сериализуемый***

Этот интерфейс расширяет RSAPrivateKey и позволяет разложить закрытый ключ на ряд чисел, которые применялись для его создания. Этот интерфейс подобен RSAPrivateCrtKey за исключением того, что он используется для представления закрытых ключей RSA, основанных на более чем двух простых множителях. Он реализует дополнительный метод getOtherPrimeInfo(), который возвращает информацию об этих числах.



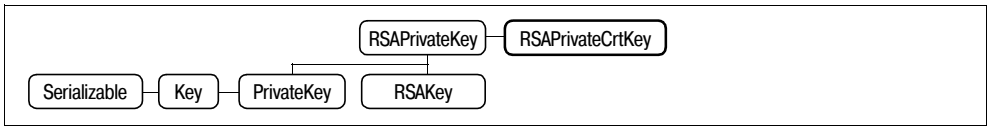

```
public interface RSAMultiPrimePrivateCrtKey extends RSAPrivateKey {
// Методы доступа к свойствам (по имени свойства)
   public abstract java.math.BigInteger getCrtCoefficient();
   public abstract java.security.spec.RSAOtherPrimeInfo[] getOtherPrimeInfo();
   public abstract java.math.BigInteger getPrimeExponentP();
   public abstract java.math.BigInteger getPrimeExponentQ();
   public abstract java.math.BigInteger getPrimeP();
   public abstract java.math.BigInteger getPrimeQ();
   public abstract java.math.BigInteger getPublicExponent();
}
```

RSAPrivateCrtKey

Java 1.2

java.security.interfaces ***сериализуемый***

Этот интерфейс расширяет RSAPrivateKey и позволяет разложить значение закрытого ключа на его составляющие (на основе теоремы Chinese remainder). Этот интерфейс будет полезен, если вы планируете реализовать ваш собственный алгоритм шифрования. Для использования этого интерфейса необходимо глубоко понимать математику, лежащую в основе шифрования RSA с закрытым ключом. Если дан объект java.security.PrivateKey, то вы можете использовать оператор instanceof для выяснения возможности приведения его к RSAPrivateCrtKey.



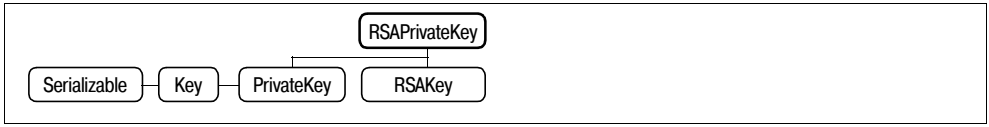

```
public interface RSAPrivateCrtKey extends RSAPrivateKey {
// Методы доступа к свойствам (по имени свойства)
   public abstract java.math.BigInteger getCrtCoefficient();
   public abstract java.math.BigInteger getPrimeExponentP();
   public abstract java.math.BigInteger getPrimeExponentQ();
   public abstract java.math.BigInteger getPrimeP();
   public abstract java.math.BigInteger getPrimeQ();
   public abstract java.math.BigInteger getPublicExponent();
}
```

RSAPrivateKey

Java 1.2

java.security.interfaces ***сериализуемый***

Этот интерфейс представляет закрытый ключ RSA и обеспечивает прямой доступ к содержащемуся в нем значению ключа. Если вы работаете с закрытым ключом, который является ключом RSA, то вы можете привести PrivateKey к RSAPrivateKey.




```
public interface RSAPrivateKey extends java.security.PrivateKey, RSAKey {
// Открытые методы экземпляра
   public abstract java.math.BigInteger getPrivateExponent();
}
```

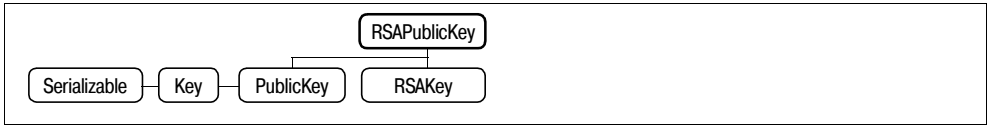
Реализации: RSAMultiPrimePrivateCrtKey, RSAPrivateCrtKey

RSAPublicKey

Java 1.2

java.security.interfaces ***сериализуемый***

Этот интерфейс представляет открытый ключ RSA и обеспечивает прямой доступ к содержащемуся в нем значению ключа. Если вы работаете с открытым ключом, который является ключом RSA, то вы можете привести PublicKey к RSAPublicKey.




```
public interface RSAPublicKey extends java.security.PublicKey, RSAKey {
// Открытые методы экземпляра
   public abstract java.math.BigInteger getPublicExponent();
}
```

Пакет java.security.spec

Java 1.2

Пакет java.security.spec содержит классы, которые определяют прозрачное представление открытого и закрытого ключей DSA и RSA и их представление в форматах X.509 и PKCS#8. Кроме того, он определяет прозрачное представление параметров алгоритма DSA. Классы из этого пакета используются вместе с java.security.KeyFactory и java.security.AlgorithmParameters для конвертации непрозрачных объектов Key и AlgorithmParameters в прозрачное представление и обратного преобразования.

Этот пакет применяется нечасто. Для работы с ним необходимо понимать математику, лежащую в основе шифрования DSA и RSA с открытым ключом, и стандарты кодирования, которые определяют правила преобразования ключей в потоки байтов.

Интерфейсы

```
public interface AlgorithmParameterSpec;
public interface KeySpec;
```

Классы

```
public class DSAParameterSpec implements AlgorithmParameterSpec,
   java.security.interfaces.DSAParams;
public class DSAPrivateKeySpec implements KeySpec;
public class DSAPublicKeySpec implements KeySpec;
```

```
public abstract class EncodedKeySpec implements KeySpec;
   └ public class PKCS8EncodedKeySpec extends EncodedKeySpec;
   └ public class X509EncodedKeySpec extends EncodedKeySpec;
public class PSSParameterSpec implements AlgorithmParameterSpec;
public class RSAKeyGenParameterSpec implements AlgorithmParameterSpec;
public class RSAOtherPrimeInfo;
public class RSAPrivateKeySpec implements KeySpec;
   └ public class RSAMultiPrimePrivateCrtKeySpec extends RSAPrivateKeySpec;
   └ public class RSAPrivateCrtKeySpec extends RSAPrivateKeySpec;
public class RSAPublicKeySpec implements KeySpec;
```

Исключения

```
public class InvalidKeySpecException extends java.security.GeneralSecurityException;
public class InvalidParameterSpecException extends java.security.GeneralSecurityException;
```

AlgorithmParameterSpec Java 1.2

java.security.spec

Этот интерфейс не определяет методов; он помечает классы, которые определяют прозрачное представление параметров шифрования. Объект `AlgorithmParameterSpec` можно применять для инициализации непрозрачного объекта `java.security.AlgorithmParameters`.

```
public interface AlgorithmParameterSpec {
}
```

Реализации: `DSAParameterSpec`, `PSSParameterSpec`, `RSAKeyGenParameterSpec`, `javax.crypto.spec.DHGenParameterSpec`, `javax.crypto.spec.DHParameterSpec`, `javax.crypto.spec.IvParameterSpec`, `javax.crypto.spec.PBEParameterSpec`, `javax.crypto.spec.RC2ParameterSpec`, `javax.crypto.spec.RC5ParameterSpec`

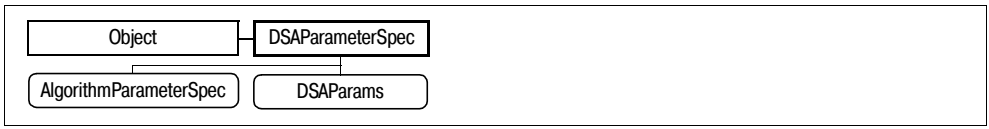
Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: `java.security.AlgorithmParameters.getParameterSpec()`, `java.security.AlgorithmParametersSpi.engineGetParameterSpec()`

DSAParameterSpec Java 1.2

java.security.spec

Этот класс представляет параметры алгоритма, используемые при шифровании DSA с открытым ключом.




```
public class DSAParameterSpec implements AlgorithmParameterSpec, java.security.interfaces.DSAParams {
// Открытые методы
   public DSAParameterSpec(java.math.BigInteger p, java.math.BigInteger q, java.math.BigInteger g);
// Методы, реализующие DSAParams
   public java.math.BigInteger getG();
   public java.math.BigInteger getP();
   public java.math.BigInteger getQ();
}
```

DSAPrivateKeySpec Java 1.2

java.security.spec

Этот класс является прозрачным представлением закрытого ключа DSA.


```
public class DSAPrivateKeySpec implements KeySpec {
// Открытые методы
   public DSAPrivateKeySpec(java.math.BigInteger x, java.math.BigInteger p,
                            java.math.BigInteger q, java.math.BigInteger g);
// Открытые методы экземпляра
   public java.math.BigInteger getG();
   public java.math.BigInteger getP();
   public java.math.BigInteger getQ();
   public java.math.BigInteger getX();
}
```

DSAPublicKeySpec Java 1.2

java.security.spec

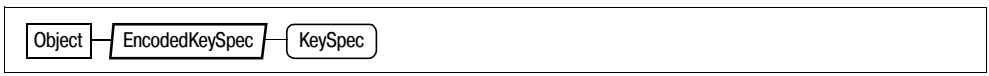
Этот класс является прозрачным представлением открытого ключа DSA.


```
public class DSAPublicKeySpec implements KeySpec {
// Открытые методы
   public DSAPublicKeySpec(java.math.BigInteger y, java.math.BigInteger p,
                           java.math.BigInteger q, java.math.BigInteger g);
// Открытые методы экземпляра
   public java.math.BigInteger getG();
   public java.math.BigInteger getP();
   public java.math.BigInteger getQ();
   public java.math.BigInteger getY();
}
```

EncodedKeySpec Java 1.2

java.security.spec

Этот абстрактный класс представляет открытый или закрытый ключ в закодированном формате. Он служит в качестве родительского класса для классов, предназначенных для кодирования.




```
public abstract class EncodedKeySpec implements KeySpec {
// Открытые методы
   public EncodedKeySpec(byte[] encodedKey);
```

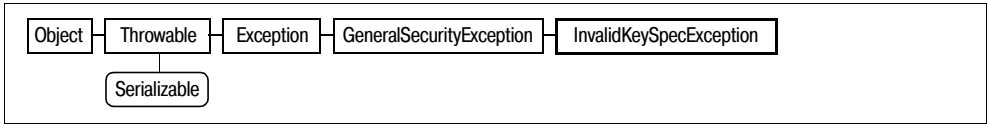
```
// Открытые методы экземпляра
   public byte[] getEncoded();
   public abstract String getFormat();
}
```

Подклассы: PKCS8EncodedKeySpec, X509EncodedKeySpec

InvalidKeySpecException Java 1.2

java.security.spec ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках с KeySpec.



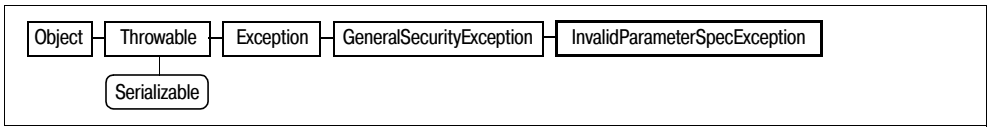

```
public class InvalidKeySpecException extends java.security.GeneralSecurityException {
// Открытые методы
   public InvalidKeySpecException();
   public InvalidKeySpecException(String msg);
}
```

Генерируется методами: java.security.KeyFactory.{generatePrivate(), generatePublic(), getKeySpec()}, java.security.KeyFactorySpi.{engineGeneratePrivate(), engineGeneratePublic(), engineGetKeySpec()}, javax.crypto.EncryptedPrivateKeyInfo.getKeySpec(), javax.crypto.SecretKeyFactory.{generateSecret(), getKeySpec()}, avax.crypto.SecretKeyFactorySpi.{engineGenerateSecret(), engineGetKeySpec()}

InvalidParameterSpecException Java 1.2

java.security.spec ***сериализуемое, проверяемое***

Это исключение сигнализирует о неполадках с AlgorithmParameterSpec.




```
public class InvalidParameterSpecException extends java.security.GeneralSecurityException {
// Открытые методы
   public InvalidParameterSpecException();
   public InvalidParameterSpecException(String msg);
}
```

Генерируется методами: java.security.AlgorithmParameters.{getParameterSpec(), init()}, java.security.AlgorithmParametersSpi.{engineGetParameterSpec(), engineInit()}

KeySpec Java 1.2

java.security.spec

Этот интерфейс не определяет методов; он помечает классы, которые определяют прозрачное представление ключа шифрования. Используйте объект `java.security.KeyFactory` для конвертации `KeySpec` в непрозрачный объект `java.security.Key` и обратного преобразования.

```
public interface KeySpec {
}
```

Реализации: `DSAPrivateKeySpec`, `DSAPublicKeySpec`, `EncodedKeySpec`, `RSAPrivateKeySpec`, `RSAPublicKeySpec`, `javax.crypto.spec.DESedeKeySpec`, `javax.crypto.spec.DESKeySpec`, `javax.crypto.spec.DHPrivateKeySpec`, `javax.crypto.spec.DHPublicKeySpec`, `javax.crypto.spec.PBEKeySpec`, `javax.crypto.spec.SecretKeySpec`

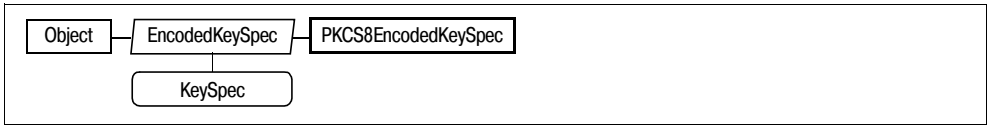
Передается методами: `java.security.KeyFactory.{generatePrivate(), generatePublic()}`, `java.security.KeyFactorySpi.{engineGeneratePrivate(), engineGeneratePublic()}`, `javax.crypto.SecretKeyFactory.generateSecret()`, `javax.crypto.SecretKeyFactorySpi.engineGenerateSecret()`

Возвращается методами: `java.security.KeyFactory.getKeySpec()`, `java.security.KeyFactorySpi.engineGetKeySpec()`, `javax.crypto.SecretKeyFactory.getKeySpec()`, `javax.crypto.SecretKeyFactorySpi.engineGetKeySpec()`

PKCS8EncodedKeySpec Java 1.2

java.security.spec

Этот класс представляет закрытый ключ, закодированный в соответствии со стандартом PKCS#8.



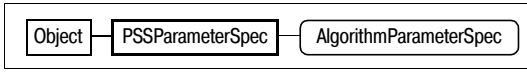

```
public class PKCS8EncodedKeySpec extends EncodedKeySpec {
// Открытые методы
   public PKCS8EncodedKeySpec(byte[] encodedKey);
// Открытые методы, замещающие EncodedKeySpec
   public byte[] getEncoded();
   public final String getFormat();
}
```

Возвращается методами: `javax.crypto.EncryptedPrivateKeyInfo.getKeySpec()`

PSSParameterSpec Java 1.4

java.security.spec

Этот класс представляет параметры алгоритма, используемые в схеме кодирования RSA PSS, определенной версией 2.1 стандарта RSA PKCS#1.




```
public class PSSParameterSpec implements AlgorithmParameterSpec {
// Открытые методы
   public PSSParameterSpec(int saltLen);
// Открытые методы экземпляра
   public int getSaltLength();
}
```

RSAKeyGenParameterSpec Java 1.3

java.security.spec

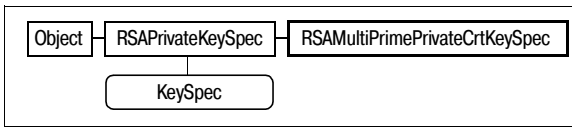
Этот класс представляет параметры, которые используются при генерации пар открытых/закрытых ключей для шифрования RSA.


```
public class RSAKeyGenParameterSpec implements AlgorithmParameterSpec {
// Открытые методы
   public RSAKeyGenParameterSpec(int keysize, java.math.BigInteger publicExponent);
// Открытые константы
   public static final java.math.BigInteger F0;
   public static final java.math.BigInteger F4;
// Открытые методы экземпляра
   public int getKeysize();
   public java.math.BigInteger getPublicExponent();
}
```

RSAMultiPrimePrivateCrtKeySpec Java 1.4

java.security.spec

Этот класс является прозрачным представлением закрытого ключа RSA с несколькими простыми числами. Он очень похож на RSAPrivateCrtKeySpec, но добавляет еще один метод для получения информации о других простых числах, ассоциированных с ключом.




```
public class RSAMultiPrimePrivateCrtKeySpec extends RSAPrivateKeySpec {
// Открытые методы
   public RSAMultiPrimePrivateCrtKeySpec(java.math.BigInteger modulus, java.math.BigInteger
        publicExponent, java.math.BigInteger privateExponent,
        java.math.BigInteger primeP, java.math.BigInteger primeQ,
        java.math.BigInteger primeExponentP, java.math.BigInteger primeExponentQ,
        java.math.BigInteger crtCoefficient, RSAOtherPrimeInfo[] otherPrimeInfo);
// Методы доступа к свойствам (по имени свойства)
   public java.math.BigInteger getCrtCoefficient();
   public RSAOtherPrimeInfo[] getOtherPrimeInfo();
```

```
   public java.math.BigInteger getPrimeExponentP();
   public java.math.BigInteger getPrimeExponentQ();
   public java.math.BigInteger getPrimeP();
   public java.math.BigInteger getPrimeQ();
   public java.math.BigInteger getPublicExponent();
}
```

RSAOtherPrimeInfo Java 1.4

java.security.spec

Этот класс представляет собой триплет (простое число, экспонента, коэффициент), который составляет структуру «OtherPrimeInfo», используемую закрытыми ключами RSA с несколькими простыми числами, как определено в версии 2.1 стандарта PKCS#1.

```
public class RSAOtherPrimeInfo {
// Открытые методы
   public RSAOtherPrimeInfo(java.math.BigInteger prime, java.math.BigInteger primeExponent,
                            java.math.BigInteger crtCoefficient);
// Открытые методы экземпляра
   public final java.math.BigInteger getCrtCoefficient();
   public final java.math.BigInteger getExponent();
   public final java.math.BigInteger getPrime();
}
```

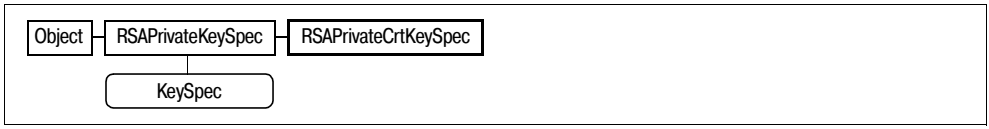
Передается методам: `RSAMultiPrimePrivateCrtKeySpec.RSAMultiPrimePrivateCrtKeySpec()`

Возвращается методами: `java.security.interfaces.RSAMultiPrimePrivateCrtKey.getOtherPrimeInfo()`, `RSAMultiPrimePrivateCrtKeySpec.getOtherPrimeInfo()`

RSAPrivateCrtKeySpec Java 1.2

java.security.spec

Этот класс является прозрачным представлением закрытого ключа RSA. Для удобства в него включены значения теоремы Chinese remainder, ассоциированные с ключом.

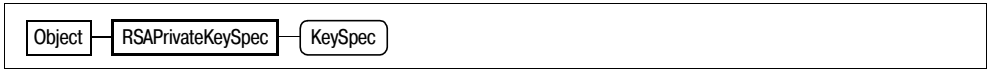



```
public class RSAPrivateCrtKeySpec extends RSAPrivateKeySpec {
// Открытые методы
   public RSAPrivateCrtKeySpec(java.math.BigInteger modulus, java.math.BigInteger publicExponent,
                               java.math.BigInteger privateExponent, java.math.BigInteger primeP,
                               java.math.BigInteger primeQ, java.math.BigInteger primeExponentP,
                               java.math.BigInteger primeExponentQ, java.math.BigInteger crtCoefficient);
// Методы доступа к свойствам (по имени свойства)
   public java.math.BigInteger getCrtCoefficient();
   public java.math.BigInteger getPrimeExponentP();
   public java.math.BigInteger getPrimeExponentQ();
   public java.math.BigInteger getPrimeP();
   public java.math.BigInteger getPrimeQ();
   public java.math.BigInteger getPublicExponent();
}
```

RSAPrivateKeySpec Java 1.2

java.security.spec

Этот класс является прозрачным представлением закрытого ключа RSA.




```
public class RSAPrivateKeySpec implements KeySpec {
// Открытые методы
   public RSAPrivateKeySpec(java.math.BigInteger modulus, java.math.BigInteger privateExponent);
// Открытые методы экземпляра
   public java.math.BigInteger getModulus();
   public java.math.BigInteger getPrivateExponent();
}
```

Подклассы: RSAMultiPrimePrivateCrtKeySpec, RSAPrivateCrtKeySpec

RSAPublicKeySpec Java 1.2

java.security.spec

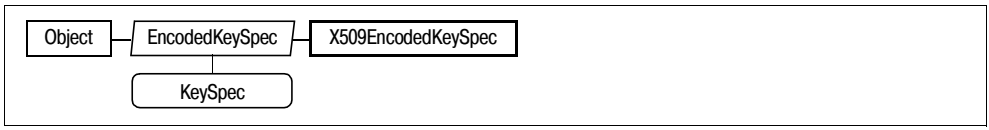
Этот класс является прозрачным представлением открытого ключа RSA.


```
public class RSAPublicKeySpec implements KeySpec {
// Открытые методы
   public RSAPublicKeySpec(java.math.BigInteger modulus, java.math.BigInteger publicExponent);
// Открытые методы экземпляра
   public java.math.BigInteger getModulus();
   public java.math.BigInteger getPublicExponent();
}
```

X509EncodedKeySpec Java 1.2

java.security.spec

Этот класс представляет открытый или закрытый ключ, закодированный в соответствии со стандартом X.509.




```
public class X509EncodedKeySpec extends EncodedKeySpec {
// Открытые методы
   public X509EncodedKeySpec(byte[] encodedKey);
// Открытые методы, замещающие EncodedKeySpec
   public byte[] getEncoded();
   public final String getFormat();
}
```

Глава 16

java.text

Пакет java.text

Java 1.1

Пакет java.text содержит классы и интерфейсы, которые используются при написании интернациональнных программ с корректной поддержкой локальных особенностей, таких как формат даты и времени и сортировка строк.

Класс NumberFormat форматирует числа, class formats numbers, денежные величины и проценты в соответствии с указанным регионом (locale) или регионом по умолчанию. DateFormat форматирует дату и время способом, специфичным для региона. Конкретные подклассы этих классов DecimalFormat и SimpleDateFormat могут быть использованы для настройки формата чисел, дат и времени. MessageFormat позволяет замещать динамические значения статическими строками, включая форматированные числа и даты. ChoiceFormat форматирует число, используя нумерованное множество строковых значений. Общее описание форматирования и разбора строк с помощью этих классов вы можете найти в разделе, посвященном родительскому классу Format. Collator сравнивает строки в соответствии со способом сортировки, принятым для данного региона. BreakIterator просматривает текст в поисках слова, строки и границ фразы, пользуясь правилами, принятыми для региона. Класс Bidi из Java 1.4 реализует «двунаправленный» алгоритм Unicode для работы с такими языками, как арабский и иврит, в которых текст пишется справа налево, а числа – слева направо.

Интерфейсы

```
public interface AttributedCharacterIterator extends CharacterIterator;
public interface CharacterIterator extends Cloneable;
```

Классы

```
public class Annotation;
public static class AttributedCharacterIterator.Attribute implements Serializable;
   └ public static class Format.Field extends AttributedCharacterIterator.Attribute;
      └ public static class DateFormat.Field extends Format.Field;
      └ public static class MessageFormat.Field extends Format.Field;
      └ public static class NumberFormat.Field extends Format.Field;
public class AttributedString;
public final class Bidi;
public abstract class BreakIterator implements Cloneable;
public class CharSet.Enumeration implements java.util.Enumeration;
```

```
public final class CollationElementIterator;
public final class CollationKey implements Comparable;
public abstract class Collator implements Cloneable, java.util.Comparator;
   └ public class RuleBasedCollator extends Collator;
public class DateFormatSymbols implements Cloneable, Serializable;
public final class DecimalFormatSymbols implements Cloneable, Serializable;
public class FieldPosition;
public abstract class Format implements Cloneable, Serializable;
   └ public abstract class DateFormat extends Format;
      └ public class SimpleDateFormat extends DateFormat;
   └ public class MessageFormat extends Format;
   └ public abstract class NumberFormat extends Format;
      └ public class ChoiceFormat extends NumberFormat;
      └ public class DecimalFormat extends NumberFormat;
public class ParsePosition;
public final class StringCharacterIterator implements CharacterIterator;
```

Исключения

```
public class ParseException extends Exception;
```

Annotation Java 1.2

java.text

Этот класс является оберткой (wrapper) для значения текстового атрибута, который представляет собой аннотацию. Аннотации отличаются от других типов текстовых атрибутов двумя особенностями. Во-первых, аннотации привязаны к тексту, в котором они используются, поэтому изменение текста делает неактуальным или искажает аннотацию. Во-вторых, аннотации не могут сливаться со смежными аннотациями, даже если они имеют одинаковое значение. Помещение значения аннотации в обертку Annotation дает возможность указать на эти специальные свойства. Обратите внимание, что два из ключей атрибутов, определенных в AttributedCharacterIterator.Attribute, – READING и INPUT_METHOD_SEGMENT – обязательно должны использоваться с объектами Annotation.

```
public class Annotation {
// Открытые конструкторы
   public Annotation(Object value);
// Открытые методы экземпляра
   public Object getValue();
// Открытые методы, замещающие Object
   public String toString();
}
```

AttributedCharacterIterator Java 1.2

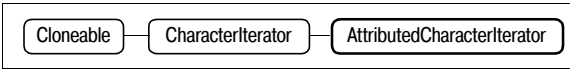
java.text ***клонируемый***

Этот интерфейс расширяет CharacterIterator для работы с текстом, который каким-либо образом размечен с помощью атрибутов. Он определяет внутренний класс AttributedCharacterIterator.Attribute, который представляет ключи атрибутов. AttributedCharacterIterator определяет методы для запроса ключей атрибутов, значений и перемещения по тексту в пределах диапазона. getAllAttributeKeys() возвращает значение типа Set, содержащее ключи всех атрибутов, встречающихся в тексте. getAttributes() возвращает значение типа Map, которое содержит ключи атрибутов и их

значения, относящиеся к текущему символу. getAttribute() возвращает значение, ассоциированное с указанным ключом атрибута для текущего символа.

getRunStart() и getRunLimit() возвращают индекс первого и последнего символов в обрабатываемом диапазоне. Диапазон – это строка смежных символов, для которых атрибут имеет одно и то же значение или неопределен (то есть имеет значение null). Диапазон также может быть определен для множества атрибутов, в этом случае он является множеством смежных символов, для которых все атрибуты во множестве имеют постоянное значение (включая null). Программы, которые обрабатывают или показывают текст с атрибутами, обычно работают с одним диапазоном в один и тот же момент времени. Версии getRunStart() и getRunLimit() без аргументов возвращают начало и конец диапазона, который включает текущий символ и все атрибуты этого символа. Другие версии этих методов возвращают начало и конец диапазона указанного атрибута или множества атрибутов, включающих текущий символ.

Класс AttributedString предоставляет простой способ определить короткие строки текста с атрибутами и получить AttributedCharacterIterator для них. Большинство приложений, которые обрабатывают текст с атрибутами, работают с текстом, получаемым из специфичных источников данных и сохраняемым в специфичном формате, поэтому они нуждаются в определении собственных специфичных реализаций AttributedCharacterIterator.




```
public interface AttributedCharacterIterator extends CharacterIterator {
// Внутренние классы
   public static class Attribute implements Serializable;
// Открытые методы экземпляра
   public abstract java.util.Set getAllAttributeKeys();
   public abstract Object getAttribute(AttributedCharacterIterator.Attribute attribute);
   public abstract java.util.Map getAttributes();
   public abstract int getRunLimit();
   public abstract int getRunLimit(java.util.Set attributes);
   public abstract int getRunLimit(AttributedCharacterIterator.Attribute attribute);
   public abstract int getRunStart();
   public abstract int getRunStart(AttributedCharacterIterator.Attribute attribute);
   public abstract int getRunStart(java.util.Set attributes);
}
```

Передается методам: Методов слишком много, чтобы их перечислить

Возвращается методами: java.awt.event.InputMethodEvent.getText(), java.awt.im.InputMethodRequests.{cancelLatestCommittedText(), getCommittedText(), getSelectedText()}, AttributedString.getIterator(), DecimalFormat.formatToCharacterIterator(), Format.formatToCharacterIterator(), MessageFormat.formatToCharacterIterator(), SimpleDateFormat.formatToCharacterIterator()

AttributedCharacterIterator.Attribute Java 1.2

java.text *сериализуемый*

Этот класс определяет типы ключей атрибутов, используемые в AttributedCharacterIterator и AttributedString. Он определяет несколько постоянных ключей Attribute, которые обычно используются с многоязычным текстом и методами ввода. Ключ LANGUAGE представляет язык соответствующего текста. Значением этого ключа должен

быть объект Locale. Ключ READING представляет произвольную поясняющую информацию, ассоциированную с текстом. Значением должен быть объект Annotation. Ключ INPUT_METHOD_SEGMENT предоставляет возможность определить элементы текста (обычно слова), с которыми работают методы ввода. Значением этого атрибута должен быть объект Annotation, содержащий null. Другие классы могут наследовать этот класс и определять другие ключи атрибутов, полезные при других обстоятельствах или в других предметных областях. См. java.awt.font.TextAttribute в книге «Java Foundation Classes in a Nutshell» (O'Reilly).

```
public static class AttributedCharacterIterator.Attribute implements Serializable {
// Защищенные конструкторы
   protected Attribute(String name);
// Открытые константы
   public static final AttributedCharacterIterator.Attribute INPUT_METHOD_SEGMENT;
   public static final AttributedCharacterIterator.Attribute LANGUAGE;
   public static final AttributedCharacterIterator.Attribute READING;
// Открытые методы, замещающие Object
   public final boolean equals(Object obj);
   public final int hashCode();
   public String toString();
// Защищенные методы экземпляра
   protected String getName();
   protected Object readResolve() throws java.io.InvalidObjectException;
}
```

Подклассы: java.awt.font.TextAttribute, Format.Field

Передается методам: java.awt.im.InputMethodRequests.{cancelLatestCommittedText(), getCommittedText(), getSelectedText()}, AttributedCharacterIterator.{getAttribute(), getRunLimit(), getRunStart()}, AttributedString.{addAttribute(), AttributedString(), getIterator()}

Возвращается методами: java.awt.Font.getAvailableAttributes()

Экземпляры: AttributedCharacterIterator.Attribute.{INPUT_METHOD_SEGMENT, LANGUAGE, READING}

AttributedString Java 1.2

java.text

Этот класс представляет текст и ассоциированные с ним атрибуты. Объект AttributedString может быть определен на основе базового объекта AttributedCharacterIterator или базового объекта String. Дополнительные атрибуты могут быть указаны с помощью методов addAttribute() и addAttributes(). getIterator() возвращает AttributedCharacterIterator для AttributedString или для указанной части этой строки. Обратите внимание, что два из методов getIterator() принимают массив ключей Attribute в качестве аргумента. Эти методы возвращают AttributedCharacterIterator, который игнорирует все атрибуты, не присутствующие в указанном массиве. Если массив, передаваемый в качестве аргумента, равен null, возвращаемый итератор содержит все атрибуты.

```
public class AttributedString {
// Открытые конструкторы
   public AttributedString(String text);
   public AttributedString(AttributedCharacterIterator text);
   public AttributedString(String text, java.util.Map attributes);
   public AttributedString(AttributedCharacterIterator text, int beginIndex, int endIndex);
```

```
   public AttributedString(AttributedCharacterIterator text, int beginIndex,
                           int endIndex, AttributedCharacterIterator.Attribute[] attributes);
// Открытые методы экземпляра
   public void addAttribute(AttributedCharacterIterator.Attribute attribute, Object value);
   public void addAttribute(AttributedCharacterIterator.Attribute attribute, Object value,
                                                              int beginIndex, int endIndex);
   public void addAttributes(java.util.Map attributes, int beginIndex, int endIndex);
   public AttributedCharacterIterator getIterator();
   public AttributedCharacterIterator getIterator(AttributedCharacterIterator.Attribute[] attributes);
   public AttributedCharacterIterator getIterator(AttributedCharacterIterator.Attribute[]
          attributes, int beginIndex, int endIndex);
}
```

Bidi

Java 1.4

java.text

Класс `Bidi` реализует «Unicode Version 3.0 Bidirectional Algorithm» для работы с текстами на арабском языке и иврите, в которых буквы пишутся справа налево, а числа – слева направо. Он назван по первым четырем буквам слова «bidirectional». Полное описание двунаправленного текста и двунаправленного алгоритма выходит за рамки данной книги, однако здесь описан простейший вариант использования этого класса. Создайте объект `Bidi`, передав `AttributedCharacterIterator` или `String` и одну из констант `DIRECTION` в конструктор `Bidi()`, чтобы указать базовое направление текста. Или воспользуйтесь методом `createLineBidi()`, чтобы вернуть подстроку существующего объекта `Bidi` (это обычно делается при форматировании параграфа текста, чтобы подогнать отдельные строки).

Когда вы получили объект `Bidi`, воспользуйтесь методами `isLeftToRight()` и `isRightToLeft()`, чтобы определить, весь ли текст имеет одинаковое направление. Если оба этих метода возвращают `false` (это эквивалентно тому, что метод `isMixed()` возвращает `true`), то вы не можете рассматривать текст как однодиапазонный или однонаправленный. В этом случае вы должны разбить текст на два или более диапазонов однонаправленного текста. `getRunCount()` возвращает количество уникальных диапазонов текста. Для каждого нумерованного диапазона метод `getRunStart()` возвращает индекс первого символа в диапазоне, а `getRunLimit()` возвращает индекс первого символа после конца этого диапазона. `getRunLevel()` возвращает уровень текста, который является целым числом, представляющим направление и уровень вложенности текста. Нечетные уровни представляют текст, написанный слева направо, а четные уровни – текст, написанный справа налево. Уровень, деленный на два, определяет вложенный уровень текста. Например, текст, написанный слева направо, вложенный в текст, написанный справа налево, имеет уровень 2.

```
public final class Bidi {
// Открытые конструкторы
   public Bidi(AttributedCharacterIterator paragraph);
   public Bidi(String paragraph, int flags);
   public Bidi(char[] text, int textStart, byte[] embeddings, int embStart,
                           int paragraphLength, int flags);
// Открытые константы
   public static final int DIRECTION_DEFAULT_LEFT_TO_RIGHT;                    // =-2
   public static final int DIRECTION_DEFAULT_RIGHT_TO_LEFT;                    // =-1
   public static final int DIRECTION_LEFT_TO_RIGHT;                             // =0
   public static final int DIRECTION_RIGHT_TO_LEFT;                             // =1
// Открытые методы класса
```

```
   public static void reorderVisually(byte[] levels, int levelStart, Object[] objects,
                                      int objectStart, int count);
   public static boolean requiresBidi(char[] text, int start, int limit);
// Методы доступа к свойствам (по имени свойства)
   public int getBaseLevel();
   public boolean isLeftToRight();
   public int getLength();
   public boolean isMixed();
   public boolean isRightToLeft();
   public int getRunCount();
// Открытые методы экземпляра
   public boolean baseIsLeftToRight();
   public Bidi createLineBidi(int lineStart, int lineLimit);
   public int getLevelAt(int offset);
   public int getRunLevel(int run);
   public int getRunLimit(int run);
   public int getRunStart(int run);
// Открытые методы, замещающие Object
   public String toString();
}
```

Возвращается методами: `Bidi.createLineBidi()`

BreakIterator

Java 1.1

java.text ***клонируемый***

Этот класс определяет символ, слово, фразу и разрывы строк в блоке текста способом, не зависящим от региона и кодировки текста. Для класса `BreakIterator` нельзя создавать экземпляров, поскольку он является абстрактным. Вместо этого вы должны использовать один из методов `getCharacterInstance()`, `getWordInstance()`, `getSentenceInstance()` или `getLineInstance()`, чтобы вернуть экземпляр неабстрактного подкласса `BreakIterator`. Эти разнообразные методы-фабрики возвращают объект `BreakIterator`, который сконфигурирован, чтобы определять запрошенные типы границ, и локализован в соответствии с указанным регионом.

Как только вы получите соответствующий объект `BreakIterator`, воспользуйтесь методом `setText()`, чтобы указать текст, в котором необходимо определять границы. Для определения границ в объекте `String` в Java просто укажите эту строку. Чтобы определить границы в тексте, который использует какую-либо другую кодировку, вы должны указать объект `CharacterIterator` для этого текста, тогда объект `BreakIterator` сможет определять отдельные символы в тексте.

Имея текст, который необходимо искать, вы можете определить позицию символов, слов, фраз или разрывов строк с помощью методов `first()`, `last()`, `next()`, `previous()`, `current()` и `following()`, которые выполняют соответствующие функции. Обратите внимание, что эти методы возвращают не найденный текст, а только позицию соответствующего слова, фразы или разрыва строки.


```
public abstract class BreakIterator implements Cloneable {
// Защищенные конструкторы
   protected BreakIterator();
// Открытые константы
```

```
   public static final int DONE;                                          // =-1
// Открытые методы класса
   public static java.util.Locale[] getAvailableLocales();                // синхронизирован
   public static BreakIterator getCharacterInstance();
   public static BreakIterator getCharacterInstance(java.util.Locale where);
   public static BreakIterator getLineInstance();
   public static BreakIterator getLineInstance(java.util.Locale where);
   public static BreakIterator getSentenceInstance();
   public static BreakIterator getSentenceInstance(java.util.Locale where);
   public static BreakIterator getWordInstance();
   public static BreakIterator getWordInstance(java.util.Locale where);
// Открытые методы экземпляра
   public abstract int current();
   public abstract int first();
   public abstract int following(int offset);
   public abstract CharacterIterator getText();
1.2 public boolean isBoundary(int offset);
   public abstract int last();
   public abstract int next();
   public abstract int next(int n);
1.2 public int preceding(int offset);
   public abstract int previous();
   public abstract void setText(CharacterIterator newText);
   public void setText(String newText);
// Открытые методы, замещающие Object
   public Object clone();
}
```

Передается методам: `java.awt.font.LineBreakMeasurer.LineBreakMeasurer()`

Возвращается методами: `BreakIterator.{getCharacterInstance(), getLineInstance(), getSentenceInstance(), getWordInstance()`

CharacterIterator

Java 1.1

java.text ***клонируемый***

Этот интерфейс определяет API для перемещения по символам, которые образуют строку текста, вне зависимости от кодировки этого текста. Этот API необходим, поскольку количество байтов на символ различно для разных кодировок, а некоторые кодировки даже используют символы переменной ширины в одной и той же строке текста. В дополнение к обеспечению возможности перемещения, класс, который реализует интерфейс `CharacterIterator` для текста без `Unicode`, выполняет преобразование кодировки символов из их родной кодировки в стандартное для Java представление в виде символов `Unicode`.

`CharacterIterator` похож на `java.util.Enumeration`, однако он немного сложнее, чем этот интерфейс. Методы `first()` и `last()` возвращают первый и последний символы в тексте, а методы `next()` и `previous()` позволяют перемещаться вперед и назад по символам текста в цикле. Эти методы возвращают константу `DONE`, когда они выходят за первый или последний символ в тексте; проверка на эту константу может использоваться для окончания цикла. Интерфейс `CharacterIterator` также позволяет выполнять произвольный доступ к символам строки текста. Методы `getBeginIndex()` и `getEndIndex()` возвращают позиции символов для начала и конца строки, а метод `setIndex()` устанавливает текущую позицию. Метод `getIndex()` возвращает индекс текущей позиции, а `current()` возвращает символ в этой позиции.


```
public interface CharacterIterator extends Cloneable {
// Открытые константы
   public static final char DONE;                                          // = '\uFFFF'
// Открытые методы экземпляра
   public abstract Object clone();
   public abstract char current();
   public abstract char first();
   public abstract int getBeginIndex();
   public abstract int getEndIndex();
   public abstract int getIndex();
   public abstract char last();
   public abstract char next();
   public abstract char previous();
   public abstract char setIndex(int position);
}
```

Реализации: AttributedCharacterIterator, StringCharacterIterator, javax.swing.text.Segment

Передается методам: java.awt.Font.{canDisplayUpTo(), createGlyphVector(), getLineMetrics(), getStringBounds()}, java.awt.FontMetrics.{getLineMetrics(), getStringBounds()}, BreakIterator.setText(), CollationElementIterator.setText(), RuleBasedCollator.getCollationElementIterator()

Возвращается методами: BreakIterator.getText()

ChoiceFormat

Java 1.1

java.text ***клонируемый, сериализуемый***

Этот класс является подклассом класса Format, который конвертирует число в объект String, используя способ, напоминающий о выражении switch и перечислимых типах. Каждый объект ChoiceFormat имеет массив чисел двойной точности, известных как его *границы* (*limits*) и массив строк, известных как его *форматы* (*formats*). Когда метод format() вызывается для форматирования числа x, ChoiceFormat ищет индекс i, такой что

```
limits[i] <= x < limits[i+1]
```

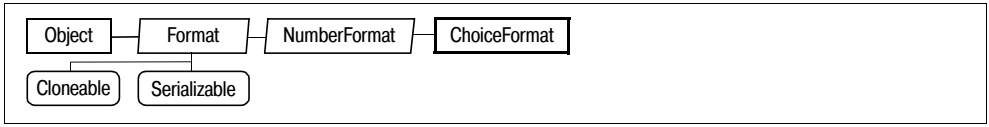
Если значение x меньше, чем первый элемент массива, то используется первый элемент, а если оно больше, чем последний элемент, то используется последний элемент массива. Как только индекс i был определен, он применяется как индекс для массива строк, а строка, соответствующая этому индексу, возвращается как результат метода format().

Объект ChoiceFormat может быть создан путем кодирования его границ и форматов в одну строку, известную как его паттерн. Типичный паттерн подобен паттерну, приведенному ниже, который применяется для возвращения единственного или множественного числа слова на основе числового значения, переданного в метод format():

```
ChoiceFormat cf = new ChoiceFormat(«0#errors|1#error|2#errors«);
```

Объект ChoiceFormat, созданный таким способом, возвращает строку «errors», когда он форматирует число 0 или любое число, большее или равное 2. Он возвращает

«error», когда форматирует число 1. В показанном здесь синтаксисе знак «решетка» (#) применяется для отделения числа от строки, соответствующей этому варианту, а вертикальная черта (|) служит для разделения вариантов. Вы можете использовать метод applyPattern(), чтобы изменить паттерн, используемый объектом ChoiceFormat, или задействовать метод toPattern(), чтобы запросить паттерн, который он использует в данный момент.




```
public class ChoiceFormat extends NumberFormat {
// Открытые конструкторы
   public ChoiceFormat(String newPattern);
   public ChoiceFormat(double[] limits, String[] formats);
// Открытые методы класса
   public static final double nextDouble(double d);
   public static double nextDouble(double d, boolean positive);
   public static final double previousDouble(double d);
// Открытые методы экземпляра
   public void applyPattern(String newPattern);
   public Object[] getFormats();
   public double[] getLimits();
   public void setChoices(double[] limits, String[] formats);
   public String toPattern();
// Открытые методы, замещающие NumberFormat
   public Object clone();
   public boolean equals(Object obj);
   public StringBuffer format(long number, StringBuffer toAppendTo, FieldPosition status);
   public StringBuffer format(double number, StringBuffer toAppendTo, FieldPosition status);
   public int hashCode();
   public Number parse(String text, ParsePosition status);
}
```

CollationElementIterator

Java 1.1

java.text

Объект CollationElementIterator возвращается методом getCollationElementIterator() объекта RuleBasedCollator. Назначение этого класса – дать возможность программе перемещаться (с помощью метода next()) по символам строки, возвращающей отсортированные значения для каждого из ключей положения в строке. Обратите внимание, что ключи положения – это не просто символы. Например, в традиционном испанском порядке сортировки двухсимвольная последовательность «ch» рассматривается как одиночный ключ положения, который расположен в алфавите между буквами «c» и «d». Значение, возвращаемое методом next(), – это порядковый номер следующего ключа положения в строке. Данное числовое значение может непосредственно сравниваться со значением, возвращаемым другими объектами CollationElementIterator. Значение, возвращаемое методом next(), может быть разложено на первичное, вторичное и третичное значения положения с помощью статических методов этого класса. Данный класс используется RuleBasedCollator для реализации его метода compare() и для создания объектов CollationKey. Некоторым приложениям может даже понадобиться использовать его непосредственно.

```
public final class CollationElementIterator {
// Конструктор отсутствует
// Открытые константы
   public static final int NULLORDER;                                    // =-1
// Открытые методы класса
   public static final int primaryOrder(int order);
   public static final short secondaryOrder(int order);
   public static final short tertiaryOrder(int order);
// Открытые методы экземпляра
1.2 public int getMaxExpansion(int order);
1.2 public int getOffset();
   public int next();
1.2 public int previous();
   public void reset();
1.2 public void setOffset(int newOffset);
1.2 public void setText(String source);
1.2 public void setText(CharacterIterator source);
}
```

Возвращается методами: RuleBasedCollator.getCollationElementIterator()

CollationKey Java 1.1

java. text ***сравнимый***

Объекты CollationKey сравнивают строки быстрее, чем это возможно сделать с помощью Collation.compare(). Объекты этого класса возвращаются методом Collation.getCollationKey(). Чтобы сравнить два объекта CollationKey, вызовите метод compareTo() ключа A, передав ключ B как аргумент (оба объекта CollationKey должны быть созданы через один и тот же объект Collation). Возвращаемое значение этого метода меньше нуля, если ключ A расположен перед ключом B. Оно равно нулю, если они эквивалентны с точки зрения положения, или больше нуля, если ключ A расположен после ключа B. Метод getSourceString() выдает строку, представленную объектом CollationKey.


```
public final class CollationKey implements Comparable {
// Конструктор отсутствует
// Открытые методы экземпляра
   public int compareTo(CollationKey target);
   public String getSourceString();
   public byte[] toByteArray();
// Методы, реализующие Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие Object
   public boolean equals(Object target);
   public int hashCode();
}
```

Передается методам: CollationKey.compareTo()

Возвращается методами: Collator.getCollationKey(), RuleBasedCollator.getCollationKey()

Collator

Java 1.1

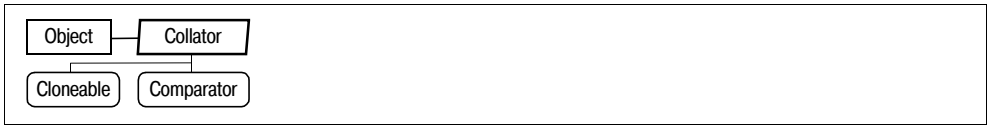
java.text ***клонируемый***

Этот класс сравнивает, упорядочивает и сортирует строки способом, соответствующим региону по умолчанию или какому-либо другому региону. Поскольку это абстрактный класс, нет возможности создавать его экземпляры непосредственно. Вместо этого вы должны использовать статический метод `getInstance()`, чтобы получить экземпляр покласса `Collator`, который соответствует региону по умолчанию или указанному региону. Вы можете использовать метод `getAvailableLocales()`, чтобы определить, доступен ли объект `Collator` для требуемого региона.

Как только соответствующий объект `Collator` получен, вы можете использовать метод `compare()` для сравнения строк. Этот метод возвращает следующие значения: –1, если первая строка располагается перед второй; 0 – обе строки эквивалентны с точки зрения расположения; 1 – первая строка расположена после второй. Метод `equals()` предоставляет удобный и быстрый способ сравнения двух строк на предмет эквивалентности расположения.

Когда сортируется массив строк, каждая строка в массиве обычно проходит процедуру сравнения больше, чем один раз. Использование метода `compare()` в этом случае неэффективно. Более эффективный метод для многократного сравнения строк – использование метода `getCollationKey()` для каждой строки при создании объектов `CollationKey`. Эти объекты затем можно сравнивать друг с другом быстрее, чем сравнивать строки.

С помощью метода `setStrength()` вы можете настроить способ, который объект `Collator` использует при сравнении. Если вы передадите в этот метод константу `PRIMARY`, сравнение будет выполняться только по первичному различию в строках, при этом сравниваются символы и игнорируются огласовки и различия в регистре. Если вы передадите константу `SECONDARY`, будут игнорироваться различия в регистре, но учитываться огласовки. И если вы передадите `TERTIARY` (по умолчанию), объект `Collator` учитывает при сравнении огласовки и различия в регистре.




```
public abstract class Collator implements Cloneable, java.util.Comparator {
// Защищенные конструкторы
   protected Collator();
// Открытые константы
   public static final int CANONICAL_DECOMPOSITION;                    // =1
   public static final int FULL_DECOMPOSITION;                         // =2
   public static final int IDENTICAL;                                  // =3
   public static final int NO_DECOMPOSITION;                           // =0
   public static final int PRIMARY;                                    // =0
   public static final int SECONDARY;                                  // =1
   public static final int TERTIARY;                                   // =2
// Открытые методы класса
   public static java.util.Locale[] getAvailableLocales();             // синхронизирован
   public static Collator getInstance();                               // синхронизирован
   public static Collator getInstance(java.util.Locale desiredLocale); // синхронизирован
// Открытые методы экземпляра
   public abstract int compare(String source, String target);
   public boolean equals(Object that);                          // Реализует:Comparator
```

```
   public boolean equals(String source, String target);
   public abstract CollationKey getCollationKey(String source);
   public int getDecomposition();                                    // синхронизирован
   public int getStrength();                                         // синхронизирован
   public void setDecomposition(int decompositionMode);              // синхронизирован
   public void setStrength(int newStrength);                         // синхронизирован
// Методы, реализующие Comparator
1.2 public int compare(Object o1, Object o2);
   public boolean equals(Object that);
// Открытые методы, замещающие Object
   public Object clone();
   public abstract int hashCode();
}
```

Подклассы: RuleBasedCollator

Возвращается методами: Collator.getInstance()

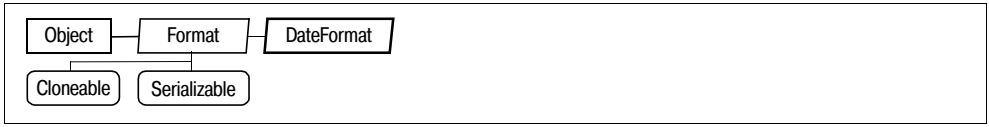
DateFormat

Java 1.1

java.text ***клонируемый, сериализуемый***

Этот класс форматирует и разбирает даты и время способом, соответствующим некоторому региону. Поскольку это абстрактный класс, его экземпляры нельзя создавать непосредственно, однако он предоставляет множество статических методов, возвращающих экземпляры конкретных подклассов, которые вы можете использовать для форматирования дат различными способами. Методы getDateInstance() возвращают объект DateFormat, пригодный для форматирования дат для региона по умолчанию или указанного региона. Может быть задан стиль форматирования; он определяет константы FULL, LONG, MEDIUM, SHORT и DEFAULT. Методы getTimeInstance() возвращают объект DateFormat, который форматирует и разбирает временны*е* значения, а методы getDateTimeInstance() возвращают объект DateFormat, который форматирует как дату, так и время. Этот метод также может получать константу, указывающую стиль форматирования и объект Locale. Наконец, метод getInstance() возвращает объект DateFormat, заданный по умолчанию, который форматирует дату и время в формате SHORT.

После создания объекта DateFormat вы сможете использовать методы setCalendar() и setTimeZone(), если вы хотите форматировать дату с использованием календаря или часового пояса, отличающихся от заданных по умолчанию. Различные методы format() конвертируют объекты java.util.Date в строки, используя формат, представленный в объекте DateFormat. Методы parse() и parseObject() производят обратную операцию; они разбирают строку, отформатированную в соответствии с правилами объекта DateFormat и преобразуют ее в объект Date. Константы DEFAULT, FULL, MEDIUM, LONG и SHORT указывают, насколько подробной или сжатой должна быть отформатированная дата или время. Остальные константы, которые заканчиваются на _FIELD, указывают различные поля отформатированной даты и времени и используются с объектом FieldPosition, который может передаваться в метод format() при необходимости.




```
public abstract class DateFormat extends Format {
// Защищенные конструкторы
   protected DateFormat();
```

```
// Открытые константы
   public static final int AM_PM_FIELD;                                    // =14
   public static final int DATE_FIELD;                                     // =3
   public static final int DAY_OF_WEEK_FIELD;                              // =9
   public static final int DAY_OF_WEEK_IN_MONTH_FIELD;                     // =11
   public static final int DAY_OF_YEAR_FIELD;                              // =10
   public static final int DEFAULT;                                        // =2
   public static final int ERA_FIELD;                                      // =0
   public static final int FULL;                                           // =0
   public static final int HOUR0_FIELD;                                    // =16
   public static final int HOUR1_FIELD;                                    // =15
   public static final int HOUR_OF_DAY0_FIELD;                             // =5
   public static final int HOUR_OF_DAY1_FIELD;                             // =4
   public static final int LONG;                                           // =1
   public static final int MEDIUM;                                         // =2
   public static final int MILLISECOND_FIELD;                              // =8
   public static final int MINUTE_FIELD;                                   // =6
   public static final int MONTH_FIELD;                                    // =2
   public static final int SECOND_FIELD;                                   // =7
   public static final int SHORT;                                          // =3
   public static final int TIMEZONE_FIELD;                                 // =17
   public static final int WEEK_OF_MONTH_FIELD;                            // =13
   public static final int WEEK_OF_YEAR_FIELD;                             // =12
   public static final int YEAR_FIELD;                                     // =1
// Внутренние классы
1.4 public static class Field extends Format.Field;
// Открытые методы класса
   public static java.util.Locale[] getAvailableLocales();
   public static final DateFormat getDateInstance();
   public static final DateFormat getDateInstance(int style);
   public static final DateFormat getDateInstance(int style, java.util.Locale aLocale);
   public static final DateFormat getDateTimeInstance();
   public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle);
   public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle,
        java.util.Locale aLocale);
   public static final DateFormat getInstance();
   public static final DateFormat getTimeInstance();
   public static final DateFormat getTimeInstance(int style);
   public static final DateFormat getTimeInstance(int style, java.util.Locale aLocale);
// Методы доступа к свойствам (по имени свойства)
   public java.util.Calendar getCalendar();
   public void setCalendar(java.util.Calendar newCalendar);
   public boolean isLenient();
   public void setLenient(boolean lenient);
   public NumberFormat getNumberFormat();
   public void setNumberFormat(NumberFormat newNumberFormat);
   public java.util.TimeZone getTimeZone();
   public void setTimeZone(java.util.TimeZone zone);
// Открытые методы экземпляра
   public final String format(java.util.Date date);
   public abstract StringBuffer format(java.util.Date date, StringBuffer toAppendTo,
        FieldPosition fieldPosition);
   public java.util.Date parse(String source) throws ParseException;
   public abstract java.util.Date parse(String source, ParsePosition pos);
// Открытые методы, замещающие Format
   public Object clone();
   public final StringBuffer format(Object obj, StringBuffer toAppendTo,
        FieldPosition fieldPosition);
```

```
   public Object parseObject(String source, ParsePosition pos);
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
// Защищенные поля экземпляра
   protected java.util.Calendar calendar;
   protected NumberFormat numberFormat;
}
```

Подклассы: SimpleDateFormat

Передается методам: javax.swing.text.DateFormatter.{DateFormatter(), setFormat()}

Возвращается методами: DateFormat.{getDateInstance(), getDateTimeInstance(), getInstance(), getTimeInstance()}

DateFormat.Field

Java 1.4

java.text ***сериализуемый***

Этот класс определяет безопасное с точки зрения типов перечисление объектов AttributedCharacterIterator.Attribute, которые могут использоваться классом AttributedCharacterIterator, возвращаемым из метода formatToCharacterIterator(), унаследованного от класса Format, или могут применяться во время создания объекта FieldPosition, с помощью которого можно получить границы указанного поля даты в форматированном выводе. Обратите внимание, что константы, определенные этим классом, практически соответствуют целым константам, определенным в java.util.Calendar, и что этот класс определяет методы для преобразования между двумя множествами констант.

```
public static class DateFormat.Field extends Format.Field {
// Защищенные конструкторы
   protected Field(String name, int calendarField);
// Открытые константы
   public static final DateFormat.Field AM_PM;
   public static final DateFormat.Field DAY_OF_MONTH;
   public static final DateFormat.Field DAY_OF_WEEK;
   public static final DateFormat.Field DAY_OF_WEEK_IN_MONTH;
   public static final DateFormat.Field DAY_OF_YEAR;
   public static final DateFormat.Field ERA;
   public static final DateFormat.Field HOUR0;
   public static final DateFormat.Field HOUR1;
   public static final DateFormat.Field HOUR_OF_DAY0;
   public static final DateFormat.Field HOUR_OF_DAY1;
   public static final DateFormat.Field MILLISECOND;
   public static final DateFormat.Field MINUTE;
   public static final DateFormat.Field MONTH;
   public static final DateFormat.Field SECOND;
   public static final DateFormat.Field TIME_ZONE;
   public static final DateFormat.Field WEEK_OF_MONTH;
   public static final DateFormat.Field WEEK_OF_YEAR;
   public static final DateFormat.Field YEAR;
// Открытые методы класса
   public static DateFormat.Field ofCalendarField(int calendarField);
// Открытые методы экземпляра
   public int getCalendarField();
// Защищенные методы, замещающие AttributedCharacterIterator.Attribute
   protected Object readResolve() throws java.io.InvalidObjectException;
}
```

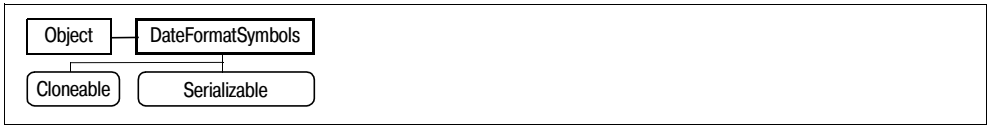
Возвращается методами: DateFormat.Field.ofCalendarField()

Экземпляры: Полей слишком много, чтобы их перечислить.

DateFormatSymbols Java 1.1

java.text ***клонируемый, сериализуемый***

Этот класс определяет методы доступа к различным данным, таким как названия месяцев и дней, используемым в SimpleDateFormat при форматировании и разборе значений даты и времени. Обычно у вас не будет необходимости использовать этот класс до тех пор, пока вам не понадобится форматировать дату для неподдерживаемого региона или применить собственный, очень специфичный способ форматирования.




```
public class DateFormatSymbols implements Cloneable, Serializable {
// Открытые конструкторы
   public DateFormatSymbols();
   public DateFormatSymbols(java.util.Locale locale);
// Методы доступа к свойствам (по имени свойства)
   public String[] getAmPmStrings();
   public void setAmPmStrings(String[] newAmpms);
   public String[] getEras();
   public void setEras(String[] newEras);
   public String getLocalPatternChars();
   public void setLocalPatternChars(String newLocalPatternChars);
   public String[] getMonths();
   public void setMonths(String[] newMonths);
   public String[] getShortMonths();
   public void setShortMonths(String[] newShortMonths);
   public String[] getShortWeekdays();
   public void setShortWeekdays(String[] newShortWeekdays);
   public String[] getWeekdays();
   public void setWeekdays(String[] newWeekdays);
   public String[][] getZoneStrings();
   public void setZoneStrings(String[][] newZoneStrings);
// Открытые методы, замещающие Object
   public Object clone();
   public boolean equals(Object obj);
   public int hashCode();
}
```

Передается методам: SimpleDateFormat.{setDateFormatSymbols(), SimpleDateFormat()}

Возвращается методами: SimpleDateFormat.getDateFormatSymbols()

DecimalFormat Java 1.1

java.text ***клонируемый, сериализуемый***

Этот конкретный подкласс класса Format, используемый в NumberFormat для всех регионов, использующих десятичные числа. Большинству приложений нет необходимос-

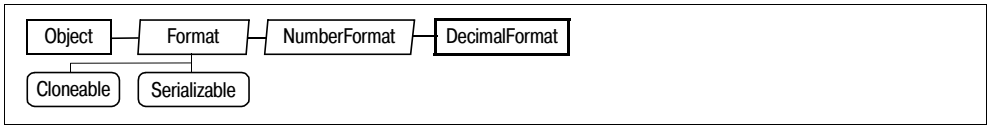
ти использовать этот класс непосредственно; они могут применять статические методы класса NumberFormat для получения объекта NumberFormat, заданного для региона по умолчанию, а затем выполнять мелкую настройку в этом объекте, не зависящую от региона.

Приложения, которым нужно особое форматирование и разбор чисел, могут создать собственные объекты DecimalFormat, передавая подходящий паттерн в конструктор DecimalFormat(). Метод applyPattern() может изменить этот паттерн. Паттерн состоит из строк символов, приведенных в нижеследующей таблице. Например:

```
"$#,##0.00;($#,##0.00)"
```

Символ	Значение
#	Разряд; нули не показываются
0	Разряд; нули показываются как 0
.	Разделитель разрядов, соответствующий региону
,	Разделитель групп разрядов (запятая), соответствующий региону
-	Префикс отрицательного числа, соответствующий региону
%	Показывает значение как проценты
;	Отделяет формат положительного числа (слева) от необязательного формата отрицательного числа (справа)
'	Заключает в кавычки зарезервированный символ, благодаря чему он передается на вывод как есть (апостроф)
другой символ	Передается на вывод как есть

При создании объекта DecimalFormat может быть задан объект DecimalFormatSymbols. Если он не задан, используется объект DecimalFormatSymbols, соответствующий региону, заданному по умолчанию.




```
public class DecimalFormat extends NumberFormat {
// Открытые конструкторы
   public DecimalFormat();
   public DecimalFormat(String pattern);
   public DecimalFormat(String pattern, DecimalFormatSymbols symbols);
// Методы доступа к свойствам (по имени свойства)
1.4 public java.util.Currency getCurrency();                    // Замещает:NumberFormat
1.4 public void setCurrency(java.util.Currency currency);       // Замещает:NumberFormat
   public DecimalFormatSymbols getDecimalFormatSymbols();
   public void setDecimalFormatSymbols(DecimalFormatSymbols newSymbols);
   public boolean isDecimalSeparatorAlwaysShown();               // по умолчанию:false
   public void setDecimalSeparatorAlwaysShown(boolean newValue);
   public int getGroupingSize();                                 // по умолчанию:3
   public void setGroupingSize(int newValue);
1.2 public void setMaximumFractionDigits(int newValue);         // Замещает:NumberFormat
1.2 public void setMaximumIntegerDigits(int newValue);          // Замещает:NumberFormat
1.2 public void setMinimumFractionDigits(int newValue);         // Замещает:NumberFormat
```

```
1.2 public void setMinimumIntegerDigits(int newValue);                    // Замещает:NumberFormat
   public int getMultiplier();                                             // по умолчанию:1
   public void setMultiplier(int newValue);
   public String getNegativePrefix();                                      // по умолчанию:"-"
   public void setNegativePrefix(String newValue);
   public String getNegativeSuffix();                                      // по умолчанию:""
   public void setNegativeSuffix(String newValue);
   public String getPositivePrefix();                                      // по умолчанию:""
   public void setPositivePrefix(String newValue);
   public String getPositiveSuffix();                                      // по умолчанию:""
   public void setPositiveSuffix(String newValue);
// Открытые методы экземпляра
   public void applyLocalizedPattern(String pattern);
   public void applyPattern(String pattern);
   public String toLocalizedPattern();
   public String toPattern();
// Открытые методы, замещающие NumberFormat
   public Object clone();
   public boolean equals(Object obj);
   public StringBuffer format(double number, StringBuffer result, FieldPosition fieldPosition);
   public StringBuffer format(long number, StringBuffer result, FieldPosition fieldPosition);
   public int hashCode();
   public Number parse(String text, ParsePosition pos);
// Открытые методы, замещающие Format
1.4 public AttributedCharacterIterator formatToCharacterIterator(Object obj);
}
```

Возвращается методами: javax.swing.JSpinner.NumberEditor.getFormat()

DecimalFormatSymbols Java 1.1

java.text ***клонируемый, сериализуемый***

Этот класс определяет различные символы и строки, такие как десятичная точка, знак процента, разделитель тысяч, используемые в DecimalFormat при форматировании чисел. Обычно вам не нужно использовать этот класс непосредственно до тех пор, пока вам не понадобится форматировать даты для неподдерживаемых регионов или использовать собственный, значительно измененный формат.


```
public final class DecimalFormatSymbols implements Cloneable, Serializable {
// Открытые конструкторы
   public DecimalFormatSymbols();
   public DecimalFormatSymbols(java.util.Locale locale);
// Методы доступа к свойствам (по имени свойства)
1.4 public java.util.Currency getCurrency();
1.4 public void setCurrency(java.util.Currency currency);
1.2 public String getCurrencySymbol();                                     // по умолчанию:"$"
1.2 public void setCurrencySymbol(String currency);
   public char getDecimalSeparator();                                      // по умолчанию:.
   public void setDecimalSeparator(char decimalSeparator);
```

```
   public char getDigit();                                          // по умолчанию:#
   public void setDigit(char digit);
   public char getGroupingSeparator();                              // по умолчанию:,
   public void setGroupingSeparator(char groupingSeparator);
   public String getInfinity();                                   // по умолчанию:"\u221E"
   public void setInfinity(String infinity);
1.2 public String getInternationalCurrencySymbol();                 // по умолчанию:"USD"
1.2 public void setInternationalCurrencySymbol(String currencyCode);
   public char getMinusSign();                                      // по умолчанию:-
   public void setMinusSign(char minusSign);
1.2 public char getMonetaryDecimalSeparator();                      // по умолчанию:.
1.2 public void setMonetaryDecimalSeparator(char sep);
   public String getNaN();                                        // по умолчанию:"\uFFFD"
   public void setNaN(String NaN);
   public char getPatternSeparator();                               // по умолчанию:;
   public void setPatternSeparator(char patternSeparator);
   public char getPercent();                                        // по умолчанию:%
   public void setPercent(char percent);
   public char getPerMill();                                        // по умолчанию:\u2030
   public void setPerMill(char perMill);
   public char getZeroDigit();                                      // по умолчанию:0
   public void setZeroDigit(char zeroDigit);
// Открытые методы, замещающие Object
   public Object clone();
   public boolean equals(Object obj);
   public int hashCode();
}
```

Передается методам: DecimalFormat.{DecimalFormat(), setDecimalFormatSymbols()}

Возвращается методами: DecimalFormat.getDecimalFormatSymbols()

FieldPosition

Java 1.1

java.text

Объекты FieldPosition могут передаваться в методы format() класса Format и его подклассов с целью получения информации о начальной и конечной позициях отдельной части или «поля» форматированной строки. Информация такого рода часто бывает полезна для выравнивания форматированных строк по столбцам – например, выравнивание десятичных точек в столбце чисел.

Интересующее поле задается при вызове конструктора FieldPosition(). Классы NumberFormat и DateFormat определяют различные целые константы (заканчиваются на _FIELD), которые могут быть здесь использованы. В Java 1.4 и последующих версиях вы можете сконструировать FieldPosition, указав объект Format.Field, идентифицирующий поле. Для того чтобы узнать больше об экземплярах констант класса Field, обратитесь к описанию DateFormat.Field, MessageFormat.Field и NumberFormat.

После создания объекта FieldPosition и его передачи в метод format() используйте методы getBeginIndex() и getEndIndex() этого класса, чтобы получить начальную и конечную позицию символов требуемого поля форматированной строки.

```
public class FieldPosition {
// Открытые конструкторы
1.4 public FieldPosition(Format.Field attribute);
   public FieldPosition(int field);
```

```
1.4 public FieldPosition(Format.Field attribute, int fieldID);
// Открытые методы экземпляра
   public int getBeginIndex();
   public int getEndIndex();
   public int getField();
1.4 public Format.Field getFieldAttribute();
1.2 public void setBeginIndex(int bi);
1.2 public void setEndIndex(int ei);
// Открытые методы, замещающие Object
1.2 public boolean equals(Object obj);
1.2 public int hashCode();
1.2 public String toString();
}
```

Передается методам: ChoiceFormat.format(), DateFormat.format(), DecimalFormat.format(), Format.format(), MessageFormat.format(),NumberFormat.format(), SimpleDateFormat.format()

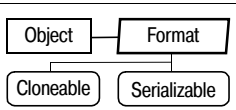
Format

Java 1.1

java.text ***клонируемый, сериализуемый***

Этот абстрактный класс является базовым для всех классов форматированных чисел, дат и строк в пакете java.text. Он определяет ключевые методы для форматирования и разбора, реализуемые всеми подклассами. Метод format() конвертирует объект в строку с использованием правил форматирования, заключенных в подклассе класса Format, и может добавлять результирующую строку к существующему объекту StringBuffer. Метод parseObject() выполняет обратную операцию; он разбирает форматированную строку и возвращает соответствующий объект. Информация о статусе этих двух операций возвращается в объектах FieldPosition и ParsePosition.

Java 1.4 определяет разновидность метода format(). Этот метод formatToCharacterIterator() выполняет такую же операцию форматирования, как и метод format(), но возвращает результат как AttributedCharacterIterator, который использует атрибуты для опознавания различных частей форматированной строки (таких как целая часть, десятичный разделитель и дробная часть форматированного числа). Все ключи атрибутов являются экземплярами внутреннего класса Format.Field. Каждый из подклассов класса Format определяет подкласс Field, реализующий множество констант Field (таких как NumberFormat.Field.DECIMAL_SEPARATOR) для использования в итераторе символов, возвращаемом этим методом. Чтобы узнать больше о подклассах, выполняющих специфические типы форматирования, обратитесь к описанию ChoiceFormat, DateFormat, MessageFormat и NumberFormat.




```
public abstract class Format implements Cloneable, Serializable {
// Открытые конструкторы
   public Format();
// Внутренние классы
1.4 public static class Field extends AttributedCharacterIterator.Attribute;
// Открытые методы экземпляра
   public final String format(Object obj);
   public abstract StringBuffer format(Object obj, StringBuffer toAppendTo, FieldPosition pos);
```

```
1.4 public AttributedCharacterIterator formatToCharacterIterator(Object obj);
   public Object parseObject(String source) throws ParseException;
   public abstract Object parseObject(String source, ParsePosition pos);
// Открытые методы, замещающие Object
   public Object clone();
}
```

Подклассы: DateFormat, MessageFormat, NumberFormat

Передается методам: MessageFormat.{setFormat(), setFormatByArgumentIndex(), setFormats(), setFormatsByArgumentIndex()}, javax.swing.JFormattedTextField.JFormattedTextField(), javax.swing.text.InternationalFormatter.{InternationalFormatter(), setFormat()}, javax.swing.text.NumberFormatter.setFormat()

Возвращается методами: MessageFormat.{getFormats(), getFormatsByArgumentIndex()}, javax.swing.text.InternationalFormatter.getFormat()

Format.Field

Java 1.4

java.text ***сериализуемый***

Этот внутренний класс расширяет AttributedCharacterIterator.Attribute; он используется как общий родительский класс для DateFormat.Field, MessageFormat.Field и NumberFormat.Field. Для получения дополнительной информации обратитесь к описанию этих классов.

```
public static class Format.Field extends AttributedCharacterIterator.Attribute {
// Защищенные конструкторы
   protected Field(String name);
}
```

Подклассы: DateFormat.Field, MessageFormat.Field, NumberFormat.Field

Передается методам: FieldPosition.FieldPosition()

Возвращается методами: FieldPosition.getFieldAttribute(), javax.swing.text.InternationalFormatter.getFields()

MessageFormat

Java 1.1

java.text ***клонируемый, сериализуемый***

Этот класс форматирует и замещает объекты в указанной позиции строки сообщения (также известной как строка паттерна). Он предоставляет Java-эквивалент функции printf() языка программирования C. Если сообщение должно показываться только один раз, то самый простой способ использования класса MessageFormat – задействовать статический метод format(). Этот метод получает сообщение или строку формата и массив объектов аргументов, которые должны форматироваться и замещаться в строке. Если сообщение будет показываться многократно, более разумно создать объект MessageFormat, передав строку паттерна, а затем вызывать метод format() этого экземпляра, передавая массив объектов, которые должны быть отформатированы и заменены в строке сообщения.

Строка сообщения, или строка паттерна, используемая в MessageFormat, содержит цифры, заключенные в фигурные скобки, для указания мест, где должны располагаться значения аргументов. Последовательность «{0}» указывает, что первый объект должен быть преобразован в строку (если это необходимо) и вставлен в выбранное место, а последовательность «{3}» указывает, что должен быть вставлен четвертый

объект. Если объект, который должен быть вставлен, не является строкой, MessageFormat проверяет, не является ли он объектом Date или подклассом Number. Если это так, то он использует объект DateFormat или NumberFormat, заданный по умолчанию, для преобразования значения в строку. Если нет, он просто вызывает метод toString() объекта для преобразования.

За цифрой в фигурных скобках в строке паттерна может следовать запятая и одно из слов: «date», «time», «number» или «choice», показывающее, что перед помещением в строку соответствующий аргумент должен быть отформатирован как дата, время, число или альтернатива. За любым из этих ключевых слов может дополнительно следовать запятая и дополнительная информация паттерна, используемая при форматировании даты, времени, числа или альтернативы. Для получения более полной информации обратитесь к описанию SimpleDateFormat, DecimalFormat и ChoiceFormat.

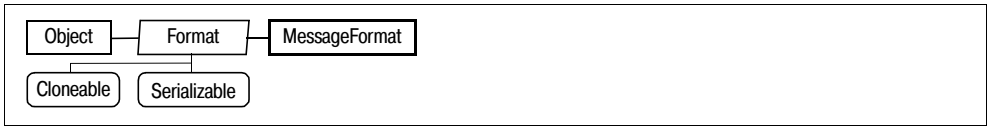
Вы можете передать объект Locale в конструктор или через вызов setLocale() для указания региона, отличного от заданного по умолчанию, который MessageFormat должен использовать при получении объектов DateFormat и NumberFormat при форматировании даты, времени и чисел, помещаемых в паттерн.

Вы можете изменить объект Format, используемый в определенной позиции паттерна, воспользовавшись методом setFormat(), или изменить все объекты Format с помощью метода setFormats(). Оба этих метода зависят от порядка, в котором аргументы будут показываться в строке паттерна. Строка паттерна часто подвергается локализации, а аргументы могут появляться в разном порядке в разных локализациях паттерна.

Поэтому в Java 1.4 и последующих версиях обычно бывает более удобно использовать версию «ByArgumentIndex» методов setFormat(), setFormats() и getFormats().

Вы можете задать новый паттерн для объекта MessageFormat, вызвав метод applyPattern(), и получить строку, представляющую текущий паттерн форматирования, с помощью метода toPattern().

MessageFormat также поддерживает метод parse(), который может разбирать массив объектов из указанной строки в соответствии с указанным паттерном.




```
public class MessageFormat extends Format {
// Открытые конструкторы
   public MessageFormat(String pattern);
1.4 public MessageFormat(String pattern, java.util.Locale locale);
// Внутренние классы
1.4 public static class Field extends Format.Field;
// Открытые методы класса
   public static String format(String pattern, Object[] arguments);
// Открытые методы экземпляра
   public void applyPattern(String pattern);
   public final StringBuffer format(Object[] arguments, StringBuffer result, FieldPosition pos);
   public Format[] getFormats();
1.4 public Format[] getFormatsByArgumentIndex();
   public java.util.Locale getLocale();
   public Object[] parse(String source) throws ParseException;
   public Object[] parse(String source, ParsePosition pos);
   public void setFormat(int formatElementIndex, Format newFormat);
```

```
1.4 public void setFormatByArgumentIndex(int argumentIndex, Format newFormat);
   public void setFormats(Format[] newFormats);
1.4 public void setFormatsByArgumentIndex(Format[] newFormats);
   public void setLocale(java.util.Locale locale);
   public String toPattern();
// Открытые методы, замещающие Format
   public Object clone();
   public final StringBuffer format(Object arguments, StringBuffer result, FieldPosition pos);
1.4 public AttributedCharacterIterator formatToCharacterIterator(Object arguments);
   public Object parseObject(String source, ParsePosition pos);
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

MessageFormat.Field

Java 1.4

java.text ***сериализуемый***

Этот класс определяет константу ARGUMENT в AttributedCharacterIterator.Attribute, которая может использоваться в AttributedCharacterIterator, возвращаемом из MessageFormat.formatToCharacterIterator(), для идентификации отдельных частей форматированного сообщения, извлеченных из аргументов, переданных в метод formatToCharacterIterator(). Значением, связанным с этим атрибутом ARGUMENT, будет целое число типа Integer, указывающее номер аргумента.

```
public static class MessageFormat.Field extends Format.Field {
// Защищенные конструкторы
   protected Field(String name);
// Открытые константы
   public static final MessageFormat.Field ARGUMENT;
// Защищенные методы, замещающие AttributedCharacterIterator.Attribute
   protected Object readResolve() throws java.io.InvalidObjectException;
}
```

Экземпляры: MessageFormat.Field.ARGUMENT

NumberFormat

Java 1.1

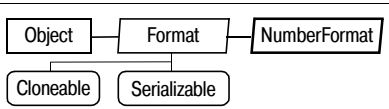
java.text ***клонируемый, сериализуемый***

Этот класс форматирует и разбирает числа способом, соответствующим региону. Поскольку это абстрактный класс, его экземпляры не могут создаваться непосредственно, однако он предоставляет множество статических методов, возвращающих экземпляры конкретных подклассов, которые вы можете использовать при форматировании. Метод getInstance() возвращает объект NumberFormat, пригодный для форматирования обычных чисел для региона по умолчанию или указанного региона. Методы getIntegerInstance(), getCurrencyInstance() и getPercentInstance() возвращают объекты NumberFormat для форматирования чисел, которые являются целыми, или представляют денежные единицы или проценты. Эти методы возвращают объекты NumberFormat для региона по умолчанию или для указанного объекта Locale. Метод getAvailableLocales() возвращает массив регионов, для которых доступны объекты NumberFormat. В Java 1.4 вы можете использовать метод setCurrency() для указания объекта java.util.Currency при форматировании денежных единиц. Обратите внимание, что класс NumberFormat не предназначен для показа очень больших или очень маленьких

чисел, требующих использования экспоненциальной нотации. Кроме того, он не может корректно обрабатывать бесконечные значения или значения NaN (нечисло).

После создания подходящего объекта NumberFormat вы можете настроить его поведение, не зависящее от конкретного региона, с помощью методов setMaximumFractionDigits(), setGroupingUsed() и им подобных. Чтобы настроить поведение, зависящее от региона, вы можете воспользоваться оператором instanceof для проверки, является ли объект NumberFormat экземпляром класса DecimalFormat, и, если это так, преобразовать его к этому типу. Класс DecimalFormat предоставляет возможность управления форматированием чисел. Однако обратите внимание, что объект NumberFormat, настроенный таким образом, не может соответствовать требуемому региону.

После создания настроенного объекта NumberFormat вы можете использовать различные методы format() для преобразования чисел в строки или строковые буферы и методы parse() или parseObject() для преобразования строк в числа. Также вы можете задействовать метод formatToCharacterIterator(), унаследованный от класса Format (и переопределенный DecimalFormat) вместо метода format(). Константы, определенные этим классом, должны использоваться объектом FieldPosition.




```
public abstract class NumberFormat extends Format {
// Открытые конструкторы
   public NumberFormat();
// Открытые константы
   public static final int FRACTION_FIELD;                                    // =1
   public static final int INTEGER_FIELD;                                     // =0
// Внутренние классы
1.4 public static class Field extends Format.Field;
// Открытые методы класса
   public static java.util.Locale[] getAvailableLocales();
   public static final NumberFormat getCurrencyInstance();
   public static NumberFormat getCurrencyInstance(java.util.Locale inLocale);
   public static final NumberFormat getInstance();
   public static NumberFormat getInstance(java.util.Locale inLocale);
1.4 public static final NumberFormat getIntegerInstance();
1.4 public static NumberFormat getIntegerInstance(java.util.Locale inLocale);
   public static final NumberFormat getNumberInstance();
   public static NumberFormat getNumberInstance(java.util.Locale inLocale);
   public static final NumberFormat getPercentInstance();
   public static NumberFormat getPercentInstance(java.util.Locale inLocale);
// Методы доступа к свойствам (по имени свойства)
1.4 public java.util.Currency getCurrency();
1.4 public void setCurrency(java.util.Currency currency);
   public boolean isGroupingUsed();
   public void setGroupingUsed(boolean newValue);
   public int getMaximumFractionDigits();
   public void setMaximumFractionDigits(int newValue);
   public int getMaximumIntegerDigits();
   public void setMaximumIntegerDigits(int newValue);
   public int getMinimumFractionDigits();
   public void setMinimumFractionDigits(int newValue);
   public int getMinimumIntegerDigits();
```

```
    public void setMinimumIntegerDigits(int newValue);
    public boolean isParseIntegerOnly();
    public void setParseIntegerOnly(boolean value);
// Открытые методы экземпляра
    public final String format(long number);
    public final String format(double number);
    public abstract StringBuffer format(long number, StringBuffer toAppendTo, FieldPosition pos);
    public abstract StringBuffer format(double number, StringBuffer toAppendTo, FieldPosition pos);
    public Number parse(String source) throws ParseException;
    public abstract Number parse(String source, ParsePosition parsePosition);
// Открытые методы, замещающие Format
    public Object clone();
    public final StringBuffer format(Object number, StringBuffer toAppendTo, FieldPosition pos);
    public final Object parseObject(String source, ParsePosition pos);
// Открытые методы, замещающие Object
    public boolean equals(Object obj);
    public int hashCode();
}
```

Подклассы: ChoiceFormat, DecimalFormat

Передается методам: DateFormat.setNumberFormat(), javax.swing.text.NumberFormatter.NumberFormatter()

Возвращается методами: DateFormat.getNumberFormat(), NumberFormat.{getCurrencyInstance(), getInstance(), getIntegerInstance(), getNumberInstance(), getPercentInstance()}

Экземпляры: DateFormat.numberFormat

NumberFormat.Field

Java 1.4

java.text ***сериализуемый***

Этот класс определяет безопасное в отношении типов перечисление объектов AttributedCharacterIterator.Attribute, которое может использоваться в AttributedCharacterIterator, возвращаемом методом formatToCharacterIterator(), который унаследован от класса Format, или использоваться при создании объекта FieldPosition для передачи в метод format() с целью получения границ конкретного числового поля (такого как десятичная точка для выравнивания чисел) в форматированном выводе.

```
public static class NumberFormat.Field extends Format.Field {
// Защищенные конструкторы
    protected Field(String name);
// Открытые константы
    public static final NumberFormat.Field CURRENCY;
    public static final NumberFormat.Field DECIMAL_SEPARATOR;
    public static final NumberFormat.Field EXPONENT;
    public static final NumberFormat.Field EXPONENT_SIGN;
    public static final NumberFormat.Field EXPONENT_SYMBOL;
    public static final NumberFormat.Field FRACTION;
    public static final NumberFormat.Field GROUPING_SEPARATOR;
    public static final NumberFormat.Field INTEGER;
    public static final NumberFormat.Field PERCENT;
    public static final NumberFormat.Field PERMILLE;
    public static final NumberFormat.Field SIGN;
// Защищенные методы, замещающие AttributedCharacterIterator.Attribute
    protected Object readResolve() throws java.io.InvalidObjectException;
}
```

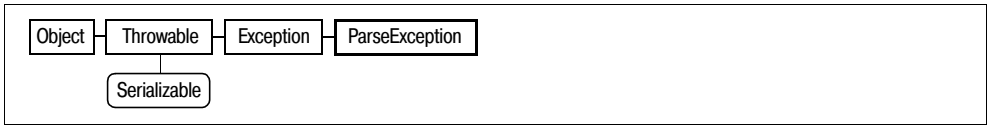
Экземпляры: NumberFormat.Field.{CURRENCY, DECIMAL_SEPARATOR, EXPONENT, EXPONENT_SIGN, EXPONENT_SYMBOL, FRACTION, GROUPING_SEPARATOR, INTEGER, PERCENT, PERMILLE, SIGN}

ParseException

Java 1.1

java.text ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что строка имеет неправильный формат и не может быть разобрана. Обычно оно генерируется методом parse() или parseObject() класса Format и его подклассов, однако может также генерироваться определенными методами в пакете java.text, которым передаются паттерны или другие правила в текстовой форме. Метод этого класса getErrorOffset() возвращает позицию символа в неправильной строке, в которой обнаружена ошибка разбора.




```
public class ParseException extends Exception {
// Открытые конструкторы
   public ParseException(String s, int errorOffset);
// Открытые методы экземпляра
   public int getErrorOffset();
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

ParsePosition

Java 1.1

java.text

Объекты ParsePosition передаются в методы parse() и parseObject() класса Format и его подклассов. Класс ParsePosition представляет позицию в строке, с которой должен начинаться или в которой должен заканчиваться разбор. Перед вызовом метода parse() вы можете указать начальную позицию разбора, передав необходимый индекс в конструктор ParsePosition() или вызвав метод setIndex() существующего объекта ParsePosition. Когда метод parse() закончит свою работу, вы сможете определить, где разбор закончился, вызвав метод getIndex(). В процессе разбора нескольких объектов или значений из строки один и тот же объект ParsePosition может быть использован последовательно.

```
public class ParsePosition {
// Открытые конструкторы
   public ParsePosition(int index);
// Открытые методы экземпляра
1.2 public int getErrorIndex();
   public int getIndex();
1.2 public void setErrorIndex(int ei);
   public void setIndex(int index);
// Открытые методы, замещающие Object
1.2 public boolean equals(Object obj);
1.2 public int hashCode();
1.2 public String toString();
}
```

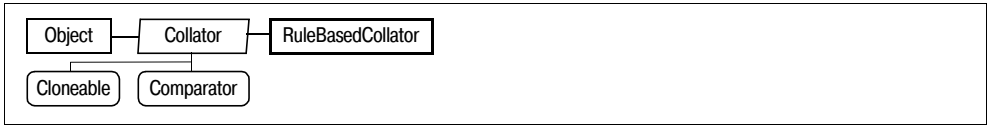
Передается методам: ChoiceFormat.parse(), DateFormat.{parse(), parseObject()}, DecimalFormat.parse(), Format.parseObject(), MessageFormat.{parse(), parseObject()}, NumberFormat.{parse(), parseObject()}, SimpleDateFormat.parse()

RuleBasedCollator

Java 1.1

java.text ***клонируемый***

Этот класс является конкретным подклассом абстрактного класса Collator. Он выполняет сравнения, пользуясь таблицей правил, заданной в текстовой форме. Большинство приложений не используют этот класс напрямую; вместо этого они вызывают метод Collator.getInstance() для получения объекта Collator (обычно это объект RuleBasedCollator), который реализует порядок сравнения по умолчанию для указанного региона или региона по умолчанию. Вам потребуется использовать этот класс, если вы сравниваете строки для региона, не поддерживаемого по умолчанию, или вам нужно реализовать собственный порядок сравнения.




```
public class RuleBasedCollator extends Collator {
// Открытые конструкторы
   public RuleBasedCollator(String rules) throws ParseException;
// Открытые методы экземпляра
1.2 public CollationElementIterator getCollationElementIterator(CharacterIterator source);
   public CollationElementIterator getCollationElementIterator(String source);
   public String getRules();
// Открытые методы, замещающие Collator
   public Object clone();
   public int compare(String source, String target);                    // синхронизирован
   public boolean equals(Object obj);
   public CollationKey getCollationKey(String source);                  // синхронизирован
   public int hashCode();
}
```

SimpleDateFormat

Java 1.1

java.text ***клонируемый, сериализуемый***

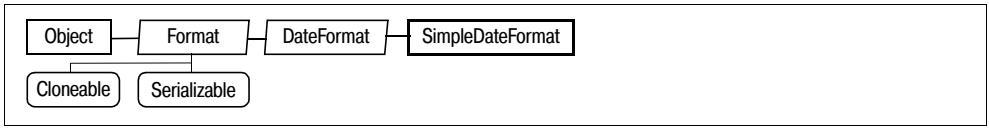
Это конкретный подкласс класса Format, используемый в DateFormat для поддержки форматирования и разбора дат. Большинство приложений не должны использовать этот класс напрямую; вместо этого они должны получить объект DateFormat для определенного региона путем вызова статических методов класса DateFormat.

SimpleDateFormat форматирует даты и время в соответствии с паттерном, указывающим позиции различных полей даты, и объектом DateFormatSymbols, указывающим важные вспомогательные данные, такие как названия месяцев. Приложения, нуждающиеся в собственном форматировании дат и времени, могут создать специальный объект SimpleDateFormat, указав требуемый паттерн. При этом создается объект SimpleDateFormat, использующий объект DateFormatSymbols для региона по умолчанию. Вы можете явно указать регион, чтобы использовать объект DateFormatSymbols для

этого региона. Кроме того, вы можете задать собственный объект DateFormatSymbols, если вам необходимо форматировать даты и время для неподдерживаемого региона.

Метод applyPattern() класса SimpleDateFormat применяется для изменения паттерна форматирования, используемого объектом. Синтаксис этого паттерна описан в нижеследующей таблице. Любые символы в строке формата, которые не описаны в этой таблице, будут появляться в форматированной дате как есть.

Поле	Полная форма	Короткая форма
Год	yyyy (четыре знака)	yy (два знака)
Месяц	MMM (название)	MM (два знака) M (один или два знака)
День недели	EEE	EE
День месяца	dd (два знака)	d (один или два знака)
Часы (1–12)	hh (два знака)	h (один или два знака)
Часы (0–23)	HH (два знака)	H (один или два знака)
Часы (0–11)	KK	K
Часы (1–24)	kk	k
Минуты	mm	
Секунды	ss	
Миллисекунды	SSS	
AM/PM	a	
Часовой пояс	zzz	zz
День недели в месяце	F (то есть 3-й четверг)	
День года	DDD (три знака)	D (один, два или три знака)
Неделя года	ww	
Эра (то есть BC/AD)	G	




```
public class SimpleDateFormat extends DateFormat {
// Открытые конструкторы
   public SimpleDateFormat();
   public SimpleDateFormat(String pattern);
   public SimpleDateFormat(String pattern, java.util.Locale locale);
   public SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols);
// Открытые методы экземпляра
   public void applyLocalizedPattern(String pattern);
   public void applyPattern(String pattern);
1.2 public java.util.Date get2DigitYearStart();
   public DateFormatSymbols getDateFormatSymbols();
1.2 public void set2DigitYearStart(java.util.Date startDate);
   public void setDateFormatSymbols(DateFormatSymbols newFormatSymbols);
   public String toLocalizedPattern();
   public String toPattern();
```

```
// Открытые методы, замещающие DateFormat
   public Object clone();
   public boolean equals(Object obj);
   public StringBuffer format(java.util.Date date, StringBuffer toAppendTo, FieldPosition pos);
   public int hashCode();
   public java.util.Date parse(String text, ParsePosition pos);
// Открытые методы, замещающие Format
1.4 public AttributedCharacterIterator formatToCharacterIterator(Object obj);
}
```

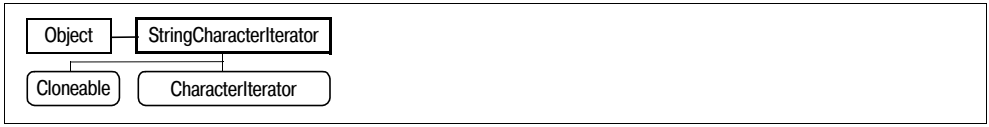
Возвращается методами: javax.swing.JSpinner.DateEditor.getFormat()

StringCharacterIterator

Java 1.1

java.text ***клонируемый***

Этот класс является простейшей реализацией интерфейса CharacterIterator**, работающей с текстом, хранящимся в объектах** String**. Для получения более полной информации обратитесь к описанию** CharacterIterator**.**




```
public final class StringCharacterIterator implements CharacterIterator {
// Открытые конструкторы
   public StringCharacterIterator(String text);
   public StringCharacterIterator(String text, int pos);
   public StringCharacterIterator(String text, int begin, int end, int pos);
// Открытые методы экземпляра
1.2 public void setText(String text);
// Методы, реализующие CharacterIterator
   public Object clone();
   public char current();
   public char first();
   public int getBeginIndex();
   public int getEndIndex();
   public int getIndex();
   public char last();
   public char next();
   public char previous();
   public char setIndex(int p);
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

Глава 17

java.util и подпакеты

Эта глава описывает пакет java.util и каждый из его подпакетов. Ниже приведен список пакетов с кратким описанием:

java.util

Этот пакет содержит определения большого количества полезных классов общего назначения, наиболее важными из которых являются разнообразные реализации Collection, Set, List и Map.

java.util.jar

Этот пакет содержит определения классов для чтения и записи файлов JAR (Java ARchive), которые базируются на классах из пакета java.util.zip.

java.util.logging

Этот пакет содержит определения мощного и гибкого API протоколирования для приложений Java.

java.util.prefs

Этот пакет позволяет приложениям устанавливать и запрашивать постоянные значения для параметров, описывающих предпочтения пользователя, и системных конфигурационных параметров.

java.util.regex

Этот пакет содержит определение API для сопоставления образцов текста с использованием регулярных выражений.

java.util.zip

Этот пакет содержит определения классов для чтения и записи файлов ZIP и для сжатия и распаковки данных с использованием формата «gzip».

Пакет java.util

Java 1.0

Пакет java.util содержит определения большого количества полезных классов, в первую очередь классов коллекций, используемых при работе с группами объектов. Данный пакет не следует рассматривать как вспомогательный пакет; это неотъемлемая и широко используемая часть платформы Java.

Наиболее важными классами в пакете java.util являются классы коллекций. До появления Java 1.2 это были класс Vector, динамически расширяемый список объектов, и класс Hashtable, отображение произвольного ключа на объекты-значения. В Java 1.2 добавлена структура коллекций, состоящая из интерфейсов Collection, Map, Set, List, SortedMap, SortedSet и классов, реализующих эти интерфейсы. Другими важными классами и интерфейсами структуры коллекций являются Comparator, Collections, Arrays, Iterator и ListIterator. Java 1.4 расширяет структуру коллекций новыми реализациями интерфейсов Map и Set и новым интерфейсом-маркером RandomAccess, применяемым в реализациях интерфейса List. BitSet — это связанный класс, который фактически не является частью структуры коллекций (и даже не представляет собой множество). Он обеспечивает очень компактное представление массивов, списков булевых значений или битов произвольного размера. Его API существенно расширен в Java 1.4.

Другие классы этого пакета тоже достаточно полезны. Date, Calendar и TimeZone работают с датой и временем. Currency представляет национальную валюту. Locale охватывает языковые соглашения и соответствующие соглашения о форматировании текста, принятые в выбранной стране, регионе или культуре. ResourceBundle и его подклассы представляют связку локализованных ресурсов, которая может быть прочитана интернациональными программами в процессе выполнения. Random генерирует и возвращает псевдослучайные числа различного вида. StringTokenizer предоставляет простой, но удивительно удобный парсер (parser), которой разбивает строку на токены.

В Java 1.3 и последующих версиях Timer и TimerTask предоставляют мощный API для составления расписания, по которому код будет запускаться в фоновом потоке один раз или периодически в указанное время.

Интерфейсы

```
public interface Comparator;
public interface Enumeration;
public interface Iterator;
public interface ListIterator extends Iterator;
public static interface Map.Entry;
public interface Observer;
public interface RandomAccess;
```

Коллекции

```
public abstract class AbstractCollection implements Collection;
   └ public abstract class AbstractList extends AbstractCollection implements List;
      └ public abstract class AbstractSequentialList extends AbstractList;
         └ public class LinkedList extends AbstractSequentialList
               implements Cloneable, List, Serializable;
      └ public class ArrayList extends AbstractList
               implements Cloneable, List, RandomAccess, Serializable;
      └ public class Vector extends AbstractList
               implements Cloneable, List, RandomAccess, Serializable;
         └ public class Stack extends Vector;
      └ public abstract class AbstractSet extends AbstractCollection implements Set;
         └ public class HashSet extends AbstractSet implements Cloneable, Serializable, Set;
            └ public class LinkedHashSet extends HashSet implements Cloneable, Serializable, Set;
         └ public class TreeSet extends AbstractSet implements Cloneable, Serializable, SortedSet;
public abstract class AbstractMap implements Map;
   └ public class HashMap extends AbstractMap implements Cloneable, Map, Serializable;
      └ public class LinkedHashMap extends HashMap;
   └ public class IdentityHashMap extends AbstractMap implements Cloneable, Map, Serializable;
```

```
   └ public class TreeMap extends AbstractMap implements Cloneable, Serializable, SortedMap;
   └ public class WeakHashMap extends AbstractMap implements Map;
public interface Collection;
public class Hashtable extends Dictionary implements Cloneable, Map, Serializable;
   └ public class Properties extends Hashtable;
public interface List extends Collection;
public interface Map;
public interface Set extends Collection;
public interface SortedMap extends Map;
public interface SortedSet extends Set;
```

События

```
public class EventObject implements Serializable;
```

Слушатели событий

```
public interface EventListener;
```

Другие классы

```
public class Arrays;
public class BitSet implements Cloneable, Serializable;
public abstract class Calendar implements Cloneable, Serializable;
   └ public class GregorianCalendar extends Calendar;
public class Collections;
public final class Currency implements Serializable;
public class Date implements Cloneable, Comparable, Serializable;
public abstract class Dictionary;
public abstract class EventListenerProxy implements EventListener;
public final class Locale implements Cloneable, Serializable;
public class Observable;
public final class PropertyPermission extends java.security.BasicPermission;
public class Random implements Serializable;
public abstract class ResourceBundle;
   └ public abstract class ListResourceBundle extends ResourceBundle;
   └ public class PropertyResourceBundle extends ResourceBundle;
public class StringTokenizer implements Enumeration;
public class Timer;
public abstract class TimerTask implements Runnable;
public abstract class TimeZone implements Cloneable, Serializable;
   └ public class SimpleTimeZone extends TimeZone;
```

Исключения

```
public class ConcurrentModificationException extends RuntimeException;
public class EmptyStackException extends RuntimeException;
public class MissingResourceException extends RuntimeException;
public class NoSuchElementException extends RuntimeException;
public class TooManyListenersException extends Exception;
```

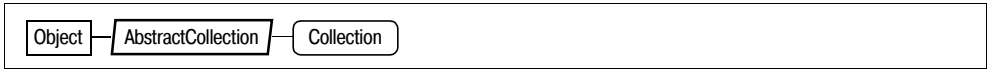
AbstractCollection Java 1.2

java.util ***коллекция***

Этот абстрактный класс представляет собой частичную реализацию интерфейса `Collection`, которая делает более простым процесс написания специальных реализаций `Collection`. Для создания неизменяемой коллекции просто подмените методы `size()` и `iterator()`. Объект `Iterator`, возвращаемый методом `iterator()`, должен поддерживать

только методы hasNext() и next(). Чтобы определить изменяемую коллекцию, вы должны дополнительно подменить метод add() в классе AbstractCollection и убедиться в том, что Iterator, возвращаемый методом iterator(), поддерживает метод remove(). Некоторые подклассы могут подменять другие методы с целью увеличения производительности. Все подклассы должны предоставлять два конструктора: конструктор, не имеющий аргументов, и конструктор, принимающий аргумент Collection, в котором представлено начальное содержимое коллекции.

Если вы напрямую создаете подкласс AbstractCollection, вы реализуете *набор* (*bag*) – неупорядоченную коллекцию, допускающую наличие повторяющихся элементов. Если ваш метод add() не допускает повторяющихся элементов, то вместо этого нужно создать подкласс AbstractSet. См. также AbstractList.




```
public abstract class AbstractCollection implements Collection {
// Защищенные конструкторы
   protected AbstractCollection();
// Методы, реализующие Collection
   public boolean add(Object o);
   public boolean addAll(Collection c);
   public void clear();
   public boolean contains(Object o);
   public boolean containsAll(Collection c);
   public boolean isEmpty();
   public abstract Iterator iterator();
   public boolean remove(Object o);
   public boolean removeAll(Collection c);
   public boolean retainAll(Collection c);
   public abstract int size();
   public Object[] toArray();
   public Object[] toArray(Object[] a);
// Открытые методы, замещающие Object
   public String toString();
}
```

Подклассы: AbstractList, AbstractSet

AbstractList

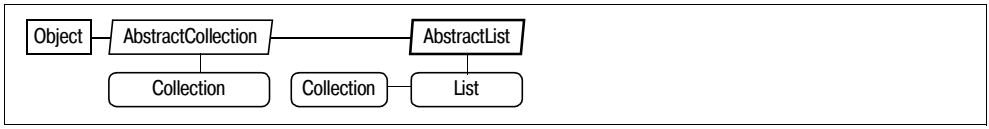
Java 1.2

java.util ***коллекция***

Этот абстрактный класс является частичной реализацией интерфейса List, которая облегчает процесс определения собственных реализаций List на базе элементов списка с произвольным доступом (таких как объекты, хранящиеся в массиве). Если при реализации List вы хотите задействовать модель данных с последовательным доступом (такой как связный список), создавайте вместо этого подкласс AbstractSequentialList.

Для создания неизменяемого List просто создайте подкласс AbstractList и подмените унаследованные методы size() и get(). Кроме того, при создании изменяемого списка необходимо подменить метод set() и, возможно, методы add() и remove(). Эти три метода являются необязательными; если их не подменить, они будут генерировать исключение UnsupportedOperationException. Все остальные методы интерфейса List реализованы на основе методов size(), get(), set(), add() и remove(). В некоторых случа-

ях для повышения производительности можно замещать другие методы. По соглашению, все реализации List должны определять два конструктора: конструктор, не принимающий никаких аргументов, и конструктор, принимающий начальные элементы списка как Collection.




```
public abstract class AbstractList extends AbstractCollection implements java.util.List {
// Защищенные конструкторы
   protected AbstractList();
// Методы, реализующие List
   public boolean add(Object o);
   public void add(int index, Object element);
   public boolean addAll(int index, Collection c);
   public void clear();
   public boolean equals(Object o);
   public abstract Object get(int index);
   public int hashCode();
   public int indexOf(Object o);
   public Iterator iterator();
   public int lastIndexOf(Object o);
   public ListIterator listIterator();
   public ListIterator listIterator(int index);
   public Object remove(int index);
   public Object set(int index, Object element);
   public java.util.List subList(int fromIndex, int toIndex);
// Защищенные методы экземпляра
   protected void removeRange(int fromIndex, int toIndex);
// Защищенные поля экземпляра
   protected transient int modCount;
}
```

Подклассы: AbstractSequentialList, ArrayList, Vector

AbstractMap Java 1.2

java.util ***коллекция***

Этот абстрактный класс является частичной реализацией интерфейса Map, которая облегчает определение простых реализаций Map. Для того чтобы определить неизменяемую таблицу (map), создайте подкласс AbstractMap и подмените метод entrySet() так, чтобы он возвращал множество объектов Map.Entry (кроме того, необходимо реализовать Map.Entry). Возвращаемое множество не должно поддерживать методы add() и remove(), а его итератор не должен поддерживать remove(). Чтобы определить изменяемый Map, вы должны подменить метод put() и предоставить поддержку метода remove() в итераторе, возвращаемом entrySet().iterator(). Все реализации Map определяют два конструктора: конструктор, не принимающий аргументов, и конструктор, принимающий исходную таблицу в виде объекта Map.

AbstractMap определяет все методы Map на основе методов entrySet() и put() и метода remove() его итератора для множества элементов. Однако реализации базируются на линейном поиске из Set, возвращаемом методом entrySet(); они неэффективны, когда

количество элементов Map больше определенного. В некоторых подклассах можно подменить дополнительные методы AbstractMap для увеличения производительности. HashMap и TreeMap используют другие, более эффективные алгоритмы.


```
public abstract class AbstractMap implements Map {
// Защищенные конструкторы
   protected AbstractMap();
// Методы, реализующие Map
   public void clear();
   public boolean containsKey(Object key);
   public boolean containsValue(Object value);
   public abstract Set entrySet();
   public boolean equals(Object o);
   public Object get(Object key);
   public int hashCode();
   public boolean isEmpty();
   public Set keySet();
   public Object put(Object key, Object value);
   public void putAll(Map t);
   public Object remove(Object key);
   public int size();
   public Collection values();
// Открытые методы, замещающие Object
   public String toString();
// Защищенные методы, замещающие Object
1.4 protected Object clone() throws CloneNotSupportedException;
}
```

Подклассы: HashMap, IdentityHashMap, TreeMap, WeakHashMap

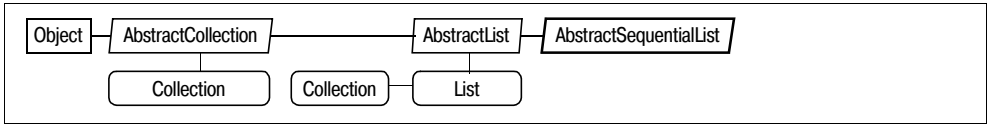
AbstractSequentialList

Java 1.2

java.util ***коллекция***

Этот абстрактный класс представляет собой частичную реализацию интерфейса List, которая облегчает создание собственных реализаций на базе модели последовательного доступа к данным, как в случае с подклассом LinkedList. Для того чтобы реализовать List на базе модели массива или другой модели с произвольным доступом, необходимо наследовать AbstractList.

Для реализации неизменяемого списка унаследуйте этот класс и подмените методы size() и listIterator(). listIterator() должен возвращать ListIterator, определяющий методы hasNext(), hasPrevious(), next(), previous() и index(). Если необходимо разрешить изменение списка, ListIterator должен поддерживать метод set() и (необязательно) методы add() и remove(). AbstractSequentialList реализует все остальные методы List на основе этих методов. В некоторых подклассах для улучшения производительности можно подменить другие методы. Все реализации List определяют два конструктора: конструктор, не имеющий аргументов, и конструктор, принимающий Collection для инициализации элементов списка.




```
public abstract class AbstractSequentialList extends AbstractList {
// Защищенные конструкторы
   protected AbstractSequentialList();
// Открытые методы, замещающие AbstractList
   public void add(int index, Object element);
   public boolean addAll(int index, Collection c);
   public Object get(int index);
   public Iterator iterator();
   public abstract ListIterator listIterator(int index);
   public Object remove(int index);
   public Object set(int index, Object element);
}
```

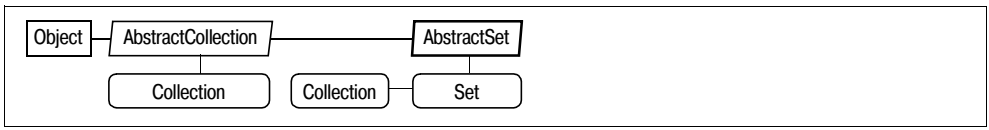
Подклассы: LinkedList

AbstractSet

Java 1.2

java.util ***коллекция***

Этот абстрактный класс является частичной реализацией интерфейса Set, облегчающей создание собственных реализаций Set. Поскольку Set определяет те же методы, что и Collection, вы можете унаследовать AbstractSet точно так же, как и AbstractCollection. Для получения дополнительной информации обратитесь к описанию AbstractCollection. При наследовании AbstractSet необходимо убедиться, что метод add() и конструкторы не допускают добавления повторяющихся элементов в множество. См. также AbstractList.




```
public abstract class AbstractSet extends AbstractCollection implements Set {
// Защищенные конструкторы
   protected AbstractSet();
// Методы, реализующие Set
   public boolean equals(Object o);
   public int hashCode();
1.3 public boolean removeAll(Collection c);
}
```

Подклассы: HashSet, TreeSet

ArrayList

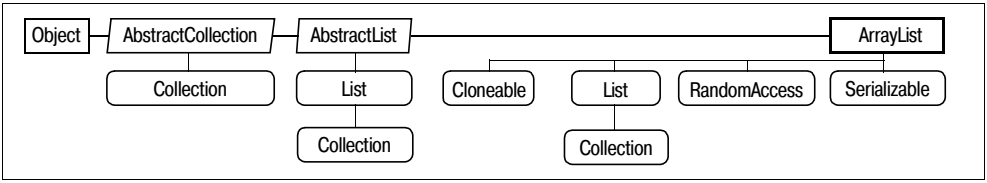
Java 1.2

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс является реализацией List на базе массива, который создается заново при расширении или сужении списка. ArrayList реализует все необязательные методы List и Collection и допускает элементы списка любого типа (включая null). Посколь-

ку ArrayList основан на массиве, методы get() и set() очень эффективны (это не так для реализации LinkedList). ArrayList представляет собой реализацию List общего назначения и применяется достаточно широко. ArrayList очень похож на класс Vector, за исключением того, что его методы не синхронизированы. Если вы используете ArrayList в многопоточном окружении, необходимо явно синхронизировать любые изменения списка или «обернуть» (wrap) этот список с помощью Collections.synchronizedList(). Дополнительную информацию о методах ArrayList можно найти в описании List и Collection. См. также LinkedList.

ArrayList обладает *вместимостью* (*capacity*). Эта характеристика представляет собой количество элементов внутреннего массива, содержащего элементы списка. Когда количество элементов превышает вместимость, необходимо создать новый массив с большей вместимостью. В дополнение к методам List и Collection, ArrayList определяет несколько методов управления вместимостью. Если заранее известно, сколько элементов будет содержать ArrayList, можно вызвать ensureCapacity(). Этот метод повышает эффективность, позволяя избежать последовательного перераспределения внутреннего массива. Кроме того, начальное значение вместимости можно передать в конструктор ArrayList(). Наконец, если ArrayList не будет изменяться в дальнейшем, вы можете вызвать trimToSize() для перераспределения внутреннего списка, указав вместимость, которая точно соответствует размеру списка. Когда ArrayList имеет долгий жизненный цикл, это может быть полезным приемом, уменьшающим использование памяти.




```
public class ArrayList extends AbstractList implements Cloneable, java.util.List, RandomAccess, Serializable {
// Открытые конструкторы
   public ArrayList();
   public ArrayList(int initialCapacity);
   public ArrayList(Collection c);
// Открытые методы экземпляра
   public void ensureCapacity(int minCapacity);
   public void trimToSize();
// Методы, реализующие List
   public boolean add(Object o);
   public void add(int index, Object element);
   public boolean addAll(Collection c);
   public boolean addAll(int index, Collection c);
   public void clear();
   public boolean contains(Object elem);
   public Object get(int index);
   public int indexOf(Object elem);
   public boolean isEmpty();                                    // по умолчанию:true
   public int lastIndexOf(Object elem);
   public Object remove(int index);
   public Object set(int index, Object element);
   public int size();
   public Object[] toArray();
   public Object[] toArray(Object[] a);
```

```
// Защищенные методы, замещающие AbstractList
   protected void removeRange(int fromIndex, int toIndex);
// Открытые методы, замещающие Object
   public Object clone();
}
```

Возвращается методами: Collections.list()

Экземпляры: java.awt.dnd.DragGestureRecognizer.events, java.beans.beancontext.BeanContextServicesSupport.bcsListeners, java.beans.beancontext.BeanContextSupport.bcmListener

Arrays Java 1.2

java.util

Этот класс определяет статические методы для сортировки, поиска и выполнения других полезных действий над массивами. Кроме того, он определяет метод asList(), возвращающий обертку List для указанного массива объектов. Любые изменения List также выполняются в базовом массиве. Это мощный метод, позволяющий манипулировать любым массивом объектов как объектом List. Он обеспечивает связь между массивами и структурой коллекций Java.

Различные методы sort() сортируют список или указанную часть списка. Варианты этого метода определены для массивов каждого из примитивных типов и массивов объектов. Сортировка массивов примитивных типов выполняется в соответствии с порядком, принятым для них. Сортировка массивов объектов выполняется в соответствии с заданным объектом Comparator, а если массив содержит только объекты java.lang.Comparable, то в соответствии с порядком, определенным этим интерфейсом. При сортировке массива объектов применяется постоянный алгоритм, поэтому относительное расположение идентичных объектов не нарушается. Это позволяет выполнять повторные сортировки, например для упорядочения объектов по ключу и подключу.

Методы binarySearch() выполняют эффективный поиск указанного значения в отсортированном списке (время поиска логарифмически убывает). Если в массиве обнаружено совпадение, binarySearch() возвращает индекс совпавшего элемента. Если совпадений не обнаружено, метод возвращает отрицательное число. В случае отрицательного возвращаемого значения r индекс -(r+1) указывает индекс массива, куда указанное значение может быть вставлено для сохранения расположения отсортированных элементов массива. Если массив, в котором выполняется поиск, является массивом объектов, то все элементы массива должны реализовывать интерфейс java.lang.Comparable. Другой вариант – предоставить объект Comparator для выполнения сравнения.

Методы equals() проверяют, являются ли два массива идентичными. Два массива примитивного типа являются идентичными, если они содержат одинаковое количество элементов, а соответствующие пары элементов идентичны в соответствии с оператором ==. Два массива объектов являются идентичными, если они содержат одинаковое количество элементов, а соответствующие пары элементов идентичны в соответствии с методом equals(), определенным для этих объектов. Методы fill() заполняют массив или указанный диапазон массива заданным значением.

```
public class Arrays {
// Конструктор отсутствует
// Открытые методы экземпляра
   public static java.util.List asList(Object[] a);
```

```
    public static int binarySearch(double[] a, double key);
    public static int binarySearch(byte[] a, byte key);
    public static int binarySearch(Object[] a, Object key);
    public static int binarySearch(float[] a, float key);
    public static int binarySearch(int[] a, int key);
    public static int binarySearch(long[] a, long key);
    public static int binarySearch(char[] a, char key);
    public static int binarySearch(short[] a, short key);
    public static int binarySearch(Object[] a, Object key, Comparator c);
    public static boolean equals(double[] a, double[] a2);
    public static boolean equals(boolean[] a, boolean[] a2);
    public static boolean equals(Object[] a, Object[] a2);
    public static boolean equals(float[] a, float[] a2);
    public static boolean equals(byte[] a, byte[] a2);
    public static boolean equals(int[] a, int[] a2);
    public static boolean equals(long[] a, long[] a2);
    public static boolean equals(char[] a, char[] a2);
    public static boolean equals(short[] a, short[] a2);
    public static void fill(short[] a, short val);
    public static void fill(char[] a, char val);
    public static void fill(long[] a, long val);
    public static void fill(int[] a, int val);
    public static void fill(byte[] a, byte val);
    public static void fill(float[] a, float val);
    public static void fill(Object[] a, Object val);
    public static void fill(boolean[] a, boolean val);
    public static void fill(double[] a, double val);
    public static void fill(short[] a, int fromIndex, int toIndex, short val);
    public static void fill(char[] a, int fromIndex, int toIndex, char val);
    public static void fill(long[] a, int fromIndex, int toIndex, long val);
    public static void fill(int[] a, int fromIndex, int toIndex, int val);
    public static void fill(byte[] a, int fromIndex, int toIndex, byte val);
    public static void fill(float[] a, int fromIndex, int toIndex, float val);
    public static void fill(Object[] a, int fromIndex, int toIndex, Object val);
    public static void fill(boolean[] a, int fromIndex, int toIndex, boolean val);
    public static void fill(double[] a, int fromIndex, int toIndex, double val);
    public static void sort(Object[] a);
    public static void sort(short[] a);
    public static void sort(char[] a);
    public static void sort(long[] a);
    public static void sort(byte[] a);
    public static void sort(float[] a);
    public static void sort(int[] a);
    public static void sort(double[] a);
    public static void sort(Object[] a, Comparator c);
    public static void sort(long[] a, int fromIndex, int toIndex);
    public static void sort(byte[] a, int fromIndex, int toIndex);
    public static void sort(char[] a, int fromIndex, int toIndex);
    public static void sort(float[] a, int fromIndex, int toIndex);
    public static void sort(int[] a, int fromIndex, int toIndex);
    public static void sort(double[] a, int fromIndex, int toIndex);
    public static void sort(Object[] a, int fromIndex, int toIndex);
    public static void sort(short[] a, int fromIndex, int toIndex);
    public static void sort(Object[] a, int fromIndex, int toIndex, Comparator c);
}
```

BitSet

Java 1.0

java.util *клонируемый, сериализуемый*

Этот класс реализует массив или список булевых значений и хранит их, используя компактное представление, требующее только 1 бит на каждое значение. Он реализует методы для следующих операций: установка, запрос и перемещение значений, хранящихся в любой заданной позиции списка; подсчет количества значений `true`, хранящихся в списке; поиск следующего значения `true` или `false` в списке. Кроме того, он определяет несколько методов, выполняющих битовые булевые операции над двумя объектами. Несмотря на свое название `BitSet` не реализует интерфейс `Set` и не характеризуется поведением, ассоциирующимся с множеством; он представляет собой список или вектор для булевых значений, но никак не относится ни к интерфейсу `List`, ни к классу `Vector`. Этот класс был представлен еще в Java 1.0, но в Java 1.4 он существенно расширен; большинство методов доступно только в Java 1.4 и последующих версиях.

Создавайте `BitSet` с помощью конструктора `BitSet()`. При желании можно указать размер (количество бит) для `BitSet`, однако это полезно только с точки зрения оптимизации, поскольку при необходимости `BitSet` будет увеличиваться, чтобы вместить любое количество булевых значений. `BitSet` не определяет точного понятия размера «множества». Метод `size()` возвращает количество булевых значений, которые можно сохранить без перестановок во внутреннем хранилище. Метод `length()` возвращает значение, которое на единицу больше самого большого индекса установленного бита (то есть значения `true`). Это означает, что `BitSet`, содержащий только значения `false`, будет иметь нулевую длину. Если вашему коду необходимо помнить наибольший индекс значения, сохраненного в `BitSet`, вне зависимости от того, было ли это `true` или `false`, то информацию о длине нужно хранить отдельно от `BitSet`.

Устанавливайте значения в `BitSet` с помощью метода `set()`. Существуют четыре версии этого метода. Две версии устанавливают значение для заданного индекса, а другие две версии устанавливают значения для диапазона индексов. Два метода `set()` не принимают аргумент с устанавливаемым значением; они «устанавливают» указанный бит или диапазон битов, присваивая значения `true`. Другие два метода принимают булевый аргумент, позволяющий установить указанное значение или диапазон значений в `true` (установленный бит) или `false` (очищенный бит). Кроме того, существуют два метода `clear()`, которые «очищают» (устанавливают в `false`) значение для указанного индекса или диапазона индексов. Методы `flip()` «переворачивают», или «переключают», значение указанного индекса или значения для диапазона (изменяют `true` на `false`, а `false` – на `true`). Подобно другим методам, оперирующим диапазоном значений, методы `set()`, `clear()` и `flip()` задают диапазон с помощью двух значений индекса. Диапазон представляет собой значения, начинающиеся со значения, хранящегося в первом указанном индексе, и заканчивающиеся на значении, хранящемся во втором указанном индексе (это значение не включается в диапазон). Большое количество методов класса `String` и родственных классов следуют такому же соглашению для указания диапазона символов.

Для проверки значения, хранящегося в указанном месте, пользуйтесь методом `get()`, который возвращает `true`, если указанный бит установлен, или `false`, если нет. Существует также метод `get()`, для которого можно указать диапазон битов и получить их состояние в форме `BitSet`; этот метод `get()` аналогичен методу `substring()` класса `String`. Поскольку `BitSet` не определяет максимальный индекс, в метод `get()` допустимо передавать любое неотрицательное число. Если указанный индекс больше или ра-

вен значению, возвращаемому методом length(), то возвращаемым значением всегда будет false.

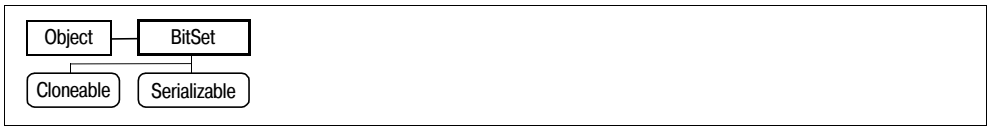
Метод cardinality() возвращает количество значений true (или установленных битов), хранящихся в BitSet. Метод isEmpty() возвращает «истину», если BitSet не хранит значений true (в этом случае и метод length() и метод cardinality() возвращают нуль). Метод nextSetBit() возвращает первый индекс после (или начиная с) указанного индекса, в котором хранится значение true или в котором бит установлен. Этот метод можно применять в цикле для перебора индексов значений true. Метод nextClearBit() подобен предыдущему, но он ищет в BitSet значения false (очищенные биты). Метод intersects() возвращает true, если целевой объект BitSet и аргумент BitSet пересекаются, то есть существует хотя бы один индекс, в котором оба объекта BitSet имеют значение true.

BitSet определяет несколько методов, выполняющих битовые булевые операции. Эти методы объединяют BitSet, для которого они вызываются («целевой» BitSet), с BitSet, передаваемым в качестве аргумента, и сохраняют результат в целевом BitSet. Если необходимо выполнить булевую операцию без изменения оригинального объекта BitSet, то нужно сначала сделать копию оригинала с помощью метода clone() и вызвать метод для копии. Метод and() выполняет битовую булевую операцию AND (И), которая очень похожа на операцию, выполняемую оператором &, когда он применяется к целочисленным аргументам. Значение в целевом BitSet будет true, если оно изначально было true, а значение с таким же индексом в аргументе BitSet тоже true. Для всех значений false аргумента BitSet метод and() устанавливает соответствующие значения в целевом BitSet в false, не изменяя другие значения. Метод andNot() объединяет булевую операцию AND (И) с булевой операцией NOT (отрицание) для аргумента BitSet, однако он не изменяет содержимое этого аргумента BitSet. В результате для всех значений true в аргументе BitSet соответствующие значения в целевом BitSet устанавливаются в false.

Метод or() выполняет битовую булевую операцию OR (ИЛИ), которая похожа на операцию, выполняемую оператором |: значение в BitSet будет установлено в true, если оригинальное значение или соответствующее значение в аргументе BitSet было true. Для всех значений true в аргументе BitSet метод or() устанавливает соответствующее значение в целевом BitSet в true, не изменяя другие значения. Метод xor() выполняет операцию «исключающее ИЛИ»: он устанавливает значение в целевом BitSet в true, если оно изначально было true или соответствующее значение в аргументе BitSet было true. Он устанавливает значение в false, если оба значения были false или оба значения были true.

И наконец, метод toString() возвращает представление BitSet в виде объекта String, которое содержит список индексов, хранящий внутри фигурных скобок значения true.

Класс BitSet не является потокобезопасным.




```
public class BitSet implements Cloneable, Serializable {
// Открытые конструкторы
   public BitSet();
   public BitSet(int nbits);
// Открытые методы экземпляра
```

```
   public void and(BitSet set);
1.2 public void andNot(BitSet set);
1.4 public int cardinality();
1.4 public void clear();
   public void clear(int bitIndex);
1.4 public void clear(int fromIndex, int toIndex);
1.4 public void flip(int bitIndex);
1.4 public void flip(int fromIndex, int toIndex);
   public boolean get(int bitIndex);
1.4 public BitSet get(int fromIndex, int toIndex);
1.4 public boolean intersects(BitSet set);
1.4 public boolean isEmpty();                                    // по умолчанию:true
1.2 public int length();
1.4 public int nextClearBit(int fromIndex);
1.4 public int nextSetBit(int fromIndex);
   public void or(BitSet set);
   public void set(int bitIndex);
1.4 public void set(int bitIndex, boolean value);
1.4 public void set(int fromIndex, int toIndex);
1.4 public void set(int fromIndex, int toIndex, boolean value);
   public int size();
   public void xor(BitSet set);
// Открытые методы, замещающие Object
   public Object clone();
   public boolean equals(Object obj);
   public int hashCode();
   public String toString();
}
```

Передается методам: `BitSet.{and(), andNot(), intersects(), or(), xor()}`, `javax.swing.text.html.parser.DTD.defineElement()`

Возвращается методами: `BitSet.get()`

Экземпляры: `javax.swing.text.html.parser.Element.{exclusions, inclusions}`

Calendar

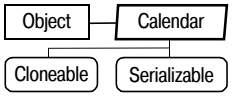
Java 1.1

java.util ***клонируемый, сериализуемый***

Этот абстрактный класс определяет методы, выполняющие арифметические действия с датой и временем. Он также включает в себя методы, которые преобразуют дату и время в миллисекундном формате, применяемом классом `Date`, в единицы, которые более удобны для людей, например минуты, часы, дни, недели, месяцы и годы. Кроме того, здесь представлены методы, выполняющие обратное преобразование. Как абстрактный класс, `Calendar` не позволяет создавать экземпляры напрямую. Вместо этого он предоставляет статические методы `getInstance()`, возвращающие экземпляры подклассов `Calendar`. Такие экземпляры соответствуют заданному региону (locale) или региону по умолчанию. При их использовании указывается часовой пояс или выбирается часовой пояс по умолчанию. См. также `Date`, `DateFormat` и `TimeZone`.

`Calendar` определяет большое количество полезных констант. Некоторые из них представляют дни недели и месяцы года. Другие константы, такие как `HOUR` и `DAY_OF_WEEK`, представляют разнообразную информацию о дате и времени. Эти поля-константы передаются во многие методы класса `Calendar`, такие как `get()` и `set()`. Они определяют, какие поля даты и времени необходимы.

Метод `add()` прибавляет значения к полям календаря, увеличивая следующее большее поле, когда в данном поле происходит переполнение (значения могут вычитаться). Метод `roll()` делает то же самое, но он может изменять только указанное поле. Методы `before()` и `after()` сравнивают два объекта `Calendar`. Многие методы класса `Calendar` являются заменой методам класса `Date`, которые признаны устаревшими в Java 1.1. Поскольку основным предназначением класса `Calendar` является преобразование значений времени в часы, дни, месяцы и другие поля, он не представляет эти поля в форме, которая удобна для показа конечному пользователю. Данную функцию выполняет класс `java.text.DateFormat`, который решает проблемы интернационализации.




```
public abstract class Calendar implements Cloneable, Serializable {
// Защищенные конструкторы
   protected Calendar();
   protected Calendar(TimeZone zone, Locale aLocale);
// Открытые константы
   public static final int AM;                                    // =0
   public static final int AM_PM;                                 // =9
   public static final int APRIL;                                 // =3
   public static final int AUGUST;                                // =7
   public static final int DATE;                                  // =5
   public static final int DAY_OF_MONTH;                          // =5
   public static final int DAY_OF_WEEK;                           // =7
   public static final int DAY_OF_WEEK_IN_MONTH;                  // =8
   public static final int DAY_OF_YEAR;                           // =6
   public static final int DECEMBER;                              // =11
   public static final int DST_OFFSET;                            // =16
   public static final int ERA;                                   // =0
   public static final int FEBRUARY;                              // =1
   public static final int FIELD_COUNT;                           // =17
   public static final int FRIDAY;                                // =6
   public static final int HOUR;                                  // =10
   public static final int HOUR_OF_DAY;                           // =11
   public static final int JANUARY;                               // =0
   public static final int JULY;                                  // =6
   public static final int JUNE;                                  // =5
   public static final int MARCH;                                 // =2
   public static final int MAY;                                   // =4
   public static final int MILLISECOND;                           // =14
   public static final int MINUTE;                                // =12
   public static final int MONDAY;                                // =2
   public static final int MONTH;                                 // =2
   public static final int NOVEMBER;                              // =10
   public static final int OCTOBER;                               // =9
   public static final int PM;                                    // =1
   public static final int SATURDAY;                              // =7
   public static final int SECOND;                                // =13
   public static final int SEPTEMBER;                             // =8
   public static final int SUNDAY;                                // =1
   public static final int THURSDAY;                              // =5
   public static final int TUESDAY;                               // =3
```

```
   public static final int UNDECIMBER;                                          // =12
   public static final int WEDNESDAY;                                           // =4
   public static final int WEEK_OF_MONTH;                                       // =4
   public static final int WEEK_OF_YEAR;                                        // =3
   public static final int YEAR;                                                // =1
   public static final int ZONE_OFFSET;                                         // =15
// Открытые методы экземпляра
   public static Locale[] getAvailableLocales();                        // синхронизирован
   public static Calendar getInstance();
   public static Calendar getInstance(TimeZone zone);
   public static Calendar getInstance(Locale aLocale);
   public static Calendar getInstance(TimeZone zone, Locale aLocale);
// Методы доступа к свойствам (по имени свойства)
   public int getFirstDayOfWeek();
   public void setFirstDayOfWeek(int value);
   public boolean isLenient();
   public void setLenient(boolean lenient);
   public int getMinimalDaysInFirstWeek();
   public void setMinimalDaysInFirstWeek(int value);
   public final java.util.Date getTime();
   public final void setTime(java.util.Date date);
   public long getTimeInMillis();
   public void setTimeInMillis(long millis);
   public TimeZone getTimeZone();
   public void setTimeZone(TimeZone value);
// Открытые методы экземпляра
   public abstract void add(int field, int amount);
   public boolean after(Object when);
   public boolean before(Object when);
   public final void clear();
   public final void clear(int field);
   public int get(int field);
1.2 public int getActualMaximum(int field);
1.2 public int getActualMinimum(int field);
   public abstract int getGreatestMinimum(int field);
   public abstract int getLeastMaximum(int field);
   public abstract int getMaximum(int field);
   public abstract int getMinimum(int field);
   public final boolean isSet(int field);
   public abstract void roll(int field, boolean up);
1.2 public void roll(int field, int amount);
   public void set(int field, int value);
   public final void set(int year, int month, int date);
   public final void set(int year, int month, int date, int hour, int minute);
   public final void set(int year, int month, int date, int hour, int minute,
        int second);
// Открытые методы, замещающие Object
   public Object clone();
   public boolean equals(Object obj);
1.2 public int hashCode();
   public String toString();
// Защищенные методы экземпляра
   protected void complete();
   protected abstract void computeFields();
   protected abstract void computeTime();
   protected final int internalGet(int field);
```

```
// Защищенные поля экземпляра
   protected boolean areFieldsSet;
   protected int[] fields;
   protected boolean[] isSet;
   protected boolean isTimeSet;
   protected long time;
}
```

Подклассы: GregorianCalendar

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.text.DateFormat.getCalendar(), Calendar.getInstance()

Экземпляры: java.text.DateFormat.calendar

Collection Java 1.2

java.util ***коллекция***

Этот интерфейс представляет собой группу, или коллекцию, объектов. Объекты могут быть упорядочены и неупорядочены. Они могут позволять или не допускать хранение повторяющихся объектов. Интерфейс Collection редко реализуется непосредственно. Вместо этого большинство классов коллекций реализуют один из более специфичных подынтерфейсов (subinterface): Set – неупорядоченная коллекция, на допускающая дубликатов, или List – упорядоченная коллекция, допускающая дубликаты.

Collection предоставляет универсальный способ доступа к любому множеству, списку или другой коллекции объектов; он определяет общие методы, работающие с любыми коллекциями. Методы contains() и containsAll() проверяют, содержится ли в данной коллекции указанный объект или группа объектов. Метод isEmpty() возвращает true, если Collection не содержит ни одного элемента, и false в противном случае. Метод size() возвращает количество элементов в Collection. Метод iterator() возвращает объект Iterator, который позволяет выполнять перемещение по объектам коллекции. Метод toArray() возвращает объекты из Collection в новом массиве типа Object. Другая версия toArray() принимает массив в качестве аргумента и сохраняет в этом массиве все элементы Collection, которые должны быть сопоставимы с массивом. Если массив имеет недостаточный размер, то метод создает новый массив такого же типа с большим размером. Если размер массива больше необходимого, метод записывает null в первый пустой элемент массива. Эта версия toArray() возвращает массив, который был передан в метод или новый массив, если он был создан.

Все предыдущие методы запрашивают или извлекают содержимое коллекции. Кроме того, интерфейс Collection определяет методы для изменения содержимого коллекции. Методы add() и addAll() доб авляют объект или коллекцию объектов в Collection. Методы remove() и removeAll() удаляют объект или коллекцию. Метод retainAll() удаляет все объекты, за исключением объектов, содержащихся в указанной Collection. Метод clear() удаляет все объекты из коллекции. Все эти модифицирующие методы, за исключением clear(), возвращают true, если коллекция была изменена в результате вызова метода. Интерфейс не может определять конструкторы, однако существует соглашение, согласно которому все реализации Collection предоставляют как минимум два стандартных конструктора: конструктор, не принимающий аргументов и создающий пустую коллекцию, и копирующий конструктор, принимающий объект Collection, который определяет начальное содержимое новой Collection.

От реализаций интерфейса Collection и его подынтерфейсов не требуется поддержка всех операций, определенных интерфейсом Collection. Все модифицирующие методы,

перечисленные выше, являются необязательными; реализация, которая не поддерживает их (например, реализация неизменяемого Set), просто генерирует java.lang.UnsupportedOperationException для этих методов. Кроме того, реализации могут устанавливать ограничения на типы объектов, которые могут являться членами коллекции. Например, в некоторых реализациях необходимо, чтобы элементы имели конкретный тип. Другие реализации могут запрещать указание null в качестве элемента.

См. также Set, List, Map и Collections.

```
public interface Collection {
// Открытые методы экземпляра
   public abstract boolean add(Object o);
   public abstract boolean addAll(Collection c);
   public abstract void clear();
   public abstract boolean contains(Object o);
   public abstract boolean containsAll(Collection c);
   public abstract boolean equals(Object o);
   public abstract int hashCode();
   public abstract boolean isEmpty();
   public abstract Iterator iterator();
   public abstract boolean remove(Object o);
   public abstract boolean removeAll(Collection c);
   public abstract boolean retainAll(Collection c);
   public abstract int size();
   public abstract Object[] toArray();
   public abstract Object[] toArray(Object[] a);
}
```

Реализации: java.beans.beancontext.BeanContext, AbstractCollection, java.util.List, Set

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: java.beans.beancontext.BeanContextMembershipEvent.children

Collections Java 1.2

java.util

Этот класс определяет статические методы и константы, используемые при работе с коллекциями и отображениями (maps). Метод sort() – один из наиболее широко используемых методов. Он сортирует список объекта List «по месту» (конечно же, список не может быть неизменяемым (immutable)). Алгоритм сортировки является стационарным, то есть идентичные элементы сохраняют свое относительное расположение. Одна из версий sort() использует указанный Comparator для выполнения сортировки; остальные версии опираются на естественный порядок элементов списка и требуют, чтобы элементы реализовывали java.lang.Comparable. Метод reverseOrder() возвращает предопределенный объект Comparator, который может располагать объекты Comparable в порядке, обратном естественному.

Метод binarySearch() имеет непосредственное отношение к sort(). За время, убывающее по логарифмическому закону, он производит поиск указанного объекта в сортированном списке (объекте List) и возвращает индекс найденного объекта. Если объект не найден, метод возвращает отрицательное число. Для отрицательного возвращаемого значения r значение -(r+1) указывает индекс списка, в соответствии с которым указанный объект может быть вставлен в список без нарушения порядка сортировки. Как и в случае с sort(), методу binarySearch() можно передать Comparator,

определяющий порядок элементов в отсортированном списке. Если Comparator не указан, все элементы списка должны реализовывать Comparable; в этом случае список будет отсортирован в соответствии с естественным порядком, определенным этим интерфейсом.

В описании Arrays представлена информация о методах, выполняющих операции сортировки и поиска в массивах, а не в коллекциях.

Разнообразные методы, имена которых начинаются с synchronized, возвращают объекты коллекций, которые пригодны для использования в многопоточной среде и являются обертками (wrappers) для указанной коллекции. Только два объекта коллекций — Vector и Hashtable — пригодны для использования в многопоточной среде по умолчанию. Применяйте эти методы для получения синхронизированного объекта-обертки, если вы используете любые другие типы Collection или Map в многопоточном окружении, где их могут изменять несколько потоков.

Разнообразные методы, имена которых начинаются с unmodifiable, работают подобно методам synchronized. Они возвращают объект Collection или Map, являющийся оберткой вокруг указанной коллекции. Поскольку возвращенный объект является неизменяемым, все его методы add(), remove(), set(), put() и т. п. генерируют java.lang.UnsupportedOperationException.

В дополнение к методам «synchronized» и «unmodifiable», класс Collections определяет большое количество других методов, которые возвращают коллекции или таблицы специального назначения. Метод singleton() возвращает неизменяемое множество, которое содержит только указанный объект. Методы singletonList() и singletonMap() соответственно возвращают неизменяемый список и неизменяемую таблицу, в которых содержится по одному элементу. Кроме того, класс Collections определяет соответствующие константы EMPTY_LIST, EMPTY_SET и EMPTY_MAP, которые представляют собой неизменяемые объекты List, Set и Map, не содержащие элементов или таблиц. Метод nCopies() создает новый неизменяемый List, который содержит указанное количество копий заданного объекта. Метод list() возвращает объект List, представляющий элементы указанного объекта Enumeration. Метод enumeration() выполняет обратную операцию: он возвращает Enumeration для Collection. Это удобно при работе с кодом, который использует старый интерфейс Enumeration вместо нового интерфейса Iterator.

Класс Collections определяет методы, изменяющие коллекцию. Эти методы генерируют UnsupportedOperationException, если целевая коллекция не допускает изменений. Метод copy() копирует элементы исходного списка в целевой список. Метод fill() заменяет все элементы указанного списка на указанный объект. Метод swap() меняет местами элементы для двух указанных индексов в List. Метод replaceAll() заменяет все идентичные элементы в List (сранение выполняется с использованием метода equals()) другим объектом и возвращает true, если замена имела место. Метод reverse() инвертирует порядок элементов в списке. Метод rotate() «разворачивает» список, добавляя указанное число к индексу каждого элемента и «заворачивая» элементы с конца списка в начало списка (если указано отрицательное значение, поворот выполняется в противоположном направлении). Метод shuffle() переставляет элементы списка в случайном порядке, используя внутренний источник случайных чисел или генератор псевдослучайных чисел Random, предоставленный вами.

И наконец, в дополнение к методам binarySearch(), описанным ранее, Collections определяет методы, которые выполняют поиск элементов коллекции; методы min() и max() выполняют поиск минимального и максимального элемента в неотсортированной коллекции в соответствии с указанным Comparator или естественным порядком,

который определен элементами, реализующими Comparable. Методы indexOfSubList() и lastIndexOfSubList() ищут в указанном списке в прямом и обратном направлении последовательность элементов, совпадающую с элементами из второго указанного списка (сравнение выполняется с помощью метода equals()). Они возвращают либо начальный индекс совпадающего подсписка, либо –1, если совпадений не было обнаружено. Эти методы похожи на методы indexOf() и lastIndexOf() класса String. В отличие от методов binarySearch(), они не требуют, чтобы список был отсортирован.

```
public class Collections {
// Конструктор отсутствует
// Открытые константы
   public static final java.util.List EMPTY_LIST;
1.3 public static final Map EMPTY_MAP;
   public static final Set EMPTY_SET;
// Открытые методы экземпляра
   public static int binarySearch(java.util.List list, Object key);
   public static int binarySearch(java.util.List list, Object key, Comparator c);
   public static void copy(java.util.List dest, java.util.List src);
   public static Enumeration enumeration(Collection c);
   public static void fill(java.util.List list, Object obj);
1.4 public static int indexOfSubList(java.util.List source, java.util.List target);
1.4 public static int lastIndexOfSubList(java.util.List source, java.util.List target);
1.4 public static ArrayList list(Enumeration e);
   public static Object max(Collection coll);
   public static Object max(Collection coll, Comparator comp);
   public static Object min(Collection coll);
   public static Object min(Collection coll, Comparator comp);
   public static java.util.List nCopies(int n, Object o);
1.4 public static boolean replaceAll(java.util.List list, Object oldVal, Object newVal);
   public static void reverse(java.util.List list);
   public static Comparator reverseOrder();
1.4 public static void rotate(java.util.List list, int distance);
   public static void shuffle(java.util.List list);
   public static void shuffle(java.util.List list, Random rnd);
   public static Set singleton(Object o);
1.3 public static java.util.List singletonList(Object o);
1.3 public static Map singletonMap(Object key, Object value);
   public static void sort(java.util.List list);
   public static void sort(java.util.List list, Comparator c);
1.4 public static void swap(java.util.List list, int i, int j);
   public static Collection synchronizedCollection(Collection c);
   public static java.util.List synchronizedList(java.util.List list);
   public static Map synchronizedMap(Map m);
   public static Set synchronizedSet(Set s);
   public static SortedMap synchronizedSortedMap(SortedMap m);
   public static SortedSet synchronizedSortedSet(SortedSet s);
   public static Collection unmodifiableCollection(Collection c);
   public static java.util.List unmodifiableList(java.util.List list);
   public static Map unmodifiableMap(Map m);
   public static Set unmodifiableSet(Set s);
   public static SortedMap unmodifiableSortedMap(SortedMap m);
   public static SortedSet unmodifiableSortedSet(SortedSet s);
}
```

Comparator Java 1.2

java.util

Этот интерфейс определяет метод compare(), который устанавливает общие правила упорядочения множества объектов, позволяя сортировать эти объекты. Comparator применяется для упорядочения объектов, которые не представлены в естественном порядке, определенном интерфейсом Comparable. Кроме того, он применяется для упорядочения объектов в соответствии с порядком, отличным от их естественного расположения.

Метод compare() принимает два объекта. Если первый аргумент меньше второго или должен быть расположен перед вторым в отсортированном списке, compare() должен вернуть отрицательное целое число. Если первый аргумент больше второго или должен быть расположен после второго в отсортированном списке, compare() должен вернуть положительное целое число. Если два объекта равнозначны или их относительное расположение в отсортированном списке не имеет значения, compare() должен вернуть нуль. Реализации Comparator могут рассчитывать на то, что оба аргумента Object имеют соответствующие типы, и приводить их к необходимым типам. Если хотя бы один из аргументов не является аргументом ожидаемого типа, метод compare() генерирует ClassCastException.

Обратите внимание, что значения чисел, возвращаемых методом compare(), не важны; важно лишь, равны ли эти значения нулю, меньше нуля или больше нуля. В большинстве случаев вам нужно реализовать Comparator таким образом, чтобы compare(o1,o2) возвращал 0 тогда и только тогда, когда o1.equals(o2) возвращает true. Это особенно важно, если Comparator применяется для упорядочения TreeSet или TreeMap.

Дополнительную информацию о разнообразных методах, использующих объекты Comparator для сортировки и поиска, можно найти в описании Collections и Arrays. См. также родственный интерфейс java.lang.Comparable.

```
public interface Comparator {
// Открытые методы экземпляра
   public abstract int compare(Object o1, Object o2);
   public abstract boolean equals(Object obj);
}
```

Реализации: java.text.Collator

Передается методам: Arrays.{binarySearch(), sort()}, Collections.{binarySearch(), max(), min(), sort()}, TreeMap.TreeMap(), TreeSet.TreeSet(), javax.swing.SortingFocusTraversalPolicy.{setComparator(), SortingFocusTraversalPolicy()}

Возвращается методами: Collections.reverseOrder(), SortedMap.comparator(), SortedSet.comparator(), TreeMap.comparator(), TreeSet.comparator(), javax.swing.SortingFocusTraversalPolicy.getComparator()

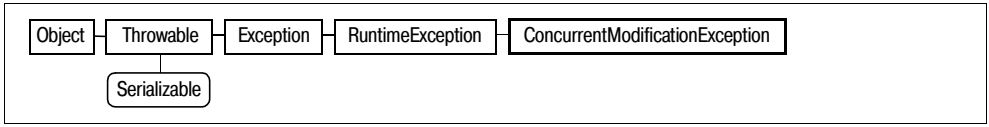
Экземпляры: String.CASE_INSENSITIVE_ORDER

ConcurrentModificationException Java 1.2

java.util ***сериализуемое, непроверяемое***

Это исключение сигнализирует о том, что изменение, которое в данный момент выполняется в структуре данных другой операцией, все еще не закончено, а значит, корректность текущей операции не может быть гарантирована. Обычно оно генерируется объектом Iterator или ListIterator для прекращения итерации, если он обна-

руживает, что обрабатываемая коллекция была изменена в процессе выполнения итерации.




```
public class ConcurrentModificationException extends RuntimeException {
// Открытые конструкторы
   public ConcurrentModificationException();
   public ConcurrentModificationException(String message);
}
```

Currency

Java 1.4

java.util ***сериализуемый***

Экземпляры этого класса представляют валюту. Для получения объекта `Currency` нужно передать в метод `getInstance()` «код валюты», например «USD» для долларов США или «EUR» для евро. После получения объекта `Currency` можно с помощью метода `getSymbol()` получить символ валюты для региональных параметров, заданных по умолчанию, или указанного объекта `Locale` (символ валюты часто отличается от кода валюты). Например, для США символом USD был бы «$», но для других регионов это может быть «US$». Если символ неизвестен, метод возвращает код валюты.

Метод `getDefaultFractionDigits()` применяется для определения количества дробных разрядов, которые принято использовать для данной валюты. Этот метод возвращает 2 для долларов США и других валют, денежные единицы которых делятся на сто. Для иорданских динаров (JOD) и других валют, денежные единицы которых традиционно делятся на тысячи, этот метод возвращает 3. Он возвращает 0 для японских йен (JPY) и других валют, имеющих маленькое значение валютной единицы, которая обычно не делится на дробные части. Коды валют регламентированы стандартом ISO 4217. Полный список валют и их кодов представлен на веб-сайте агенства, поддерживающего этот стандарт: *http://www.bsi-global.com/Technical+Information/Publications/_Publications/tig90.xalter*.


```
public final class Currency implements Serializable {
// Конструктор отсутсвует
// Открытые методы экземпляра
   public static Currency getInstance(String currencyCode);
   public static Currency getInstance(Locale locale);
// Открытые методы экземпляра
   public String getCurrencyCode();
   public int getDefaultFractionDigits();
   public String getSymbol();
   public String getSymbol(Locale locale);
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методам: java.text.DecimalFormat.setCurrency(), java.text.DecimalFormatSymbols.setCurrency(), java.text.NumberFormat.setCurrency()

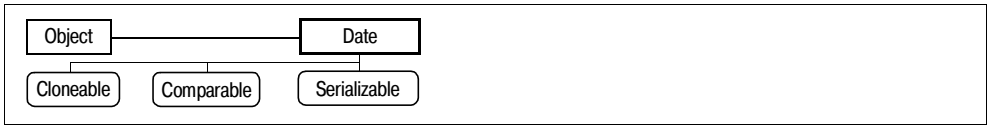
Возвращается методами: java.text.DecimalFormat.getCurrency(), java.text.DecimalFormatSymbols.getCurrency(), java.text.NumberFormat.getCurrency(), Currency.getInstance()

Date

Java 1.0

java.util ***клонируемый, сериализуемый, сравнимый***

Этот класс представляет даты и значения времени. Он дает возможность работать с ними, применяя метод, который не зависит от системы. Вы можете создать Date, указав количество миллисекунд, прошедшее от начала отсчета (полночь 1 января 1970 года по GMT), или год, месяц, число и (необязательно) часы, минуты, секунды. Годы представлены количеством лет, прошедших с 1900 года. Если вызвать конструктор Date без аргументов, объект Date инициализируется текущим временем и датой. Методы экземпляра этого класса позволяют получать и устанавливать разнообразные поля даты и времени, сравнивать даты, время, а также преобразовывать даты в строковое представление и обратно в машинное. В Java 1.1 многие из этих методов признаны устаревшими, а предпочтение отдано методам класса Calendar.




```
public class Date implements Cloneable, Comparable, Serializable {
// Открытые конструкторы
  public Date();
  public Date(long date);
# public Date(String s);
# public Date(int year, int month, int date);
# public Date(int year, int month, int date, int hrs, int min);
# public Date(int year, int month, int date, int hrs, int min, int sec);
// Методы доступа к свойствам (по имени свойства)
  public long getTime();
  public void setTime(long time);
// Открытые методы экземпляра
  public boolean after(java.util.Date when);
  public boolean before(java.util.Date when);
1.2 public int compareTo(java.util.Date anotherDate);
// Методы, реализующие Comparable
1.2 public int compareTo(Object o);
// Открытые методы, замещающие Object
1.2 public Object clone();
  public boolean equals(Object obj);
  public int hashCode();
  public String toString();
// Устаревшие открытые методы
# public int getDate();
# public int getDay();
# public int getHours();
# public int getMinutes();
# public int getMonth();
```

```
# public int getSeconds();
# public int getTimezoneOffset();
# public int getYear();
# public static long parse(String s);
# public void setDate(int date);
# public void setHours(int hours);
# public void setMinutes(int minutes);
# public void setMonth(int month);
# public void setSeconds(int seconds);
# public void setYear(int year);
# public String toGMTString();
# public String toLocaleString();
# public static long UTC(int year, int month, int date, int hrs, int min, int sec);
}
```

Подклассы: java.sql.Date, java.sql.Time, java.sql.Timestamp

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Dictionary

Java 1.0

java.util

Этот абстрактный класс является родительским классом Hashtable. Другие структуры данных, похожие на хеш-таблицу, тоже расширяют этот класс. Для получения дополнительной информации обратитесь к описанию Hashtable. В Java 1.2 интерфейс Map заменил этот класс с точки зрения функциональности.

```
public abstract class Dictionary {
// Открытые конструкторы
   public Dictionary();
// Открытые методы экземпляра
   public abstract Enumeration elements();
   public abstract Object get(Object key);
   public abstract boolean isEmpty();
   public abstract Enumeration keys();
   public abstract Object put(Object key, Object value);
   public abstract Object remove(Object key);
   public abstract int size();
}
```

Подклассы: Hashtable

Передается методам: javax.swing.JSlider.setLabelTable(), javax.swing.text.AbstractDocument.setDocumentProperties()

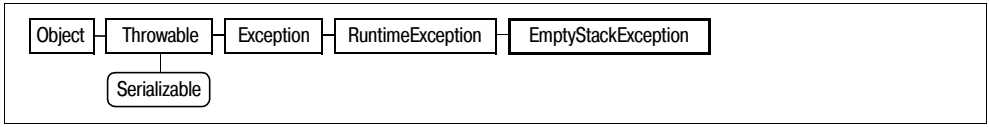
Возвращается методами: javax.swing.JSlider.getLabelTable(), javax.swing.text.AbstractDocument.getDocumentProperties()

EmptyStackException

Java 1.0

java.util ***сериализуемое, непроверяемое***

Это исключение сигнализирует о том, что объект Stack пуст.




```
public class EmptyStackException extends RuntimeException {
// Открытые конструкторы
   public EmptyStackException();
}
```

Генерируется методами: java.awt.EventQueue.pop()

Enumeration Java 1.0

java.util

Этот интерфейс определяет методы, необходимые для перечисления или последовательного прохождения по множеству значений, например по значениям, содержащимся в хеш-таблице или бинарном дереве. Он особенно удобен для структур данных, таких как хеш-таблицы, в которых элементы не могут быть получены по индексу, как в случае массивов. Обычно экземпляры Enumeration не создаются непосредственно; они создаются объектом, которому необходимо сделать свои значения перечислимыми. Большое количество классов, таких как Vector и Hashtable, имеют методы, возвращающие объекты Enumeration. В Java 1.2 вместо Enumeration предпочтительнее использовать новый интерфейс Iterator.

При использовании объекта Enumeration нужно вызывать два его метода в цикле. Метод hasMoreElements() возвращает true, если существуют значения, которые подлежат перечислению. Он позволяет определить необходимость продолжения цикла. Внутри цикла вызов метода nextElement() возвращает значение из перечисления. Enumeration не дает никаких гарантий относительно порядка, в котором будут возвращаться значения. По значениям в объекте Enumeration можно пройти (iterate through) только один раз; нет способа восстановить его первоначальное состояние.

```
public interface Enumeration {
// Открытые методы экземпляра
   public abstract boolean hasMoreElements();
   public abstract Object nextElement();
}
```

Реализации: java.text.CharSet.Enumeration, StringTokenizer, javax.naming.NamingEnumeration

Передается методам: java.io.SequenceInputStream.SequenceInputStream(), Collections.list(), javax.naming.CompositeName.CompositeName(), javax.naming.CompoundName.CompoundName(), javax.swing.JTree.removeDescendantToggledPaths(), javax.swing.text.AbstractDocument.AbstractElement.removeAttributes(), javax.swing.text.AbstractDocument.AttributeContext.removeAttributes(), javax.swing.text.MutableAttributeSet.removeAttributes(), javax.swing.text.SimpleAttributeSet.removeAttributes(), javax.swing.text.StyleContext.removeAttributes(), javax.swing.text.StyleContext.NamedStyle.removeAttributes(), javax.swing.text.html.StyleSheet.removeAttributes()

Возвращается методами: Слишком много методов, чтобы их перечислить.

Экземпляры: javax.swing.tree.DefaultMutableTreeNode.EMPTY_ENUMERATION

EventListener Java 1.0

java.util *слушатель событий*

EventListener является базовым интерфейсом для модели событий, применяемых AWT и Swing в Java 1.1 и последующих версиях. Этот интерфейс не определяет методов и констант; он просто служит меткой, идентифицирующей объекты, которые действуют как слушатели событий. Интерфейсы слушателей событий в пакетах java.awt.event, java.beans и javax.swing.event расширяют этот интерфейс.

```
public interface EventListener {
}
```

Реализации: Классов слишком много, чтобы их перечислить.

Передается методам: java.awt.AWTEventMulticaster.{addInternal(), AWTEventMulticaster(), getListeners(), remove(), removeInternal(), save()}, EventListenerProxy.EventListenerProxy(), javax.swing.event.EventListenerList.{add(), remove()}

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: java.awt.AWTEventMulticaster.{a, b}

EventListenerProxy Java 1.4

java.util

Этот абстрактный класс служит родительским классом для объектов-посредников слушателей событий. Подклассы этого класса реализуют интерфейс слушателя событий и служат обертками (wrappers) вокруг слушателей событий соответствующего типа. Они определяют методы, предоставляющие дополнительную информацию о слушателе. В описании java.beans.PropertyChangeListenerProxy поясняется, как применять объекты-посредники слушателей событий.


```
public abstract class EventListenerProxy implements java.util.EventListener {
// Открытые конструкторы
   public EventListenerProxy(java.util.EventListener listener);
// Открытые методы экземпляра
   public java.util.EventListener getListener();
}
```

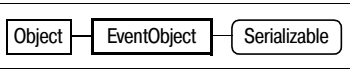
Подклассы: java.awt.event.AWTEventListenerProxy, java.beans.PropertyChangeListenerProxy, java.beans.VetoableChangeListenerProxy

EventObject Java 1.4

java.util *сериализуемый, событие*

EventObject служит родительским классом для всех объектов событий, которые применяются в модели событий, представленной в Java 1.1 для AWT и JavaBeans и в Java 1.2 для Swing. Этот класс определяет базовый тип события; он расширяется более специфичными классами событий в пакетах java.awt, java.awt.event, java.beans и javax.swing.event. Все события задействуют объект-источник, который является объек-

том, сгенерировавшим событие. Объект-источник передается в конструктор `EventObject()` и возвращается методом `getSource()`.




```
public class EventObject implements Serializable {
// Открытые конструкторы
   public EventObject(Object source);
// Открытые методы экземпляра
   public Object getSource();
// Открытые методы, замещающие Object
   public String toString();
// Защищенные поля экземпляра
   protected transient Object source;
}
```

Подклассы: Классов слишком много, чтобы их перечислить.

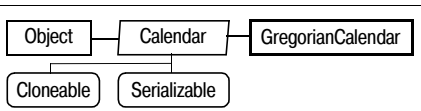
Передается методам: `javax.swing.AbstractCellEditor.{isCellEditable(), shouldSelectCell()}`, `javax.swing.CellEditor.{isCellEditable(), shouldSelectCell()}`, `javax.swing.DefaultCellEditor.{isCellEditable(), shouldSelectCell()}`, `javax.swing.DefaultCellEditor.EditorDelegate.{isCellEditable(), shouldSelectCell(), startCellEditing()}`, `javax.swing.JTable.editCellAt()`, `javax.swing.tree.DefaultTreeCellEditor.{canEditImmediately(), isCellEditable(), shouldSelectCell(), shouldStartEditingTimer()`

GregorianCalendar

Java 1.1

java.util ***клонируемый, сериализуемый***

Этот подкласс класса `Calendar` реализует стандартный солнечный календарь с годами, нумеруемыми от Рождества Христова. Он применяется в большинстве наборов региональных параметров по всему миру. Обычно этот класс не применяется непосредственно. Вместо этого с помощью метода `Calendar.getInstance()` можно получать объект `Calendar`, соответствующий региону по умолчанию. В описании `Calendar` представлена дополнительная информация о работе с объектами `Calendar`. В григорианском календаре существует разрыв, который представляет исторический переход с юлианского календаря на григорианский. По умолчанию `GregorianCalendar` предполагает, что этот переход произошел 15 октября 1582 года. Большинству программ нет необходимости заботиться об этом переходе.




```
public class GregorianCalendar extends Calendar {
// Открытые конструкторы
   public GregorianCalendar();
   public GregorianCalendar(TimeZone zone);
   public GregorianCalendar(Locale aLocale);
   public GregorianCalendar(TimeZone zone, Locale aLocale);
   public GregorianCalendar(int year, int month, int date);
```

```
   public GregorianCalendar(int year, int month, int date, int hour, int minute);
   public GregorianCalendar(int year, int month, int date, int hour, int minute, int second);
// Открытые константы
   public static final int AD;                                                    // =1
   public static final int BC;                                                    // =0
// Открытые методы экземпляра
   public final java.util.Date getGregorianChange();
   public boolean isLeapYear(int year);
   public void setGregorianChange(java.util.Date date);
// Открытые методы, замещающие Calendar
   public void add(int field, int amount);
   public boolean equals(Object obj);
1.2 public int getActualMaximum(int field);
1.2 public int getActualMinimum(int field);
   public int getGreatestMinimum(int field);
   public int getLeastMaximum(int field);
   public int getMaximum(int field);
   public int getMinimum(int field);
   public int hashCode();
1.2 public void roll(int field, int amount);
   public void roll(int field, boolean up);
// Защищенные методы, замещающие Calendar
   protected void computeFields();
   protected void computeTime();
}
```

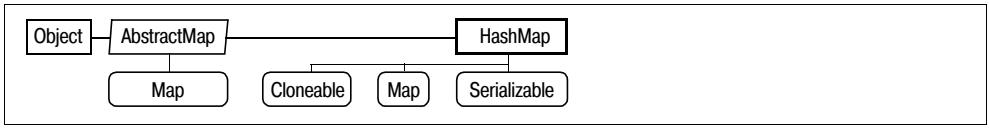
HashMap

Java 1.2

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс реализует интерфейс `Map`, используя внутреннюю хеш-таблицу. Он поддерживает все необязательные методы `Map`, допускает любые типы для объектов ключей и значений, а также позволяет использовать `null` как ключ или значение. Поскольку `HashMap` базируется на структуре данных хеш-таблицы, методы `get()` и `put()` являются очень эффективными. `HashMap` очень похож на класс `Hashtable`, за исключением того, что методы `HashMap` не синхронизированы (соответственно, они являются более быстрыми). Кроме того, `HashMap` разрешает использовать `null` в качестве ключа или значения. `Hashtable` полезен в многопоточном окружении и для обеспечения совместимости с предыдущими версиями Java. В других случаях применяется `HashMap`.

Если вы приблизительно знаете, сколько элементов будет содержать `HashMap`, то можно повысить эффективность, указав *`initialCapacity`* при вызове конструктора `HashMap()`. Произведение аргументов *`initialCapacity`* и *`loadFactor`* должно быть больше количества элементов, которое будет содержаться в `HashMap`. Хорошее значение для *`loadFactor`* – 0.75; оно же является значением по умолчанию. Для получения подробной информации о методах `HashMap` обратитесь к описанию `Map`. См. также `TreeMap` и `HashSet`.




```
public class HashMap extends AbstractMap implements Cloneable, Map, Serializable {
// Открытые конструкторы
   public HashMap();
```

```
   public HashMap(int initialCapacity);
   public HashMap(Map m);
   public HashMap(int initialCapacity, float loadFactor);
// Методы, реализующие Map
   public void clear();
   public boolean containsKey(Object key);
   public boolean containsValue(Object value);
   public Set entrySet();
   public Object get(Object key);
   public boolean isEmpty();                                    // по умолчанию: true
   public Set keySet();
   public Object put(Object key, Object value);
   public void putAll(Map t);
   public Object remove(Object key);
   public int size();
   public Collection values();
// Открытые методы, замещающие AbstractMap
   public Object clone();
}
```

Подклассы: LinkedHashMap, javax.print.attribute.standard.PrinterStateReasons

Экземпляры: java.beans.beancontext.BeanContextServicesSupport.services, java.beans.beancontext.BeanContextSupport.children

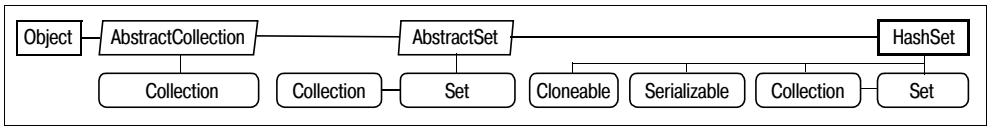
HashSet

Java 1.2

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс реализует интерфейс Set, используя внутреннюю хеш-таблицу. Он поддерживает все необязательные методы Set и Collection. В качестве члена множества может выступать объект любого типа или null. Поскольку HashSet основан на хеш-таблице, все базовые методы add(), remove() и contains() являются достаточно эффективными. HashSet не предоставляет никаких гарантий в отношении порядка, в котором элементы множества будут перечисляться с помощью Iterator, возвращаемого методом iterator(). Методы HashSet не синхронизированы. В многопоточной среде необходимо явно синхронизировать весь код, изменяющий множество, или получить синхронизированную обертку для него, вызвав Collections.synchronizedSet().

Если вам заранее известно, сколько отображений будет содержать HashSet, вы можете улучшить эффективность, указав *initialCapacity* при вызове конструктора HashSet(). Произведение аргументов *initialCapacity* и *loadFactor* должно быть больше количества элементов, которое будет содержаться в HashSet. Хорошее значение для *loadFactor* — 0.75; оно же является значением по умолчанию. В описании Set и Collection можно получить подробную информацию о методах HashSet. См. также TreeSet и HashMap.




```
public class HashSet extends AbstractSet implements Cloneable, Serializable, Set {
// Открытые конструкторы
   public HashSet();
   public HashSet(Collection c);
   public HashSet(int initialCapacity);
```

```
   public HashSet(int initialCapacity, float loadFactor);
// Методы, реализующие Set
   public boolean add(Object o);
   public void clear();
   public boolean contains(Object o);
   public boolean isEmpty();                                    // по умолчанию: true
   public Iterator iterator();
   public boolean remove(Object o);
   public int size();
// Открытые методы, замещающие Object
   public Object clone();
}
```

Подклассы: LinkedHashSet, javax.print.attribute.standard.JobStateReasons

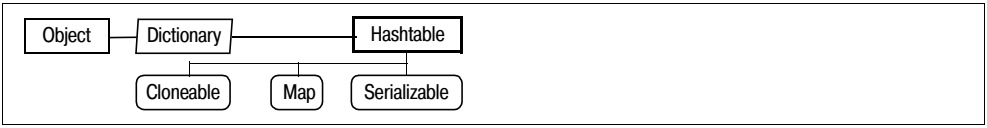
Hashtable

Java 1.0

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс реализует структуру данных хеш-таблицы, которая отображает объекты-ключи на объекты-значения и дает возможность выполнять эффективный поиск значения, ассоциированного с данным ключом. Метод put() ассоциирует значение с ключом в Hashtable. Метод get() получает значение для указанного ключа. Метод remove() удаляет пару «ключ/значение». Методы keys() и elements() возвращают объекты Enumeration, позволяющие перемещаться по множествам ключей и значений, хранящихся в таблице. Объекты, используемые в качестве ключей в Hashtable, должны иметь полноценные методы equals() и hashCode() (версии, унаследованные от Object вполне подойдут). null не допустим в качестве ключа или значения в Hashtable.

Hashtable представляет собой широко используемый класс, являющийся частью Java API начиная с Java 1.0. В Java 1.2 он был расширен реализацией интерфейса Map, который определяет некоторую функциональность в дополнение к методам Hashtable в Java 1.0. Hashtable очень похож на класс HashMap, однако он имеет синхронизированные методы, которые делают его пригодным для использования в многопоточной среде. При этом накладные расходы увеличиваются. Если вам необходима поточная безопасность или совместимость с Java 1.0 или Java 1.1, используйте Hashtable. В противном случае применяйте HashMap.




```
public class Hashtable extends Dictionary implements Cloneable, Map, Serializable {
// Открытые конструкторы
   public Hashtable();
1.2 public Hashtable(Map t);
   public Hashtable(int initialCapacity);
   public Hashtable(int initialCapacity, float loadFactor);
// Открытые методы экземпляра
   public void clear();                        // Реализует:Map; синхронизирован
   public boolean contains(Object value);      // синхронизирован
   public boolean containsKey(Object key);     // Реализует:Map; синхронизирован
   public Object get(Object key);              // Реализует:Map; синхронизирован
   public boolean isEmpty();                   // Реализует:Map; синхронизирован; по умолчанию:true
```

```
   public Object put(Object key, Object value);              // Реализует:Map; синхронизирован
   public Object remove(Object key);                         // Реализует:Map; синхронизирован
   public int size();                                        // Реализует:Map; синхронизирован
// Методы, реализующие Map
   public void clear();                                      // синхронизирован
   public boolean containsKey(Object key);                   // синхронизирован
1.2 public boolean containsValue(Object value);
1.2 public Set entrySet();
1.2 public boolean equals(Object o);                         // синхронизирован
   public Object get(Object key);                            // синхронизирован
1.2 public int hashCode();                                   // синхронизирован
   public boolean isEmpty();                                 // синхронизирован; по умолчанию:true
1.2 public Set keySet();
   public Object put(Object key, Object value);              // синхронизирован
1.2 public void putAll(Map t);                               // синхронизирован
   public Object remove(Object key);                         // синхронизирован
   public int size();                                        // синхронизирован
1.2 public Collection values();
// Открытые методы, замещающие Dictionary
   public Enumeration elements();                            // синхронизирован
   public Enumeration keys();                                // синхронизирован
// Открытые методы, замещающие Object
   public Object clone();                                    // синхронизирован
   public String toString();                                 // синхронизирован
// Защищенные методы экземпляра
   protected void rehash();
}
```

Подклассы: Properties, javax.swing.UIDefaults

Передается методам: **Методов слишком много, чтобы их перечислить.**

Возвращается методами: javax.naming.CannotProceedException.getEnvironment(), javax.naming.Context.getEnvironment(), javax.naming.InitialContext.getEnvironment(), javax.swing.JLayeredPane.getComponentToLayer(), javax.swing.JSlider.createStandardLabels()

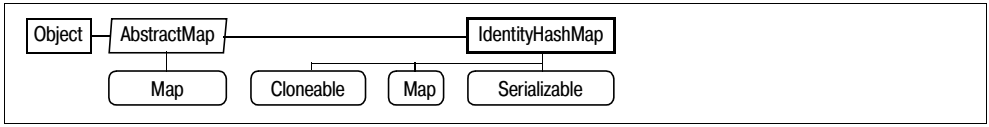
Экземпляры: java.awt.GridBagLayout.comptable, java.text.RuleBasedBreakIterator.Builder.expressions, javax.naming.CannotProceedException.environment, javax.naming.InitialContext.myProps, javax.swing.JTable.{defaultEditorsByColumnClass, defaultRenderersByColumnClass}, javax.swing.plaf.basic.BasicFileChooserUI.BasicFileView.iconCache, javax.swing.plaf.basic.BasicTreeUI.drawingCache, javax.swing.text.html.parser.DTD.{elementHash, entityHash}, javax.swing.undo.StateEdit.{postState, preState}

IdentityHashMap Java 1.4

java.util ***клонируемый, сериализуемый, коллекция***

Эта реализация Map имеет API, похожий на HashMap, и использует внутреннюю хеш-таблицу, подобно HashMap. Однако у них есть очень важное отличие. Если два ключа проверяются на совпадение, то HashMap, LinkedHashMap и TreeMap используют метод equals() для сравнения содержимого или состояния двух объектов. IdentityHashMap отличается в этом плане: он использует оператор == при определении того, идентичны ли два объекта-ключа, то есть являются ли они одним и тем же объектом. Это единственное отличие в проверке эквивалентности ключей; оно вызывает появление глубоких различий в поведении Map. Проверка эквивалентности в HashMap, LinkedHashMap или TreeMap подходит в большинстве случаев, и вы должны использовать один из этих

классов. Однако для определенных задач требуется проверка идентичности, как в IdentityHashMap.




```
public class IdentityHashMap extends AbstractMap implements Cloneable, Map, Serializable {
// Открытые конструкторы
   public IdentityHashMap();
   public IdentityHashMap(int expectedMaxSize);
   public IdentityHashMap(Map m);
// Методы, реализующие Map
   public void clear();
   public boolean containsKey(Object key);
   public boolean containsValue(Object value);
   public Set entrySet();
   public boolean equals(Object o);
   public Object get(Object key);
   public int hashCode();
   public boolean isEmpty();                                    // по умолчанию: true
   public Set keySet();
   public Object put(Object key, Object value);
   public void putAll(Map t);
   public Object remove(Object key);
   public int size();
   public Collection values();
// Открытые методы, замещающие AbstractMap
   public Object clone();
}
```

Iterator

Java 1.2

java.util

Этот интерфейс определяет методы для перебора или перечисления элементов коллекции. Метод hasNext() возвращает true, если существуют еще элементы, которые можно перечислять, или false, если все элементы были возвращены. Метод next() возвращает следующий элемент. Эти два метода облегчают создание циклов через итератор с использованием следующего кода:

```
for(Iterator i = c.iterator(); i.hasNext(); )
  processObject(i.next());
```

Интерфейс Iterator очень похож на интерфейс Enumeration. В Java 1.2 предпочтительнее использовать Iterator, а не Enumeration, потому что Iterator предоставляет хорошо определенный способ для безопасного удаления элементов из коллекции в процессе перебора. Из коллекции, в которой производится перебор, метод remove() удаляет объект, возвращенный при последнем вызове метода next(). Однако поддержка remove() является необязательной; если Iterator не поддерживает remove(), он генерирует java.lang.UnsupportedOperationException при вызове этого метода. При выполнении перебора в коллекции вы можете изменять коллекцию только через вызов метода remove() в Iterator. Если во время перебора коллекция изменится любым другим спосо-

бом, Iterator может потерпеть неудачу при попытке выполнить операцию. Кроме того, он может генерировать ConcurrentModificationException.

```
public interface Iterator {
// Открытые методы экземпляра
   public abstract boolean hasNext();
   public abstract Object next();
   public abstract void remove();
}
```

Реализации: java.beans.beancontext.BeanContextSupport.BCSIterator, ListIterator

Передается методам: javax.imageio.ImageReader.{getDestination(), readAll()}, javax.imageio.spi.ServiceRegistry.{registerServiceProviders(), ServiceRegistry()}

Возвращается методами: Методов слишком много, чтобы их перечислить.

LinkedHashMap

Java 1.4

java.util ***клонируемый, сериализуемый, коллекция***

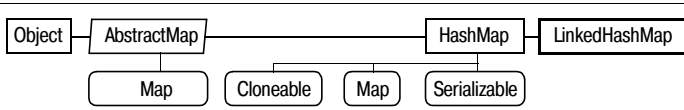
Этот класс, подобно его родительскому классу HashMap, является реализацией Map, базирующейся на хеш-таблице. Он не определяет новых открытых методов; его можно применять так же, как HashMap. В этой реализации Map уникально то, что в дополнение к структуре данных в виде хеш-таблицы она применяет двусвязный список для соединения ключей Map во внутренний список, определяющий предсказуемый порядок перебора.

Вы можете осуществлять перебор по ключам или значениям LinkedHashMap, вызывая методы entrySet(), keySet() или values(), а затем получая Iterator для возвращенной коллекции, как в HashMap. Однако в этом случае ключи и/или значения возвращаются в хорошо определенном порядке, а не в случайном порядке, предоставляемом HashMap. Порядком, принятым по умолчанию для LinkedHashMap, является порядок вставки ключа: ключу, который первым вставлен в Map, присваивается первый номер (как и значению, ассоциированному с ним), а элементу, вставленному последним, присваивается последний номер. Обратите внимание, что на этот порядок не влияют повторные вставки. Так, если LinkedHashMap содержит проекцию ключа k на значение v1, а вы вызываете метод put(), чтобы отобразить k на новое значение v2, то такая операция не изменит порядок вставки или порядок перебора ключа k. Порядок перебора для значения в таблице – это порядок перебора для ключа, с которым это значение ассоциировано.

Порядок вставки является порядком перебора по умолчанию для этого класса, но если при создании экземпляра LinkedHashMap используется конструктор с тремя аргументами, а в качестве третьего аргумента передается true, то порядок перебора будет основываться на порядке доступа: первым ключом, возвращаемым итератором, будет ключ, который дольше всех не участвовал в операции get() или put(). Последним возвращенным ключом будет ключ, который использовался последним. Как и в случае с порядком вставки, коллекция values() перебирается в порядке, определенном ключами, с которыми эти значения ассоциированы.

«Порядок при доступе» особенно удобен при реализации кэша LRU, из которого периодически вычищаются дольше всех не использовавшиеся элементы. Для упрощения этой операции в классе LinkedHashMap определен защищенный метод removeEldestEntry(). При каждом вызове метода put() (или для каждой проекции, добавляемой методом putAll()), LinkedHashMap вызывает removeEldestEntry() и передает объект Map.Entry, который дольше всех не используется или вставлен первым, если выбран порядок

при вставке. Если метод возвращает `true`, то этот элемент будет удален из проекции. В `LinkedHashMap` метод `removeEldestEntry()` всегда возвращает `false`; старые элементы никогда не удаляются автоматически, но вы можете изменить это поведение в подклассе. Решение об удалении старого элемента может основываться на содержимом элемента, а в более простом случае – на размере `LinkedHashMap`, который можно узнать, вызвав метод `size()`. Обратите внимание, что от метода `removeEldestEntry()` требуется просто вернуть `true` или `false`; он не должен удалять элемент самостоятельно.



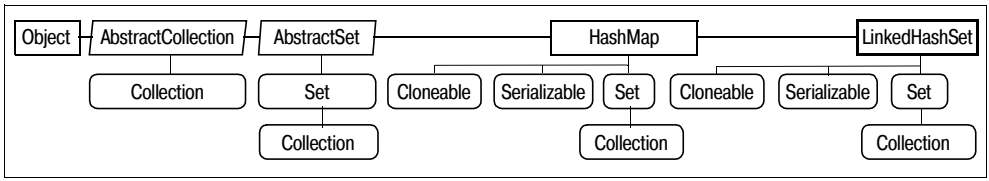

```
public class LinkedHashMap extends HashMap {
// Открытые конструкторы
   public LinkedHashMap();
   public LinkedHashMap(int initialCapacity);
   public LinkedHashMap(Map m);
   public LinkedHashMap(int initialCapacity, float loadFactor);
   public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder);
// Открытые методы, замещающие HashMap
   public void clear();
   public boolean containsValue(Object value);
   public Object get(Object key);
// Защищенные методы экземпляра
   protected boolean removeEldestEntry(Map.Entry eldest);  // константа
}
```

LinkedHashSet

Java 1.4

java.util ***клонируемый, сериализуемый, коллекция***

Этот подкласс `HashSet` является реализацией `Set`, основанной на хеш-таблице. Он не определяет новых методов и используется так же, как и `HashSet`. В `LinkedHashSet` уникально то, что в дополнение к структуре данных в виде хеш-таблицы он использует двусвязный список для соединения элементов множества во внутренний список в соответствии с порядком их вставки. Это означает, что `Iterator`, возвращаемый унаследованным методом `iterator()`, всегда нумерует элементы множества в порядке их вставки. Элементы `HashSet` нумеруются в случайном порядке. На порядок перебора не влияет повторная вставка элементов множества. Это означает, что при попытке добавить уже существующий во множестве элемент порядок перебора множества не изменяется. Если вы удаляете элемент, а затем повторно его вставляете, то порядок вставки и, следовательно, порядок перебора изменятся.




```
public class LinkedHashSet extends HashSet implements Cloneable, Serializable, Set {
// Открытые конструкторы
```

```
    public LinkedHashSet();
    public LinkedHashSet(Collection c);
    public LinkedHashSet(int initialCapacity);
    public LinkedHashSet(int initialCapacity, float loadFactor);
}
```

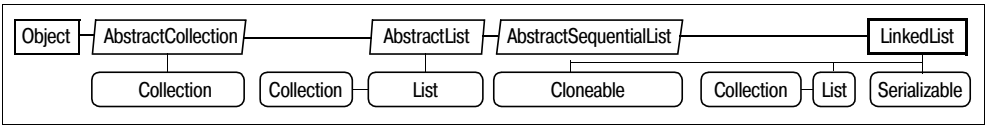
LinkedList

Java 1.2

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс реализует интерфейс `List` в терминах двусвязного списка. Он поддерживает все необязательные методы `List` и `Collection` и позволяет использовать элементы списка любого типа, включая `null`. Поскольку `LinkedList` реализован на основе структуры данных в виде связного списка, его методы `get()` и `set()` менее эффективны, чем аналогичные методы в `ArrayList`. Однако `LinkedList` может быть более эффективным при частом использовании методов `add()` и `remove()`. Методы `LinkedList` не синхронизированы. Если применять `LinkedList` в многопоточном окружении, то необходимо явно синхронизировать любой код, изменяющий список, или получить синхронизированный объект-обертку с помощью метода `Collections.synchronizedList()`.

В дополнение к методам, определенным интерфейсом `List`, `LinkedList` определяет методы для получения первого и последнего элементов списка, добавления элемента в начало или конец списка и удаления первого или последнего элементов списка. Это удобные и эффективные методы, благодаря которым `LinkedList` можно применять в качестве стека или очереди. В описании `List` и `Collection` можно найти дополнительную информацию о методах `LinkedList`. См. также `ArrayList`.




```
public class LinkedList extends AbstractSequentialList implements Cloneable, java.util.List,
Serializable {
// Открытые конструкторы
    public LinkedList();
    public LinkedList(Collection c);
// Открытые методы экземпляра
    public void addFirst(Object o);
    public void addLast(Object o);
    public Object getFirst();
    public Object getLast();
    public Object removeFirst();
    public Object removeLast();
// Методы, реализующие List
    public boolean add(Object o);
    public void add(int index, Object element);
    public boolean addAll(Collection c);
    public boolean addAll(int index, Collection c);
    public void clear();
    public boolean contains(Object o);
    public Object get(int index);
    public int indexOf(Object o);
    public int lastIndexOf(Object o);
    public ListIterator listIterator(int index);
```

```
    public boolean remove(Object o);
    public Object remove(int index);
    public Object set(int index, Object element);
    public int size();
    public Object[] toArray();
    public Object[] toArray(Object[] a);
// Открытые методы, замещающие Object
    public Object clone();
}
```

List

Java 1.2

java.util *коллекция*

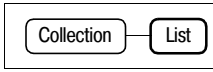
Этот интерфейс представляет упорядоченную коллекцию объектов. Каждый элемент в `List` имеет индекс (позицию) в списке; элементы можно вставлять, получать и удалять по индексу. Первый элемент в `List` имеет индекс `0`. Последний элемент списка имеет индекс, равный `size()-1`.

В дополнение к методам, определенным родительским интерфейсом `Collection`, `List` определяет большое количество методов для работы с индексированными элементами. Методы `get()` и `set()` соответственно запрашивают и устанавливают значение объекта по данному индексу. Версии методов `add()` и `addAll()`, получающие индекс в качестве аргумента, вставляют объект или коллекцию объектов по указанному индексу. Версии `add()` и `addAll()`, не получающие индекс как аргумент, вставляют объект или коллекцию объектов в конец списка. `List` определяет версию метода `remove()`, удаляющую объект по указанному индексу.

Метод `iterator()` похож на метод `iterator()` в `Collection`, за исключением того, что `Iterator`, который он возвращает, гарантирует упорядоченное перечисление элементов `List`. Метод `listIterator()` возвращает объект `ListIterator`, который является более мощным, чем обычный `Iterator`, и дает возможность изменять список в процессе перебора. `listIterator()` может принимать в качестве аргумента индекс, который указывает, в каком месте списка должен начаться перебор.

Методы `indexOf()` и `lastIndexOf()` выполняют линейный поиск указанного объекта от начала и конца списка соответственно. Каждый из этих методов возвращает индекс первого подходящего объекта или –1, если совпадений не обнаружено. И наконец, метод `subList()` возвращает `List`, содержащий только элементы списка в указанном непрерывном диапазоне. Возвращаемый список является просто представлением оригинального списка, поэтому изменения в оригинальном списке видны в возвращенном. Этот метод `subList()` особенно удобен, если нужно сортировать, искать, очищать область большого списка или каким-либо другим способом манипулировать ею.

Интерфейс не может описывать конструкторы, но по соглашению все реализации `List` предоставляют как минимум два стандартных конструктора: конструктор, не принимающий аргументов и создающий пустой список, и копирующий конструктор, принимающий произвольный объект `Collection`, определяющий начальное содержимое нового `List`. Как и в случае с `Collection`, все методы `List`, изменяющие содержимое списка, являются необязательными, а реализации, не поддерживающие их, просто генерируют `java.lang.UnsupportedOperationException`. Эффективность реализаций `List` может существенно различаться. Например, методы `get()` и `set()` в `ArrayList` гораздо эффективнее, чем в `LinkedList`. С другой стороны, методы `add()` и `remove()` в `LinkedList` могут быть более эффективными, чем такие же методы в `ArrayList`. См. также `Collection`, `Set`, `Map`, `ArrayList` и `LinkedList`.




```
public interface List extends Collection {
// Открытые методы экземпляра
   public abstract boolean add(Object o);
   public abstract void add(int index, Object element);
   public abstract boolean addAll(Collection c);
   public abstract boolean addAll(int index, Collection c);
   public abstract void clear();
   public abstract boolean contains(Object o);
   public abstract boolean containsAll(Collection c);
   public abstract boolean equals(Object o);
   public abstract Object get(int index);
   public abstract int hashCode();
   public abstract int indexOf(Object o);
   public abstract boolean isEmpty();
   public abstract Iterator iterator();
   public abstract int lastIndexOf(Object o);
   public abstract ListIterator listIterator();
   public abstract ListIterator listIterator(int index);
   public abstract boolean remove(Object o);
   public abstract Object remove(int index);
   public abstract boolean removeAll(Collection c);
   public abstract boolean retainAll(Collection c);
   public abstract Object set(int index, Object element);
   public abstract int size();
   public abstract java.util.List subList(int fromIndex, int toIndex);
   public abstract Object[] toArray();
   public abstract Object[] toArray(Object[] a);
}
```

Реализации: AbstractList, ArrayList, LinkedList, Vector

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: Collections.EMPTY_LIST, javax.imageio.IIOImage.thumbnails, javax.imageio.ImageReader.{progressListeners, updateListeners, warningListeners, warningLocales}, javax.imageio.ImageWriter.{progressListeners, warningListeners, warningLocales}

ListIterator Java 1.2

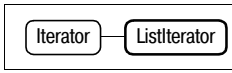
java.util

Этот интерфейс является расширением интерфейса Iterator для использования с упорядоченными коллекциями. Он определяет методы для прохождения по списку в прямом и обратном направлении и методы для определения индекса элементов. Кроме того, в нем определены методы для изменяемых списков и методы для безопасной вставки, удаления и редактирования элементов списка в процессе перебора. Для некоторых списков, в том числе для LinkedList, использование итератора для перечисления элементов списка может быть значительно более эффективно, чем перемещение по списку в цикле с повторяющимся вызовом get().

Методы hasNext() и next() – это наиболее широко используемые методы ListIterator; они выполняют перебор по списку в прямом направлении. Для получения дополни-

тельной информации обратитесь к описанию Iterator. В дополнение к этим двум методам ListIterator также определяет методы hasPrevious() и previous(), позволяющие выполнять перебор по списку в обратном направлении. Метод previous() возвращает предыдущий элемент списка или генерирует NoSuchElementException, если такого элемента нет. Метод hasPrevious() возвращает true, если последующий вызов previous() возвращает объект. Методы nextIndex() и previousIndex() возвращают индекс объекта, который будет возвращен последующим вызовом next() или previous(). Если next() или previous() генерирует NoSuchElementException, то метод nextIndex() возвращает размер списка, а previousIndex() возвращает –1.

ListIterator определяет три необязательных метода, предоставляющих безопасный способ изменения содержимого обрабатываемого списка в процессе перебора. Метод add() вставляет новый объект в список непосредственно перед объектом, который был бы возвращен последующим вызовом next(). Однако вызов add() не влияет на значение, возвращаемое next(). Если вызвать previous() непосредственно после вызова add(), то этот метод вернет только что добавленный объект. Метод remove() удаляет из списка объект, возвращенный последним при вызове метода next() или previous(). Метод remove() можно вызывать только один раз на каждый вызов next() или previous(). Если вы вызвали add(), то перед вызовом remove() нужно снова вызвать next() или previous(). Метод set() заменяет объект, возвращенный последним при вызове метода next() или previous(), указанным объектом. Если вы вызвали add() или remove(), то перед вызовом set() вы должны вызвать next() или previous(). Помните, что поддержка методов add(), remove() и set() является необязательной. Конечно, итераторы для неизменяемых списков никогда не поддерживают их. Во время вызова неподдерживаемые методы генерируют java.lang.UnsupportedOperationException. Кроме того, когда используется итератор, все изменения должны выполняться через него, а не непосредственно в списке. Если обрабатываемый список будет изменен в процессе перебора, правильная работа ListIterator будет нарушена либо он сгенерирует ConcurrentModificationException.




```
public interface ListIterator extends Iterator {
// Открытые методы экземпляра
   public abstract void add(Object o);
   public abstract boolean hasNext();
   public abstract boolean hasPrevious();
   public abstract Object next();
   public abstract int nextIndex();
   public abstract Object previous();
   public abstract int previousIndex();
   public abstract void remove();
   public abstract void set(Object o);
}
```

Возвращается методами: AbstractList.listIterator(), AbstractSequentialList.listIterator(), LinkedList.listIterator(), java.util.List.listIterator()

ListResourceBundle Java 1.1

java.util

Этот абстрактный класс предоставляет простой способ для определения ResourceBundle. Возможно, вам будет проще наследовать ListResourceBundle, чем непосредственно наследовать ResourceBundle. ListResourceBundle предоставляет реализации для абстрактных методов handleGetObject() и getKeys(), определенных в ResourceBundle, и добавляет собственный абстрактный метод getContents(), который должен замещаться потомками. Метод getContents() возвращает Object[][] – массив массивов объектов. Этот массив может иметь любое количество элементов. Каждый элемент этого массива должен представлять собой массив из двух элементов: первым элементом каждого подмассива должен быть объект String, определяющий имя ресурса, а вторым элементом – значение этого ресурса; данное значение может быть объектом любого типа. См. также ResourceBundle и PropertyResourceBundle.


```
public abstract class ListResourceBundle extends ResourceBundle {
// Открытые конструкторы
   public ListResourceBundle();
// Открытые методы, замещающие ResourceBundle
   public Enumeration getKeys();
   public final Object handleGetObject(String key);
// Защищенные методы экземпляра
   protected abstract Object[][] getContents();
}
```

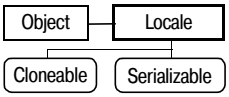
Подклассы: javax.accessibility.AccessibleResourceBundle

Locale Java 1.1

java.util ***клонируемый, сериализуемый***

Класс Locale представляет набор региональных параметров (locale): политический, географический или культурный регион, обычно имеющий уникальный язык, уникальные традиции и соглашения для дат, времени и чисел. Класс Locale определяет большое количество констант, представляющих наиболее широко используемые наборы региональных параметров. Кроме того, Locale определяет статический метод getDefault(), возвращающий объект Locale по умолчанию, который представляет регион, унаследованный от базовой системы. Метод getAvailableLocales() возвращает список всех регионов, поддерживаемых этой системой. Если ни один из методов получения объекта Locale вам не подходит, можно явно создать собственный объект Locale. Для этого необходимо указать код языка и (в необязательном порядке) – код страны и строку варианта. Методы getISOCountries() и getISOLanguages() возвращают список поддерживаемых кодов стран и языков.

Класс Locale не реализует никаких функций, имеющих отношение к интернационализации; он только предоставляет идентификатор региона тем классам, которые могут локализовать свое поведение. Получив объект Locale, вы можете вызвать разнообразные методы getDisplay для получения описания региона, которое пригодно для показа пользователю. Эти методы могут получать аргумент Locale, поэтому названия языков и стран могут быть локализованы соответствующим образом.




```
public final class Locale implements Cloneable, Serializable {
// Открытые конструкторы
1.4 public Locale(String language);
   public Locale(String language, String country);
   public Locale(String language, String country, String variant);
// Открытые константы
   public static final Locale CANADA;
   public static final Locale CANADA_FRENCH;
   public static final Locale CHINA;
   public static final Locale CHINESE;
   public static final Locale ENGLISH;
   public static final Locale FRANCE;
   public static final Locale FRENCH;
   public static final Locale GERMAN;
   public static final Locale GERMANY;
   public static final Locale ITALIAN;
   public static final Locale ITALY;
   public static final Locale JAPAN;
   public static final Locale JAPANESE;
   public static final Locale KOREA;
   public static final Locale KOREAN;
   public static final Locale PRC;
   public static final Locale SIMPLIFIED_CHINESE;
   public static final Locale TAIWAN;
   public static final Locale TRADITIONAL_CHINESE;
   public static final Locale UK;
   public static final Locale US;
// Открытые методы экземпляра
1.2 public static Locale[] getAvailableLocales();
   public static Locale getDefault();
1.2 public static String[] getISOCountries();
1.2 public static String[] getISOLanguages();
   public static void setDefault(Locale newLocale);                    // синхронизирован
// Методы доступа к свойствам (по имени свойства)
   public String getCountry();
   public final String getDisplayCountry();
   public String getDisplayCountry(Locale inLocale);
   public final String getDisplayLanguage();
   public String getDisplayLanguage(Locale inLocale);
   public final String getDisplayName();
   public String getDisplayName(Locale inLocale);
   public final String getDisplayVariant();
   public String getDisplayVariant(Locale inLocale);
   public String getISO3Country() throws MissingResourceException;
   public String getISO3Language() throws MissingResourceException;
   public String getLanguage();
   public String getVariant();
// Открытые методы, замещающие Object
   public Object clone();
   public boolean equals(Object obj);
   public int hashCode();                                              // синхронизирован
```

```
    public final String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: Полей слишком много, чтобы их перечислить.

Map

Java 1.2

java.util ***коллекция***

Этот интерфейс представляет коллекцию отображений (ассоциаций) между объектами-ключами и объектами-значениями. Примерами ассоциативных таблиц (maps) являются хеш-таблицы и ассоциативные массивы. Множество объектов-ключей в `Map` не должно содержать повторяющихся объектов; на коллекцию объектов-значений это ограничение не распространяется. Объекты-ключи должны быть неизменяемыми объектами, или необходимо позаботиться о том, чтобы они не изменялись во время нахождения в `Map`. В Java 1.2 интерфейс `Map` заменяет абстрактный класс `Dictionary`. Несмотря на то что `Map` не является коллекцией, интерфейс `Map` по-прежнему рассматривается как неотъемлемая часть структуры коллекций Java вместе с `Set`, `List` и другими классами и интерфейсами.

Вы можете добавить ассоциацию «ключ/значение» в `Map` с помощью метода `put()`. Метод `putAll()` применяется для копирования всех отображений из одного объекта `Map` в другой. Метод `get()` позволяет найти объект, ассоциированный с указанным объектом-ключом. С помощью метода `remove()` можно удалить ассоциацию между указанным ключом и его значением, а метод `clear()` позволяет удалить все ассоциации из `Map`. Метод `size()` возвращает количество ассоциаций в `Map`, а `isEmpty()` проверяет, содержит ли `Map` хотя бы одну ассоциацию. Метод `containsKey()` проверяет наличие в `Map` указанного объекта-ключа, а `containsValue()` проверяет, содержит ли он заданное значение. (Однако для большинства реализаций метод `containsValue()` является гораздо более дорогой операцией, чем `containsKey()`). Метод `keySet()` возвращает множество объектов-ключей из `Map` в виде `Set`. Метод `values()` возвращает множество всех объектов-значений из отображения в виде `Collection`, а не в виде `Set`, поскольку это множество может содержать повторяющиеся элементы. Метод `entrySet()` возвращает множество всех отображений из `Map` в виде `Set`. Элементы возвращенного множества `Set` являются объектами `Map.Entry`. Коллекции, возвращаемые методами `values()`, `keySet()` и `entrySet()`, базируются на исходном объекте `Map`, поэтому изменения, вносимые в `Map`, отражаются на этих коллекциях.

Интерфейс не может описывать конструкторы, но в соответствии с соглашением все реализации `Map` предоставляют как минимум два стандартных конструктора: конструктор, не принимающий аргументов и создающий пустую таблицу, и копирующий конструктор, который принимает объект `Map`, определяющий начальное содержимое нового объекта `Map`.

Реализации должны поддерживать все методы, запрашивающие содержимое `Map`, но поддержка методов, изменяющих содержимое, является необязательной. Если реализация не поддерживает конкретный метод, то вызов этого метода просто генерирует `ajava.lang.UnsupportedOperationException`. **См. также** `Collection`, `Set`, `List`, `HashMap`, `Hashtable`, `WeakHashMap`, `SortedMap` и `TreeMap`.

```
public interface Map {
// Внутренние классы
```

```
   public static class Entry;
// Открытые методы экземпляра
   public abstract void clear();
   public abstract boolean containsKey(Object key);
   public abstract boolean containsValue(Object value);
   public abstract Set entrySet();
   public abstract boolean equals(Object o);
   public abstract Object get(Object key);
   public abstract int hashCode();
   public abstract boolean isEmpty();
   public abstract Set keySet();
   public abstract Object put(Object key, Object value);
   public abstract void putAll(Map t);
   public abstract Object remove(Object key);
   public abstract int size();
   public abstract Collection values();
}
```

Реализации: java.awt.RenderingHints, AbstractMap, HashMap, Hashtable, IdentityHashMap, SortedMap, WeakHashMap, java.util.jar.Attributes

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: java.awt.Toolkit.desktopProperties, Collections.EMPTY_MAP, java.util.jar.Attributes.map

Map.Entry Java 1.2

java.util

Этот интерфейс представляет одиночное отображение, или ассоциацию, между объектом-ключом и объектом-значением в Map. Метод entrySet() в Map возвращает объекты Map.Entry, представляющие множество ассоциаций в виде Set. Метод iterator() этого Set применяется для перечисления объектов Map.Entry. Методы getKey() и getValue() позволяют получить объекты-ключи и объекты-значения для элемента. Метод setValue() используется для изменения значения элемента (этот метод может не поддерживаться). Данный метод генерирует java.lang.UnsupportedOperationException, если он не поддерживается реализацией.

```
public static interface Map.Entry {
// Открытые методы экземпляра
   public abstract boolean equals(Object o);
   public abstract Object getKey();
   public abstract Object getValue();
   public abstract int hashCode();
   public abstract Object setValue(Object value);
}
```

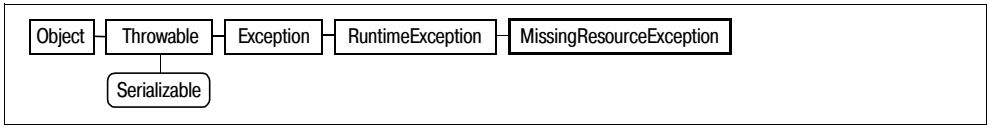
Передается методам: LinkedHashMap.removeEldestEntry()

MissingResourceException Java 1.1

java.util ***сериализуемое, непроверяемое***

Это исключение сигнализирует о том, что для выбранного набора региональных параметров не найдено ResourceBundle, или о том, что указанный ресурс не найден в дан-

ном ResourceBundle. Метод getClassName() возвращает имя класса ResourceBundle, в котором возникла исключительная ситуация, а getKey() возвращает имя ресурса, который не может быть найден.




```
public class MissingResourceException extends RuntimeException {
// Открытые конструкторы
   public MissingResourceException(String s, String className, String key);
// Открытые методы экземпляра
   public String getClassName();
   public String getKey();
}
```

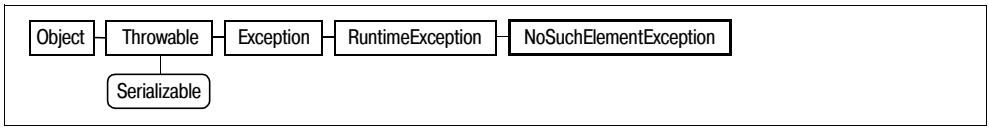
Генерируется методами: Locale.{getISO3Country(), getISO3Language()}

NoSuchElementException

Java 1.0

java.util ***сериализуемое, непроверяемое***

Это исключение сигнализирует о том, что в объекте (таком как Vector) нет элементов, или о том, что в объекте (например Enumeration) больше нет элементов, которые можно перечислить.




```
public class NoSuchElementException extends RuntimeException {
// Открытые конструкторы
   public NoSuchElementException();
   public NoSuchElementException(String s);
}
```

Observable

Java 1.0

java.util

Этот класс является родительским классом для классов, которые хотят оповещать об изменении состояния объекты Observer. Observer, который должен получать оповещения, можно зарегистрировать, передав его в метод addObserver() объекта Observable, а для удаления Observer нужно передать его в метод deleteObserver(). С помощью метода deleteObservers() можно удалить всех наблюдателей, зарегистрированных в Observable, а метод countObservers() позволяет узнать, сколько наблюдателей было добавлено. Обратите внимание, что не существует метода, перечисляющего отдельные объекты Observer, которые были добавлены.

Подкласс Observable должен вызывать защищенный метод setChanged(), когда его состояние изменяется каким-либо образом. Этот вызов устанавливает флаг «состояние изменено». После операции или серии операций, которые могут вызвать изменение

состояния, подкласс Observable должен вызвать метод notifyObservers(), передавая ему произвольный аргумент Object (передача этого аргумента необязательна). Если флаг «состояние изменено» установлен, метод notifyObservers() вызывает метод update() каждого зарегистрированного Observer (в произвольном порядке), передавая объект Observable и необязательный аргумент, если он есть. После вызова метода update() всех Observable метод notifyObservers() вызывает метод clearChanged(), чтобы очистить флаг «состояние изменено». Если метод notifyObservers() вызывается, когда флаг «состояние изменено» не установлен, то он не выполняет никаких действий. Метод hasChanged() можно применять для запроса текущего состояния этого флага.

Класс Observable и интерфейс Observer не применяются широко. Большинство приложений предпочитают событийно-ориентированную модель оповещений, определенную структурой компонентов JavaBeans, а также классом EventObject и интерфейсом EventListener этого пакета.

```
public class Observable {
// Открытые конструкторы
   public Observable();
// Открытые методы экземпляра
   public void addObserver(Observer o);                    // синхронизирован
   public int countObservers();                            // синхронизирован
   public void deleteObserver(Observer o);                 // синхронизирован
   public void deleteObservers();                          // синхронизирован
   public boolean hasChanged();                            // синхронизирован
   public void notifyObservers();
   public void notifyObservers(Object arg);
// Защищенные методы экземпляра
   protected void clearChanged();                          // синхронизирован
   protected void setChanged();                            // синхронизирован
}
```

Передается методам: Observer.update()

Observer Java 1.0

java.util

Этот интерфейс определяет метод update(), необходимый объекту для наблюдения за подклассами Observable. Observer объявляет о своей заинтересованности в объекте Observable, вызывая метод addObserver() этого объекта. Методы update() объектов Observer, зарегистрированных таким образом, будут вызываться объектом Observable после изменения этого объекта.

Этот интерфейс подобен интерфейсу EventListener и его разнообразным подынтерфейсам, специфичным для конкретных событий, но он используется менее широко.

```
public interface Observer {
// Открытые методы экземпляра
   public abstract void update(Observable o, Object arg);
}
```

Передается методам: Observable.{addObserver(), deleteObserver()}

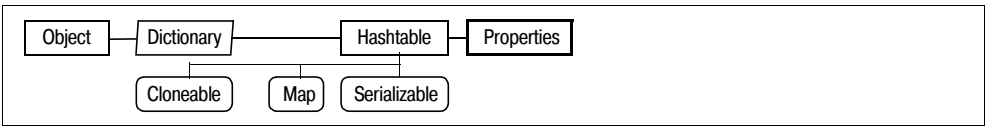
Properties Java 1.0

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс является расширением Hashtable, позволяющим читать пары «ключ/значение» из потока и записывать их в поток. Класс Properties реализует список системных свойств, который поддерживает пользовательские настройки, предоставляя программам возможность получать значения указанных ресурсов. Благодаря тому что методы load() и store() предоставляют простой способ чтения свойств из текстового потока и записи свойств в поток, этот класс предоставляет удобный способ реализации конфигурационного файла приложения.

При создании объекта Properties можно указать другой объект Properties, содержащий значения по умолчанию. Ключи (имена свойств) и значения ассоциируются в объекте Properties с помощью метода put() из Hashtable. Значения получаются с помощью метода getProperty(); если этот метод не находит ключ в текущем объекте Properties, он продолжает поиск в объекте Properties по умолчанию, переданном в конструктор. Кроме того, значение по умолчанию может быть задано для случая, когда ключ вообще не найден. Метод setProperty() позволяет добавлять пары «имя/значение» свойства в объект Properties. Этот метод, появившийся в Java 1.2, предпочтительнее унаследованного метода put(), поскольку он накладывает ограничение на имена свойств и их значения – они должны быть строковыми значениями.

Метод propertyNames() возвращает список всех имен свойств (ключей), хранящихся в объекте Properties и (рекурсивно) всех имен свойств, хранящихся в ассоциированном с ним объекте Properties по умолчанию. Метод list() выводит свойства, хранящиеся в объекте Properties. Это может быть полезно для отладки. Метод store() помещает объект Properties в поток, записывая каждое свойство в строку в формате имя=значение. Начиная с Java 1.2 store() предпочтительнее устаревшего метода save(), который записывает свойства таким же образом, но подавляет любые исключения ввода-вывода, которые могут возникнуть в процессе записи. Второй аргумент методов store() и save() – это комментарий, записываемый в начале файла свойств. И наконец, load() считывает пары «ключ/значение» из потока и сохраняет их в объекте Properties. Он подходит для чтения свойств, записанных с помощью метода store(), и свойств, отредактированных вручную.




```
public class Properties extends Hashtable {
// Открытые конструкторы
   public Properties();
   public Properties(Properties defaults);
// Открытые методы экземпляра
   public String getProperty(String key);
   public String getProperty(String key, String defaultValue);
1.1 public void list(java.io.PrintWriter out);
   public void list(java.io.PrintStream out);
   public void load(java.io.InputStream inStream) throws java.io.IOException;  // синхронизирован
   public Enumeration propertyNames();
1.2 public Object setProperty(String key, String value);                       // синхронизирован
1.2 public void store(java.io.OutputStream out, String header) throws java.io.IOException;
                                                                               // синхронизирован
```

```
// Защищенные поля экземпляра
   protected Properties defaults;
// Устаревшие открытые методы
# public void save(java.io.OutputStream out, String header);                    // синхронизирован
}
```

Подклассы: java.security.Provider

Передается методам: java.awt.Toolkit.getPrintJob(), System.setProperties(), java.rmi.activation.ActivationGroupDesc.ActivationGroupDesc(), java.sql.Driver.{connect(), getPropertyInfo()}, java.sql.DriverManager.getConnection(), Properties.Properties(), javax.naming.CompoundName.CompoundName(), javax.xml.transform.Transformer.setOutputProperties(), org.omg.CORBA.ORB.{init(), set_parameters()}

Возвращается методами: System.getProperties(), java.rmi.activation.ActivationGroupDesc.getPropertyOverrides(), javax.xml.transform.Templates.getOutputProperties(), javax.xml.transform.Transformer.getOutputProperties()

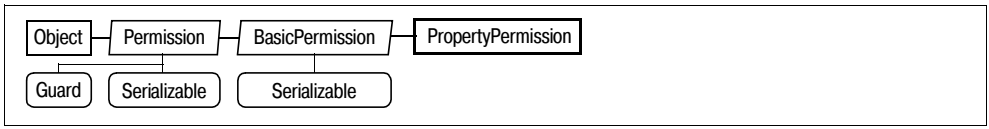
Экземпляры: Properties.defaults, javax.naming.CompoundName.mySyntax

PropertyPermission Java 1.2

java.util ***сериализуемый***

Данный класс представляет собой java.security.Permission, который управляет доступом к системным свойствам на чтение и запись, выполняемым с помощью методов System.getProperty() и System.setProperty(). Объект PropertyPermission имеет имя (целевой объект) и список действий, разделенный запятыми. Имя права (permission) – это имя выбранного свойства. Строка действия может принимать значение «read» для доступа с помощью getProperty(), «write» для доступа с помощью setProperty() или «read,write» для обоих типов доступа. PropertyPermission расширяет java.security.BasicPermission, поэтому имя свойства поддерживает простые групповые символы. Имя «*» представляет имя любого свойства. Если имя заканчивается на «.*», оно представляет имена любых свойств, использующих указанный префикс. Например, имя «java.*» представляет «java.version», «java.vendor», «java.vendor.url» и все другие свойства, начинающиеся с «java».

Предоставление доступа к системным свойствам не слишком опасно, но осторожность по-прежнему необходима. Некоторые свойства, например «user.home», раскрывают сведения о системе, которые злонамеренный код может использовать для организации атаки. Этот класс может понадобиться программистам, пишущим код системного уровня, и системным администраторам, настраивающим правила безопасности, но у приложений никогда не возникает в нем необходимости.




```
public final class PropertyPermission extends java.security.BasicPermission {
// Открытые конструкторы
   public PropertyPermission(String name, String actions);
// Открытые методы, замещающие BasicPermission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
```

```
    public boolean implies(java.security.Permission p);
    public java.security.PermissionCollection newPermissionCollection();
}
```

PropertyResourceBundle

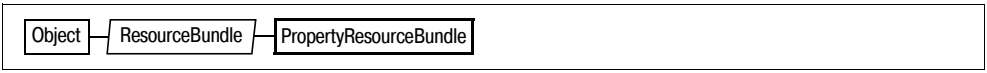
Java 1.1

java.util

Этот класс является конкретным подклассом ResourceBundle. Он читает файл Properties из указанного InputStream и реализует API ResourceBundle для извлечения указанных ресурсов из получившегося объекта Properties. Файл Properties содержит строки следующего формата:

```
имя=значение
```

Каждая такая строка определяет именованное значение с указанным значением типа String. Несмотря на то что вы можете создавать экземпляры PropertyResourceBundle самостоятельно, чаще применяется такой способ: просто определить файл Properties, а затем дать возможность ResourceBundle.getBundle() найти этот файл и вернуть необходимый объект PropertyResourceBundle. См. также Properties и ResourceBundle.




```
public class PropertyResourceBundle extends ResourceBundle {
// Открытые конструкторы
    public PropertyResourceBundle(java.io.InputStream stream) throws java.io.IOException;
// Открытые методы, замещающие ResourceBundle
    public Enumeration getKeys();
    public Object handleGetObject(String key);
}
```

Random

Java 1.0

java.util ***сериализуемый***

Этот класс реализует генератор псевдослучайных чисел, который полезен для создания игр и подобных приложений. Если вам нужен источник криптостойких псевдослучайных чисел, обратитесь к java.security.SecureRandom. Методы nextDouble() и nextFloat() возвращают значение между 0.0 и 1.0. Метод nextLong() и версия метода nextInt(), не принимающая аргументов, возвращают значения типа long и int, распределенные в диапазонах этих типов. Если в Java 1.2 передать аргумент в nextInt(), то он вернет значение между нулем (включительно) и указанным числом (исключая это число). Метод nextGaussian() возвращает псевдослучайные значения с плавающей точкой в соответствии с гауссовым распределением; среднее значение здесь – 0.0, а среднеквадратичное отклонение – 1.0. Метод nextBoolean() возвращает псевдослучайное булевое значение, а nextBytes() заполняет указанный массив байтов псевдослучайными байтами. Метод setSeed() или необязательный аргумент конструктора можно использовать для инициализации генератора псевдослучайных чисел некоторым переменным начальным значением, отличным от текущего времени (используется по умолчанию), или константой, чтобы гарантировать повторяемость псевдослучайной последовательности.


```
public class Random implements Serializable {
// Открытые конструкторы
   public Random();
   public Random(long seed);
// Открытые методы экземпляра
1.2 public boolean nextBoolean();
1.1 public void nextBytes(byte[] bytes);
   public double nextDouble();
   public float nextFloat();
   public double nextGaussian();                                    // синхронизирован
   public int nextInt();
1.2 public int nextInt(int n);
   public long nextLong();
   public void setSeed(long seed);                                  // синхронизирован
// Защищенные методы экземпляра
1.1 protected int next(int bits);                                   // синхронизирован
}
```

Подклассы: java.security.SecureRandom

Передается методам: java.math.BigInteger.{BigInteger(), probablePrime()}, Collections.shuffle()

RandomAccess

Java 1.4

java.util

Этот интерфейс-маркер образуется реализациями List, чтобы проинформировать о том, что они предоставляют эффективный произвольный доступ (обычно с постоянным временем) ко всем элементам списка. ArrayList и Vector реализуют этот интерфейс, а LinkedList – нет. Классы, обрабатывающие базовые объекты List, могут выполнить проверку наличия этого интерфейса с помощью instanceof. Они могут использовать разные алгоритмы для списков, предоставляющих эффективный произвольный доступ, и для списков, в которых более эффективен последовательный доступ.

```
public interface RandomAccess {
}
```

Реализации: ArrayList, Vector

ResourceBundle

Java 1.1

java.util

Этот абстрактный класс дает возможность подклассам определять множества локализованных ресурсов, которые можно динамически подгрузить в соответствии с потребностями интернациональных программ. Такие ресурсы могут включать текст, видимый пользователю, и изображения, показываемые в приложении, а также более сложные компоненты, например объекты Menu. Метод getBundle() применяется для загрузки подкласса ResourceBundle, подходящего для заданного региона или региона по умолчанию. Методы getObject(), getString() и getStringArray() применяются для поиска указанного ресурса в пакете. Для определения пакета предоставьте реализации методов handleGetObject() и getKeys(). Однако зачастую проще создать подкласс

ListResourceBundle или предоставить файл Properties, используемый в PropertyResourceBundle. Имя любого локализованного класса ResourceBundle, который вы определите, должно включать код языка региона и может включать код страны.

```
public abstract class ResourceBundle {
// Открытые конструкторы
   public ResourceBundle();
// Открытые методы экземпляра
   public static final ResourceBundle getBundle(String baseName);
   public static final ResourceBundle getBundle(String baseName, Locale locale);
1.2 public static ResourceBundle getBundle(String baseName, Locale locale, ClassLoader loader);
// Открытые методы экземпляра
   public abstract Enumeration getKeys();
1.2 public Locale getLocale();
   public final Object getObject(String key);
   public final String getString(String key);
   public final String[] getStringArray(String key);
// Защищенные методы экземпляра
   protected abstract Object handleGetObject(String key);
   protected void setParent(ResourceBundle parent);
// Защищенные поля экземпляра
   protected ResourceBundle parent;
}
```

Подклассы: ListResourceBundle, PropertyResourceBundle

Передается методам: java.awt.ComponentOrientation.getOrientation(), java.awt.Window.applyResourceBundle(), ResourceBundle.setParent(), java.util.logging.LogRecord.setResourceBundle()

Возвращается методами: ResourceBundle.getBundle(), java.util.logging.Logger.getResourceBundle(), java.util.logging.LogRecord.getResourceBundle()

Экземпляры: ResourceBundle.parent

Set

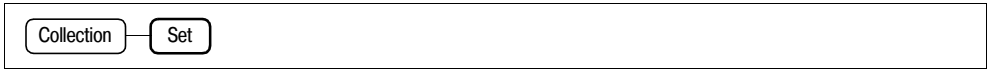
Java 1.2

java.util ***коллекция***

Этот интерфейс представляет неупорядоченную коллекцию объектов, которая не содержит повторяющихся элементов. Это означает, что Set не может содержать двух элементов e1 и e2, для которых e1.equals(e2). В нем может быть не более одного элемента null. Интерфейс Set определяет такие же методы, как и его родительский интерфейс Collection. Он ограничивает методы add() и addAll(), не позволяя добавлять повторяющиеся элементы в Set.

Интерфейс не может описывать конструкторы, но по соглашению все реализации Set предоставляют как минимум два стандартных конструктора: конструктор, не принимающий аргументов и создающий пустое множество, и копирующий конструктор, который принимает объект Collection, определяющий начальное содержимое нового Set. Конечно, копирующий конструктор должен не допускать того, чтобы повторяющиеся элементы были добавлены в Set.

Как и в случае с Collection, методы Set, изменяющие содержимое множества, являются необязательными, а реализации, не поддерживающие эти методы, просто генерируют java.lang.UnsupportedOperationException. См. также Collection, List, Map, SortedSet, HashSet и TreeSet.




```
public interface Set extends Collection {
// Открытые методы экземпляра
   public abstract boolean add(Object o);
   public abstract boolean addAll(Collection c);
   public abstract void clear();
   public abstract boolean contains(Object o);
   public abstract boolean containsAll(Collection c);
   public abstract boolean equals(Object o);
   public abstract int hashCode();
   public abstract boolean isEmpty();
   public abstract Iterator iterator();
   public abstract boolean remove(Object o);
   public abstract boolean removeAll(Collection c);
   public abstract boolean retainAll(Collection c);
   public abstract int size();
   public abstract Object[] toArray();
   public abstract Object[] toArray(Object[] a);
}
```

Реализации: AbstractSet, HashSet, LinkedHashSet, SortedSet

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

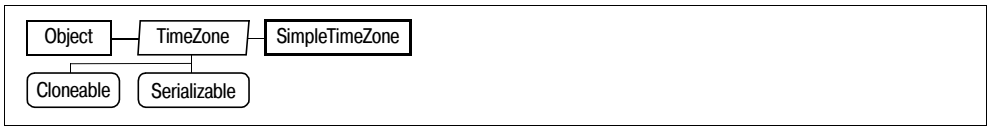
Экземпляры: Collections.EMPTY_SET

SimpleTimeZone

Java 1.1

java.util ***клонируемый, сериализуемый***

Этот конкретный подкласс TimeZone, который является простой реализацией абстрактного класса, пригодной для регионов, использующих григорианский календарь. Обычно программам не нужно создавать экземпляры этого класса непосредственно; вместо этого они используют один из статических методов-фабрик TimeZone для получения подходящего подкласса TimeZone. Единственной причиной для создания экземпляров этого класса может быть использование часового пояса с нестандартными правилами перехода на летнее время. В этом можно вызвать setStartRule() и setEndRule() для указания начальной и конечной даты перехода на летнее время.




```
public class SimpleTimeZone extends TimeZone {
// Открытые конструкторы
   public SimpleTimeZone(int rawOffset, String ID);
   public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDay,
      int startDayOfWeek, int startTime, int endMonth, int endDay, int endDayOfWeek, int endTime);
1.2 public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDay,
      int startDayOfWeek, int startTime, int endMonth, int endDay, int endDayOfWeek, int endTime,
      int dstSavings);
```

```
1.4 public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDay,
      int startDayOfWeek, int startTime, int startTimeMode, int endMonth, int endDay,
      int endDayOfWeek, int endTime, int endTimeMode, int dstSavings);
// Открытые константы
1.4 public static final int STANDARD_TIME;                                    // =1
1.4 public static final int UTC_TIME;                                         // =2
1.4 public static final int WALL_TIME;                                        // =0
// Открытые методы экземпляра
1.2 public void setDSTSavings(int millisSavedDuringDST);
1.2 public void setEndRule(int endMonth, int endDay, int endTime);
   public void setEndRule(int endMonth, int endDay, int endDayOfWeek, int endTime);
1.2 public void setEndRule(int endMonth, int endDay, int endDayOfWeek, int endTime, boolean after);
1.2 public void setStartRule(int startMonth, int startDay, int startTime);
   public void setStartRule(int startMonth, int startDay, int startDayOfWeek, int startTime);
1.2 public void setStartRule(int startMonth, int startDay, int startDayOfWeek,
                             int startTime, boolean after);
   public void setStartYear(int year);
// Открытые методы, замещающие TimeZone
   public Object clone();
1.2 public int getDSTSavings();
1.4 public int getOffset(long date);
   public int getOffset(int era, int year, int month, int day, int dayOfWeek, int millis);
   public int getRawOffset();
1.2 public boolean hasSameRules(TimeZone other);
   public boolean inDaylightTime(java.util.Date date);
   public void setRawOffset(int offsetMillis);
   public boolean useDaylightTime();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();                                                     // синхронизирован
   public String toString();
}
```

SortedMap

Java 1.2

java.util ***коллекция***

Этот интерфейс представляет объект Map, который хранит множество объектов-ключей, отсортированных в определенном порядке. Как и в случае с Map, существует соглашение о том, что все реализации этого интерфейса определяют конструктор без аргументов, который применяется для создания пустой таблицы, и копирующий конструктор, получающий объект Map, который определяет начальное содержимое SortedMap. Кроме того, при создании SortedMap нужно указать объект Comparator для сортировки объектов-ключей таблицы. Если Comparator не указан, все объекты-ключи должны реализовывать интерфейс java.lang.Comparable, благодаря чему они могут быть отсортированы в соответствии с их естественным порядком. См. также Map, TreeMap и SortedSet.

Унаследованные методы keySet(), values() и entrySet() возвращают коллекции, которые можно перебирать в порядке сортировки. Методы firstKey() и lastKey() возвращают наименьшее и наибольшее значения ключей в SortedMap. Метод subMap() возвращает SortedMap, содержащий только отображения для ключей, начиная с первого указанного ключа (он включается в набор) и заканчивая вторым указанным ключом (он не включается в набор). Метод headMap() возвращает SortedMap, содержащий отображения, ключи которых меньше указанного ключа. Метод tailMap() возвращает

SortedMap, содержащий отображения, ключи которых больше указанного ключа или равны ему. Методы subMap(), headMap() и tailMap() возвращают объекты SortedMap, которые являются отображением оригинального SortedMap; любые изменения в оригинальной таблице отражаются в возвращенной таблице, и наоборот.


```
public interface SortedMap extends Map {
// Открытые методы экземпляра
   public abstract Comparator comparator();
   public abstract Object firstKey();
   public abstract SortedMap headMap(Object toKey);
   public abstract Object lastKey();
   public abstract SortedMap subMap(Object fromKey, Object toKey);
   public abstract SortedMap tailMap(Object fromKey);
}
```

Реализации: TreeMap

Передается методам: Collections.{synchronizedSortedMap(), unmodifiableSortedMap()}, TreeMap.TreeMap()

Возвращается методами: java.nio.charset.Charset.availableCharsets(), ollections.{synchronizedSortedMap(), unmodifiableSortedMap()}, SortedMap.{headMap(), subMap(), tailMap()}, TreeMap.{headMap(), subMap(), tailMap()}

SortedSet

Java 1.2

java.util ***коллекция***

Данный интерфейс представляет собой Set, который сортирует свои элементы и гарантирует, что его метод iterator() возвращает Iterator, перечисляющий элементы множества в порядке сортировки. Как и в случае с интерфейсом Set, существует соглашение, согласно которому все реализации SortedSet должны предоставлять конструктор без аргументов, создающий пустое множество, и копирующий конструктор, ожидающий объект Collection, который определяет начальное (неотсортированное) содержимое множества. Более того, при создании SortedSet нужно указать объект Comparator, сравнивающий и сортирующий элементы множества. Если Comparator не указан, все элементы должны реализовывать java.lang.Comparable, благодаря чему они могут быть отсортированы в соответствии с их естественным порядком. См. также Set, TreeSet и SortedMap.

SortedSet определяет несколько методов в дополнение к методам, унаследованным от интерфейса Set. Методы first() и last() возвращают наименьший и наибольший объекты в множестве. Метод headSet() возвращает все элементы от начала множества до указанного элемента (но не включая его). Метод tailSet() возвращает все элементы между указанным элементом (включая его) и концом множества. subSet() возвращает все элементы множества с первого указанного элемента (включая его) и до второго указанного элемента (не включая его). Обратите внимание, что все три метода возвращают SortedSet, реализованный как отображение оригинального SortedSet. Изменения в оригинальном множестве видимы через возвращенное множество, и наоборот.

```
public interface SortedSet extends Set {
// Открытые методы экземпляра
   public abstract Comparator comparator();
   public abstract Object first();
   public abstract SortedSet headSet(Object toElement);
   public abstract Object last();
   public abstract SortedSet subSet(Object fromElement, Object toElement);
   public abstract SortedSet tailSet(Object fromElement);
}
```

Реализации: TreeSet

Передается методам: Collections.{synchronizedSortedSet(), unmodifiableSortedSet()}, TreeSet.TreeSet()

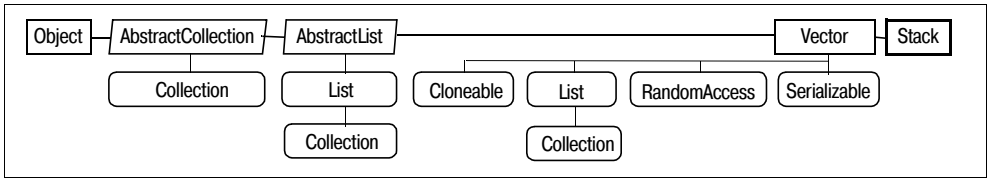
Возвращается методами: Collections.{synchronizedSortedSet(), unmodifiableSortedSet()}, SortedSet.{headSet(), subSet(), tailSet()}, TreeSet.{headSet(), subSet(), tailSet()}

Stack

Java 1.0

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс реализует стек объектов типа «последним вошел – первым обслужен» (LIFO). Метод push() помещает объект на вершину стека. Метод pop() возвращает объект с вершины стека и удаляет его из стека. Метод peek() возвращает объект с вершины стека без его удаления. В Java 1.2 в качестве стека можно задействовать объект LinkedList.




```
public class Stack extends Vector {
// Открытые конструкторы
   public Stack();
// Открытые методы экземпляра
   public boolean empty();
   public Object peek();                                    // синхронизирован
   public Object pop();                                     // синхронизирован
   public Object push(Object item);
   public int search(Object o);                             // синхронизирован
}
```

Экземпляры: java.text.RuleBasedBreakIterator.Builder.decisionPointStack

StringTokenizer

Java 1.0

java.util

Когда объект StringTokenizer создается с аргументом String, он разбивает строку на лексемы, разделяемые любым символом из указанной строки разделителей (например, слова, разделенные пробелом и символом табуляции, являются лексемами). Методы hasMoreTokens() и nextToken() получают лексемы по порядку. Метод countTokens()

возвращает количество лексем в строке. StringTokenizer реализует интерфейс Enumeration, поэтому вы можете получить доступ к лексемам с помощью знакомых методов hasMoreElements() и nextElement(). При создании StringTokenizer вы можете указать строку разделительных символов для использования с обрабатываемой строкой или сослаться на пробельные разделители по умолчанию. Кроме того, можно указать, должны ли разделители возвращаться как лексемы. И наконец, при вызове nextToken() можно задать новую строку разделительных символов.


```
public class StringTokenizer implements Enumeration {
// Открытые конструкторы
   public StringTokenizer(String str);
   public StringTokenizer(String str, String delim);
   public StringTokenizer(String str, String delim, boolean returnDelims);
// Открытые методы экземпляра
   public int countTokens();
   public boolean hasMoreTokens();
   public String nextToken();
   public String nextToken(String delim);
// Методы, реализующие Enumeration
   public boolean hasMoreElements();
   public Object nextElement();
}
```

Timer Java 1.3

java.util

Этот класс реализует таймер: его методы позволяют запланировать один или несколько запускаемых объектов TimerTask, которые должны быть выполнены (один раз или периодически) фоновым потоком в указанное время. Кроме того, вы можете создать таймер с помощью конструктора Timer(). Версия этого конструктора, не содержащая аргументов, создает обычный фоновый поток (не демон), означающий, что виртуальная машина Java не завершит свое выполнение, пока поток таймера будет выполняться. Передайте true в конструктор, если вы хотите, чтобы фоновый поток был потоком-демоном.

После создания Timer вы можете с помощью разнообразных методов schedule() и scheduleAtFixedRate() планировать запуск объектов TimerTask. Для того чтобы запланировать задачу для однократного исполнения, воспользуйтесь одним из двух методов schedule(), принимающих два аргумента, и укажите требуемое время исполнения – количество миллисекунд, которые должны пройти, или абсолютную дату (в виде объекта Date). Если количество миллисекунд равно нулю или объект Date представляет прошедшее время, то задача планируется к незамедлительному исполнению.

Чтобы запланировать повторяющуюся задачу, воспользуйтесь версиями методов schedule() или scheduleAtFixedRate(), принимающими три аргумента. Этим методам передается аргумент, указывающий время первого исполнения задачи (количество миллисекунд или объект Date), и еще один аргумент – период, указывающий количество миллисекунд между повторными исполнениями задачи. Методы schedule() планируют исполнение задачи *через* фиксированные интервалы. Это означает, что каждое исполнение планируется через интервал в миллисекундах после *окончания* пре-

дыдущего исполнения. Метод schedule() полезен для решения задач, в которых важно иметь постоянный интервал между исполнениями (например, в анимации). С другой стороны, методы scheduleAtFixedRate() планируют выполнение задач с фиксированным интервалом. Это означает, что каждое повторное выполнение задачи планируется через период в миллисекундах после *начала* предыдущего исполнения. Метод scheduleAtFixedRate() применяется для решения задач, в которых события должны происходить в конкрентные абсолютные моменты времени, а не через фиксированные интервалы (например, обновление циферблата часов).

Один объект Timer может планировать множество объектов TimerTask, однако все задачи, запланированные одним объектом Timer, разделяют один поток. Если вы запланируете большое количество повторяющихся задач или если некоторые задачи будут выполняться долго, то исполнение других задач может быть задержано.

Когда вы закончили работу с Timer, вызовите cancel(), чтобы остановить выполнение потока, ассоциированного с ним. Это очень важно, когда вы используете таймер, чей ассоциированный поток не является демоном, поскольку в противном случае поток таймера может не позволить виртуальной машине Java завершить работу. Для отмены исполнения отдельной задачи применяйте метод cancel() из TimerTask.

```
public class Timer {
// Открытые конструкторы
   public Timer();
   public Timer(boolean isDaemon);
// Открытые методы экземпляра
   public void cancel();
   public void schedule(TimerTask task, long delay);
   public void schedule(TimerTask task, java.util.Date time);
   public void schedule(TimerTask task, java.util.Date firstTime, long period);
   public void schedule(TimerTask task, long delay, long period);
   public void scheduleAtFixedRate(TimerTask task, java.util.Date firstTime, long period);
   public void scheduleAtFixedRate(TimerTask task, long delay, long period);
}
```

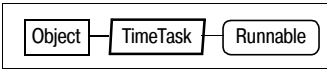
TimerTask

Java 1.3

java.util ***запускаемый***

Этот абстрактный запускаемый (Runnable) класс представляет задачу, которая планируется объектом Timer для одного или повторяющихся исполнений. Вы можете определить задачу, создав подкласс TimerTask и реализовав абстрактный метод run(). Чтобы запланировать будущее выполнение задачи, передайте экземпляр вашего подкласса в один из методов объекта Timer — schedule() или scheduleAtFixedRate(). Затем объект Timer вызовет метод run() в запланированное время.

Вызовите cancel() для отмены однократного или повторяющегося исполнения TimerTask. Этот метод возвращает true, если исполнение было действительно отменено. Он возвращает false, если задача уже была отменена, никогда не была запланирована или была запланирована для однократного исполнения и уже выполнена. Метод scheduledExecutionTime() возвращает время в миллисекундах, на которое было запланировано ближайшее исполнение TimerTask. Когда система сильно загружена, метод run() может не быть вызван в точно запланированное время. Некоторые задачи могут не выполнять никаких действий, если они были вызваны не вовремя. Метод run() может сравнивать возвращаемые значения scheduledExecutionTime() и System.currentTimeMillis(), чтобы определить, был ли текущий вызов выполнен вовремя.




```
public abstract class TimerTask implements Runnable {
// Защищенные конструкторы
   protected TimerTask();
// Открытые методы экземпляра
   public boolean cancel();
   public long scheduledExecutionTime();
// Методы, реализующие Runnable
   public abstract void run();
}
```

Передается методам: java.util.Timer.{schedule(), scheduleAtFixedRate()}

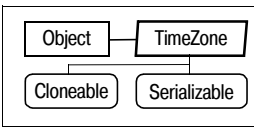
TimeZone

Java 1.1

java.util ***клонируемый, сериализуемый***

Класс TimeZone **представляет часовой пояс; он используется с классами** Calendar **и** DateFormat**. Как у абстрактного класса, экземпляры** TimeZone **не могут быть созданы напрямую. Вместо этого необходимо вызвать статический метод** getDefault() **для получения объекта** TimeZone**, который представляет часовой пояс, унаследованный от основной операционной системы. Кроме того, можно вызвать статический метод** getTimeZone() **с именем требуемого пояса. Список имен поддерживаемых часовых поясов можно получить с помощью статического метода** getAvailableIDs()**.**

После получения объекта TimeZone **вы можете вызвать** inDaylightTime()**, чтобы определить, в каком времени, «летнем» или «зимнем», находится данный объект** Date**. Метод** getID() **позволяет получить имя часового пояса. Для заданной даты метод** getOffset() **позволяет определить количество миллисекунд, которое необходимо добавить к времени по Гринвичу (GMT), чтобы «перейти» в указанный часовой пояс.**




```
public abstract class TimeZone implements Cloneable, Serializable {
// Открытые конструкторы
   public TimeZone();
// Открытые константы
1.2 public static final int LONG;                                    // =1
1.2 public static final int SHORT;                                   // =0
// Открытые методы экземпляра
   public static String[] getAvailableIDs();                         // синхронизирован
   public static String[] getAvailableIDs(int rawOffset);            // синхронизирован
   public static TimeZone getDefault();                              // синхронизирован
   public static TimeZone getTimeZone(String ID);                    // синхронизирован
   public static void setDefault(TimeZone zone);                     // синхронизирован
// Методы доступа к свойствам (по имени свойства)
1.2 public final String getDisplayName();
1.2 public final String getDisplayName(Locale locale);
1.2 public final String getDisplayName(boolean daylight, int style);
1.2 public String getDisplayName(boolean daylight, int style, Locale locale);
```

```
1.4 public int getDSTSavings();
   public String getID();
   public void setID(String ID);
   public abstract int getRawOffset();
   public abstract void setRawOffset(int offsetMillis);
// Открытые методы экземпляра
1.4 public int getOffset(long date);
   public abstract int getOffset(int era, int year, int month, int day, int dayOfWeek,
        int milliseconds);
1.2 public boolean hasSameRules(TimeZone other);
   public abstract boolean inDaylightTime(java.util.Date date);
   public abstract boolean useDaylightTime();
// Открытые методы, замещающие Object
   public Object clone();
}
```

Подклассы: SimpleTimeZone

Передается методам: java.text.DateFormat.setTimeZone(), Calendar.{Calendar(), getInstance(), setTimeZone()}, GregorianCalendar.GregorianCalendar(), SimpleTimeZone.hasSameRules(), TimeZone.{hasSameRules(), setDefault()}

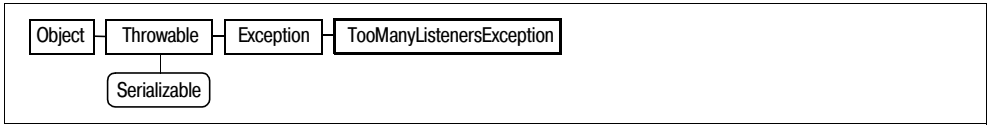
Возвращается методами: java.text.DateFormat.getTimeZone(), Calendar.getTimeZone(), TimeZone.{getDefault(), getTimeZone()}

TooManyListenersException

Java 1.1

java.util ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что компонент AWT, JavaBeans или Swing может иметь только один объект EventListener, зарегистрированный для некоторого конкретного типа событий. Оно оповещает о том, что конкретное событие является однонаправленным, а не широковещательным. В модели событий Java этот тип исключения служит совершенно определенной цели; его присутствие в выражении throws в методе регистрации EventListener (даже если метод никогда не генерирует это исключение) сигнализирует о том, что событие является однонаправленным.




```
public class TooManyListenersException extends Exception {
// Открытые конструкторы
   public TooManyListenersException();
   public TooManyListenersException(String s);
}
```

Генерируется методами: java.awt.dnd.DragGestureRecognizer.addDragGestureListener(), java.awt.dnd.DragSourceContext.addDragSourceListener(), java.awt.dnd.DropTarget.addDropTargetListener(), java.beans.beancontext.BeanContextServices.getService(), java.beans.beancontext.BeanContextServicesSupport.getService()

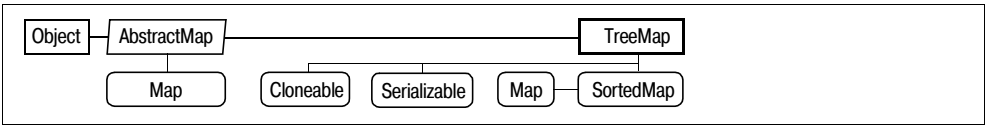
TreeMap Java 1.2

java.util *клонируемый, сериализуемый, коллекция*

Этот класс реализует интерфейс SortedMap, используя внутреннюю структуру бинарного дерева. Он гарантирует, что ключи и значения в отображении будут перечисляться по возрастанию ключей. TreeMap поддерживает все необязательные методы Map. Все объекты, используемые в качестве ключей в TreeMap, должны быть взаимно сравнимыми. В качестве варианта при создании TreeMap можно предоставить соответствующий объект Comparator. Поскольку TreeMap строится на структуре данных бинарного дерева, методы get(), put(), remove() и containsKey() выполняются за относительно небольшое время, изменяющееся по логарифмическому закону. Однако если от TreeMap не требуется возможность сортировки, то вместо TreeMap можно использовать HashMap, поскольку этот класс более эффективен. Для получения подробной информации о методах TreeMap обратитесь к описанию Map и SortedMap. См. также родственный класс TreeSet.

Чтобы TreeMap работал правильно, метод сравнения из интерфейса Comparable или Comparator должен быть совместим с методом equals(). Это означает, что метод equals() должен считать два объекта идентичными, если и только если метод сравнения тоже показывает, что эти объекты идентичны.

Методы TreeMap не синхронизированы. В многопоточной среде нужно явно синхронизировать весь код, изменяющий TreeMap, или получить синхронизированную обертку (wrapper) с помощью Collections.symchronizedMap().




```
public class TreeMap extends AbstractMap implements Cloneable,  Serializable, SortedMap {
// Открытые конструкторы
   public TreeMap();
   public TreeMap(Comparator c);
   public TreeMap(SortedMap m);
   public TreeMap(Map m);
// Методы, реализующие Map
   public void clear();
   public boolean containsKey(Object key);
   public boolean containsValue(Object value);
   public Set entrySet();
   public Object get(Object key);
   public Set keySet();
   public Object put(Object key, Object value);
   public void putAll(Map map);
   public Object remove(Object key);
   public int size();
   public Collection values();
// Методы, реализующие SortedMap
   public Comparator comparator();
   public Object firstKey();
   public SortedMap headMap(Object toKey);
   public Object lastKey();
   public SortedMap subMap(Object fromKey, Object toKey);
   public SortedMap tailMap(Object fromKey);
```

```
// Открытые методы, замещающие AbstractMap
   public Object clone();
}
```

TreeSet

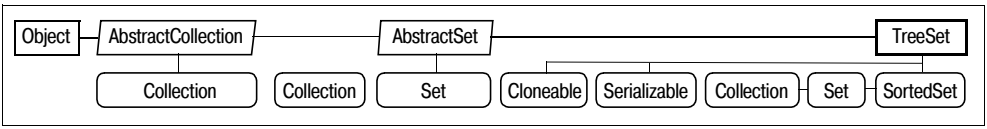
Java 1.2

java.util ***клонируемый, сериализуемый, коллекция***

Этот класс реализует SortedSet, предоставляя поддержку для всех необязательных методов и гарантируя, что элементы будут перечисляться в порядке возрастания. Чтобы элементы множества можно было сортировать, они должны быть взаимно сравнимыми объектами либо быть совместимыми с объектом Comparator, который указан при создании TreeSet. TreeSet реализован поверх TreeMap, поэтому все его методы add(), remove() и contains() выполняются за относительно небольшое время, изменяемое по логарифмическому закону. Однако если сортировка TreeSet не нужна, вместо данного класса можно применять HashSet, поскольку этот класс значительно эффективнее. Для получения подробной информации о методах TreeSet обратитесь к описанию Set, SortedSet и Collection.

Чтобы TreeSet работал правильно, метод сравнения из Comparable или Comparator должен быть совместим с методом equals(). Это значит, что метод equals() должен считать два объекта идентичными, если и только если сравнивающий метод тоже показывает, что эти объекта идентичны.

Методы TreeSet не синхронизированы. В многопоточной среде вы должны явно синхронизировать код, изменяющий содержимое множества, или получить синхронизированную обертку с помощью Collections.synchronizedSet().




```
public class TreeSet extends AbstractSet implements Cloneable,  Serializable, SortedSet {
// Открытые конструкторы
   public TreeSet();
   public TreeSet(Comparator c);
   public TreeSet(SortedSet s);
   public TreeSet(Collection c);
// Методы, реализующие Set
   public boolean add(Object o);
   public boolean addAll(Collection c);
   public void clear();
   public boolean contains(Object o);
   public boolean isEmpty();                                    // по умолчанию:true
   public Iterator iterator();
   public boolean remove(Object o);
   public int size();
// Методы, реализующие SortedSet
   public Comparator comparator();
   public Object first();
   public SortedSet headSet(Object toElement);
   public Object last();
   public SortedSet subSet(Object fromElement, Object toElement);
   public SortedSet tailSet(Object fromElement);
```

```
// Открытые методы, замещающие Object
   public Object clone();
}
```

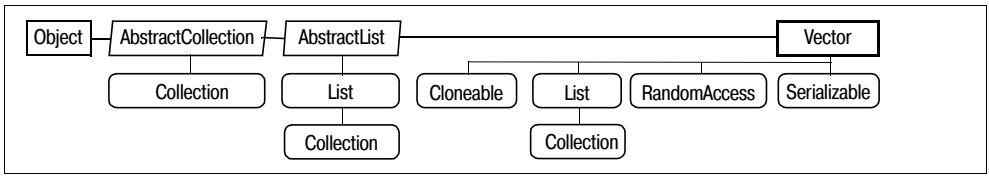
Vector

Java 1.0

java.util *клонируемый, сериализуемый, коллекция*

Этот класс реализует упорядоченную коллекцию (по существу, массив) объектов, которая при необходимости может расти или уменьшаться. `Vector` полезен, когда вам нужно отслеживать количество объектов, но заранее неизвестно, сколько их будет. Метод `setElementAt()` применяется для того, чтобы установить объект в данном индексе `Vector`. Метод `elementAt()` позволяет получить объект, хранимый по указанному индексу. `Object`, возвращенный методом `elementAt()`, обычно необходимо приводить к требуемому типу. Метод `add()` позволяет добавить объект в конец `Vector` или вставить его в любую указанную позицию. Метод `removeElementAt()` применяется для удаления элемента по заданному индексу, а метод `removeElement()` – для удаления указанного объекта из вектора. Метод `size()` возвращает количество объектов, содержащихся в `Vector` на данный момент. Метод `elements()` возвращает объект `Enumeration`, позволяющий осуществлять перебор по этим объектам. Метод `capacity()` не является аналогом `size()`; он возвращает максимальное количество объектов, которое `Vector` может хранить без изменения размера его внутреннего хранилища. `Vector` автоматически изменяет размер своего внутреннего хранилища, но если заранее известно, сколько объектов будет содержать `Vector`, вы можете увеличить его эффективность, предварительно выделив место под это количество элементов с помощью `ensureCapacity()`.

Начиная с Java 1.0 `Vector` является частью пакета `java.util`, но в Java 1.2 он был расширен реализацией интерфейса `List`. `List` определяет новые имена для большого количества методов, уже присутствующих в `Vector`; для получения подробной информации об этих методах обратитесь к описанию `List`. `Vector` похож на класс `ArrayList`, за исключением того, что методы `Vector` синхронизированы. Синхронизация делает методы потокобезопасными, но увеличивает затраты времени на их вызов. Если в многопоточной среде необходима безопасность или совместимость с Java 1.0 или Java 1.1, применяйте `Vector`; в противном случае используйте `ArrayList`.




```
public class Vector extends AbstractList implements Cloneable, java.util.List, RandomAccess,
Serializable {
// Открытые конструкторы
   public Vector();
1.2 public Vector(Collection c);
   public Vector(int initialCapacity);
   public Vector(int initialCapacity, int capacityIncrement);
// Открытые методы экземпляра
   public void addElement(Object obj);                                    // синхронизирован
   public int capacity();                                                 // синхронизирован
   public boolean contains(Object elem);                                  // Реализует:List
   public void copyInto(Object[] anArray);                                // синхронизирован
```

```
   public Object elementAt(int index);                                    // синхронизирован
   public Enumeration elements();
   public void ensureCapacity(int minCapacity);                           // синхронизирован
   public Object firstElement();                                          // синхронизирован
   public int indexOf(Object elem);                                       // Реализует:List
   public int indexOf(Object elem, int index);                            // синхронизирован
   public void insertElementAt(Object obj, int index);                    // синхронизирован
   public boolean isEmpty();              // Реализует:List; синхронизирован; по умолчанию:true
   public Object lastElement();                                           // синхронизирован
   public int lastIndexOf(Object elem);                     // Реализует:List; синхронизирован
   public int lastIndexOf(Object elem, int index);                        // синхронизирован
   public void removeAllElements();                                       // синхронизирован
   public boolean removeElement(Object obj);                              // синхронизирован
   public void removeElementAt(int index);                                // синхронизирован
   public void setElementAt(Object obj, int index);                       // синхронизирован
   public void setSize(int newSize);                                      // синхронизирован
   public int size();                                       // Реализует:List; синхронизирован
   public void trimToSize();                                              // синхронизирован
// Методы, реализующие List
1.2 public boolean add(Object o);                                         // синхронизирован
1.2 public void add(int index, Object element);
1.2 public boolean addAll(Collection c);                                  // синхронизирован
1.2 public boolean addAll(int index, Collection c);                       // синхронизирован
1.2 public void clear();
   public boolean contains(Object elem);
1.2 public boolean containsAll(Collection c);                             // синхронизирован
1.2 public boolean equals(Object o);                                      // синхронизирован
1.2 public Object get(int index);                                         // синхронизирован
1.2 public int hashCode();                                                // синхронизирован
   public int indexOf(Object elem);
   public boolean isEmpty();                                // синхронизирован; по умолчанию:true
   public int lastIndexOf(Object elem);                                   // синхронизирован
1.2 public boolean remove(Object o);
1.2 public Object remove(int index);                                      // синхронизирован
1.2 public boolean removeAll(Collection c);                               // синхронизирован
1.2 public boolean retainAll(Collection c);                               // синхронизирован
1.2 public Object set(int index, Object element);                         // синхронизирован
   public int size();                                                     // синхронизирован
1.2 public java.util.List subList(int fromIndex, int toIndex);            // синхронизирован
1.2 public Object[] toArray();                                            // синхронизирован
1.2 public Object[] toArray(Object[] a);                                  // синхронизирован
// Защищенные методы, замещающие AbstractList
1.2 protected void removeRange(int fromIndex, int toIndex);
// Открытые методы, замещающие AbstractCollection
   public String toString();                                              // синхронизирован
// Открытые методы, замещающие Object
   public Object clone();                                                 // синхронизирован
// Защищенные поля экземпляра
   protected int capacityIncrement;
   protected int elementCount;
   protected Object[] elementData;
}
```

Подклассы: Stack

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: java.awt.image.BufferedImage.getSources(), java.awt.image.RenderedImage.getSources(), java.awt.image.renderable.ParameterBlock.{getParameters(), getSources()}, java.awt.image.renderable.RenderableImage.getSources(), java.awt.image.renderable.RenderableImageOp.getSources(), javax.swing.plaf.basic.BasicDirectoryModel.{getDirectories(), getFiles()}, javax.swing.table.DefaultTableModel.{convertToVector(), getDataVector()}, javax.swing.text.GapContent.getPositionsInRange(), javax.swing.text.StringContent.getPositionsInRange()

Экземпляры: Полей слишком много, чтобы их перечислить

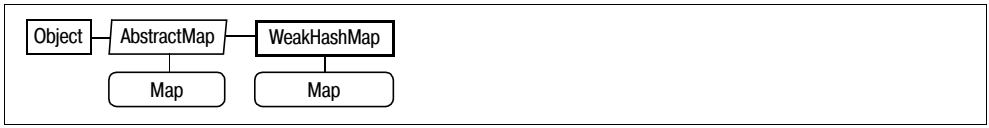
WeakHashMap

Java 1.2

java.util ***коллекция***

Этот класс реализует Map, используя внутреннюю хеш-таблицу. По возможностям и производительности он похож на HashMap, за исключением того, что WeakHashMap использует возможности пакета java.lang.ref: отображения «ключ/значение», которые он поддерживает, не препятствуют удалению объектов-ключей сборщиком мусора. Когда на объект-ключ не остается ссылок, кроме слабой (weak) ссылки, поддерживаемой WeakHashMap, сборщик мусора утилизирует объект, а WeakHashMap удаляет отображение утилизированного ключа и ассоциированного с ним значения. Если на объект-значение не остается ссылок, за исключением ссылки, поддерживаемой WeakHashMap, объект-значение также становится доступным для сборки мусора. Соответственно, WeakHashMap можно использовать для связывания дополнительного значения с объектом, не препятствуя утилизации объекта (ключа) или дополнительного значения. Возможности реализации этого класса обсуждаются в описании HashMap. Методы этого класса представлены в описании Map.

WeakHashMap пригоден для объектов, у которых методы equals() используют оператор == для выполнения сравнения. Он менее полезен для объектов-ключей типа String, поскольку в приложении может существовать множество объектов String, идентичных друг другу. Даже если значение оригинального ключа было утилизировано сборщиком мусора, всегда остается возможность передать объект String с таким же значением в метод get().




```
public class WeakHashMap extends AbstractMap implements Map {
// Открытые конструкторы
   public WeakHashMap();
   public WeakHashMap(int initialCapacity);
1.3 public WeakHashMap(Map t);
   public WeakHashMap(int initialCapacity, float loadFactor);
// Методы, реализующие Map
   public void clear();
   public boolean containsKey(Object key);
1.4 public boolean containsValue(Object value);
   public Set entrySet();
   public Object get(Object key);
   public boolean isEmpty();                                    // по умолчанию:true
```

```
1.4 public Set keySet();
   public Object put(Object key, Object value);
1.4 public void putAll(Map t);
   public Object remove(Object key);
   public int size();
1.4 public Collection values();
}
```

Пакет java.util.jar

java 1.2

Пакет java.util.jar содержит классы для чтения и записи файлов JAR (Java ARchive). Он основан на пакете java.util.zip. Файл JAR – это файл ZIP, чьим первым элементом является файл манифеста со специальным именем, который содержит атрибуты и цифровые подписи для элементов файла ZIP, следующих за ним. Многие классы этого пакета представляют собой сравнительно простые расширения классов из пакета java.util.zip.

Простейший способ прочитать файл JAR – использовать класс JarFile, обеспечивающий произвольный доступ. Данный класс позволяет получить объект JarEntry, который описывает любой указанный файл внутри архива JAR. Кроме того, он позволяет получить перечисление всех элементов архива и InputStream для чтения байтов конкретного JarEntry. Каждый JarEntry описывает единичный элемент архива и обеспечивает доступ к объекту Attributes и цифровым подписям, связанными с этим элементом. JarFile также предоставляет доступ к объекту Manifest архива JAR; данный объект содержит атрибуты для всех элементов в файле JAR. Attributes – это отображение пар «имя/значение» атрибута, а внутренний класс Attributes.Name определяет константы для разнообразных стандартных имен атрибутов.

Кроме того, файл JAR можно прочитать с помощью JarInputStream. Однако этот класс требует последовательного прочтения каждого элемента файла JAR. И наконец, с помощью объекта java.net.JarURLConnection вы можете прочитать элемент внутри файла JAR и атрибуты манифеста для этого элемента.

Коллекции

```
public class Attributes implements Cloneable, java.util.Map;
```

Другие классы

```
public static class Attributes.Name;
public class JarEntry extends java.util.zip.ZipEntry;
public class JarFile extends java.util.zip.ZipFile;
public class JarInputStream extends java.util.zip.ZipInputStream;
public class JarOutputStream extends java.util.zip.ZipOutputStream;
public class Manifest implements Cloneable;
```

Исключения

```
public class JarException extends java.util.zip.ZipException;
```

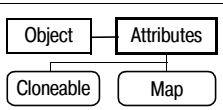
Attributes

Java 1.2

java.util.jar ***клонируемый, коллекция***

Этот класс представляет собой java.util.Map, отображающий имена атрибутов манифеста файла JAR на произвольные строковые значения. Формат манифеста JAR опре-

деляет, что имена атрибутов могут содержать только символы ASCII от A до Z (верхнего и нижнего регистров), цифры от 0 до 9, символы дефиса и подчеркивания. Соответственно, этот класс использует Attributes.Name как тип имен атрибутов в дополнение к более общему классу String. Несмотря на то что вы можете создавать собственные объекты Attributes, эти объекты, как правило, получают из объекта Manifest.




```
public class Attributes implements Cloneable, java.util.Map {
// Открытые конструкторы
   public Attributes();
   public Attributes(java.util.jar.Attributes attr);
   public Attributes(int size);
// Внутренние классы
   public static class Name;
// Открытые методы экземпляра
   public String getValue(String name);
   public String getValue(Attributes.Name name);
   public String putValue(String name, String value);
// Методы, реализующие Map
   public void clear();
   public boolean containsKey(Object name);
   public boolean containsValue(Object value);
   public java.util.Set entrySet();
   public boolean equals(Object o);
   public Object get(Object name);
   public int hashCode();
   public boolean isEmpty();  // по умолчанию: true
   public java.util.Set keySet();
   public Object put(Object name, Object value);
   public void putAll(java.util.Map attr);
   public Object remove(Object name);
   public int size();
   public java.util.Collection values();
// Открытые методы, замещающие Object
   public Object clone();
// Защищенные поля экземпляра
   protected java.util.Map map;
}
```

Передается методам: java.util.jar.Attributes.Attributes()

Возвращается методами: java.net.JarURLConnection.{getAttributes(), getMainAttributes()}, JarEntry.getAttributes(), Manifest.{getAttributes(), getMainAttributes()}

Attributes.Name Java 1.2

java.util.jar

Этот класс представляет имя атрибута в объекте Attributes. Он определяет константы для разнообразных стандартных имен атрибутов, используемых в манифестах файлов JAR. Имя атрибута может содержать только буквы из набора ASCII, цифры, а также символы дефиса и подчеркивания. Любые другие символы Unicode недопустимы.

```
public static class Attributes.Name {
// Открытые конструкторы
   public Name(String name);
// Открытые константы
   public static final Attributes.Name CLASS_PATH;
   public static final Attributes.Name CONTENT_TYPE;
1.3 public static final Attributes.Name EXTENSION_INSTALLATION;
1.3 public static final Attributes.Name EXTENSION_LIST;
1.3 public static final Attributes.Name EXTENSION_NAME;
   public static final Attributes.Name IMPLEMENTATION_TITLE;
1.3 public static final Attributes.Name IMPLEMENTATION_URL;
   public static final Attributes.Name IMPLEMENTATION_VENDOR;
1.3 public static final Attributes.Name IMPLEMENTATION_VENDOR_ID;
   public static final Attributes.Name IMPLEMENTATION_VERSION;
   public static final Attributes.Name MAIN_CLASS;
   public static final Attributes.Name MANIFEST_VERSION;
   public static final Attributes.Name SEALED;
   public static final Attributes.Name SIGNATURE_VERSION;
   public static final Attributes.Name SPECIFICATION_TITLE;
   public static final Attributes.Name SPECIFICATION_VENDOR;
   public static final Attributes.Name SPECIFICATION_VERSION;
// Открытые методы, замещающие Object
   public boolean equals(Object o);
   public int hashCode();
   public String toString();
}
```

Передается методам: `java.util.jar.Attributes.getValue()`

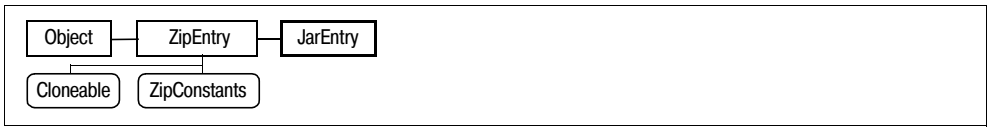
Экземпляры: Полей слишком много, чтобы их перечислить.

JarEntry

Java 1.2

java.util.jar ***клонируемый***

Этот класс расширяет `java.util.zip.ZipEntry`; он представляет единичный файл в архиве JAR, а также атрибуты манифеста и цифровые подписи, связанные с этим файлом. Объекты `JarEntry` можно прочитать из файла JAR с помощью `JarFile` или `JarInputStream` и записать в файл JAR с помощью `JarOutputStream`. Метод `getAttributes()` позволяет получить объект `Attributes` для элемента. Метод `getCertificates()` применяется для получения массива `java.security.cert.Certificate`, содержащего цепочки сертификатов для всех цифровых подписей, связанных с файлом.




```
public class JarEntry extends java.util.zip.ZipEntry {
// Открытые конструкторы
   public JarEntry(JarEntry je);
   public JarEntry(String name);
   public JarEntry(java.util.zip.ZipEntry ze);
// Открытые методы экземпляра
   public java.util.jar.Attributes getAttributes() throws java.io.IOException;
   public java.security.cert.Certificate[] getCertificates();
}
```

Передается методам: JarEntry.JarEntry()

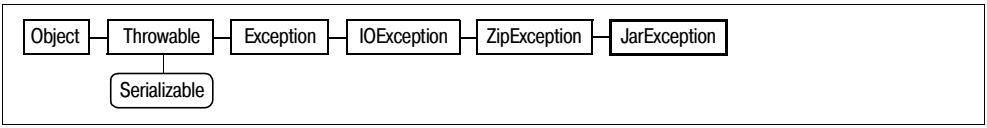
Возвращается методами: java.net.JarURLConnection.getJarEntry(), JarFile.getJarEntry(), JarInputStream.getNextJarEntry()

JarException

Java 1.2

java.util.jar ***сериализуемое, проверяемое***

Это исключение сообщает об ошибке чтения/записи в jar-файл.




```
public class JarException extends java.util.zip.ZipException {
// Открытые конструкторы
   public JarException();
   public JarException(String s);
}
```

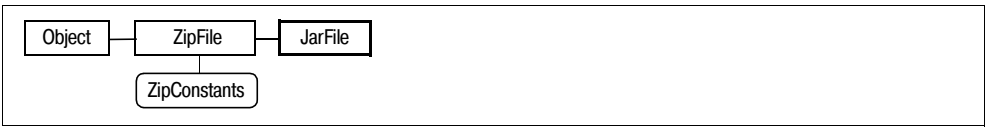
JarFile

Java 1.2

java.util.jar

Данный класс представляет файл JAR и позволяет прочитать из этого файла манифест, список файлов и отдельные файлы. Он расширяет java.util.zip.ZipFile, а использование данного класса аналогично использованию его родительского класса. JarFile можно создать, указав имя файла или объект File. Если вы не хотите, чтобы объект JarFile пытался проверять какие-либо цифровые подписи, содержащиеся в JarFile, передайте булевый аргумент со значением false в конструктор JarFile(). В Java 1.3 временные файлы JAR могут автоматически удаляться при их закрытии. Для того чтобы воспользоваться этой возможностью, передайте ZipFile.OPEN_READ|ZipFile.OPEN_DELETE в конструктор JarFile() как аргумент *mode*.

После создания объекта JarFile манифест JAR можно получить с помощью метода getManifest(). Перечисление объектов java.util.zip.ZipEntry в файле можно получить с помощью entries(), а JarEntry для указанного файла в файле JAR – с помощью getJarEntry(). Чтобы прочитать содержимое элемента файла JAR, получите объект JarEntry или ZipEntry, представляющий этот элемент, передайте его в getInputStream(), а затем читайте до завершения потока. JarFile не поддерживает создание новых файлов JAR и изменение существующих файлов.




```
public class JarFile extends java.util.zip.ZipFile {
// Открытые конструкторы
   public JarFile(String name) throws java.io.IOException;
   public JarFile(java.io.File file) throws java.io.IOException;
   public JarFile(String name, boolean verify) throws java.io.IOException;
```

```
   public JarFile(java.io.File file, boolean verify) throws java.io.IOException;
1.3 public JarFile(java.io.File file, boolean verify, int mode) throws java.io.IOException;
// Открытые константы
   public static final String MANIFEST_NAME;                    // ="META-INF/MANIFEST.MF"
// Открытые методы экземпляра
   public JarEntry getJarEntry(String name);
   public Manifest getManifest() throws java.io.IOException;
// Открытые методы, замещающие ZipFile
   public java.util.Enumeration entries();
   public java.util.zip.ZipEntry getEntry(String name);
   public java.io.InputStream getInputStream(java.util.zip.ZipEntry ze)
      throws java.io.IOException;                                  // синхронизирован
}
```

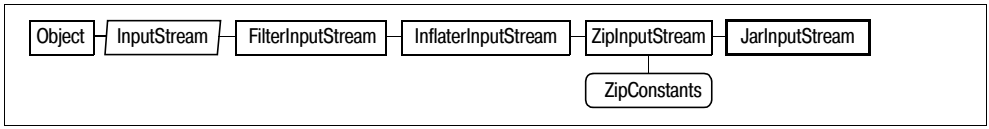
Возвращается методами: java.net.JarURLConnection.getJarFile()

JarInputStream Java 1.2

java.util.jar

Этот класс позволяет читать файл JAR из потока ввода. Он расширяет java.util.ZipInputStream **и используется почти так же, как и этот класс. Для создания** JarInputStream **нужно указать** InputStream**, из которого необходимо читать файл. Если объекту** JarInputStream **не следует проверять цифровые подписи, содержащиеся в файле JAR, передайте** false **как второй аргумент в конструктор** JarInputStream()**. Конструктор** JarInputStream() **прежде всего считывает манифест JAR, если он существует. Манифест должен быть первым элементом файла JAR. Метод** getManifest() **возвращает объект** Manifest **для файла JAR.**

После создания JarInputStream **вызовите** getNextJarEntry() **или** getNextEntry() **для получения объекта** JarEntry **или** java.util.zip.ZipEntry**, описывающего следующий элемент файла JAR. Затем вызовите метод** read() **(включая унаследованные версии), чтобы прочитать содержимое этого элемента. Когда поток достигнет конца файла, снова вызовите** getNextJarEntry()**, чтобы начать чтение следующего элемента файла. Когда все элементы прочитаны из файла JAR, методы** getNextJarEntry() **и** getNextEntry() **возвратят** null**.**



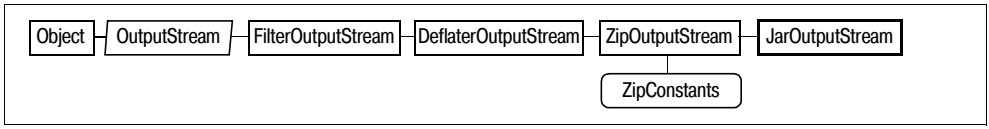

```
public class JarInputStream extends java.util.zip.ZipInputStream {
// Открытые конструкторы
   public JarInputStream(java.io.InputStream in) throws java.io.IOException;
   public JarInputStream(java.io.InputStream in, boolean verify) throws java.io.IOException;
// Открытые методы экземпляра
   public Manifest getManifest();
   public JarEntry getNextJarEntry() throws java.io.IOException;
// Открытые методы, замещающие ZipInputStream
   public java.util.zip.ZipEntry getNextEntry() throws java.io.IOException;
   public int read(byte[] b, int off, int len) throws java.io.IOException;
// Защищенные методы, замещающие ZipInputStream
   protected java.util.zip.ZipEntry createZipEntry(String name);
}
```

JarOutputStream Java 1.2

java.util.jar

Этот класс может записывать файл JAR в произвольный OutputStream. Класс JarOutputStream расширяет java.util.zip.ZipOutputStream и используется практически так же, как и этот класс. Создайте JarOutputStream, указав поток для записи и при необходимости объект Manifest для файла JAR. Конструктор JarOutputStream() начинает работу с записи содержимого объекта Manifest в соответствующий элемент файла JAR. На плечи программиста ложится забота о том, чтобы элементы JAR, записанные в дальнейшем, соответствовали сведениям, представленным в объекте Manifest. Этот класс не предоставляет явной поддержки для присоединения цифровых подписей к элементам в файле JAR.

После создания JarOutputStream вызовите putNextEntry() для указания объекта JarEntry или java.util.zip.ZipEntry, который необходимо записать в поток. Затем вызовите любой из унаследованных методов write() для записи содержимого элемента в поток. После этого вызовите putNextEntry() еще раз для начала записи следующего элемента. После записи всех элементов файла JAR вызовите close(). Перед записью любого элемента можно вызвать унаследованные методы setMethod() и setLevel(), чтобы указать, каким методом элемент должен быть сжат. См. java.util.zip.ZipOutputStream.




```
public class JarOutputStream extends java.util.zip.ZipOutputStream {
// Открытые конструкторы
   public JarOutputStream(java.io.OutputStream out) throws java.io.IOException;
   public JarOutputStream(java.io.OutputStream out, Manifest man) throws java.io.IOException;
// Открытые методы, замещающие ZipOutputStream
   public void putNextEntry(java.util.zip.ZipEntry ze) throws java.io.IOException;
}
```

Manifest Java 1.2

java.util.jar ***клонируемый***

Этот класс представляет манифест файла JAR. Метод getMainAttributes() возвращает объект Attributes, представляющий атрибуты манифеста, которые применяются ко всему файлу JAR. Метод getAttributes() возвращает объект Attributes, представляющий атрибуты манифеста, указанные для отдельного файла в файле JAR. Метод getEntries() возвращает java.util.Map, отображающий имена элементов файла JAR на объекты Attributes, связанные с этими элементами. Метод getEntries() возвращает объект Map, используемый объектом Manifest. Вы можете редактировать содержимое Manifest, добавляя, удаляя или редактируя элементы Map. Метод read() читает элементы манифеста из потока ввода, объединяя их в текущее множество элементов. Метод write() записывает объект Manifest в поток вывода.

```
public class Manifest implements Cloneable {
// Открытые конструкторы
   public Manifest();
   public Manifest(Manifest man);
   public Manifest(java.io.InputStream is) throws java.io.IOException;
// Открытые методы экземпляра
   public void clear();
   public java.util.jar.Attributes getAttributes(String name);
   public java.util.Map getEntries();                          // по умолчанию:HashMap
   public java.util.jar.Attributes getMainAttributes();
   public void read(java.io.InputStream is) throws java.io.IOException;
   public void write(java.io.OutputStream out) throws java.io.IOException;
// Открытые методы, замещающие Object
   public Object clone();
   public boolean equals(Object o);
   public int hashCode();
}
```

Передается методам: `java.net.URLClassLoader.definePackage()`, `JarOutputStream.JarOutputStream()`, `Manifest.Manifest()`

Возвращается методами: `java.net.JarURLConnection.getManifest()`, `JarFile.getManifest()`, `JarInputStream.getManifest()`

Пакет java.util.logging

Java 1.4

Пакет `java.util.logging` определяет мощные и гибкие средства протоколирования, которые могут использоваться приложениями Java для порождения, фильтрации, форматирования и вывода предупреждающих, диагностических, трассировочных и отладочных сообщений. Приложение генерирует регистрационные сообщения (log messages), вызывая разнообразные методы объекта `Logger`. Содержимое регистрационного сообщения (с другими деталями, такими как время и порядковый номер) помещается в объект `LogRecord`, генерируемый объектом `Logger`. Объект `Handler` представляет место назначения для объектов `LogRecord`. Конкретные подклассы `Handler` поддерживают такие точки назначения, как файлы и сокеты. Большинство объектов `Handler` имеют связанный объект `Formatter`, конвертирующий объект `LogRecord` в реальный текст, который заносится в журнал событий. Подклассы `SimpleFormatter` и `XMLFormatter` соответственно предоставляют простые регистрационные сообщения в текстовом формате и подробные журналы в формате XML.

Каждое регистрационное сообщение имеет уровень серьезности (severity level). Класс `Level` определяет независимое от типа перечисление определенных уровней. Оба объекта `Logger` и `Handler` имеют объект `Level` и отбрасывают любые регистрационные сообщения, чей уровень серьезности меньше указанного. В дополнение к средствам фильтрации, основанной на уровне серьезности, объекты `Logger` и `Handler` могут иметь объект `Filter`, который служит для фильтрации регистрационных сообщений, основанной на выбранном критерии.

Приложения, которым необходим полный контроль над журналами, могут создать объект `Logger` вместе с объектами `Handler`, `Formatter` и `Filter`, которые управляют расположением, содержимым и внешним видом журнала. Более простым приложениям нужно создавать только `Logger`, а всю остальную работу они могут оставить классу `LogManager`. `LogManager` читает системный конфигурационный файл (или конфигураци-

онный класс) и автоматически направляет регистрационные сообщения в место назначения, установленное в системе.

Интерфейсы

```
public interface Filter;
```

Классы

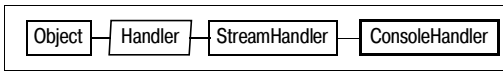
```
public class ErrorManager;
public abstract class Formatter;
   └ public class SimpleFormatter extends Formatter;
   └ public class XMLFormatter extends Formatter;
public abstract class Handler;
   └ public class MemoryHandler extends Handler;
   └ public class StreamHandler extends Handler;
      └ public class ConsoleHandler extends StreamHandler;
      └ public class FileHandler extends StreamHandler;
      └ public class SocketHandler extends StreamHandler;
public class Level implements Serializable;
public class Logger;
public final class LoggingPermission extends java.security.BasicPermission;
public class LogManager;
public class LogRecord implements Serializable;
```

ConsoleHandler Java 1.4

java.util.logging

Этот подкласс класса `Handler` форматирует объекты `LogRecord` и выводит результирующую строку в выходной поток `System.err`. При создании `ConsoleHandler` инициализуются разнообразные свойства, унаследованные от `Handler`. При этом учитываются предопределенные системные значения, получаемые с помощью `LogManager.getProperty()`. В следующей таблице перечисляются эти свойства: значение, передаваемое методу `getProperty()`, и значение по умолчанию, которое используется, если метод `getProperty()` возвращает `null`. Дополнительную информацию можно найти в описании `Handler`.

Свойство Handler	Имя свойства LogManager	Значение по умолчанию
level	java.util.logging.ConsoleHandler.level	Level.INFO
filter	java.util.logging.ConsoleHandler.filter	null
formatter	java.util.logging.ConsoleHandler.formatter	SimpleFormatter
encoding	java.util.logging.ConsoleHandler.encoding	по умолчанию для платформы




```
public class ConsoleHandler extends StreamHandler {
// Открытые конструкторы
   public ConsoleHandler();
// Открытые методы, замещающие StreamHandler
   public void close();
```

```
    public void publish(LogRecord record);
}
```

ErrorManager

Java 1.4

java.util.logging

Важная особенность API протоколирования заключается в следующем: методы протоколирования, вызываемые приложениями, никогда не генерируют исключений: бессмысленно ожидать от программистов, что они будут помещать регистрационные вызовы в блоки try/catch. Кроме того, не существует подходящего способа, с помощью которого приложение может восстановить работу после исключительной ситуации в подсистеме протоколирования. Поскольку классы обработчиков, например FileHandler, генерируют исключения ввода/вывода, ErrorManager предоставляет обработчику возможность сообщить об исключительной ситуации, вместо того чтобы просто проигнорировать ее.

Все объекты Handler имеют экземпляр класса ErrorManager, связанный с ними. Если в обработчике происходит исключительная ситуация, он передает в метод error() исключение вместе с сообщением и одной из констант кода ошибки, определяемых в ErrorManager. Метод error() записывает в System.err сообщение, описывающее исключительную ситуацию, но делает это только при первом вызове: ожидается, что Handler, сгенерировавший исключение один раз, будет генерировать аналогичные исключения с каждым последующим регистрационным сообщением, поэтому было бы неправильно наводнять System.err повторяющимися сообщениями об ошибке. Разумеется, вы можете определить подклассы для ErrorManager, замещающие метод error(), чтобы предоставить другой механизм оповещения. В этом случае зарегистрируйте экземпляр вашего собственного класса ErrorManager, вызвав метод setErrorManager() объекта Handler.

```
public class ErrorManager {
// Открытые конструкторы
    public ErrorManager();
// Открытые константы
    public static final int CLOSE_FAILURE;                          // =3
    public static final int FLUSH_FAILURE;                          // =2
    public static final int FORMAT_FAILURE;                         // =5
    public static final int GENERIC_FAILURE;                        // =0
    public static final int OPEN_FAILURE;                           // =4
    public static final int WRITE_FAILURE;                          // =1
// Открытые методы экземпляра
    public void error(String msg, Exception ex, int code);          // синхронизирован
}
```

Передается методам: Handler.setErrorManager()

Возвращается методами: Handler.getErrorManager()

FileHandler

Java 1.4

java.util.logging

Этот подкласс класса Handler форматирует объекты LogRecord и выводит получившиеся строки в файл или множество файлов. Аргументы, передаваемые в конструктор FileHandler(), определяют порядок использования файлов. Аргументы необязатель-

ны; если они не указаны, то с помощью LogManager.getProperty() извлекаются значения по умолчанию. Аргументы конструктора следующие:

pattern

Строка, содержащая символы подмены, которые задают один или несколько файлов. Преобразование паттерна в имена файлов описано ниже.

limit

Приблизительный максимальный размер файла журнала; значение равно 0, если ограничение отсутствует. Если параметр *count* больше единицы, то в момент, когда размер файла журнала достигает этого максимума, FileHandler закрывает файл, переименовывает его, а затем начинает новый журнал с тем же именем файла.

count

Когда параметр *limit* установлен в ненулевое значение, этот аргумент определяет количество сохраняемых старых файлов журнала.

append

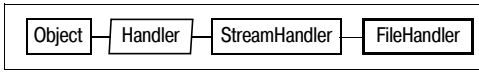
Принимает значение true, если FileHandler должен дописывать регистрационные сообщения в существующий файл с указанным именем, и false, если файл должен быть перезаписан.

Аргумент *pattern* является наиболее важным; он определяет, в какой файл или файлы FileHandler должен производить запись. При преобразовании паттерна в имена файлов FileHandler выполняет следующие замены:

Символы	Заменяются на
/	Символ разделения каталогов для данной платформы. При записи паттернов нужно всегда использовать простую косую черту – даже на платформе Windows, где применяется обратная косая черта.
%%	Одиночный знак процента.
%h	Домашний каталог пользователя: значение системного свойства «user.home».
%t	Каталог для временных файлов системы.
%u	Уникальный номер, который применяется для отличия данного log-файла от других файлов с таким же паттерном (это может понадобиться, если несколько Java-программ одновременно создают журналы).
%g	«Номер поколения» («generation number») старых log-файлов, когда аргумент *limit* не равен 0, а аргумент *count* больше 1. В файле, в который FileHandler помещает регистрационные записи, символы %g заменяются на 0. Когда этот файл заполнится, он будет закрыт, а в его имени 0 будет заменен на 1. Более старые файлы переименовываются таким же образом: их «номер поколения» увеличивается на 1. Когда количество файлов журнала достигнет величины, заданной агрументом *count*, самый старый файл удаляется, освобождая место для нового.

Когда FileHandler создан, метод LogManager.getProperty() применяется для извлечения значений по умолчанию для любого аргумента конструктора, а также для получения начальных значений разнообразных свойств, унаследованных от класса Handler. В таблице, приведенной ниже, перечислены аргументы и свойства, а также значение, передаваемое в getProperty(), и значение по умолчанию, которое используется, если getProperty() возвращает null. Дополнительные сведения можно найти в описании Handler.

Свойство или аргумент	Имя свойства LogManager	Значение по умолчанию
level	java.util.logging.FileHandler.level	Level.ALL
filter	java.util.logging.FileHandler.filter	null
formatter	java.util.logging.FileHandler.formatter	XMLFormatter
encoding	java.util.logging.FileHandler.encoding	**по умолчанию для платформы**
pattern	java.util.logging.FileHandler.pattern	%h/java%u.log
limit	java.util.logging.FileHandler.limit	**0 (нет ограничения)**
count	java.util.logging.FileHandler.count	**1**
append	java.util.logging.FileHandler.append	false




```
public class FileHandler extends StreamHandler {
// Открытые конструкторы
   public FileHandler() throws java.io.IOException, SecurityException;
   public FileHandler(String pattern) throws java.io.IOException, SecurityException;
   public FileHandler(String pattern, boolean append) throws java.io.IOException, SecurityException;
   public FileHandler(String pattern, int limit, int count) throws java.io.IOException,
                                                                   SecurityException;
   public FileHandler(String pattern, int limit, int count, boolean append)
        throws java.io.IOException, SecurityException;
// Открытые методы, замещающие StreamHandler
   public void close() throws SecurityException;                          // синхронизирован
   public void publish(LogRecord record);                                 // синхронизирован
}
```

Filter Java 1.4

java.util.logging

Этот интерфейс определяет метод, который должен быть реализован классом, фильтрующим регистрационные сообщения для классов Logger или Handler. Метод isLoggable() должен возвращать true, если указанный объект LogRecord содержит информацию, которую необходимо протоколировать. Он должен возвращать false, если фильтруемый LogRecord не отображается ни в одном журнале. Обратите внимание, что классы Logger и Handler предоставляют встроенную фильтрацию, основанную на уровне серьезности объекта LogRecord. Интерфейс Filter предоставляет возможность нестандартной фильтрации.

```
public interface Filter {
// Открытые методы экземпляра
   public abstract boolean isLoggable(LogRecord record);
}
```

Передается методам: Handler.setFilter(), Logger.setFilter()

Возвращается методами: Handler.getFilter(), Logger.getFilter()

Formatter Java 1.4

java.util.logging

Объект Formatter используется объектом Handler для преобразования объекта LogRecord в объект String перед его протоколированием. Большинство приложений могут просто использовать один из предопределенных подклассов: SimpleFormatter или XMLFormatter. Приложениям, которым необходимо нестандартное форматирование регистрационных сообщений, нужно создать подкласс этого класса и определить метод format() для выполнения требуемого преобразования. В таких подклассах может быть удобен метод formatMessage(); с помощью объекта java.util.ResourceBundle он выполняет локализацию, а с помощью средств пакета java.text – форматирование. Методы getHead() и getTail() возвращают префикс и суффикс для файла журнала (например, открывающий и закрывающий теги XML).

```
public abstract class Formatter {
// Защищенные конструкторы
   protected Formatter();
// Открытые методы экземпляра
   public abstract String format(LogRecord record);
   public String formatMessage(LogRecord record);                    // синхронизирован
   public String getHead(Handler h);
   public String getTail(Handler h);
}
```

Подклассы: SimpleFormatter, XMLFormatter

Передается методам: Handler.setFormatter(), StreamHandler.StreamHandler()

Возвращается методами: Handler.getFormatter()

Handler Java 1.4

java.util.logging

Handler берет объекты LogRecord из объекта Logger; если их уровень серьезности достаточно высок, то он форматирует их и выводит в определенное место назначения (например, в файл или сокет). Подклассы этого абстрактного класса поддерживают разнообразные пункты назначения и реализуют специфичные методы publish(), flush() и close().

В дополнение к абстрактным методам, специфичным для пункта назначения, этот класс определяет конкретные методы, используемые большинством подклассов Handler: методы получения и установки значений свойств для определения уровня серьезности регистрируемых сообщений; необязательный фильтр; объект Formatter для преобразования регистрационных сообщений из объектов LogRecord в текст; методы кодировки выводимого текста и объект ErrorManager для обработки любых исключительных ситуаций, возникающих при занесении сообщений в журнал. Значения по умолчанию, специфичные для подклассов, для каждого из этих свойств обычно определяются как свойства LogManager и считываются из системного конфигурационного файла протоколирования.

В простейшем случае Logger отсылает регистрационные сообщения одному или нескольким обработчикам, которые определены классом LogManager для «корневого регистратора» («root logger»). В этом случае приложению не нужно создавать экземпляры или использовать Handler напрямую. Приложения, которым необходим особый контроль над местом расположения их журнала, создают и настраивают экземпляр

подкласса Handler, однако у них никогда не возникает необходимости вызывать напрямую методы publish(), flush() или close() — это выполняется объектом Logger.

```
public abstract class Handler {
// Защищенные конструкторы
   protected Handler();
// Методы доступа к свойствам (по имени свойства)
   public String getEncoding();
  public void setEncoding(String encoding) throws SecurityException, java.io.UnsupportedEncodingException;
   public ErrorManager getErrorManager();
   public void setErrorManager(ErrorManager em);
   public Filter getFilter();
   public void setFilter(Filter newFilter) throws SecurityException;
   public Formatter getFormatter();
   public void setFormatter(Formatter newFormatter) throws SecurityException;
   public Level getLevel();                                          // синхронизирован
   public void setLevel(Level newLevel) throws SecurityException;    // синхронизирован
// Открытые методы экземпляра
   public abstract void close() throws SecurityException;
   public abstract void flush();
   public boolean isLoggable(LogRecord record);
   public abstract void publish(LogRecord record);
// Защищенные методы экземпляра
   protected void reportError(String msg, Exception ex, int code);
}
```

Подклассы: MemoryHandler, StreamHandler

Передается методам: Formatter.{getHead(), getTail()}, Logger.{addHandler(), removeHandler()}, MemoryHandler.MemoryHandler(), XMLFormatter.{getHead(), getTail()}

Экземпляры: Logger.getHandlers()

Level

Java 1.4

java.util.logging ***сериализуемый***

Этот класс определяет константы, представляющие семь стандартных уровней серьезности для регистрационных сообщений, плюс константы, запрещающие и разрешающие протоколирование на любом уровне. Когда протоколирование разрешено на одном уровне серьезности, оно разрешено на всех более высоких уровнях. Вот семь констант уровня в порядке расположения от более важного уровня к менее важному: SEVERE, WARNING, INFO, CONFIG, FINE, FINER и FINEST. Константа ALL разрешает протоколирование любого сообщения безотносительно к его уровню. Константа OFF полностью запрещает протоколирование. Все эти константы являются объектами Level, а не целыми числами. За счет этого достигается безопасность типов (type safety).

Коду приложений редко требуются методы этого класса; они могут вообще не потребоваться. Вместо этого можно просто использовать константы, определяемые данным классом.


```
public class Level implements Serializable {
// Защищенные конструкторы
   protected Level(String name, int value);
```

```
   protected Level(String name, int value, String resourceBundleName);
// Открытые константы
   public static final Level ALL;
   public static final Level CONFIG;
   public static final Level FINE;
   public static final Level FINER;
   public static final Level FINEST;
   public static final Level INFO;
   public static final Level OFF;
   public static final Level SEVERE;
   public static final Level WARNING;
// Открытые методы экземпляра
   public static Level parse(String name) throws IllegalArgumentException;    // синхронизирован
// Открытые методы экземпляра
   public String getLocalizedName();
   public String getName();
   public String getResourceBundleName();
   public final int intValue();
// Открытые методы, замещающие Object
   public boolean equals(Object ox);
   public int hashCode();
   public final String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: `Handler.getLevel()`, `Level.parse()`, `Logger.getLevel()`, `LogRecord.getLevel()`, `MemoryHandler.getPushLevel()`

Экземпляры: `Level.{ALL, CONFIG, FINE, FINER, FINEST, INFO, OFF, SEVERE, WARNING}`

Logger Java 1.4

java.util.logging

Объект `Logger` используется для порождения регистрационных сообщений. `Logger` не имеет открытого конструктора, однако существует несколько способов получения объекта `Logger`:

- Обычно приложения вызывают статический метод `getLogger()` для создания или поиска именованного объекта `Logger` в иерархии именованных регистраторов. Регистраторы имеют иерархические имена, разделенные точками. Эти имена должны основываться на имени класса или пакета, который их использует. Регистраторы, полученные таким способом, наследуют уровень регистрации, пакет ресурсов (для локализации) и объекты `Handler` от их предков в иерархии и, в конечном итоге, от корневого объекта `Logger`, определенного глобальным объектом `LogManager`.
- Апплеты, которым необходим регистратор без защитных ограничений, должны вызывать статический метод `getAnonymousLogger()` для создания безымянного объекта `Logger`. Этот объект не является частью иерархии именованных объектов `Logger`, управляемых объектом `LogManager`. `Logger`, созданный этим методом, имеет корневой регистратор `LogManager` в качестве своего предка и наследует уровень регистрации и обработчики данного корневого регистратора.
- Наконец, статическое поле `Logger.global` ссылается на предопределенный объект `Logger`, называемый «global»; программисты могут ощутить удобство предопреде-

ленного регистратора на ранних этапах разработки приложения, однако его не следует применять в готовом коде.

Как только получен подходящий `Logger`, приложению доступно множество методов, которые можно использовать для создания регистрационного сообщения:

- Методы `log()` протоколируют указанное сообщение на заданном уровне. Необязательные параметры можно использовать при локализации сообщения. Эти методы обращаются к стеку вызовов и пытаются определить имена класса и метода, из которых вызван метод. Однако они не всегда могут получить эту информацию из-за проведенной оптимизации кода и синхронной компиляции.
- Методы `logp()` («log precise») подобны методам `log()`, но они позволяют явно указать имя класса и метода, которые генерируют регистрационное сообщение.
- Методы `logrb()` подобны методам `logp()`, но дополнительно они принимают имя пакета ресурсов для использования при локализации сообщения.
- Методы `entering()`, `exiting()` и `throwing()` являются удобными методами для генерации регистрационных сообщений, комментирующих выполнение программы. Эти методы используют уровень протоколирования `Level.FINER`. Существуют разновидности методов `entering()` и `exiting()`, позволяющие указывать аргументы метода и возвращаемые значения.
- Наконец, `Logger` определяет набор простых и удобных методов для регистрации простого сообщения на указанном уровне протоколирования. Эти методы имеют такие же имена, как и уровни протоколирования: `severe()`, `warning()`, `info()`, `config()`, `fine()`, `finer()` и `finest()`.

`Logger` имеет определенный уровень протоколирования и игнорирует любые регистрационные сообщения с меньшим уровнем серьезности. Уровень серьезности инициализируется из системного конфигурационного файла, который обычно определяет требуемое протоколирование. Такие установки можно изменить с помощью метода `setLevel()`. Его вызов может потребоваться при создании `Logger` с помощью `getAnonymousLogger()` и считывании уровня протоколирования из собственного конфигурационного файла. Если фильтрация регистрационных сообщений, определяемая заданным уровнем, недостаточна, вы можете связать `Filter` с вашим объектом `Logger`, вызвав `setFilter()`. В этом случае любые регистрационные сообщения, отвергнутые фильтром, будут игнорироваться.

`Logger` отсылает свои регистрационные сообщения во все объекты `Handler`, которые были зарегистрированы с помощью метода `addHandler()`. Метод `getHandlers()` позволяет получить массив зарегистрированных обработчиков, а метод `removeHandler()` — удалить обработчик из списка. По умолчанию все регистрационные сообщения дополнительно отсылаются обработчикам родительского регистратора и любых других регистраторов-предков. Поскольку корневой регистратор `LogManager` является родительским регистратором или регистратором-предком для всех именованных регистраторов, то все они по умолчанию отсылают регистрационные сообщения обработчикам, определенным в системном конфигурационном файле протоколирования. Более подробные сведения можно получить в описании `LogManager`. Если вы не хотите, чтобы `Logger` использовал обработчиков его предков, передайте `false` в метод `setUseParentHandlers()`.

Методы `getLogger()` и `getAnonymousLogger()` позволяют вам задать имя `java.util.ResourceBundle` для использования при локализации регистрационных сообщений, а методы `logrb()` позволяют указать имя пакета ресурсов, необходимого при локализации определенного регистрационного сообщения. Если пакет ресурсов указан для объекта

Logger или для определенного регистрационного сообщения, то аргумент «сообщение» в разнообразных регистрационных методах рассматривается не как текстовое сообщение, а как ключ локализации, определяющий локализованную версию в пакете ресурсов. Локализация подразумевает, что параметры, определяемые аргументами param1 и params для метода log(), заменяются на локализованную строку сообщения с помощью java.text.MessageFormat. (Обратите внимание, однако, что локализация и форматирование не выполняются объектом Logger самостоятельно: вместо этого он просто сохраняет ResourceBundle и параметры в LogRecord. В действительности локализация производится объектом Formatter, связанным с выходным объектом Handler.)

Все методы этого класса являются потокобезопасными и не требуют внешней синхронизации.

```
public class Logger {
// Защищенные конструкторы
   protected Logger(String name, String resourceBundleName);
// Открытые константы
   public static final Logger global;
// Открытые методы экземпляра
   public static Logger getAnonymousLogger();                                  // синхронизирован
   public static Logger getAnonymousLogger(String resourceBundleName);         // синхронизирован
   public static Logger getLogger(String name);                                // синхронизирован
   public static Logger getLogger(String name, String resourceBundleName);     // синхронизирован
// Методы доступа к свойствам (по имени свойства)
   public Filter getFilter();
   public void setFilter(Filter newFilter) throws SecurityException;
   public Handler[] getHandlers();                                             // синхронизирован
   public Level getLevel();
   public void setLevel(Level newLevel) throws SecurityException;
   public String getName();
   public Logger getParent();
   public void setParent(Logger parent);
   public java.util.ResourceBundle getResourceBundle();
   public String getResourceBundleName();
   public boolean getUseParentHandlers();                                      // синхронизирован
   public void setUseParentHandlers(boolean useParentHandlers);                // синхронизирован
// Открытые методы экземпляра
   public void addHandler(Handler handler) throws SecurityException;          // синхронизирован
   public void config(String msg);
   public void entering(String sourceClass, String sourceMethod);
   public void entering(String sourceClass, String sourceMethod, Object param1);
   public void entering(String sourceClass, String sourceMethod, Object[] params);
   public void exiting(String sourceClass, String sourceMethod);
   public void exiting(String sourceClass, String sourceMethod, Object result);
   public void fine(String msg);
   public void finer(String msg);
   public void finest(String msg);
   public void info(String msg);
   public boolean isLoggable(Level level);
   public void log(LogRecord record);
   public void log(Level level, String msg);
   public void log(Level level, String msg, Throwable thrown);
   public void log(Level level, String msg, Object param1);
   public void log(Level level, String msg, Object[] params);
   public void logp(Level level, String sourceClass, String sourceMethod, String msg);
   public void logp(Level level, String sourceClass, String sourceMethod, String msg, Object param1);
   public void logp(Level level, String sourceClass, String sourceMethod, String msg, Object[] params);
```

```
   public void logp(Level level, String sourceClass, String sourceMethod, String msg, Throwable thrown);
   public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName, String msg);
   public void logrb(Level level, String sourceClass, String sourceMethod,
                     String bundleName, String msg, Object param1);
   public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName,
                     String msg, Throwable thrown);
   public void logrb(Level level, String sourceClass, String sourceMethod, String bundleName,
                     String msg, Object[] params);
   public void removeHandler(Handler handler) throws SecurityException;      // синхронизирован
   public void severe(String msg);
   public void throwing(String sourceClass, String sourceMethod, Throwable thrown);
   public void warning(String msg);
}
```

Передается методам: `Logger.setParent()`, `LogManager.addLogger()`

Возвращается методами: `Logger.{getAnonymousLogger(), getLogger(), getParent()}`, `LogManager.getLogger()`

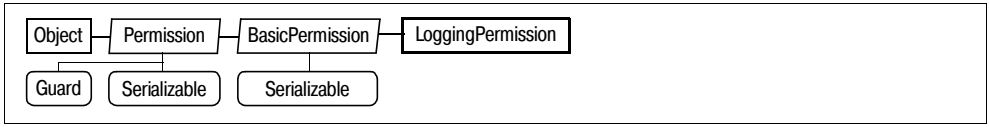
Экземпляры: `Logger.global`

LoggingPermission

Java 1.4

java.util.logging ***сериализуемый***

Этот класс – потомок `java.security.Permission`, управляющий использованием методов протоколирования, зависящих от настроек системы безопасности. Единственное определенное имя (или целевой объект) для `LoggingPermission` – это «control». Оно представляет право на вызов разнообразных методов управления протоколированием, таких как `Logger.setLevel()` и `LogManager.readConfiguration()`. Для методов этого пакета, генерирующих `SecurityException`, необходим объект `LoggingPermission` с именем «control». У программистов приложений никогда не возникает необходимости использовать этот класс, а системным администраторам, настраивающим правила безопасности, следует его изучить.




```
public final class LoggingPermission extends java.security.BasicPermission {
// Открытые конструкторы
   public LoggingPermission(String name, String actions) throws IllegalArgumentException;
}
```

LogManager

Java 1.4

java.util.logging

Как явствует из его имени, данный класс является менеджером для API `java.util.logging`. Он имеет три варианта применения.

1. Чтение конфигурационного файла протоколирования и создание объектов `Handler` по умолчанию, определенных в этом файле.
2. Управление набором объектов `Logger` путем компоновки их в дерево на основе иерархических имен.

3. Создание и управление безымянным объектом Logger, который служит родителем или предком для всех других объектов Logger.

Этот класс управляет важными закулисными действиями, которые обеспечивают работоспособность API протоколирования. Типичные приложения могут выполнять протоколирование без непосредственного применения этого класса. Хотя программисты нечасто применяют этот API, работу класса LogManager полезно понимать. Именно поэтому здесь представлено его подробное описание.

Единственный глобальный экземпляр LogManager можно получить с помощью статического метода getLogManager(). По умолчанию глобальный объект менеджера журнала является экземпляром класса LogManager. Вместо этого можно создать экземпляр подкласса LogManager, указав полное имя подкласса как значение системного свойства java.util.logging.manager.

Одно из основных назначений класса LogManager – чтение файла java.util.Properties, определяющего конфигурацию системы протоколирования. По умолчанию этот файл называется logging.properties; он хранится в каталоге *jre/lib* инсталляции Java. Если необходимо запустить приложение Java с использованием другой конфигурации системы протоколирования, то можно отредактировать конфигурационный файл по умолчанию. Но обычно проще создать новый конфигурационный файл и уведомить об этом JVM, присвоив системному свойству java.util.logging.config.file имя этого конфигурационного файла.

Наиболее важное назначение конфигурационного файла – определять набор объектов Handler, которым отсылаются регистрационные сообщения. Для этого свойству handlers присваивается список имен класса Handler, разделенных пробелами. LogManager загрузит указанные классы и создаст экземпляр каждого класса с помощью конструктора без аргументов, определенного по умолчанию. Затем он зарегистрирует эти объекты Handler в корневом объекте Logger, откуда они унаследуются всеми другими регистраторами (корневой регистратор будет рассмотрен далее). Затем каждый объект Handler «настроит» себя, прочитав дополнительные свойства из конфигурационного файла, как описано в документации каждого класса обработчика.

Конфигурационный файл может содержать имя свойства, сформированное добавлением «.level» к имени регистратора. Значение любого свойства воспринимается как имя уровня регистрации для именованного регистратора. Когда именованный регистратор создан и зарегистрирован с помощью LogManager (описано ниже), ему автоматически присваивается указанный уровень регистрации.

Приложения, любые специальные подклассы Handler и Formatter, а также реализации Filter могут считывать собственные свойства из конфигурационного файла протоколирования, используя метод getProperty() из LogManager. Это удобный способ настройки для классов, имеющих отношение к протоколированию.

В дополнение к управлению свойствами конфигурационного файла вторым назначением LogManager является поддержка дерева объектов Logger, организованных в иерархию на основе иерархических имен, разделенных точками. Метод addLogger() регистрирует новый объект Logger с помощью LogManager и вставляет его в дерево. Впрочем, этот метод вызывается автоматически методом-фабрикой Logger.getLogger(), поэтому вам никогда не понадобится вызывать его самостоятельно. Метод getLogger() в LogManager ищет в дереве и возвращает именованный объект Logger. Метод getLoggerNames() применяется для получения перечисления (Enumeration) имен всех вовлеченных регистраторов.

В корне дерева находится корневой регистратор, создаваемый объектом LogManager и инициализируемый объектами Handler по умолчанию, которые указаны в конфигу-

рационном файле протоколирования. Корневой регистратор не имеет имени; вы можете получить ссылку на него, передав пустую строку в метод getLogger(). За исключением этого корневого регистратора и анонимных регистраторов (см. Logger.getAnonymousLogger()), все остальные регистраторы имеют имена; обычно они именуются в соответствии с именем пакета или класса, для которых они производят регистрацию. Когда именованный регистратор попадает к LogManager, LogManager изучает его имя и вставляет его в соответствующее место дерева регистраторов: регистратор, названный «java.util.logging», будет потомком регистратора «java.util», если такой регистратор существует, или потомком регистратора с именем «java» в противном случае. Если регистраторов с такими именами нет, он будет вставлен как потомок корневого регистратора с именем «». Когда LogManager определил позицию регистратора в дереве регистраторов, он вызывает метод setParent() для вновь зарегистрированного объекта Logger, чтобы уведомить этот объект о его предке. Это важно, поскольку по умолчанию регистраторы наследуют уровень протоколирования и обработчики от своего предка. Несмотря на то что метод Logger.setParent() является открытым, он предназначен только для класса LogManager.

Анонимные регистраторы, создаваемые с помощью Logger.getAnonymousLogger(), не имеют имен и не являются частью дерева регистраторов. Однако когда они создаются, их предком назначается корневой регистратор, определенный в LogManager. По этой причине анонимные регистраторы наследуют обработчики по умолчанию, указанные в конфигурационном файле протоколирования.

Методы readConfiguration() заставляют LogManager перечитать системный конфигурационный файл или прочитать новый конфигурационный файл из указанного потока. Обе версии этого метода генерируют java.beans.PropertyChangeEvent и используют его для оповещения любых слушателей, зарегистрированных с помощью addPropertyChangeListener(). Оба метода вначале вызывают метод reset(), который сбрасывает свойства текущего конфигурационного файла, удаляет и закрывает все обработчики для всех регистраторов, а затем устанавливает уровень протоколирования для всех регистраторов в null, за исключением уровня протоколирования для корневого регистратора, который устанавливается в Level.INFO. Маловероятно, что вам может потребоваться вызывать reset() самостоятельно. Некоторые методы LogManager генерируют SecurityException, если вызывающий модуль не имеет соответствующих прав. Метод checkAccess() проверяет, имеет ли текущий контекст вызова требуемый объект LoggingPermission с именем «control».

Все методы LogManager могут безопасно использоваться несколькими потоками.

```
public class LogManager {
// Защищенные конструкторы
   protected LogManager();
// Открытые методы экземпляра
   public static LogManager getLogManager();
// Методы регистрации событий (по имени события)
   public void addPropertyChangeListener(java.beans.PropertyChangeListener l)
        throws SecurityException;
   public void removePropertyChangeListener(java.beans.PropertyChangeListener l)
        throws SecurityException;
// Открытые методы экземпляра
   public boolean addLogger(Logger logger);                                    // синхронизирован
   public void checkAccess() throws SecurityException;
   public Logger getLogger(String name);                                       // синхронизирован
   public java.util.Enumeration getLoggerNames();                              // синхронизирован
   public String getProperty(String name);
```

```
    public void readConfiguration() throws java.io.IOException, SecurityException;
    public void readConfiguration(java.io.InputStream ins) throws java.io.IOException, SecurityException;
    public void reset() throws SecurityException;
}
```

Возвращается методами: `LogManager.getLogManager()`

LogRecord

Java 1.4

java.util.logging ***сериализуемый***

Экземпляры этого класса используются для представления регистрационных сообщений, когда они перемещаются между объектами `Logger`, `Handler`, `Filter` и `Formatter`. `LogRecord` определяет несколько JavaBeans-подобных методов установки и получения свойств. Значения разнообразных свойств определяют все детали регистрационного сообщения. Конструктор `LogRecord()` получает аргументы для двух наиболее важных свойств: уровня протоколирования и регистрационного сообщения (ключа локализации). Этот конструктор инициализирует свойство `millis` текущим временем, а свойство `sequenceNumber` – уникальным (в пределах виртуальной машины) значением, которое можно применять для сравнения порядка двух регистрационных сообщений. Кроме того, данный конструктор присваивает свойству `threadID` уникальный идентификатор текущего потока. У всех остальных свойств `LogRecord` остаются значения по умолчанию (`null`).


```
public class LogRecord implements Serializable {
// Открытые конструкторы
    public LogRecord(Level level, String msg);
// Методы доступа к свойствам (по имени свойства)
    public Level getLevel();
    public void setLevel(Level level);
    public String getLoggerName();
    public void setLoggerName(String name);
    public String getMessage();
    public void setMessage(String message);
    public long getMillis();
    public void setMillis(long millis);
    public Object[] getParameters();
    public void setParameters(Object[] parameters);
    public java.util.ResourceBundle getResourceBundle();
    public void setResourceBundle(java.util.ResourceBundle bundle);
    public String getResourceBundleName();
    public void setResourceBundleName(String name);
    public long getSequenceNumber();
    public void setSequenceNumber(long seq);
    public String getSourceClassName();
    public void setSourceClassName(String sourceClassName);
    public String getSourceMethodName();
    public void setSourceMethodName(String sourceMethodName);
    public int getThreadID();
    public void setThreadID(int threadID);
    public Throwable getThrown();
    public void setThrown(Throwable thrown);
}
```

Передается методам: ConsoleHandler.publish(), FileHandler.publish(), Filter.isLoggable(), Formatter.{format(), formatMessage()}, Handler.{isLoggable(), publish()}, Logger.log(), MemoryHandler.{isLoggable(), publish()}, SimpleFormatter.format(), SocketHandler.publish(), StreamHandler.{isLoggable(), publish()}, XMLFormatter.format()

MemoryHandler Java 1.4

java.util.logging

MemoryHandler сохраняет объекты LogRecord в буфере фиксированного размера в памяти. Когда буфер заполняется, то при поступлении новой записи он отбрасывает самую старую запись. Он сохраняет ссылку на другой объект Handler. Когда вызывается метод push() или приходит LogRecord с уровнем, который не меньше порога pushLevel, он «выталкивает» все буферизованные объекты LogRecord в объект Handler. Этот объект форматирует их и размещает в определенном месте. Поскольку MemoryHandler никогда не выводит регистрационные записи самостоятельно, он не использует свойства formatter и encoding, унаследованные от родительского класса.

При создании MemoryHandler можно указать целевой объект Handler, размер буфера в памяти и значение свойства pushLevel. Эти аргументы конструктора можно опустить и положиться на предопределенные системные значения, полученные с помощью LogManager.getProperty(). Кроме того, MemoryHandler использует LogManager.getProperty() для получения начальных значений свойств level и filter, унаследованных от класса Handler. В нижеследующей таблице перечисляются эти свойства. Кроме них в таблице представлены аргументы конструктора target, size и pushLevel, а также значение, передаваемое в getProperty(), и значение по умолчанию, которое используется, если getProperty() возвращает null. Более подробная информация представлена в описании Handler.

Свойство или аргумент	Имя свойства LogManager	Значение по умолчанию
level	java.util.logging.MemoryHandler.level	Level.ALL
filter	java.util.logging.MemoryHandler.filter	null
target	java.util.logging.MemoryHandler.target	Нет значения по умолчанию
size	java.util.logging.MemoryHandler.size	1000 регистрационных записей
pushLevel	java.util.logging.MemoryHandler.push	Level.SEVERE


```
public class MemoryHandler extends Handler {
// Открытые конструкторы
   public MemoryHandler();
   public MemoryHandler(Handler target, int size, Level pushLevel);
// Открытые методы экземпляра
   public Level getPushLevel();                                    // синхронизирован
   public void push();                                             // синхронизирован
   public void setPushLevel(Level newLevel) throws SecurityException;
// Открытые методы, замещающие Handler
   public void close() throws SecurityException;
   public void flush();
```

```
   public boolean isLoggable(LogRecord record);
   public void publish(LogRecord record);                          // синхронизирован
}
```

SimpleFormatter

Java 1.4

java.util.logging

Этот подкласс класса `Formatter` преобразует объект `LogRecord` в регистрационное сообщение, которое удобно для восприятия человеком. Обычно его длина от одной до двух строк. См. также `XMLFormatter`.


```
public class SimpleFormatter extends Formatter {
// Открытые конструкторы
   public SimpleFormatter();
// Открытые методы, замещающие Formatter
   public String format(LogRecord record);                         // синхронизирован
}
```

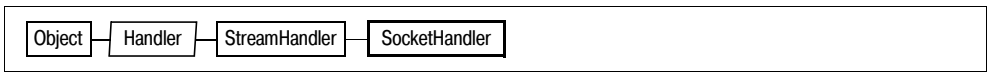
SocketHandler

Java 1.4

java.util.logging

Этот подкласс класса `Handler` форматирует объекты `LogRecord` и выводит результирующие строки в сетевой сокет. При создании `SocketHandler` вы можете передать конструктору имя хоста и порт сокета или положиться на предопределенные системные значения, получаемые с помощью `LogManager.getProperty()`. Кроме того, `SocketHandler` использует `LogManager.getProperty()` для получения начальных значений, унаследованных от класса `Handler`. В нижеследующей таблице перечисляются эти свойства, а также аргументы `host` и `port`, значение, передаваемое в `getProperty()`, и значение по умолчанию, которое используется, если `getProperty()` возвращает `null`. Для получения более подробной информации обратитесь к описанию `Handler`.

Свойство Handler	Имя свойства LogManager	Значение по умолчанию
`level`	`java.util.logging.SocketHandler.level`	`Level.ALL`
`filter`	`java.util.logging.SocketHandler.filter`	`null`
`formatter`	`java.util.logging.SocketHandler.formatter`	`XMLFormatter`
`encoding`	`java.util.logging.SocketHandler.encoding`	По умолчанию для платформы
`hostname`	`java.util.logging.SocketHandler.host`	Нет значения по умолчанию
`port`	`java.util.logging.SocketHandler.port`	Нет значения по умолчанию




```
public class SocketHandler extends StreamHandler {
// Открытые конструкторы
   public SocketHandler() throws java.io.IOException;
```

```
   public SocketHandler(String host, int port) throws java.io.IOException;
// Открытые методы, замещающие StreamHandler
   public void close() throws SecurityException;                          // синхронизирован
   public void publish(LogRecord record);                                 // синхронизирован
}
```

StreamHandler Java 1.4

java.util.logging

Этот подкласс класса `Handler` посылает регистрационные сообщения в произвольный `java.io.OutputStream`. Он служит общим родительским классом для классов `ConsoleHandler`, `FileHandler` и `SocketHandler`.


```
public class StreamHandler extends Handler {
// Открытые конструкторы
   public StreamHandler();
   public StreamHandler(java.io.OutputStream out, Formatter formatter);
// Открытые методы, замещающие Handler
   public void close() throws SecurityException;                          // синхронизирован
   public void flush();                                                   // синхронизирован
   public boolean isLoggable(LogRecord record);
   public void publish(LogRecord record);                                 // синхронизирован
   public void setEncoding(String encoding) throws SecurityException, java.io.UnsupportedEncodingException;
// Защищенные методы экземпляра
   protected void setOutputStream(java.io.OutputStream out) throws SecurityException;
                                                                          // синхронизирован
}
```

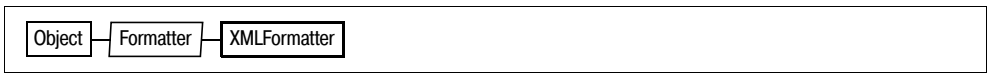
Подклассы: `ConsoleHandler`, `FileHandler`, `SocketHandler`

XMLFormatter Java 1.4

java.util.logging

Этот подкласс класса `Formatter` преобразует `LogRecord` в строку, форматированную в виде XML. Метод `format()` возвращает элемент `<record>`, который всегда содержит теги `<date>`, `<millis>`, `<sequence>`, `<level>` и `<message>`. Кроме того, он может содержать теги `<logger>`, `<class>`, `<method>`, `<thread>`, `<key>`, `<catalog>`, `<param>` и `<exception>`. DTD для выходного документа можно получить по адресу *http://java.sun.com/dtd/logger.dtd*.

Методы `getHead()` и `getTail()` подменяются, чтобы возвращать открывающие и закрывающие теги `<log>` и `</log>` для окружения всех выводимых тегов `<record>`. Однако если работа приложения завершается аварийно, то средство регистрации (logging facility) может оказаться не в состоянии поставить в конце файла журнала закрывающий тег `<log>`.




```
public class XMLFormatter extends Formatter {
// Открытые конструкторы
```

```
   public XMLFormatter();
// Открытые методы, замещающие Formatter
   public String format(LogRecord record);
   public String getHead(Handler h);
   public String getTail(Handler h);
}
```

Пакет java.util.prefs

Java 1.4

Пакет java.util.prefs содержит классы и интерфейсы для управления постоянными (persistent) свойствами пользовательского и системного уровней для приложений и классов Java. Большинству приложений необходим только класс Preferences. Некоторые приложения будут также использовать объекты событий и интерфейсы слушателей событий, определенные этим пакетом. Отдельным приложениям может понадобиться явно перехватывать типы исключений, определяемые этим пакетом. Прикладным программистам не пригодится интерфейс PreferencesFactory или класс AbstractPreferences, которые предназначены только для реализаций Preferences.

При работе с классом Preferences в первую очередь используйте статический метод для получения соответствующего объекта или объектов Preferences, а затем вызовите метод get(), чтобы получить значение свойства, или метод put() для установки значения свойства. Следующий код демонстрирует работу с этим классом. Более подробная информация представлена в описании Preferences.

```
import java.util.prefs.Preferences;
public class TextEditor {
  // Некоторые константы, определяющие значения свойств, заданные по умолчанию
  public static final int WIDTH_DEFAULT = 80;
  public static final String DICTIONARY_DEFAULT = "";

  // Поля, которые должны быть инициализированы значениями свойств
  public int width;                 // Ширина экрана в столбцах
  public String dictionary;         // Имя словаря для проверки правописания

  public void initPrefs() {
    // Получаем объекты Preferences пользовательского и системного уровня для этого пакета
    Preferences userprefs = Preferences.userNodeForPackage(TextEditor.class);
    Preferences sysprefs = Preferences.systemNodeForPackage(TextEditor.class);

    // Ищем значения свойств.  Обратите внимание, что вы всегда передаете
    // значение по умолчанию.
    width = userprefs.getInt("width", WIDTH_DEFAULT);
    // Ищем пользовательское свойство, используя системное свойство как значение по умолчанию
    dictionary = userprefs.get("dictionary",
                               sysprefs.get("dictionary",
                                            DICTIONARY_DEFAULT));
  }
}
```

Интерфейс

```
public interface PreferencesFactory;
```

События

```
public class NodeChangeEvent extends java.util.EventObject;
public class PreferenceChangeEvent extends java.util.EventObject;
```

Слушатели событий

```
public interface NodeChangeListener extends java.util.EventListener;
public interface PreferenceChangeListener extends java.util.EventListener;
```

Другие классы

```
public abstract class Preferences;
   └ public abstract class AbstractPreferences extends Preferences;
```

Исключения

```
public class BackingStoreException extends Exception;
public class InvalidPreferencesFormatException extends Exception;
```

AbstractPreferences Java 1.4

java.util.prefs

Этот класс реализует все абстрактные методы класса Preferences поверх небольшого набора абстрактных методов. Программисты, создающие реализации Preferences (или «поставщики услуг»), могут создавать подклассы этого класса; им потребуется определить девять методов, имена которых заканчиваются на «Spi». Прикладные программисты никогда не нуждаются в использовании этого класса.


```
public abstract class AbstractPreferences extends Preferences {
// Защищенные конструкторы
   protected AbstractPreferences(AbstractPreferences parent, String name);
// Методы регистрации событий (по имени события)
   public void addNodeChangeListener(NodeChangeListener ncl);             // Замещает:Preferences
   public void removeNodeChangeListener(NodeChangeListener ncl);          // Замещает:Preferences
   public void addPreferenceChangeListener(PreferenceChangeListener pcl);  // Замещает:Preferences
   public void removePreferenceChangeListener(PreferenceChangeListener pcl); // Замещает:Preferences
// Открытые методы, замещающие Preferences
   public String absolutePath();
   public String[] childrenNames() throws BackingStoreException;
   public void clear() throws BackingStoreException;
   public void exportNode(java.io.OutputStream os) throws java.io.IOException, BackingStoreException;
   public void exportSubtree(java.io.OutputStream os) throws java.io.IOException, BackingStoreException;
   public void flush() throws BackingStoreException;
   public String get(String key, String def);
   public boolean getBoolean(String key, boolean def);
   public byte[] getByteArray(String key, byte[] def);
   public double getDouble(String key, double def);
   public float getFloat(String key, float def);
   public int getInt(String key, int def);
   public long getLong(String key, long def);
   public boolean isUserNode();
   public String[] keys() throws BackingStoreException;
   public String name();
```

```
   public Preferences node(String path);
   public boolean nodeExists(String path) throws BackingStoreException;
   public Preferences parent();
   public void put(String key, String value);
   public void putBoolean(String key, boolean value);
   public void putByteArray(String key, byte[] value);
   public void putDouble(String key, double value);
   public void putFloat(String key, float value);
   public void putInt(String key, int value);
   public void putLong(String key, long value);
   public void remove(String key);
   public void removeNode() throws BackingStoreException;
   public void sync() throws BackingStoreException;
   public String toString();
// Защищенные методы экземпляра
   protected final AbstractPreferences[] cachedChildren();
   protected abstract String[] childrenNamesSpi() throws BackingStoreException;
   protected abstract AbstractPreferences childSpi(String name);
   protected abstract void flushSpi() throws BackingStoreException;
   protected AbstractPreferences getChild(String nodeName) throws BackingStoreException;
   protected abstract String getSpi(String key);
   protected boolean isRemoved();
   protected abstract String[] keysSpi() throws BackingStoreException;
   protected abstract void putSpi(String key, String value);
   protected abstract void removeNodeSpi() throws BackingStoreException;
   protected abstract void removeSpi(String key);
   protected abstract void syncSpi() throws BackingStoreException;
// Защищенные поля экземпляра
   protected final Object lock;
   protected boolean newNode;
}
```

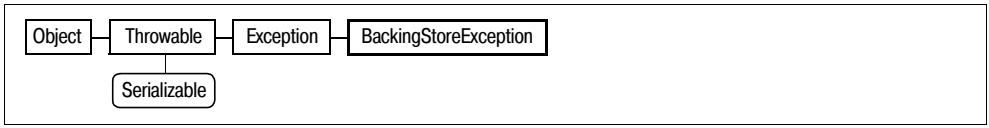
Передается методам: AbstractPreferences.AbstractPreferences()

Возвращается методами: AbstractPreferences.{cachedChildren(), childSpi(), getChild()}

BackingStoreException Java 1.4

java.util.prefs ***сериализуемое, проверяемое***

Это исключение сигнализирует о том, что метод Preferences не может успешно завершиться из-за проблемы с базой данных свойств, возникшей на уровне реализации. Наиболее широко применяются методы класса Preferences, которые не генерируют это исключение и гарантируют успешность их выполнения, даже если свойства уровня реализации недоступны. Несмотря на то что этот класс наследует интерфейс Serializable, реализации могут не быть сериализуемыми.



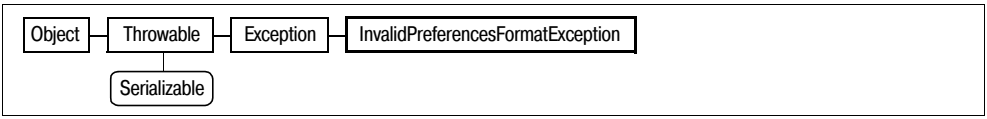

```
public class BackingStoreException extends Exception {
// Открытые конструкторы
   public BackingStoreException(Throwable cause);
   public BackingStoreException(String s);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

InvalidPreferencesFormatException Java 1.4

java.util.prefs ***сериализуемое, проверяемое***

Это исключение сигнализирует о синтаксической ошибке в XML-данных свойств. Несмотря на то что этот класс наследует инетрфейс `Serializable`, реализации могут не быть сериализуемыми.




```
public class InvalidPreferencesFormatException extends Exception {
// Открытые конструкторы
   public InvalidPreferencesFormatException(String message);
   public InvalidPreferencesFormatException(Throwable cause);
   public InvalidPreferencesFormatException(String message, Throwable cause);
}
```

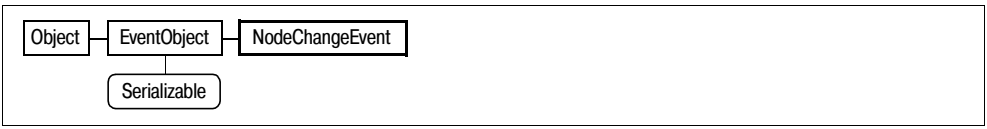
Генерируется методами: `Preferences.importPreferences()`

NodeChangeEvent Java 1.4

java.util.prefs ***сериализуемый, событие***

Объект `NodeChangeEvent` передается методам любых объектов `NodeChangeListener`, зарегистрированных в объекте `Preferences`, когда удаляется или добавляется дочерний узел `Preferences`. Метод `getChild()` возвращает объект `Preferences`, который был добавлен или удален. Метод `getParent()` возвращает родительский узел `Preferences`, дочерний узел которого был добавлен или удален. Этот родительский объект `Preferences` является объектом, в котором был зарегистрирован `NodeChangeListener`.

Несмотря на то что этот класс наследует интерфейс `Serializable`, в действительности он не является сериализуемым.




```
public class NodeChangeEvent extends java.util.EventObject {
// Открытые конструкторы
   public NodeChangeEvent(Preferences parent, Preferences child);
// Открытые методы экземпляра
   public Preferences getChild();
   public Preferences getParent();
}
```

Передается методам: `NodeChangeListener.{childAdded(), childRemoved()}`

NodeChangeListener Java 1.4

java.util.prefs *слушатель события*

Этот интерфейс определяет методы, которые должен реализовывать объект, если он хочет получать уведомления, когда дочерний узел свойств добавляется к объекту Preferences или удаляется из него. Когда происходит такое добавление или удаление, родительский объект Preferences передает объект NodeChangeEvent в соответствующий метод любого объекта NodeChangeListener, который был зарегистрирован с помощью метода Preferences.addNodeChangeListener().


```
public interface NodeChangeListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void childAdded(NodeChangeEvent evt);
   public abstract void childRemoved(NodeChangeEvent evt);
}
```

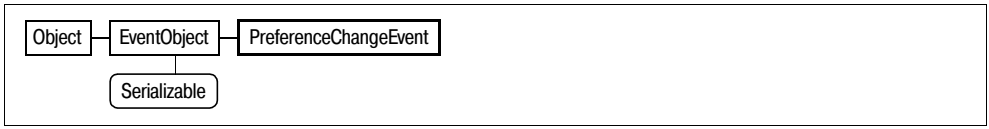
Передается методам: AbstractPreferences.{addNodeChangeListener(), removeNodeChangeListener()}, Preferences.{addNodeChangeListener(), removeNodeChangeListener()}

PreferenceChangeEvent Java 1.4

java.util.prefs *сериализуемый, событие*

Объект PreferenceChangeEvent передается методу preferenceChange() любого объекта PreferenceChangeListener, зарегистрированного в объекте Preferences, когда значения свойств добавляются в узел Preferences, удаляются из него или изменяются. Метод getNode() возвращает вовлеченный объект Preferences. Метод getKey() возвращает имя измененного свойства. Если значение свойства было добавлено или изменено, метод getNewValue() возвращает это значение. Если свойство было удалено, метод getNewValue() возвращает null.

Несмотря на то что этот класс наследует интерфейс Serializable, в действительности он не является сериализуемым.




```
public class PreferenceChangeEvent extends java.util.EventObject {
// Открытые конструкторы
   public PreferenceChangeEvent(Preferences node, String key, String newValue);
// Открытые методы экземпляра
   public String getKey();
   public String getNewValue();
   public Preferences getNode();
}
```

Передается методам: PreferenceChangeListener.preferenceChange()

PreferenceChangeListener Java 1.4

java.util.prefs *слушатель события*

Этот интерфейс определяет метод, который должен быть реализован объектом, если он хочет получать уведомления о добавлении, удалении или изменении пары «ключ/значение» свойства в объекте Preferences. После любого такого изменения объект Preferences передает объект PreferenceChangeEvent, описывающий изменение, в метод preferenceChange() любого объекта PreferenceChangeListener, который был зарегистрирован с помощью метода Preferences.addPreferenceChangeListener().


```
public interface PreferenceChangeListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void preferenceChange(PreferenceChangeEvent evt);
}
```

Передается методам: AbstractPreferences.{addPreferenceChangeListener(), removePreferenceChangeListener()}, Preferences.{addPreferenceChangeListener(), removePreferenceChangeListener()}

Preferences Java 1.4

java.util.prefs

Объект Preferences представляет отображения имен свойств, которые являются строками, чувствительными к регистру, и соответствующих значений свойств. Метод get() дает возможность получить строковое значение именованного свойства, а метод put() позволяет установить строковое значение для именованного свойства. Все значения свойств хранятся как строки, однако существуют разнообразные удобные методы, которые служат для преобразования значений свойств типа boolean, byte[], double, float, int и long в строковое представление, а также для обратного преобразования. Имена этих методов начинаются с «get» и «put».

Метод remove() позволяет удалить именованное свойство, а метод clear() удаляет все значения свойств, хранящихся в объекте Preferences. Метод keys() возвращает массив строк, определяющих имена всех свойств в объекте Preferences.

Значения свойств хранятся в некотором хранилище, определяемом реализацией. Таким хранилищем может быть сервер каталогов LDAP, реестр Windows или любое другое долговременное хранилище. Обратите внимание, что все методы get() этого класса требуют указания значений по умолчанию. Они возвращают значение по умолчанию, если в именованном свойстве не хранится никакого значения или хранилище недоступно. Класс Preferences не зависит от реализации нижнего уровня, за исключением того, что он устанавливает 80-символьное ограничение для имен свойств и имен узлов Preference (см. далее), а также ограничение в 8192 символа для значений свойств.

Класс Preferences не имеет открытого конструктора. Для получения объекта Preferences необходимо использовать один из статических методов, описанных ниже. Каждый объект Preferences является узлом в иерархии узлов Preferences. Существует две независимые иерархии: одна хранит пользовательские свойства, а вторая – системные. Все узлы Preferences (в любой иерархии) имеют уникальные имена и исполь-

зуют такие же соглашения об именовании, как и файловая система Unix. Приложения и классы могут хранить их свойства в узле Preferences с любым именем, но по соглашению должны применяться имена узлов, соответствующие имени пакета приложения или класса; при этом все символы «.» в имени пакета преобразуются в символы «/». Например, узел свойств, используемый java.lang.System, должен называться «/java/lang».

Класс Preferences определяет статические методы, которые можно использовать для получения объектов Preferences. Передайте объект Class в методы systemNodeForPackage() и userNodeForPackage() для получения системного и пользовательского объектов Preferences, определенных для пакета этого класса. Если вам нужен узел Preferences, определенный для отдельного класса, а не для пакета, то вы можете передать имя класса в метод node() узла, возвращенного методом systemNodeForPackage() или userNodeForPackage(). При перемещении по дереву узлов свойств вызывайте методы systemRoot() и userRoot() для получения корневого узла в обеих иерархиях, а затем используйте метод node() для поиска дочерних узлов этих корней.

Разнообразные методы класса Preferences позволяют перемещаться по иерархии свойств. Метод parent() возвращает родительский узел Preferences. Метод childrenNames() возвращает массив относительных имен всех дочерних узлов узла Preferences. Метод node() возвращает именованный объект Preferences из иерархии. Если указанное имя узла начинается с косой черты, то данное имя является абсолютным именем, которое интерпретируется относительно корня иерархии. В противном случае это относительное имя, которое интерпретируется относительно объекта Preferences, у которого был вызван метод node(). Метод nodeExists() позволяет проверить наличие именованного узла. Метод removeNode() удаляет целый узел Preferences из иерархии (это удобно при деинсталляции приложения). Метод name() возвращает простое имя узла Preferences относительно его родительского узла. Метод absoutePath() возвращает полное абсолютное имя узла относительно корня иерархии. И наконец, метод isUserNode() позволяет определить, является ли объект Preferences частью пользовательской или системной иерархии.

Большинство приложений будут однократно считывать значения свойств при запуске. Приложения, которые должны динамически реагировать на изменения свойств (например приложения, тесно интегрированные с рабочим столом), могут использовать метод addPreferenceChangeListener() для регистрации PreferenceChangeListener, чтобы получать уведомления об изменениях свойств в виде объектов PreferenceChangeEvent. Приложения, которым необходимо отслеживать изменения в самой иерархии Preferences, могут регистрировать NodeChangeListener.

Метод put() и разнообразные вспомогательные методы put...(), специфичные для того или иного типа, могут завершать работу асинхронно еще до реального сохранения нового значения свойства в хранилище. Метод flush() принудительно сохраняет любые изменения свойств текущего узла Preferences (и всех его наследников в иерархии). Обратите внимание, что нет необходимости вызывать flush() перед завершением приложения; все свойства со временем будут сохранены в постоянном хранилище. В нескольких виртуальных машинах Java несколько приложений могут устанавливать значения свойств в одном и том же узле Preferences в одно и то же время. Вызов метода sync() гарантирует, что будущие вызовы метода get() и связанных с ним вспомогательных методов получат текущие значения свойств, установленные той или иной виртуальной машиной. Операции flush() и sync() обычно гораздо более трудоемкие, чем операции get() и put(), поэтому приложениям не следует часто ими пользоваться.

Реализации Preferences гарантируют, что все методы этого класса являются потокобезопасными. Если несколько потоков или несколько виртуальных машин парал-

лельно сохраняют одни и те же свойства, то их значения могут перезаписывать друг друга, но данные свойств не будут разрушены. В целях упрощения работы класс Preferences не определяет никаких способов для установки нескольких свойств в одиночной атомарной транзакции. Если нужно гарантировать атомарность для нескольких значений свойств, определите формат данных, позволяющий сохранять все необходимые значения в одной строке. Устанавливайте и получайте эти значения одним вызовом put() или get().

Содержимое узла Preferences или содержимое узла и всех его наследников можно экспортировать как файл XML с помощью методов exportNode() и exportSubtree(). Статический метод importPreferences() считывает экспортированный файл XML в иерархию свойств. Эти методы позволяют резервировать данные свойств и переносить свойства между системами или пользователями.

До Java 1.4 свойства приложений иногда обрабатывались с помощью объекта java.util.Properties.

```
public abstract class Preferences {
// Защищенные конструкторы
   protected Preferences();
// Открытые константы
   public static final int MAX_KEY_LENGTH;                                    // =80
   public static final int MAX_NAME_LENGTH;                                   // =80
   public static final int MAX_VALUE_LENGTH;                                  // =8192
// Открытые методы экземпляра
   public static void importPreferences(java.io.InputStream is)
        throws java.io.IOException, InvalidPreferencesFormatException;
   public static Preferences systemNodeForPackage(Class c);
   public static Preferences systemRoot();
   public static Preferences userNodeForPackage(Class c);
   public static Preferences userRoot();
// Методы регистрации событий (по имени события)
   public abstract void addNodeChangeListener(NodeChangeListener ncl);
   public abstract void removeNodeChangeListener(NodeChangeListener ncl);
   public abstract void addPreferenceChangeListener(PreferenceChangeListener pcl);
   public abstract void removePreferenceChangeListener(PreferenceChangeListener pcl);
// Открытые методы экземпляра
   public abstract String absolutePath();
   public abstract String[] childrenNames() throws BackingStoreException;
   public abstract void clear() throws BackingStoreException;
   public abstract void exportNode(java.io.OutputStream os) throws java.io.IOException,
                                                                   BackingStoreException;
   public abstract void exportSubtree(java.io.OutputStream os)
                                      throws java.io.IOException,BackingStoreException;
   public abstract void flush() throws BackingStoreException;
   public abstract String get(String key, String def);
   public abstract boolean getBoolean(String key, boolean def);
   public abstract byte[] getByteArray(String key, byte[] def);
   public abstract double getDouble(String key, double def);
   public abstract float getFloat(String key, float def);
   public abstract int getInt(String key, int def);
   public abstract long getLong(String key, long def);
   public abstract boolean isUserNode();
   public abstract String[] keys() throws BackingStoreException;
   public abstract String name();
   public abstract Preferences node(String pathName);
   public abstract boolean nodeExists(String pathName) throws BackingStoreException;
```

```
   public abstract Preferences parent();
   public abstract void put(String key, String value);
   public abstract void putBoolean(String key, boolean value);
   public abstract void putByteArray(String key, byte[] value);
   public abstract void putDouble(String key, double value);
   public abstract void putFloat(String key, float value);
   public abstract void putInt(String key, int value);
   public abstract void putLong(String key, long value);
   public abstract void remove(String key);
   public abstract void removeNode() throws BackingStoreException;
   public abstract void sync() throws BackingStoreException;
// Открытые методы, замещающие Object
   public abstract String toString();
}
```

Подклассы: AbstractPreferences

Передается методам: NodeChangeEvent.NodeChangeEvent(), PreferenceChangeEvent.PreferenceChangeEvent()

Возвращается методами: AbstractPreferences.{node(), parent()}, NodeChangeEvent.{getChild(), getParent()}, PreferenceChangeEvent.getNode(), Preferences.{node(), parent(), systemNodeForPackage(), systemRoot(), userNodeForPackage(), userRoot()}, PreferencesFactory.{systemRoot(), userRoot()}

PreferenceFactory

Java 1.4

java.util.prefs

Интерфейс PreferencesFactory определяет методы-фабрики, используемые статическими методами класса Preferences, чтобы получить корневые узлы Preferences для пользовательских и системных иерархий свойств. Прикладным программистам никогда не нужно использовать этот интерфейс.

Реализация API свойств для конкретного хранилища данных должна содержать реализацию данного интерфейса, которая работает с этим хранилищем. Рализации Java от Sun включают реализацию по умолчанию, основанную на файловой системе. Эту реализацию можно подменить, указав имя реализации PreferencesFactory как значение системного свойства java.util.prefs.PreferencesFactory.

```
public interface PreferencesFactory {
// Открытые методы экземпляра
   public abstract Preferences systemRoot();
   public abstract Preferences userRoot();
}
```

Пакет java.util.regex

Java 1.4

Этот маленький пакет предоставляет средства для сравнения текста и паттернов с помощью регулярных выражений. Объекты Pattern представляют регулярные выражения, которые строятся на основе синтаксиса, очень близкого к синтаксису языка программирования Perl. Класс Matcher инкапсулирует Pattern и строку текста; он определяет разнообразные методы для сравнения паттерна с текстом. Все методы классов Pattern и Matcher, задающие текст для сравнения, задают этот текст в виде

java.lang.CharSequence. Интерфейс CharSequence появился в Java 1.4. Он реализуется классами String и StringBuffer, а также классом java.nio.CharBuffer из нового API ввода/вывода.

В дополнение к методам сравнения текста и паттернов, определенным в этом пакете, в Java 1.4 класс java.lang.String расширен большим количеством полезных методов для сравнения строк с регулярными выражениями, представленными в текстовом виде (строками), а не в скомпилированном виде (объектами Pattern). Приложения, которым нужны простые сравнения, могут использовать эти методы; им может никогда не понадобиться напрямую использовать класс Pattern или Matcher.

Классы

```
public final class Matcher;
public final class Pattern implements Serializable;
```

Исключения

```
public class PatternSyntaxException extends IllegalArgumentException;
```

Matcher Java 1.4

java.util.regex

Объект Matcher инкапсулирует регулярное выражение и строку текста (Pattern и java.lang.CharSequence) и определяет методы для сравнения паттерна и текста несколькими различными способами, а также методы для получения подробностей о сравнении текста с паттерном и методы для выполнения операций поиска/замены в тексте. Matcher не имеет ни одного открытого конструктора. Matcher можно получить, передав последовательность символов для сравнения в метод matcher() требуемого объекта Pattern. Существующий объект Matcher можно повторно использовать с новой последовательностью символов (но с тем же объектом Pattern), передав новый объект CharSequence методу reset() модуля сравнения.

После создания или восстановления Matcher можно выполнять различные сравнения регулярных выражений и последовательностей символов. Простейшее сравнение выполняется методом matches(). Он возвращает true, если паттерн соответствует последовательности символов, и false в противном случае. Метод lookingAt() похож на предыдущий: он возвращает true, если паттерн соответствует всей последовательности символов или некоторой подпоследовательности в начале текста. Если паттерн не соответствует началу текста, метод lookingAt() возвращает false. Метод matches() требует, чтобы паттерн соответствовал началу и концу текста, а метод lookingAt() проверяет только соответствие паттерна началу текста. Метод find() не предъявляет ни одного из этих требований: он возвращает true, если паттерн соответствует любой части текста. Как будет показано далее, у метода find() есть особенности, благодаря которым его можно применять в цикле для поиска всех совпадений в тексте.

Если matches(), lookingAt() или find() возвращают true, то несколько других методов класса Matcher можно использовать для получения подробностей о совпавшем тексте. Метод start(), не принимающий аргументов, возвращает индекс первого символа, соответствующего паттерну (конечно, для методов matches() и lookingAt() это должен быть нуль). Метод end(), не принимающий аргументов, возвращает индекс последнего символа, соответствующего паттерну, увеличенный на единицу (или индекс первого символа, следующего за совпавшим текстом). Некоторые регулярные выражения могут соответствовать пустой строке. В таких случаях метод end() возвращает такое же значение, как и start().

Регулярные выражения могут содержать подвыражения, группируемые с помощью круглых скобок, которые известны как «группы захвата» («capturing groups»). В процессе работы с регулярными выражениями часто бывает удобно получить информацию о тексте, соответствующем этим группам. Версии методов `start()` и `end()`, не принимающие аргументов, получают номер группы и возвращают индекс первого символа, соответствующего этой группе, и индекс первого символа, следующего за текстом, соответствующим данной группе. Аналогично, метод `group()` получает номер группы и возвращает строку, которая содержит текст, соответствующий группе (версия метода `group()`, не принимающая аргументов, возвращает текст, соответствующий регулярному выражению). Группы нумеруются, начиная с 1 (группа 0 – это полное регулярное выражение), и располагаются слева направо внутри регулярного выражения. Если существуют вложенные группы, их расположение основывается на позиции открывающей левой скобки, начинающей группу.

У версии метода `find()` есть особенности, которые делают ее удобной для использования в цикле при поиске всех совпадений паттерна со строкой. Во время первого вызова метода `find()` после создания объекта `Matcher` или после вызова метода `reset()` инициируется поиск с начала строки. Если метод находит совпадение, он сохраняет начальную и конечную позиции совпавшего текста. Если в это время не вызывается метод `reset()`, то следующий вызов `find()` снова выполняет поиск, начиная с первого символа после совпадения – в позиции, возвращенной методом `end()`. Если при предыдущем вызове метода `find()` найдена пустая строка, то следующий вызов начинает поиск с позиции `end()+1`. Таким образом можно найти все совпадения с паттерном, многократно вызывая `find()`, пока он не вернет `false`, указывая на то, что совпадений не найдено. После каждого повторного вызова `find()` можно использовать методы `start()`, `end()` и `group()` для получения дополнительной информации о тексте, совпавшем с паттерном и любым его подпаттерном.

`Matcher` также определяет методы, выполняющие операции поиска/замены. Метод `replaceFirst()` ищет последовательность символов для первой подпоследовательности, совпавшей с паттерном. Затем он возвращает строку, являющуюся последовательностью символов, в которой совпавший текст заменен указанной строкой. Метод `replaceAll()` похож на предыдущий, но заменяет все совпадающие подпоследовательности в последовательности символов, а не только первую из них. Строка замены, передаваемая в методы `replaceFirst()` и `replaceAll()`, не всегда вставляется буквально. Если строка замены содержит знак доллара, за которым следует целое число, являющееся допустимым номером группы, то знак доллара и номер заменяются текстом, совпадающим с нумерованной группой. Если знак доллара необходимо включить в строку замены, поставьте перед ним обратную косую черту.

Методы `replaceFirst()` и `replaceAll()` являются удобными методами, охватывающими наиболее типичные случаи поиска/замены. Кроме того, `Matcher` определяет низкоуровневые методы, которые можно использовать для выполнения специальных операций поиска/замены в совокупности с вызовами `find()` и сборкой модифицированной строки в объекте `StringBuffer`. Для того чтобы понять процедуру поиска/замены, вам нужно знать, что `Matcher` поддерживает «позицию сцепления», которой присваивается нуль при создании `Matcher`. Нулевое значение этой позиции восстанавливается методом `reset()`. Метод `appendReplacement()` предназначен для использования после успешного вызова метода `find()`. Он копирует в указанный строковый буфер весь текст между позицией сцепления и символом, стоящим перед позицией `start()` для последнего совпадения. Затем он добавляет указанный текст замены в этот строковый буфер, выполняя такую же замену, как и метод `replaceAll()`. И наконец, он устанавливает позицию сцепления в `end()` для последнего совпадения, поэтому последую-

щие вызовы appendReplacement() начинают работу с нового символа. Метод appendReplacement() предназначен для использования после вызова find(), вернувшего true. Когда метод find() не может найти других совпадений и возвращает false, нужно закончить операцию замены, вызвав appendTail(): этот метод копирует весь текст между позицией end() последнего совпадения и концом последовательности символов в указанный StringBuffer.

Метод reset() упоминался несколько раз. Он стирает любую сохраненную информацию о последнем совпадении и восстанавливает Matcher в исходное состояние так, что последующие вызовы методов find() и appendReplacement() приступают к работе с начала последовательности символов. Версия метода reset(), не принимающая аргументов, позволяет задать новую последовательность символов для выполнения повторного поиска совпадения. Важно понимать, что несколько других методов класса Matcher вызывают reset() самостоятельно перед непосредственным выполнением операций. Вот эти методы: matches(), lookingAt(), версия find() без аргументов, replaceAll() и replaceFirst().

Класс Matcher не является потокобезопасным; нескольким потокам не следует обращаться к нему в одно и то же время.

```
public final class Matcher {
// Конструктор отсутствует
// Открытые методы экземпляра
   public Matcher appendReplacement(StringBuffer sb, String replacement);
   public StringBuffer appendTail(StringBuffer sb);
   public int end();
   public int end(int group);
   public boolean find();
   public boolean find(int start);
   public String group();
   public String group(int group);
   public int groupCount();
   public boolean lookingAt();
   public boolean matches();
   public Pattern pattern();
   public String replaceAll(String replacement);
   public String replaceFirst(String replacement);
   public Matcher reset();
   public Matcher reset(CharSequence input);
   public int start();
   public int start(int group);
}
```

Возвращается методами: Matcher.{appendReplacement(), reset()}, Pattern.matcher()

Pattern

Java 1.4

java.util.regex ***сериализуемый***

Этот класс представляет регулярное выражение. Он не имеет открытых конструкторов; Pattern можно получить вызовом одного из статических методов compile(), передавая ему строковое представление регулярного выражения и необязательную битовую маску флагов, изменяющую поведение регулярного выражения. Методы pattern() и flags() возвращают строковое представление регулярного выражения и битовую маску, переданную методу compile().

Если необходимо выполнить только одну операцию сравнения с регулярным выражением и не нужны какие-либо флаги, то можно не создавать объект Pattern: просто передайте в статический метод matches() строковое представление паттерна и CharSequence для сравнения. Метод возвращает true, если указанный паттерн полностью соответствует указанному тексту, или возвращает false в противном случае.

Pattern представляет регулярное выражение, но в действительности он не определяет никаких примитивных методов для сравнения регулярных выражений с текстом. Для выполнения сравнения нужно создать объект Matcher, инкапсулирующий паттерн и текст для сравнения. При вызове метода matcher() следует указать объект CharSequence, с которым нужно выполнять сравнение. Для получения дополнительной информации обратитесь к описанию Matcher.

Методы split() являются исключением из правила, согласно которому вы должны получить Matcher для работы с объектом Pattern. Тем не менее эти методы создают и используют Matcher внутри себя. Они получают CharSequence на входе и разбивают его на подстроки, используя в качестве разделителя текст, соответствующий регулярному выражению, и возвращая подстроки в виде String[]. Версия split(), принимающая два аргумента, получает целое число, определяющее максимальное количество подстрок, на которые можно разбить исходный текст.

Pattern определяет флаги, управляющие разнообразными аспектами сравнения с регулярным выражением. Вот эти флаги:

CANON_EQ

Стандарт Unicode дает возможность использовать несколько способов представления одного и того же символа. Если этот флаг установлен, символы сравниваются на основе их полного канонического разложения. Таким образом, символы будут совпадать, даже если они представлены разными способами. Установка этого флага обычно снижает производительность. В отличие от всех остальных флагов, этот флаг нельзя временно установить внутри паттерна.

CASE_INSENSITIVE

Сравнивать буквы, не учитывая регистр. По умолчанию влияние этого флага распространяется только на сравнение букв ASCII. Флаг UNICODE_CASE позволяет игнорировать регистр всех символов Unicode. Установить этот флаг внутри паттерна можно с помощью (?i).

COMMENTS

Если этот флаг установлен, то пробелы и комментарии внутри паттерна игнорируются. Комментариями являются все символы между # и концом строки. Установить этот флаг внутри паттерна можно с помощью (?x).

DOTALL

Если этот флаг установлен, то выражение . (точка) соответствует любому символу. Если он не установлен, то данное выражение не соответствует символам завершения строки. Такой режим называется однострочным. Вы можете установить этот флаг внутри паттерна с помощью (?s).

MULTILINE

Если этот флаг установлен, то якоря ^ и $ соответствуют не только началу и концу входной строки, но и началу и концу любой строки внутри этой строки. В паттерне можно установить этот флаг с помощью (?m).

UNICODE_CASE

Если этот флаг установлен вместе с флагом CASE_INSENSITIVE, то сравнение без учета регистра выполняется для всех букв Unicode, а не только для букв ASCII. Вы можете установить оба флага внутри паттерна с помощью (?iu).

UNIX_LINES

Если этот флаг установлен, то для выражений . (точка), ^ и $ только символ новой строки считается признаком конца строки. Если флаг не установлен, то признаком конца строки являются символ новой строки (\n), символ возврата каретки (\r), последовательность символов возврата каретки и новой строки (\r\n), а также символы Unicode \u0085 («новая строка«), \u2028 («разделитель строк») и \u2029 («разделитель абзацев»). Вы можете включить этот флаг внутри паттерна с помощью (?d).

API класса Pattern достаточно прост, однако синтаксис текстового представления регулярных выражений довольно сложен. Полное рассмотрение регулярных выражений выходит за рамки этой книги. Таблица 17.1 является лишь кратким справочником по синтаксису регулярных выражений. Он очень похож на синтаксис, используемый языком Perl. Многие элементы синтаксиса содержат символ обратной косой черты, например выражение \d для сравнения с одной из цифр 0–9. Поскольку строки Java используют символ обратной косой черты в качестве управляющего символа, то для представления регулярных выражений в виде строковых литералов данный символ необходимо удваивать: «\\d». Для получения более полной информации по регулярным выражениям обратитесь к книгам «Programming Perl»[1] и «Mastering Regular Expressions»[2] (обе книги выпущены издательством O'Reilly).

Таблица 17.1. Обзор регулярных выражений в Java

Синтаксис	Соответствие
Одиночные символы	
x	Символ x, если x – не символ пунктуации со специальным значением в синтаксисе регулярного выражения.
\p	Символ пунктуации p.
\\	Символ обратной косой черты.
\n	Символ новой строки \u000A.
\t	Символ табуляции \u0009.
\r	Символ возврата каретки \u000D.
\f	Символ перевода страницы \u000C.
\e	Управляющий символ \u001B.
\a	Символ звонка (сигнала) \u0007.
\u*xxxx*	Символ Unicode с шестнадцатеричным кодом *xxxx*.
\x*xx*	Символ с шестнадцатеричным кодом *xx*.
\0*n*	Символ с восьмеричным кодом *n*.
\0*nn*	Символ с восьмеричным кодом *nn*.

[1] Ларри Уолл, Том Кристиансен и Джон Орвант «Программирование на Perl», 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2002.

[2] Дж. Фридл «Регулярные выражения». – Пер. с англ. – СПб: Питер, 2003.

Синтаксис	Соответствие
\0*nnn*	Символ с восьмеричным кодом *nnn*, для которого *nnn* <= 377.
\c*x*	Управляющий символ ^*x*.
Классы символов	
[...]	Один из символов внутри скобок. Символы могут быть заданы буквально, но синтаксис также позволяет задавать диапазоны символов с операторами пересечения, объединения и вычитания. См. следующие примеры.
[^...]	Любой символ, не присутствующий в скобках.
[a-z0-9]	Диапазон символов: символ между a и z или 0 и 9 (границы включаются в диапазон).
[0-9[a-fA-F]]	Объединение классов: то же, что и [0-9a-fA-F].
[a-z&&[aeiou]]	Пересечение классов: то же, что и [aeiou].
[a-z&&[^aeiou]]	Вычитание: символы от a до z, исключая гласные.
.	Любой символ, кроме символа конца строки. Если флаг DOTALL установлен, то символ . соответствует любому символу, включая символ конца строки.
\d	Цифра ASCII: [0-9].
\D	Любой символ, кроме цифр ASCII: [^\d].
\s	Пробельный символ ASCII: [\t\n\f\r\x0B].
\S	Любой символ, кроме пробельного символа ASCII: [^\s].
\w	Алфавитно-цифровой символ ASCII: [a-zA-Z0-9_].
\W	Любой символ, кроме алфавитно-цифрового символа ASCII: [^\w].
\p{*group*}	Любой символ в именованной группе. См. далее имена групп. Многие имена групп взяты из стандарта POSIX, поэтому для обозначения этого класса символов используется p.
\P{*group*}	Любой символ, не принадлежащий к именованной группе.
\p{Lower}	Буквы нижнего регистра ASCII: [a-z].
\p{Upper}	Буквы верхнего регистра ASCII: [A-Z].
\p{ASCII}	Любой символ ASCII: [\x00-\x7f].
\p{Alpha}	Буква ASCII: [a-zA-Z].
\p{Digit}	Цифра ASCII: [0-9].
\p{XDigit}	Шестнадцатеричная цифра: [0-9a-fA-F].
\p{Alnum}	Буква или цифра ASCII: [\p{Alpha}\p{Digit}].
\p{Punct}	Знак пунктуации ASCII: один из символов !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~].
\p{Graph}	Видимый символ ASCII: [\p{Alnum}\p{Punct}].
\p{Print}	Видимый символ ASCII: то же, что и \p{Graph}.
\p{Blank}	Пробел или табуляция ASCII: [\t].
\p{Space}	Пробельный символ ASCII: [\t\n\f\r\x0b].
\p{Cntrl}	Управляющий символ ASCII: [\x00-\x1f\x7f].

Таблица 17.1 (продолжение)

Синтаксис	Соответствие
\p{*category*}	Любой символ в именованной категории Unicode. Имена категорий представлены одно- или двухбуквенными кодами, которые определены стандартом Unicode. Однобуквенные коды включают в себя L для буквы, N для числа, S для символа, Z для разделителя и P для пунктуации. Двухбуквенные коды представляют подкатегории: Lu для буквы верхнего регистра, Nd для десятичной цифры, Sc для символа валюты, Sm для математического символа и Zs для пробельного разделителя. В описании java.lang.Character представлена информация о наборе констант, соответствующих этим подкатегориям. Обратите внимание на то, что в данной книге не приведен полный набор одно- и двухбуквенных кодов.
\p{*block*}	Любой символ в именованном блоке Unicode. В регулярных выражениях Java имена блоков начинаются с «In»; за этим символом следует имя блока Unicode, записанное заглавными и прописными буквами без пробелов и подчеркиваний. Например: \p{InOgham} или \p{InMathematicalOperators}. В описании java.lang.Character.UnicodeBlock представлен полный список имен блоков Unicode.
Последовательности, альтернативы, группы и ссылки	
xy	Соответствует *x* с последующим *y*.
x|*y*	Соответствует *x* или *y*.
(...)	Группировка. Группирует подвыражение в круглых скобках в единый модуль, который можно применять с *, +, ?, | и т. д. Также «захватывает» символы, соответствующие этой группе, для последующего использования.
(?:...)	Только группировка. Подобно (), группирует подвыражение, но не захватывает соответствующий текст.
n	Соответствует символам, которые совпали, когда впервые совпала «группа захвата» с номером *n*. Будьте внимательны, если за *n* следует другая цифра: при этом будет задействован самый большой из допустимых номеров группы.
Повторяемость[a]	
x?	Не более одного вхождения *x*; то есть *x* является необязательным.
*x**	Ноль или более вхождений *x*.
x+	Одно или более вхождений *x*.
x{*n*}	*n* вхождений *x*.
x{*n*,}	*n* или более вхождений *x*.
x{*n*,*m*}	Как минимум *n* и как максимум *m* вхождений *x*.
Якори[b]	
^	Начало входной строки или, если указан флаг MULTILINE, начало всей строки или любой новой строки.
$	Конец входной строки или, если указан флаг MULTILINE, конец всей строки или строки внутри этой строки.
\b	Граница слова: позиция в строке между алфавитно-цифровым и не алфавитно-цифровым символом.
\B	Позиция в строке, не являющаяся границей слова.

Синтаксис	Соответствие
\A	Начало входной строки. Похоже на ^, но никогда не соответствует началу новой строки, вне зависимости от установленных флагов.
\Z	Конец входной строки, игнорирующий любой завершающий символ конца строки.
\z	Конец входной строки, включая любой символ конца строки.
\G	Конец предыдущего совпадения.
(?=*x*)	Положительное опережающее утверждение. Требует, чтобы последующие символы соответствовали *x*, но не включает эти символы в строку совпадения.
(?!*x*)	Отрицательное опережающее утверждение. Требует, чтобы последующие символы не соответствовали паттерну *x*.
(?<=*x*)	Положительное запаздывающее утверждение. Требует, чтобы символы непосредственно перед этой позицией соответствовали *x*, но не включает эти символы в совпадение. *x* должен быть паттерном с фиксированным количеством символов.
(?<!*x*)	Отрицательное запаздывающее утверждение. Требует, чтобы символы непосредственно перед этой позицией не соответствовали *x*. *x* должен быть паттерном с фиксированным количеством символов.
Разное	
(?>*x*)	Соответствует *x* вне зависимости от остальной части выражения. Не учитывается, вызовет ли это совпадение отсутствие совпадений в оставшейся части регулярного выражения. Удобно для оптимизации сложных регулярных выражений. Эта разновидность группы не захватывает совпавший текст.
(?*onflags-offflags*)	Не соответствует никаким символам, но устанавливает флаги, указанные как *onflags*, и снимает флаги, указанные как *offflags*. Эти две строки представляют собой комбинации букв, расположенных в произвольном порядке и соответствующих нижеприведеннымм константам класса Pattern: i (CASE_INSENSITIVE), d (UNIX_LINES), m (MULTILINE), s (DOTALL), u (UNICODE_CASE) и x (COMMENTS). Установки флагов, представленные таким образом, оказывают влияние с той позиции в выражении, где они появляются. Данные флаги действуют до конца выражения или до конца группы в круглых скобках, частью которой они являются, или до тех пор, пока они не будут отменены другим выражением установки флагов.
(?*onflags-offflags*:*x*)	Соответствует подвыражению *x*. Указанные флаги применяются только к этому подвыражению. Подобно (?:...), это незахватывающая группа, но у нее есть дополнительные флаги.
\Q	Не соответствует никаким символам, но экранирует (quote) весь последующий текст паттерна до появления \E. При сравнении все символы внутри этой секции интерпретируются как буквенные символы, ни один из которых не имеет специального значения (за исключением \E).
\E	Не соответствует никаким символам; завершает строку, начатую с помощью \Q.
#*comment*	Если флаг COMMENT установлен, текст паттерна между # и концом строки считается комментарием и игнорируется.

[a] Эти символы повторения известны как «жадные квантификаторы», поскольку они соответствуют максимально возможному количеству вхождений *X*, пока для остальной части регулярного выражения сохраняется возможность совпадения с паттерном. Поместите после этих квантификаторов знак вопроса, если вам необходим «ленивый квантификатор», который соответствует минимальному количеству вхождений, пока для остальной части регулярного выражения сохраняется возможность совпадения с паттерном. Например, используйте `*?` вместо `*` и `{2,}?` вместо `{2,}`. Если после квантификатора поставить знак «плюс» вместо знака вопроса, то будет определен «притяжательный квантификатор», который соответствует максимально возможному количеству вхождений *X*, даже если это будет означать отсутствие совпадения в оставшейся части регулярного выражения. Притяжательные квантификаторы полезны, когда вы уверены, что они не окажут отрицательного влияния на остальную часть соответствия, поскольку они могут быть реализованы более эффективно, чем обычные жадные квантификаторы.

[b] Якори не соответствуют символам; они находят поля нулевой ширины между символами и ставят «на якорь» соответствие позиции, в которой выполняется конкретное условие.


```
public final class Pattern implements Serializable {
// Конструктор отсутствует
// Открытые константы
   public static final int CANON_EQ;                                    // =128
   public static final int CASE_INSENSITIVE;                            // =2
   public static final int COMMENTS;                                    // =4
   public static final int DOTALL;                                      // =32
   public static final int MULTILINE;                                   // =8
   public static final int UNICODE_CASE;                                // =64
   public static final int UNIX_LINES;                                  // =1
// Открытые методы экземпляра
   public static Pattern compile(String regex);
   public static Pattern compile(String regex, int flags);
   public static boolean matches(String regex, CharSequence input);
// Открытые методы экземпляра
   public int flags();
   public Matcher matcher(CharSequence input);
   public String pattern();
   public String[] split(CharSequence input);
   public String[] split(CharSequence input, int limit);
}
```

Возвращается методами: `Matcher.pattern()`, `Pattern.compile()`

PatternSyntaxException

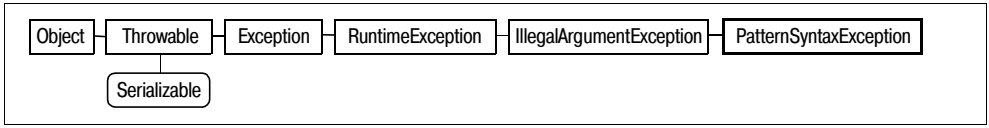
Java 1.4

java.util.regex ***сериализуемое, непроверяемое***

Это исключение сигнализирует о синтаксической ошибке в текстовом представлении регулярного выражения. Исключение этого типа может быть сгенерировано методами `Pattern.compile()` и `Pattern.matches()`, а также методами `matches()`, `replaceFirst()`, `replaceAll()` и `split()` класса `String`, которые вызывают эти методы класса `Pattern`.

Метод `getPattern()` возвращает текст, содержащий синтаксическую ошибку, а `getIndex()` выдает приблизительное место появления ошибки в тексте или –1, если место не известно. Метод `getDescription()` возвращает сообщение об ошибке, которое предо-

ставляет дополнительную информацию о ней. Унаследованный метод `getMessage()` объединяет информацию, предоставляемую этими тремя методами, в одно многострочное сообщение.




```
public class PatternSyntaxException extends IllegalArgumentException {
// Открытые конструкторы
   public PatternSyntaxException(String desc, String regex, int index);
// Открытые методы экземпляра
   public String getDescription();
   public int getIndex();
   public String getPattern();
// Открытые методы, замещающие Throwable
   public String getMessage();
}
```

Пакет java.util.zip

Java 1.1

Пакет `java.util.zip` содержит классы для сжатия и распаковки данных. Классы `Deflater` и `Inflater` выполняют сжатие и распаковку данных. `DeflaterOutputStream` и `InflaterInputStream` применяют эту функциональность к потокам байтов; подклассы этих потоков реализуют два формата упаковки – GZIP и ZIP. Классы `Adler32` и `CRC32` реализуют интерфейс `Checksum` и вычисляют контрольные суммы, необходимые для сжатия данных.

Интерфейсы

```
public interface Checksum;
```

Классы

```
public class Adler32 implements Checksum;
public class CheckedInputStream extends java.io.FilterInputStream;
public class CheckedOutputStream extends java.io.FilterOutputStream;
public class CRC32 implements Checksum;
public class Deflater;
public class DeflaterOutputStream extends java.io.FilterOutputStream;
   └ public class GZIPOutputStream extends DeflaterOutputStream;
   └ public class ZipOutputStream extends DeflaterOutputStream;
public class Inflater;
public class InflaterInputStream extends java.io.FilterInputStream;
   └ public class GZIPInputStream extends InflaterInputStream;
   └ public class ZipInputStream extends InflaterInputStream;
public class ZipEntry implements Cloneable;
public class ZipFile;
```

Исключения

```
public class DataFormatException extends Exception;
public class ZipException extends java.io.IOException;
```

Adler32 Java 1.1

java.util.zip

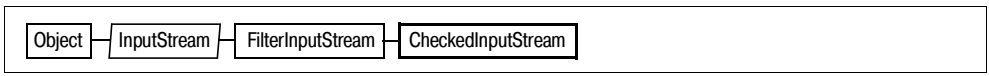
Этот класс реализует интерфейс Checksum и вычисляет контрольную сумму потока данных, используя алгоритм Adler-32. Данный алгоритм намного быстрее алгоритма CRC-32 и почти так же надежен. Классы CheckedInputStream и CheckedOutputStream предоставляют высокоуровневый интерфейс для вычисления контрольных сумм потоков данных.


```
public class Adler32 implements Checksum {
// Открытые конструкторы
   public Adler32();
// Открытые методы экземпляра
   public void update(byte[] b);
// Методы, реализующие Checksum
   public long getValue();  // по умолчанию:1
   public void reset();
   public void update(int b);
   public void update(byte[] b, int off, int len);
}
```

CheckedInputStream Java 1.1

java.util.zip

Этот класс является подклассом java.io.FilterInputStream; он позволяет одновременно читать поток и вычислять контрольную сумму для его содержимого. Это удобно, когда вы хотите проверить целостность потока данных по известному значению контрольной суммы. Для создания CheckedInputStream нужно указать поток для чтения и объект Checksum, например CRC32, который реализует выбранный алгоритм вычисления контрольной суммы. Методы read() и skip() аналогичны соответствующим методам в других потоках ввода. Прочитанные байты задействуются при вычислении контрольной суммы. Метод getChecksum() возвращает не значение контрольной суммы, а объект Checksum. Чтобы получить значение контрольной суммы, необходимо вызвать метод getValue() этого объекта.

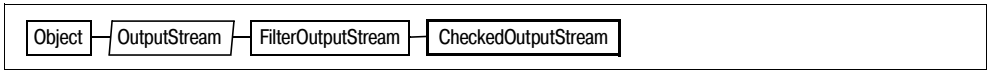



```
public class CheckedInputStream extends java.io.FilterInputStream {
// Открытые конструкторы
   public CheckedInputStream(java.io.InputStream in, Checksum cksum);
// Открытые методы экземпляра
   public Checksum getChecksum();
// Открытые методы, замещающие FilterInputStream
   public int read() throws java.io.IOException;
   public int read(byte[] buf, int off, int len) throws java.io.IOException;
   public long skip(long n) throws java.io.IOException;
}
```

CheckedOutputStream Java 1.1

java.util.zip

Этот класс является подклассом java.io.FilterOutputStream; он позволяет одновременно записывать данные в поток и вычислять контрольную сумму для этих данных. При создании CheckedOutputStream необходимо указать выходной поток для записи и объект Checksum, например экземпляр класса Adler32, который реализует выбранный алгоритм вычисления контрольной суммы. Методы write() идентичны соответствующим методам других классов OutputStream. Метод getChecksum() возвращает объект Checksum. Для получения реального значения контрольной суммы следует вызвать метод getValue() этого объекта.




```
public class CheckedOutputStream extends java.io.FilterOutputStream {
// Открытые конструкторы
   public CheckedOutputStream(java.io.OutputStream out, Checksum cksum);
// Открытые методы экземпляра
   public Checksum getChecksum();
// Открытые методы, замещающие FilterOutputStream
   public void write(int b) throws java.io.IOException;
   public void write(byte[] b, int off, int len) throws java.io.IOException;
}
```

Checksum Java 1.1

java.util.zip

Этот интерфейс определяет методы, необходимые при вычислении контрольной суммы для потока данных. Контрольная сумма вычисляется на основе байтов данных, предоставляемых методами update(); текущее значение контрольной суммы можно получить с помощью метода getValue(). Метод reset() присваивает контрольной сумме значение по умолчанию; этот метод следует применять перед организацией нового потока данных. Значение контрольной суммы, вычисляемое объектом Checksum и возвращаемое через метод getValue(), должно помещаться в переменную типа long. Поэтому данный интерфейс не удобен для криптографических алгоритмов вычисления контрольной суммы, используемых в криптографии и при защите информации. Классы CheckedInputStream и CheckedOutputStream предоставляют высокоуровневый API для вычисления контрольной суммы потока данных. См. также java.security.MessageDigest.

```
public interface Checksum {
// Открытые методы экземпляра
   public abstract long getValue();
   public abstract void reset();
   public abstract void update(int b);
   public abstract void update(byte[] b, int off, int len);
}
```

Реализации: Adler32, CRC32

Передается методам: CheckedInputStream.CheckedInputStream(), CheckedOutputStream.CheckedOutputStream()

Возвращается методами: `CheckedInputStream.getChecksum()`, `CheckedOutputStream.getChecksum()`

CRC32 Java 1.1

java.util.zip

Этот класс реализует интерфейс `Checksum` и вычисляет контрольную сумму для потока данных, используя алгоритм CRC-32. Классы `CheckedInputStream` и `CheckedOutputStream` предоставляют высокоуровневый интерфейс для вычисления контрольных сумм потоков данных.

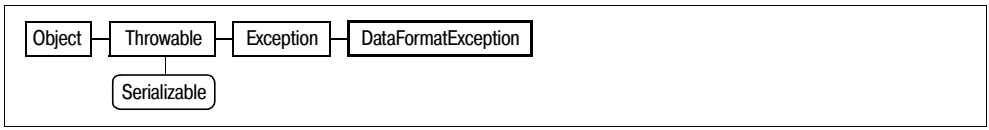

```
public class CRC32 implements Checksum {
// Открытые конструкторы
   public CRC32();
// Открытые методы экземпляра
   public void update(byte[] b);
// Методы, реализующие Checksum
   public long getValue();                                    // по умолчанию:0
   public void reset();
   public void update(int b);
   public void update(byte[] b, int off, int len);
}
```

Экземпляры: `GZIPInputStream.crc`, `GZIPOutputStream.crc`

DataFormatException Java 1.1

java.util.zip ***сериализуемое, проверяемое***

Сигнализирует о том, что в процессе распаковки были обнаружены неправильные или поврежденные данные.




```
public class DataFormatException extends Exception {
// Открытые конструкторы
   public DataFormatException();
   public DataFormatException(String s);
}
```

Генерируется методами: `Inflater.inflate()`

Deflater Java 1.1

java.util.zip

Этот класс реализует общий алгоритм сжатия данных ZLIB, используемый программами сжатия *gzip* и *PKZip*. Константы, определенные этим классом, применяются

для задания стратегии сжатия и уровня скорости/плотности сжатия. Если аргумент nowrap в конструкторе равен true, то заголовок ZLIB и данные контрольной суммы не включаются в сжатый вывод. Такой способ соответствует формату, используемому в *gzip* и *PKZip*.

Важными методами этого класса являются setInput(), который определяет входные данные для сжатия, и deflate(), который сжимает данные и возвращает сжатый вывод. Остальные методы позволяют применять Deflater для потокоориентированного сжатия, которое выполняется с помощью высокоуровневых классов, таких как GZIPOutputStream и ZipOutputStream. Этих классов, основанных на потоках, достаточно в большинстве случаев. Многим приложениям не нужно использовать Deflater напрямую. Класс Inflater распаковывает данные, сжатые с помощью объекта Deflater.

```
public class Deflater {
// Открытые конструкторы
   public Deflater();
   public Deflater(int level);
   public Deflater(int level, boolean nowrap);
// Открытые константы
   public static final int BEST_COMPRESSION;                  // =9
   public static final int BEST_SPEED;                        // =1
   public static final int DEFAULT_COMPRESSION;               // =-1
   public static final int DEFAULT_STRATEGY;                  // =0
   public static final int DEFLATED;                          // =8
   public static final int FILTERED;                          // =1
   public static final int HUFFMAN_ONLY;                      // =2
   public static final int NO_COMPRESSION;                    // =0
// Методы доступа к свойствам (по имени свойства)
   public int getAdler();                                     // синхронизирован, по умолчанию:1
   public int getTotalIn();                                   // синхронизирован, по умолчанию:0
   public int getTotalOut();                                  // синхронизирован, по умолчанию:0
// Открытые методы экземпляра
   public int deflate(byte[] b);
   public int deflate(byte[] b, int off, int len);            // синхронизирован
   public void end();                                         // синхронизирован
   public void finish();                                      // синхронизирован
   public boolean finished();                                 // синхронизирован
   public boolean needsInput();
   public void reset();                                       // синхронизирован
   public void setDictionary(byte[] b);
   public void setDictionary(byte[] b, int off, int len);     // синхронизирован
   public void setInput(byte[] b);
   public void setInput(byte[] b, int off, int len);          // синхронизирован
   public void setLevel(int level);                           // синхронизирован
   public void setStrategy(int strategy);                     // синхронизирован
// Защищенные методы, замещающие Object
   protected void finalize();
}
```

Передается методам: DeflaterOutputStream.DeflaterOutputStream()

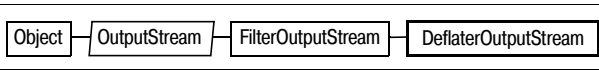
Экземпляры: DeflaterOutputStream.def

DeflaterOutputStream

Java 1.1

java.util.zip

Этот класс является подклассом java.io.FilterOutputStream; он фильтрует поток данных, сжимая его и записывая сжатые данные в другой выходной поток. При создании DeflaterOutputStream необходимо указать поток для записи и объект Deflater для выполнения сжатия. Вы можете установить разнообразные параметры объекта Deflater для того, чтобы указать, какой тип сжатия будет выполняться. Методы write() и close() созданного DeflaterOutputStream аналогичны соответствующим методам других выходных потоков. Класс InflaterInputStream может считывать данные, записанные с помощью DeflaterOutputStream. DeflaterOutputStream записывает «сырые» сжатые данные; зачастую в приложениях лучше использовать один из его подклассов – GZIPOutputStream или ZipOutputStream. Эти подклассы «обертывают» сжатые данные согласно стандартному формату файла.




```
public class DeflaterOutputStream extends java.io.FilterOutputStream {
// Открытые конструкторы
   public DeflaterOutputStream(java.io.OutputStream out);
   public DeflaterOutputStream(java.io.OutputStream out, Deflater def);
   public DeflaterOutputStream(java.io.OutputStream out, Deflater def, int size);
// Открытые методы экземпляра
   public void finish() throws java.io.IOException;
// Открытые методы, замещающие FilterOutputStream
   public void close() throws java.io.IOException;
   public void write(int b) throws java.io.IOException;
   public void write(byte[] b, int off, int len) throws java.io.IOException;
// Защищенные методы экземпляра
   protected void deflate() throws java.io.IOException;
// Защищенные поля экземпляра
   protected byte[] buf;
   protected Deflater def;
}
```

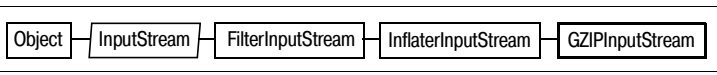
Подклассы: GZIPOutputStream, ZipOutputStream

GZIPInputStream

Java 1.1

java.util.zilp

Этот класс является подклассом класса InflaterInputStream, который читает и распаковывает данные, сжатые по формату *gzip*. При создании GZIPInputStream просто укажите InputStream для считывания упакованных данных и, в необязательном порядке, размер внутреннего буфера распаковки. После создания GZIPInputStream вы можете использовать методы read() и close() так же, как и в случае с любым другим входным потоком.




```
public class GZIPInputStream extends InflaterInputStream {
// Открытые конструкторы
```

```
   public GZIPInputStream(java.io.InputStream in) throws java.io.IOException;
   public GZIPInputStream(java.io.InputStream in, int size) throws java.io.IOException;
// Открытые константы
   public static final int GZIP_MAGIC;        // =35615
// Открытые методы, замещающие InflaterInputStream
   public void close() throws java.io.IOException;
   public int read(byte[] buf, int off, int len) throws java.io.IOException;
// Защищенные поля экземпляра
   protected CRC32 crc;
   protected boolean eos;
}
```

GZIPOutputStream

Java 1.1

java.util.zip

Этот класс является подклассом DeflaterOutputStream, который сжимает и записывает данные, используя формат файла *gzip*. При создании GZIPOutputStream укажите OutputStream для записи и, в необязательном порядке, размер внутреннего буфера сжатия. После создания GZIPOutputStream вы можете использовать методы write() и close() так же, как и в случае с любым другим выходным потоком.


```
public class GZIPOutputStream extends DeflaterOutputStream {
// Открытые конструкторы
   public GZIPOutputStream(java.io.OutputStream out) throws java.io.IOException;
   public GZIPOutputStream(java.io.OutputStream out, int size) throws java.io.IOException;
// Открытые методы, замещающие DeflaterOutputStream
   public void finish() throws java.io.IOException;
   public void write(byte[] buf, int off, int len) throws java.io.IOException;  // синхронизирован
// Защищенные поля экземпляра
   protected CRC32 crc;
}
```

Inflater

Java 1.1

java.util.zip

Этот класс реализует общий алгоритм распаковки данных ZLIB, используемый в *gzip*, *PKZip* и других приложениях сжатия данных. Он распаковывает данные, сжатые с помощью класса Deflater. Важными методами этого класса являются setInput(), который указывает входные данные для распаковки, и inflate(), распаковывающий входные данные в выходной буфер. Существует множество других методов, поэтому данный класс можно применять для потокоориентированной распаковки наравне с высокоуровневыми классами GZIPInputStream и ZipInputStream. Этих классов, основанных на потоках, достаточно в большинстве случаев. Многим приложениям не нужно использовать Inflater напрямую.

```
public class Inflater {
// Открытые конструкторы
   public Inflater();
   public Inflater(boolean nowrap);
```

```
// Методы доступа к свойствам (по имени свойства)
   public int getAdler();                                    // синхронизирован, по умолчанию: 1
   public int getRemaining();                                // синхронизирован, по умолчанию: 0
   public int getTotalIn();                                  // синхронизирован, по умолчанию: 0
   public int getTotalOut();                                 // синхронизирован, по умолчанию: 0
// Открытые методы экземпляра
   public void end();                                                          // синхронизирован
   public boolean finished();                                                  // синхронизирован
   public int inflate(byte[] b) throws DataFormatException;
   public int inflate(byte[] b, int off, int len) throws DataFormatException;  // синхронизирован
   public boolean needsDictionary();                                           // синхронизирован
   public boolean needsInput();                                                // синхронизирован
   public void reset();                                                        // синхронизирован
   public void setDictionary(byte[] b);
   public void setDictionary(byte[] b, int off, int len);                      // синхронизирован
   public void setInput(byte[] b);
   public void setInput(byte[] b, int off, int len);                           // синхронизирован
// Защищенные методы, замещающие Object
   protected void finalize();
}
```

Передается методам: `InflaterInputStream.InflaterInputStream()`

Экземпляры: `InflaterInputStream.inf`

InflaterInputStream

Java 1.1

java.util.zip

Этот класс является подклассом `java.io.FilterInputStream`**; он считывает указанный поток сжатых входных данных, который обычно записан с помощью** `DeflaterOutputStream` **или его подкласса, и фильтрует эти данные, распаковывая их. При создании** `InflaterInputStream` **укажите входной поток для чтения и объект** `Inflater` **для выполнения распаковки. После создания** `InflaterInputStream` **методы** `read()` **и** `skip()` **можно применять так же, как и в случае других входных потоков.** `InflaterInputStream` **распаковывает «сырые» данные. Зачастую в приложениях лучше использовать один из его подклассов,** `GZIPInputStream` **или** `ZipInputStream`**, которые работают с упакованными данными, записанными в стандартном формате файлов *gzip* и *PKZip*.**


```
public class InflaterInputStream extends java.io.FilterInputStream {
// Открытые конструкторы
   public InflaterInputStream(java.io.InputStream in);
   public InflaterInputStream(java.io.InputStream in, Inflater inf);
   public InflaterInputStream(java.io.InputStream in, Inflater inf, int size);
// Открытые методы, замещающие FilterInputStream
1.2 public int available() throws java.io.IOException;
1.2 public void close() throws java.io.IOException;
   public int read() throws java.io.IOException;
   public int read(byte[] b, int off, int len) throws java.io.IOException;
   public long skip(long n) throws java.io.IOException;
// Защищенные методы экземпляра
   protected void fill() throws java.io.IOException;
// Защищенные поля экземпляра
```

```
   protected byte[] buf;
   protected Inflater inf;
   protected int len;
}
```

Подклассы: GZIPInputStream, ZipInputStream

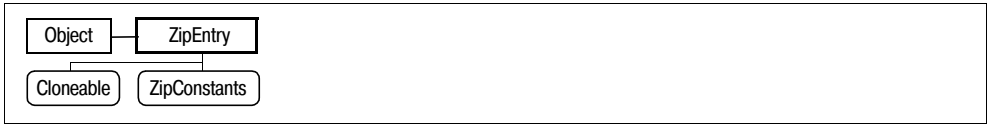
ZipEntry

Java 1.1

java.util.zip ***клонируемый***

Этот класс описывает одиночный элемент (обычно сжатый файл), хранящийся в файле ZIP. Разнообразные методы получают и устанавливают различные фрагменты информации об элементе. Класс ZipEntry применяется классами ZipFile и ZipInputStream, которые читают файлы ZIP, и классом ZipOutputStream, который записывает файлы ZIP.

Когда вы читаете файл ZIP, объект ZipEntry, возвращаемый объектом ZipFile или ZipInputStream, содержит имя, размер, дату изменения и другую информацию об элементе файла. С другой стороны, при записи файла ZIP необходимо создать собственные объекты ZipEntry и инициализировать их: они должны содержать имя элемента и другую соответствующую информацию перед записью содержимого элемента.




```
public class ZipEntry implements Cloneable {
// Открытые конструкторы
   public ZipEntry(String name);
1.2 public ZipEntry(ZipEntry e);
// Открытые константы
   public static final int DEFLATED;                                    // =8
   public static final int STORED;                                      // =0
// Методы доступа к свойствам (по имени свойства)
   public String getComment();
   public void setComment(String comment);
   public long getCompressedSize();
1.2 public void setCompressedSize(long csize);
   public long getCrc();
   public void setCrc(long crc);
   public boolean isDirectory();
   public byte[] getExtra();
   public void setExtra(byte[] extra);
   public int getMethod();
   public void setMethod(int method);
   public String getName();
   public long getSize();
   public void setSize(long size);
   public long getTime();
   public void setTime(long time);
// Открытые методы, замещающие Object
1.2 public Object clone();
1.2 public int hashCode();
   public String toString();
}
```

Подклассы: java.util.jar.JarEntry

Передается методам: java.util.jar.JarEntry.JarEntry(), java.util.jar.JarFile.getInputStream(), java.util.jar.JarOutputStream.putNextEntry(), ZipEntry.ZipEntry(), ZipFile.getInputStream(), ZipOutputStream.putNextEntry()

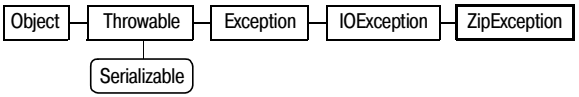
Возвращается методами: java.util.jar.JarFile.getEntry(), java.util.jar.JarInputStream.{createZipEntry(), getNextEntry()}, ZipFile.getEntry(), ZipInputStream.{createZipEntry(), getNextEntry()}

ZipException

Java 1.1

java.util.zip ***сериализуемое, проверяемое***

Сигнализирует о том, что в процессе чтения или записи файла ZIP произошла ошибка.




```
public class ZipException extends java.io.IOException {
// Открытые конструкторы
   public ZipException();
   public ZipException(String s);
}
```

Подклассы: java.util.jar.JarException

Генерируется методами: ZipFile.ZipFile()

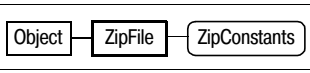
ZipFile

Java 1.1

java.util.zip

Этот класс считывает содержимое файлов ZIP. Он использует файл с произвольным доступом, поэтому элементы файла ZIP не обязательно считывать последовательно, как в случае с классом ZipInputStream. При создании объекта ZipFile нужно указать файл ZIP для чтения – либо имя файла как объект String, либо объект File. В Java 1.3 временные файлы ZIP можно пометить для автоматического удаления после закрытия. Для этого передайте ZipFile.OPEN_READ|ZipFile.OPEN_DELETE как аргумент *mode* в конструктор ZipFile().

После создания ZipFile метод getEntry() возвращает объект ZipEntry для именованного элемента, а метод entries() возвращает объект Enumeration, позволяющий перебирать объекты ZipEntry в цикле. Для считывания содержимого конкретного ZipEntry в файле ZIP передайте ZipEntry методу getInputStream(); он вернет объект InputStream, из которого можно прочитать содержимое элемента.




```
public class ZipFile {
// Открытые конструкторы
   public ZipFile(String name) throws java.io.IOException;
   public ZipFile(java.io.File file) throws ZipException, java.io.IOException;
```

```
1.3 public ZipFile(java.io.File file, int mode) throws java.io.IOException;
// Открытые константы
1.3 public static final int OPEN_DELETE;                                    // =4
1.3 public static final int OPEN_READ;                                      // =1
// Открытые методы экземпляра
   public void close() throws java.io.IOException;
   public java.util.Enumeration entries();
   public ZipEntry getEntry(String name);
   public java.io.InputStream getInputStream(ZipEntry entry) throws java.io.IOException;
   public String getName();
1.2 public int size();
// Защищенные методы, замещающие Object
1.3 protected void finalize() throws java.io.IOException;
}
```

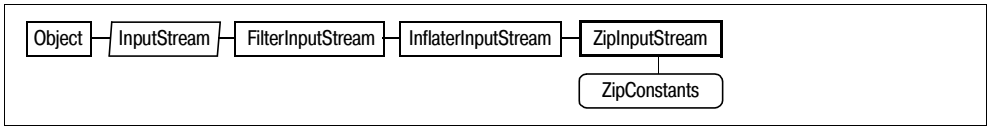
Подклассы: java.util.jar.JarFile

ZipInputStream Java 1.1

java.util.zip

Этот класс является подклассом InflaterInputStream, который последовательно считывает элементы файла ZIP. При создании ZipInputStream нужно указать InputStream, из которого следует считывать содержимое файла ZIP. После создания ZipInputStream метод getNextEntry() можно применять для того, чтобы начать чтение данных из следующего элемента файла ZIP. Для прочтения первого элемента данный метод следует вызвать перед вызовом метода read(). Метод getNextEntry() возвращает объект ZipEntry, описывающий считываемый элемент, или null, если больше нет элементов для чтения из файла ZIP.

Методы read() класса ZipInputStream выполняют чтение до конца текущего элемента, а затем возвращают значение –1, указывающее на то, что больше нет данных для чтения. Для работы со следующим элементом в файле ZIP нужно снова вызвать метод getNextEntry(). Подобным образом метод skip() пропускает только байты из текущего элемента. Метод closeEntry() можно вызвать для того, чтобы пропустить оставшиеся данные в текущем элементе, но обычно проще вызвать метод getNextEntry(), чтобы перейти к следующему элементу.




```
public class ZipInputStream extends InflaterInputStream {
// Открытые конструкторы
   public ZipInputStream(java.io.InputStream in);
// Открытые методы экземпляра
   public void closeEntry() throws java.io.IOException;
   public ZipEntry getNextEntry() throws java.io.IOException;
// Открытые методы, замещающие InflaterInputStream
1.2 public int available() throws java.io.IOException;
   public void close() throws java.io.IOException;
   public int read(byte[] b, int off, int len) throws java.io.IOException;
   public long skip(long n) throws java.io.IOException;
// Защищенные методы экземпляра
```

```
1.2 protected ZipEntry createZipEntry(String name);
}
```

Подклассы: java.util.jar.JarInputStream

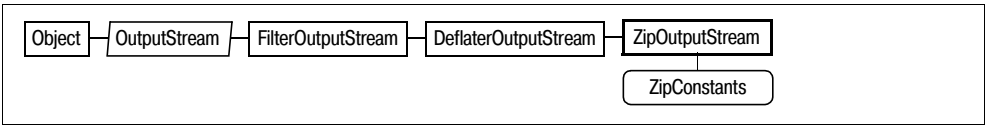
ZipOutputStream Java 1.1

java.util.zip

Этот класс является подклассом DeflaterOutputStream, который записывает в выходной поток данные в формате файла ZIP. Перед записью любых данных в ZipOutputStream необходимо обратиться к элементу внутри файла ZIP с помощью метода putNextEntry(). Объект ZipEntry, передаваемый в этот метод, должен как минимум представлять имя элемента. После обращения к элементу с помощью putNextEntry() вы можете записывать содержимое этого элемента с помощью методов write(). По достижении конца элемента можно приступить к следующему, снова вызвав putNextEntry(). Текущий элемент можно закрыть с помощью метода closeEntry(), а весь поток – с помощью метода close().

Перед тем как обратиться к элементу с помощью putNextEntry(), можно задать метод и уровень сжатия с помощью setMethod() и setLevel(). Двумя допустимыми значениями для setMethod() являются константы DEFLATED и STORED. При задании STORED элемент сохраняется в файле ZIP безо всякого сжатия. Если вы используете DEFLATED, то можно указать скорость/плотность сжатия, передав число от 1 до 9 в метод setLevel(), где 9 означает наиболее сильное и медленное сжатие. Кроме того, с методом setLevel() можно использовать константы Deflater.BEST_SPEED, Deflater.BEST_COMPRESSION и Deflater.DEFAULT_COMPRESSION.

Согласно формату ZIP-файла при сохранении элемента без сжатия необходимо дополнительно указать размер элемента и контрольную сумму CRC-32 в объекте ZipEntry. Если эти значения не указаны или указаны неверно, генерируется исключение.




```
public class ZipOutputStream extends DeflaterOutputStream {
// Открытые конструкторы
   public ZipOutputStream(java.io.OutputStream out);
// Открытые константы
   public static final int DEFLATED;                                                    // =8
   public static final int STORED;                                                      // =0
// Открытые методы экземпляра
   public void closeEntry() throws java.io.IOException;
   public void putNextEntry(ZipEntry e) throws java.io.IOException;
   public void setComment(String comment);
   public void setLevel(int level);
   public void setMethod(int method);
// Открытые методы, замещающие DeflaterOutputStream
   public void close() throws java.io.IOException;
   public void finish() throws java.io.IOException;
   public void write(byte[] b, int off, int len) throws java.io.IOException;   // синхронизирован
}
```

Подклассы: java.util.jar.JarOutputStream

Глава 18

javax.crypto и подпакеты

В этой главе описываются средства криптографии пакета javax.crypto и его подпакетов, в том числе средства шифрования и дешифрования. До того как эти пакеты были интегрированы в Java 1.4, они являлись частью расширения Java Cryptography Extension (JCE), чем и объясняется префикс «javax» в их названиях. Все наиболее распространенные криптографические классы находятся в самом пакете javax.crypto. В подпакете javax.crypto.interfaces определены алгоритмо-зависимые интерфейсы для определенных типов криптографических ключей. Эти интерфейсы являются «прозрачным», переносимым и независимым от реализации представлением криптографических ключей и связанных с ними объектов.

Пакет javax.crypto

Java 1.4

В пакете javax.crypto определены классы и интерфейсы для различных криптографических операций. Cipher – главный класс этого пакета, предназначенный для шифрования и дешифрования данных. Еще один важный класс, SealedObject, использует объект Cipher для шифрования сериализуемых объектов Java.

Класс KeyGenerator создает объекты SecretKey, которые применяются классом Cipher в процессе шифрования и дешифрования. SecretKeyFactory кодирует и декодирует объекты SecretKey. Класс KeyAgreement позволяет двум и более сторонам согласовывать ключ SecretKey таким образом, чтобы этот ключ нельзя было перехватить. Класс Mac вычисляет аутентификационный код сообщения (MAC), который подтверждает целостность данных при передаче их между сторонами, совместно использующими ключ SecretKey. Код MAC в целом похож на цифровую подпись, за исключением того, что в нем используется секретный ключ вместо открытого и закрытого ключей.

Как и пакет java.security, javax.crypto основан на системе поставщиков (providers), поэтому любая реализация криптографических функций может встраиваться в любую реализацию Java. Имена некоторых классов из этого пакета заканчиваются на «Spi». Эти классы представляют собой интерфейс поставщика сервисов. Их нужно реализовать, когда необходимо предоставить определенный криптографический сервис или алгоритм.

Изначально этот пакет был частью расширения Java Cryptography Extension (JCE), но в Java 1.4 его поместили в ядро платформы. Сейчас расширение JCE доступно в стандартных расширениях для Java 1.2 и Java 1.3 (см. *http://java.sun.com/security/*).

Этот пакет распространяется вместе с криптографическим поставщиком «SunJCE», который содержит довольно мощные реализации классов Cipher, KeyAgreement, Mac и других. Этот поставщик установлен в Java 1.4 системными свойствами java.security.

Полное изложение теории криптографии выходит за рамки данной книги. Для работы с этим пакетом необходимо поверхностное понимание криптографических алгоритмов, таких как DES. Чтобы использовать все возможности этого пакета, требуется глубокое осмысление таких понятий криптографии, как режимы обратной связи (feedback modes), схемы заполнения (padding schemes), протокол согласования ключей Диффи-Хеллмана (Diffie-Hellman key-agreement Protocol) и т. д. Введение в современную криптографию Java содержится в книге «Java Cryptography» (O'Reilly). Более подробное описание криптографии без привязки к Java представлено в книге «Applied Cryptography» (Wiley).

Интерфейс

```
public interface SecretKey extends java.security.Key;
```

Классы

```
public class Cipher;
   └ public class NullCipher extends Cipher;
public class CipherInputStream extends java.io.FilterInputStream;
public class CipherOutputStream extends java.io.FilterOutputStream;
public abstract class CipherSpi;
public class EncryptedPrivateKeyInfo;
public class ExemptionMechanism;
public abstract class ExemptionMechanismSpi;
public class KeyAgreement;
public abstract class KeyAgreementSpi;
public class KeyGenerator;
public abstract class KeyGeneratorSpi;
public class Mac implements Cloneable;
public abstract class MacSpi;
public class SealedObject implements Serializable;
public class SecretKeyFactory;
public abstract class SecretKeyFactorySpi;
```

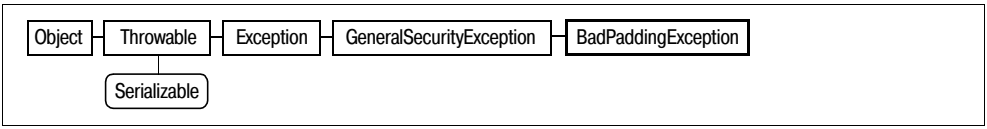
Исключения

```
public class BadPaddingException extends java.security.GeneralSecurityException;
public class ExemptionMechanismException extends java.security.GeneralSecurityException;
public class IllegalBlockSizeException extends java.security.GeneralSecurityException;
public class NoSuchPaddingException extends java.security.GeneralSecurityException;
public class ShortBufferException extends java.security.GeneralSecurityException;
```

BadPaddingException Java 1.4

javax.crypto ***сериализуемое, проверяемое***

Это исключение сообщает о том, что входные данные класса Cipher представлены некорректно.



```
public class BadPaddingException extends java.security.GeneralSecurityException {
// Открытые конструкторы
   public BadPaddingException();
   public BadPaddingException(String msg);
}
```

Генерируется методами: `Cipher.doFinal()`, `CipherSpi.engineDoFinal()`, `SealedObject.getObject()`

Cipher Java 1.4

javax.crypto

Этот класс предназначен для шифрования и дешифрования массивов байтовых значений. `Cipher` основан на системе поставщиков. Чтобы получить объект этого класса, нужно вызвать статический метод-фабрику `getInstance()`. Этому методу в качестве аргумента передается строка, определяющая требуемый тип шифрования. Можно также передать строку с именем необходимого поставщика. Тип шифрования можно указать через имя алгоритма (например, «DES»). Можно воспользоваться строкой, состоящей из имени алгоритма, режима работы и схемы заполнения, разделенных символами косой черты (например, «DES/CBC/PKCS5Padding»). Наконец, если нужно использовать алгоритм блочного шифрования в потоковом режиме, то после имени режима обратной связи указывается количество битов, обрабатываемых за один раз (например, «DES/CFB8/NoPadding»).

Поставщик «SunJCE» поддерживает следующие криптографические алгоритмы:

«DES»

Digital Encryption Standard.

«DESede»

Тройное шифрование по алгоритму DES, также называемое «TripleDES» (тройной DES).

«Blowfish»

Блочное шифрование «Blowfish», разработанное Брюсом Шнайером (Bruce Schneier).

«PBEWithMD5AndDES»

Основанная на паролях схема шифрования, которая описана в PKCS#5. Этот алгоритм неявно использует режим «CBC» и тип заполнения «PKCS5Padding»; нельзя указывать другие режимы и схемы заполнения.

«PBEWithMD5AndTripleDES»

Способ шифрования, основанный на паролях. Он аналогичен схеме «PBEWithMD5AndDES», но вместо DES применяется алгоритм DESede.

SunJCE поддерживает следующие режимы работы:

«ECB»

Режим Electronic Codebook («электронная шифровальная книга»).

«CBC»

Режим Cipher Block Chaining («сцепление блоков шифра»).

«CFB»

Режим обратной связи (Cipher Feedback).

«PCBC»

Режим сцепления блоков шифра открытого текста (Plaintext Cipher Block Chaining).

Наконец, поставщик «SunJCE» поддерживает две схемы заполнения: «NoPadding» и «PKCS5Padding». Имя «SSL3Padding» зарезервировано, но эта схема заполнения не реализована в текущей версии «SunJCE».

После того как создан объект `Cipher` с указанным криптографическим алгоритмом, режимом и схемой заполнения, его нужно инициализировать, вызвав один из методов `init()`. В качестве первого аргумента должна стоять одна из констант: `ENCRYPT_MODE` или `DECRYPT_MODE`. Второй аргумент – объект `java.security.Key`, который будет выполнять шифрование или дешифрование. Если применяется один из алгоритмов с симметричным ключом (то есть алгоритмов, не использующих открытый ключ), которые поддерживаются в поставщике «SunJCE», то этот объект `Key` должен быть реализацией `SecretKey`. В качестве аргумента можно также передать объект `java.security.SecureRandom`, предоставляющий источник случайных чисел, который будет использовать метод `init()`. Если этот аргумент не указан, класс `Cipher` задействует собственный генератор псевдослучайных чисел.

Для некоторых криптографических алгоритмов требуются дополнительные параметры инициализации. Эти параметры можно передать методу `init()` в виде объекта `java.security.AlgorithmParameters` или `java.security.spec.AlgorithmParameterSpec`. Для шифрования эти параметры можно не указывать, а объект `Cipher` использует значения по умолчанию или генерирует случайные значения соответствующих параметров. В этом случае после выполнения шифрования нужно вызвать метод `getParameters()`, чтобы получить параметры (в объекте `AlgorithmParameters`), которые использовались при шифровании. Эти параметры необходимы для дешифрования, поэтому их нужно сохранить или переслать вместе с зашифрованными данными. Алгоритмы блочного шифрования, поддерживаемые поставщиком «SunJCE», а именно «DES», «DESede» и «Blowfish», требуют при инициализации вектор параметров, если они используются в режимах «CBC», «CFB», «OFB» или «PCBC». Вектор инициализации можно представить в виде объекта `javax.crypto.spec.IvParameterSpec`, а получить этот вектор в виде последовательности байтов можно с помощью метода `getIV()`. Алгоритм «PBEWithMD5AndDES» требует в качестве параметров количество итераций и salt. Их можно указать в виде объекта `javax.crypto.spec.PBEParameterSpec`.

После создания и инициализации объекта `Cipher` его можно применять для шифрования или дешифрования. Если необходимо выполнить эти операции только с одним массивом байтовых значений, его нужно передать одному из методов `doFinal()`. Одни варианты этого метода возвращают выходные байты, а другие сохраняют выходные данные в указанном массиве типа `byte`. Если выбран последний способ, то сначала нужно вызывать метод `getOutputSize()` для определения требуемого размера выходного массива. Если нужно зашифровать или расшифровать данные из потока или данные из нескольких массивов, то эти данные нужно передать одному из методов `update()`, вызвав его необходимое количество раз. Если вы работаете с потоковыми данными, применяйте классы `CipherInputStream` и `CipherOutputStream`.

```
public class Cipher {
// Защищенные конструкторы
   protected Cipher(CipherSpi cipherSpi, java.security.Provider provider, String transformation);
// Открытые константы
   public static final int DECRYPT_MODE;                                          // =2
   public static final int ENCRYPT_MODE;                                          // =1
   public static final int PRIVATE_KEY;                                           // =2
```

```
   public static final int PUBLIC_KEY;                                                    // =1
   public static final int SECRET_KEY;                                                    // =3
   public static final int UNWRAP_MODE;                                                   // =4
   public static final int WRAP_MODE;                                                     // =3
// Открытые методы класса
   public static final Cipher getInstance(String transformation)
        throws java.security.NoSuchAlgorithmException, NoSuchPaddingException;
   public static final Cipher getInstance(String transformation, String provider)
        throws java.security.NoSuchAlgorithmException,
        java.security.NoSuchProviderException, NoSuchPaddingException;
   public static final Cipher getInstance(String transformation, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException, NoSuchPaddingException;
// Методы доступа к свойствам (по имени свойства)
   public final String getAlgorithm();
   public final int getBlockSize();
   public final ExemptionMechanism getExemptionMechanism();
   public final byte[] getIV();
   public final java.security.AlgorithmParameters getParameters();
   public final java.security.Provider getProvider();
// Открытые методы экземпляра
   public final byte[] doFinal()
        throws IllegalStateException, IllegalBlockSizeException, BadPaddingException;
   public final byte[] doFinal(byte[] input)
        throws IllegalStateException, IllegalBlockSizeException, BadPaddingException;
   public final int doFinal(byte[] output, int outputOffset) throws IllegalStateException,
        IllegalBlockSizeException, ShortBufferException, BadPaddingException;
   public final byte[] doFinal(byte[] input, int inputOffset, int inputLen)
        throws IllegalStateException,IllegalBlockSizeException, BadPaddingException;
   public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output)
     throws IllegalStateException,ShortBufferException, IllegalBlockSizeException,BadPaddingException;
   public final int doFinal(byte[] input, int inputOffset, int inputLen, byte[] output,
        int outputOffset) throws IllegalStateException,
        ShortBufferException, IllegalBlockSizeException, BadPaddingException;
   public final int getOutputSize(int inputLen) throws IllegalStateException;
   public final void init(int opmode, java.security.cert.Certificate certificate)
        throws java.security.InvalidKeyException;
   public final void init(int opmode, java.security.Key key) throws java.security.InvalidKeyException;
   public final void init(int opmode, java.security.cert.Certificate certificate,
        java.security.SecureRandom random) throws java.security.InvalidKeyException;
   public final void init(int opmode, java.security.Key key,
        java.security.AlgorithmParameters params) throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException;
   public final void init(int opmode, java.security.Key key, java.security.SecureRandom random)
        throws java.security.InvalidKeyException;
   public final void init(int opmode, java.security.Key key,
        java.security.spec.AlgorithmParameterSpec params) throws
        java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
   public final void init(int opmode, java.security.Key key,
        java.security.spec.AlgorithmParameterSpec params, java.security.SecureRandom random)
        throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException;
   public final void init(int opmode, java.security.Key key,
        java.security.AlgorithmParameters params,java.security.SecureRandom random)
       throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
   public final java.security.Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm,
        int wrappedKeyType) throws IllegalStateException,
```

```
        java.security.InvalidKeyException, java.security.NoSuchAlgorithmException;
    public final byte[] update(byte[] input) throws IllegalStateException;
    public final byte[] update(byte[] input, int inputOffset, int inputLen)
        throws IllegalStateException;
    public final int update(byte[] input, int inputOffset, int inputLen, byte[] output)
        throws IllegalStateException, ShortBufferException;
    public final int update(byte[] input, int inputOffset, int inputLen, byte[] output,
        int outputOffset) throws IllegalStateException, ShortBufferException;
    public final byte[] wrap(java.security.Key key) throws IllegalStateException,
       IllegalBlockSizeException, java.security.InvalidKeyException;
}
```

Подклассы: NullCipher

Передается методам: CipherInputStream.CipherInputStream(), CipherOutputStream.CipherOutputStream(), EncryptedPrivateKeyInfo.getKeySpec(), SealedObject.{getObject(), SealedObject()}

Возвращается методами: Cipher.getInstance()

CipherInputStream

Java 1.4

javax.crypto

Этот класс является входным потоком, использующим объект Cipher для шифрования и дешифрования байтовых последовательностей, которые читаются из другого потока. Инициализировать объект Cipher нужно до передачи его конструктору CipherInputStream().


```
public class CipherInputStream extends java.io.FilterInputStream {
// Открытые конструкторы
    public CipherInputStream(java.io.InputStream is, Cipher c);
// Защищенные конструкторы
    protected CipherInputStream(java.io.InputStream is);
// Открытые методы, замещающие FilterInputStream
    public int available() throws java.io.IOException;
    public void close() throws java.io.IOException;
    public boolean markSupported();  // константа
    public int read() throws java.io.IOException;
    public int read(byte[] b) throws java.io.IOException;
    public int read(byte[] b, int off, int len) throws java.io.IOException;
    public long skip(long n) throws java.io.IOException;
}
```

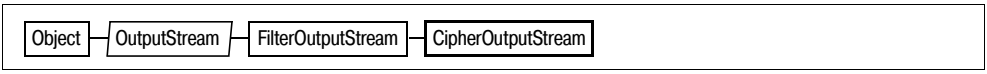
CipherOutputStream

Java 1.4

javax.crypto

Этот класс является выходным потоком, использующим объект Cipher для шифрования и дешифрования байтовых последовательностей, которые затем записываются в другой выходной поток. Объект Cipher нужно инициализировать до передачи его конструктору CipherOutputStream(). Если объект Cipher применяет заполнение (pad-

ding), вы не должны вызывать метод flush() до того, как в поток будут записаны все данные; в противном случае дешифрование завершится неудачей.




```
public class CipherOutputStream extends java.io.FilterOutputStream {
// Открытые конструкторы
   public CipherOutputStream(java.io.OutputStream os, Cipher c);
// Защищенные конструкторы
   protected CipherOutputStream(java.io.OutputStream os);
// Открытые методы, замещающие FilterOutputStream
   public void close() throws java.io.IOException;
   public void flush() throws java.io.IOException;
   public void write(int b) throws java.io.IOException;
   public void write(byte[] b) throws java.io.IOException;
   public void write(byte[] b, int off, int len) throws java.io.IOException;
```

CipherSpi

Java 1.4

javax.crypto

Этот абстрактный класс определяет интерфейс поставщика сервисов для объекта Cipher. В поставщике должен быть реализован свой потомок этого класса для каждого поддерживаемого алгоритма. Поставщик может реализовать отдельный класс для каждого сочетания алгоритма, режима и схемы заполнения. Или же можно реализовать общие классы, предполагая, что режим и схема заполнения будут указаны методами engineSetMode() и engineSetPadding(). В приложениях никогда не приходится использовать или наследовать этот класс.

```
public abstract class CipherSpi {
// Открытые конструкторы
   public CipherSpi();
// Защищенные методы экземпляра
   protected abstract byte[] engineDoFinal(byte[] input, int inputOffset, int inputLen)
       throws IllegalBlockSizeException, BadPaddingException;
   protected abstract int engineDoFinal(byte[] input, int inputOffset, int inputLen, byte[] output,
       int outputOffset) throws ShortBufferException, IllegalBlockSizeException, BadPaddingException;
   protected abstract int engineGetBlockSize();
   protected abstract byte[] engineGetIV();
   protected int engineGetKeySize(java.security.Key key)
       throws java.security.InvalidKeyException;
   protected abstract int engineGetOutputSize(int inputLen);
   protected abstract java.security.AlgorithmParameters engineGetParameters();
   protected abstract void engineInit(int opmode, java.security.Key key,
       java.security.SecureRandom random) throws java.security.InvalidKeyException;
   protected abstract void engineInit(int opmode, java.security.Key key,
       java.security.AlgorithmParameters params, java.security.SecureRandom random)
       throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
   protected abstract void engineInit(int opmode, java.security.Key key,
       java.security.spec.AlgorithmParameterSpec params, java.security.SecureRandom random)
       throws java.security.InvalidKeyException, java.security.InvalidAlgorithmParameterException;
   protected abstract void engineSetMode(String mode) throws java.security.NoSuchAlgorithmException;
   protected abstract void engineSetPadding(String padding) throws NoSuchPaddingException;
   protected java.security.Key engineUnwrap(byte[] wrappedKey, String wrappedKeyAlgorithm,
```

```
      int wrappedKeyType) throws java.security.InvalidKeyException,
      java.security.NoSuchAlgorithmException;
   protected abstract byte[] engineUpdate(byte[] input, int inputOffset, int inputLen);
   protected abstract int engineUpdate(byte[] input, int inputOffset, int inputLen, byte[] output,
      int outputOffset) throws ShortBufferException;
   protected byte[] engineWrap(java.security.Key key)
      throws IllegalBlockSizeException, java.security.InvalidKeyException;
}
```

Передается методам: `Cipher.Cipher()`

EncryptedPrivateKeyInfo

Java 1.4

javax.crypto

Этот класс представляет зашифрованный закрытый ключ. Метод `getEncryptedData()` возвращает зашифрованную последовательность байтов. Методы `getAlgName()` и `getAlgParameters()` возвращают имя алгоритма и параметры шифрования. Чтобы расшифровать ключ, нужно передать объект `Cipher` методу `getKeySpec()`.

```
public class EncryptedPrivateKeyInfo {
// Открытые конструкторы
   public EncryptedPrivateKeyInfo(byte[] encoded) throws java.io.IOException;
   public EncryptedPrivateKeyInfo(java.security.AlgorithmParameters algParams,
        byte[] encryptedData) throws java.security.NoSuchAlgorithmException;
   public EncryptedPrivateKeyInfo(String algName, byte[] encryptedData)
        throws java.security.NoSuchAlgorithmException;
// Открытые методы экземпляра
   public String getAlgName();
   public java.security.AlgorithmParameters getAlgParameters();
   public byte[] getEncoded() throws java.io.IOException;
   public byte[] getEncryptedData();
   public java.security.spec.PKCS8EncodedKeySpec getKeySpec(Cipher c)
        throws java.security.spec.InvalidKeySpecException;
}
```

ExemptionMechanism

Java 1.4

javax.crypto

В некоторых странах закон накладывает ограничения на использование криптографических алгоритмов. В отдельных случаях программа может быть освобождена от этих ограничений, если она реализует один из так называемых «механизмов освобождения от ограничений», например восстановление ключа (key recovery), условное депонирование ключа (key escrow) или ослабление ключа (key weakening). В этом классе определены общие API для таких механизмов. Он редко применяется и не поддерживается в предустановленной реализации поставщика криптографических служб фирмы Sun. Использовать этот класс очень сложно, а его описание выходит за рамки данного справочника. Дополнительную информацию по этой теме можно найти в разделе «How to Make Applicaions Exempt» справочника «JCE Reference Guide», который входит в комплект стандартной документации, поставляемой с JDK.

```
public class ExemptionMechanism {
// Защищенные конструкторы
   protected ExemptionMechanism(ExemptionMechanismSpi exmechSpi, java.security.Provider provider,
                                                                 String mechanism);
```

```
// Открытые методы класса
   public static final ExemptionMechanism getInstance(String mechanism)
        throws java.security.NoSuchAlgorithmException;
   public static final ExemptionMechanism getInstance(String mechanism, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
   public static final ExemptionMechanism getInstance(String mechanism,
        java.security.Provider provider) throws java.security.NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final byte[] genExemptionBlob() throws IllegalStateException, ExemptionMechanismException;
   public final int genExemptionBlob(byte[] output) throws IllegalStateException,
        ShortBufferException, ExemptionMechanismException;
   public final int genExemptionBlob(byte[] output, int outputOffset)
        throws IllegalStateException, ShortBufferException, ExemptionMechanismException;
   public final String getName();
   public final int getOutputSize(int inputLen) throws IllegalStateException;
   public final java.security.Provider getProvider();
   public final void init(java.security.Key key)
        throws java.security.InvalidKeyException, ExemptionMechanismException;
   public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException,ExemptionMechanismException;
   public final void init(java.security.Key key, java.security.AlgorithmParameters params)
        throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException, ExemptionMechanismException;
   public final boolean isCryptoAllowed(java.security.Key key) throws ExemptionMechanismException;
// Защищенные методы, замещающие Object
   protected void finalize();
}
```

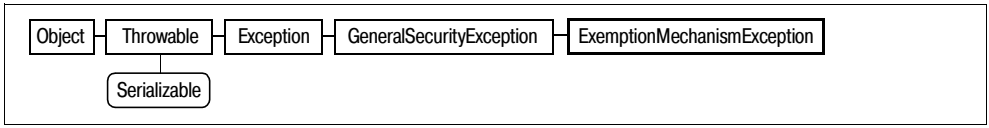
Возвращается методами: `Cipher.getExemptionMechanism()`, `ExemptionMechanism.getInstance()`

ExemptionMechanismException

Java 1.4

javax.crypto ***сериализуемое, проверяемое***

Это исключение сообщает о возникновении особой ситуации при исполнении одного из методов класса `ExemptionMechanism`.




```
public class ExemptionMechanismException extends java.security.GeneralSecurityException {
// Открытые конструкторы
   public ExemptionMechanismException();
   public ExemptionMechanismException(String msg);
}
```

Генерируется методами: `ExemptionMechanism.{genExemptionBlob(), init(), isCryptoAllowed()}`, `ExemptionMechanismSpi.{engineGenExemptionBlob(), engineInit()}`

ExemptionMechanismSpi Java 1.4

javax.crypto

Этот абстрактный класс определяет интерфейс поставщика сервисов (Service Provider Interface) для класса ExemptionMechanism. Поставщики системы безопасности могут реализовать этот интерфейс, но в приложениях его никогда не приходится использовать. Обратите внимание, что предустановленный поставщик «SunJCE» не реализует этот класс.

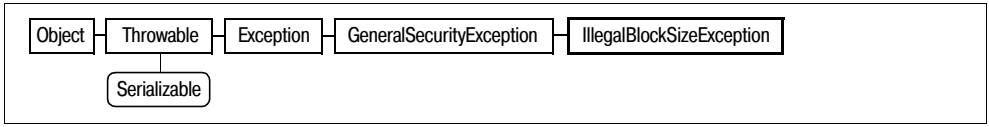
```
public abstract class ExemptionMechanismSpi {
// Открытые конструкторы
   public ExemptionMechanismSpi();
// Защищенные методы экземпляра
   protected abstract byte[] engineGenExemptionBlob() throws ExemptionMechanismException;
   protected abstract int engineGenExemptionBlob(byte[] output, int outputOffset)
        throws ShortBufferException, ExemptionMechanismException;
   protected abstract int engineGetOutputSize(int inputLen);
   protected abstract void engineInit(java.security.Key key)
        throws java.security.InvalidKeyException, ExemptionMechanismException;
   protected abstract void engineInit(java.security.Key key, java.security.AlgorithmParameters params)
        throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException, ExemptionMechanismException;
   protected abstract void engineInit(java.security.Key key,
        java.security.spec.AlgorithmParameterSpec params) throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException, ExemptionMechanismException;
}
```

Передается методам: ExemptionMechanism.ExemptionMechanism()

IllegalBlockSizeException Java 1.4

javax.crypto ***сериализуемое, проверяемое***

Это исключение возникает, если размер данных, переданных механизму блочного шифрования, который реализован в классах Cipher и SealedObject, не совпадает с размером блока.




```
public class IllegalBlockSizeException extends java.security.GeneralSecurityException {
// Открытые конструкторы
   public IllegalBlockSizeException();
   public IllegalBlockSizeException(String msg);
}
```

Генерируется методами: Cipher.{doFinal(), wrap()}, CipherSpi.{engineDoFinal(), engineWrap()}, SealedObject.{getObject(), SealedObject()}

KeyAgreement Java 1.4

javax.crypto

Этот класс содержит API протокола согласования ключей, позволяющего двум и более сторонам согласовывать секретный ключ без обмена секретными данными таким образом, чтобы этот ключ нельзя было перехватить. Класс `KeyAgreement` не зависит от алгоритма. Он основан на системе поставщиков, поэтому для получения объекта `KeyAgreement` нужно вызвать один из статических методов-фабрик `getInstance()`. При этом нужно указать имя используемого алгоритма согласования ключей и (необязательно) имя поставщика этого алгоритма. Поставщик «SunJCE» реализует единственный алгоритм согласования ключей, который называется «DiffieHellman».

Перед использованием объекта `KeyAgreement` каждая сторона должна сначала вызвать метод `init()` и предоставить собственный объект `Key`. Затем каждая сторона получает от другой стороны, участвующей в согласовании, объект `Key` и передает его методу `doPhase()`. Каждая сторона получает промежуточный объект `Key`, который возвращается методом `doPhase()`, после чего снова происходит обмен этими ключами, которые передаются методу `doPhase()`. Как правило, этот процесс повторяется `n-1` раз, где `n` – это количество сторон, но в действительности количество повторений зависит от типа алгоритма. Когда метод `doPhase()` вызывается в последний раз, второй аргумент должен быть `true`, чтобы указать на последний шаг согласования. После этого все стороны вызывают метод `generateSecret()`, чтобы получить массив байтовых значений или объект `SecretKey` для указанного типа алгоритма. Всем сторонам этот метод возвращает одинаковые байтовые значения или объекты `SecretKey`. Класс `KeyAgreement` не отвечает за передачу объектов `Key` между сторонами и за их взаимное опознание. Эту задачу должен выполнять внешний механизм.

Наиболее распространенный тип согласования ключей – это согласование ключей «DiffieHellman» между двумя сторонами. Он работает следующим образом: сначала обе стороны получают объект `java.security.KeyPairGenerator` для алгоритма «DiffieHellman», а затем используют его для создания закрытого и открытого ключа в виде объекта `java.security`. Каждая сторона передает закрытый ключ методу `init()` объекта `KeyAgreement`. Методу `init()` можно передавать объект `java.security.spec.AlgorithmParameterSpec`, но протокол Диффи-Хеллмана не требует дополнительных параметров. Далее обе стороны обмениваются открытыми ключами, обычно используя один из сетевых механизмов (класс `KeyAgreement` не отвечает за фактический обмен ключами). Каждая сторона передает открытый ключ другой стороны методу `doPhase()` объекта `KeyAgreement`. В данном случае участвуют две стороны, поэтому метод `doPhase()` требуется вызвать только один раз, а его второй аргумент должен быть `true`. В этот момент обе стороны вызывают метод `generateSecret()`, чтобы получить общий секретный ключ.

Согласование ключей по методу Диффи-Хеллмана между тремя сторонами требует двух повторений. Данный процесс немного сложнее. Пусть есть три стороны – Элис, Боб и Кэрол. Как и в предыдущем случае, каждый из них создает пару ключей и использует закрытый ключ для инициализации объекта `KeyAgreement`. Затем Элис передает свой открытый ключ Бобу, Боб передает свой ключ Кэрол, а Кэрол передает свой ключ Элис. Каждая сторона передает этот открытый ключ методу `doPhase()`. Поскольку это не последний вызов метода `doPhase()`, то второй аргумент равен `false`, и этот метод возвращает промежуточный объект `Key`. На втором этапе три стороны снова обмениваются промежуточными ключами: Элис передает свой ключ Бобу, Боб передает Кэрол, а Кэрол – Элис. После этого каждый участник передает полученный промежуточный ключ методу `doPhase()`. Второй аргумент этого метода равен `true`,

указывая на то, что это последний этап. Наконец, все трое могут вызвать generateSecret(), чтобы получить общий ключ для расшифровки последующих данных.

```
public class KeyAgreement {
// Защищенные конструкторы
   protected KeyAgreement(KeyAgreementSpi keyAgreeSpi, java.security.Provider provider, String algorithm);
// Открытые методы класса
   public static final KeyAgreement getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static final KeyAgreement getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
  public static final KeyAgreement getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final java.security.Key doPhase(java.security.Key key, boolean lastPhase)
        throws java.security.InvalidKeyException, IllegalStateException;
   public final byte[] generateSecret() throws IllegalStateException;
   public final SecretKey generateSecret(String algorithm) throws IllegalStateException,
        java.security.NoSuchAlgorithmException, java.security.InvalidKeyException;
   public final int generateSecret(byte[] sharedSecret, int offset)
        throws IllegalStateException, ShortBufferException;
   public final String getAlgorithm();
   public final java.security.Provider getProvider();
   public final void init(java.security.Key key) throws java.security.InvalidKeyException;
   public final void init(java.security.Key key, java.security.SecureRandom random
        throws java.security.InvalidKeyException;
   public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidKeyException,java.security.InvalidAlgorithmParameterException;
   public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params,
        java.security.SecureRandom random) throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException;
}
```

Возвращается методами: KeyAgreement.getInstance()

KeyAgreementSpi

Java 1.4

javax.crypto

Этот абстрактный класс определяет интерфейс поставщика сервисов для объекта KeyAgreement. В криптографическом поставщике для каждого поддерживаемого алгоритма должен быть реализован свой класс – потомок данного класса. В приложениях никогда не требуется использовать или наследовать этот класс.

```
public abstract class KeyAgreementSpi {
// Открытые конструкторы
   public KeyAgreementSpi();
// Защищенные методы экземпляра
   protected abstract java.security.Key engineDoPhase(java.security.Key key, boolean lastPhase)
       throws java.security.InvalidKeyException, IllegalStateException;
   protected abstract byte[] engineGenerateSecret() throws IllegalStateException;
   protected abstract SecretKey engineGenerateSecret(String algorithm)
       throws IllegalStateException,
       java.security.NoSuchAlgorithmException, java.security.InvalidKeyException;
   protected abstract int engineGenerateSecret(byte[] sharedSecret, int offset)
       throws IllegalStateException, ShortBufferException;
   protected abstract void engineInit(java.security.Key key, java.security.SecureRandom random)
```

```
        throws java.security.InvalidKeyException;
    protected abstract void engineInit(java.security.Key key,
        java.security.spec.AlgorithmParameterSpec params, java.security.SecureRandom random)
        throws java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException;
}
```

Передается методам: KeyAgreement.KeyAgreement()

KeyGenerator Java 1.4

javax.crypto

Этот класс содержит API создания секретных ключей для симметричной криптографии. Он похож на класс java.security.KeyPairGenerator, который создает пару ключей (закрытый и открытый) для асимметричных алгоритмов, или алгоритмов с открытым ключом. Объект KeyGenerator не зависит от алгоритма. Он основан на поставщике, поэтому получить экземпляр этого класса можно с помощью одного из статических методов-фабрик getInstance(), указав имя алгоритма и (необязательно) имя поставщика системы безопасности, который должен применяться для генерации ключей. Поставщик «SunJCE» содержит реализации класса KeyGenerator для алгоритмов шифрования «DES», «DESede» и «Blowfish», а также поддерживает алгоритмы аутентификации сообщений «HmacMD5» и «HmacSHA1».

После получения объекта KeyGenerator его нужно проинициализировать методом init(). Можно передать объект java.security.spec.AlgorithmParameterSpec, содержащий алгоритмо-зависимые параметры инициализации, или просто указать желаемый размер создаваемого ключа (в битах). Кроме того, можно указать объект SecureRandom, генерирующий случайные параметры. Если он не указан, объект KeyGenerator создает собственный генератор. Ни один из алгоритмов, поддерживаемых поставщиком «SunJCE», не требует специальных параметров.

После того как был вызван метод getInstance(), получен объект KeyGenerator и вызван метод init(), инициализирующий его, нужно вызвать метод generateKey(), чтобы создать новый ключ SecretKey. Необходимо помнить о том, что этот ключ нужно держать в секрете. При хранении и передаче этого ключа нужно предпринимать меры предосторожности, чтобы он не стал доступен посторонним. Для хранения ключа под защитой пароля можно использовать объект java.security.KeyStore.

```
public class KeyGenerator {
// Защищенные конструкторы
    protected KeyGenerator(KeyGeneratorSpi keyGenSpi, java.security.Provider provider, String algorithm);
// Открытые методы класса
    public static final KeyGenerator getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static final KeyGenerator getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
    public static final KeyGenerator getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
// Открытые методы экземпляра
    public final SecretKey generateKey();
    public final String getAlgorithm();
    public final java.security.Provider getProvider();
    public final void init(int keysize);
    public final void init(java.security.spec.AlgorithmParameterSpec params)
        throws java.security.InvalidAlgorithmParameterException;
```

```
    public final void init(java.security.SecureRandom random);
    public final void init(int keysize, java.security.SecureRandom random);
    public final void init(java.security.spec.AlgorithmParameterSpec params,
        java.security.SecureRandom random) throws java.security.InvalidAlgorithmParameterException;
}
```

Возвращается методами: KeyGenerator.getInstance()

KeyGeneratorSpi

Java 1.4

javax.crypto

Этот абстрактный класс определяет интерфейс поставщика сервисов для объекта KeyGenerator. Для каждого поддерживаемого алгоритма генерации ключей криптографический поставщик должен реализовать определенный потомок этого класса. В приложениях никогда не требуется использовать или наследовать этот класс.

```
public abstract class KeyGeneratorSpi {
// Открытые конструкторы
    public KeyGeneratorSpi();
// Защищенные методы экземпляра
    protected abstract SecretKey engineGenerateKey();
    protected abstract void engineInit(java.security.SecureRandom random);
    protected abstract void engineInit(int keysize, java.security.SecureRandom random);
    protected abstract void engineInit(java.security.spec.AlgorithmParameterSpec params,
        java.security.SecureRandom random) throws java.security.InvalidAlgorithmParameterException;
}
```

Передается методам: KeyGenerator.KeyGenerator()

Mac

Java 1.4

javax.crypto ***клонируемый***

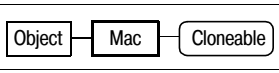
В этом классе содержится API для вычисления так называемого *кода аутентификации сообщения* (message authentication code, MAC), который предназначен для проверки целостности данных, передаваемых между двумя сторонами, которые используют общий секретный ключ. MAC подобен цифровой подписи, за исключением того, что он создается с помощью секретного ключа, а не двух ключей – открытого и закрытого. Класс Mac не зависит от конкретного алгоритма и основан на системе поставщиков. Чтобы получить объект этого класса, нужно вызвать один из статических методов-фабрик getInstance(), указав имя требуемого алгоритма создания объекта Mac и (необязательно) имя поставщика. Поставщик «SunJCE» реализует два алгоритма: «HmacMD5» и «HmacSHA1». Они основаны на криптографических алгоритмах хеширования MD5 и SHA-1.

После получения объекта Mac его нужно проинициализировать методом init(), указав ключ SecretKey и (необязательно) объект java.security.spec.AlgorithmParameterSpec. Алгоритмы «HmacMD5» и «HmacSHA1» могут работать с любым видом ключей SecretKey; они не ограничены использованием определенного криптографического алгоритма. В этих алгоритмах не требуется указывать объект AlgorithmParameterSpec.

После создания и инициализации объекта Mac нужно указать данные, для которых будет вычислен MAC-код. Если данные содержатся в одном массиве байтовых значений, то нужно просто передать этот массив методу doFinal(). Если данные принимаются из потока или хранятся в нескольких местах, то нужно вызвать метод update()

необходимое количество раз. После каждой серии вызовов этого метода нужно вызвать метод doFinal(). Обратите внимание на то, что одни виды метода doFinal() возвращают код MAC, а другие записывают его в указанный байтовый массив. В последнем случае применяйте метод getMacLength() для создания массива правильного размера.

Вызов метода doFinal() приводит объект Mac в исходное состояние. Если нужно вычислить код MAC сначала для части данных, а затем продолжить вычисление для всех данных, то перед вызовом doFinal() нужно создать копию объекта Mac методом clone(). Заметьте, что при создании классов Mac не требуется реализовывать интерфейс Cloneable.




```
public class Mac implements Cloneable {
// Защищенные конструкторы
   protected Mac(MacSpi macSpi, java.security.Provider provider, String algorithm);
// Открытые методы класса
   public static final Mac getInstance(String algorithm) throws java.security.NoSuchAlgorithmException;
   public static final Mac getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException,java.security.NoSuchProviderException;
   public static final Mac getInstance(String algorithm, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
// Открытые методы экземпляра
   public final byte[] doFinal() throws IllegalStateException;
   public final byte[] doFinal(byte[] input) throws IllegalStateException;
   public final void doFinal(byte[] output, int outOffset)
        throws ShortBufferException, IllegalStateException;
   public final String getAlgorithm();
   public final int getMacLength();
   public final java.security.Provider getProvider();
   public final void init(java.security.Key key) throws java.security.InvalidKeyException;
   public final void init(java.security.Key key, java.security.spec.AlgorithmParameterSpec params)
       throws java.security.InvalidKeyException,java.security.InvalidAlgorithmParameterException;
   public final void reset();
   public final void update(byte[] input) throws IllegalStateException;
   public final void update(byte input) throws IllegalStateException;
   public final void update(byte[] input, int offset, int len) throws IllegalStateException;
// Открытые методы, замещающие Object
   public final Object clone() throws CloneNotSupportedException;
}
```

Возвращается методами: Mac.getInstance()

MacSpi Java 1.4

javax.crypto

Этот абстрактный класс определяет интерфейс поставщика сервисов для объекта Mac. Для каждого поддерживаемого алгоритма создания MAC-кода криптографический поставщик должен реализовать определенного потомка этого класса. В обычных приложениях никогда не требуется использовать или наследовать этот класс.

```
public abstract class MacSpi {
// Открытые конструкторы
   public MacSpi();
```

```
// Открытые методы, замещающие Object
   public Object clone() throws CloneNotSupportedException;
// Защищенные методы экземпляра
   protected abstract byte[] engineDoFinal();
   protected abstract int engineGetMacLength();
   protected abstract void engineInit(java.security.Key key,
       java.security.spec.AlgorithmParameterSpec params)
      throws java.security.InvalidKeyException,java.security.InvalidAlgorithmParameterException;
   protected abstract void engineReset();
   protected abstract void engineUpdate(byte input);
   protected abstract void engineUpdate(byte[] input, int offset, int len);
}
```

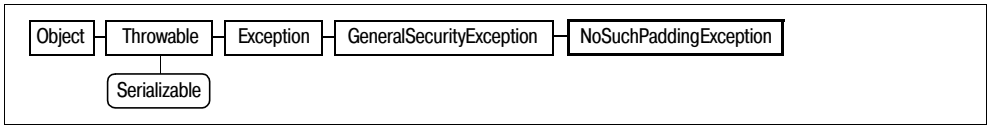
Передается методам: `Mac.Mac()`

NoSuchPaddingException

Java 1.4

javax.crypto ***сериализуемое, проверяемое***

Это исключение сообщает о том, что для запрошенной схемы заполнения не найдена ее реализация.




```
public class NoSuchPaddingException extends java.security.GeneralSecurityException {
// Открытые конструкторы
   public NoSuchPaddingException();
   public NoSuchPaddingException(String msg);
}
```

Генерируется методами: `Cipher.getInstance(), CipherSpi.engineSetPadding()`

NullCipher

Java 1.4

javax.crypto

Этот простейший потомок класса `Cipher` производит «тождественное» шифрование, то есть не преобразует входные данные. В отличие от объектов `Cipher`, возвращаемых методом `Cipher.getInstance()`, объект `NullCipher` нужно создавать с помощью конструктора `NullCipher()`.


```
public class NullCipher extends Cipher {
// Открытые конструкторы
   public NullCipher();
}
```

SealedObject

Java 1.4

javax.crypto ***сериализуемый***

Этот класс является оберткой (wrapper) вокруг сериализуемого объекта. Он сериализует объект и зашифровывает итоговый поток данных, защищая этот объект. При создании объекта данного класса нужно указать объект, который требуется сериализовать, и Cipher, который будет производить шифрование. Получить сериализованный и зашифрованный объект можно методом getObject(), указав в качестве параметра объект Cipher или java.security.Key, который будет выполнять дешифрование. Объект SealedObject запоминает алгоритм шифрования и его параметры, поэтому объект Key может самостоятельно расшифровать объект.


```
public class SealedObject implements Serializable {
// Открытые конструкторы
   public SealedObject(Serializable object, Cipher c) throws java.io.IOException,
      IllegalBlockSizeException;
// Защищенные конструкторы
   protected SealedObject(SealedObject so);
// Открытые методы экземпляра
   public final String getAlgorithm();
   public final Object getObject(java.security.Key key) throws java.io.IOException,
      ClassNotFoundException, java.security.NoSuchAlgorithmException,
      java.security.InvalidKeyException;
   public final Object getObject(Cipher c) throws java.io.IOException,
      ClassNotFoundException, IllegalBlockSizeException, BadPaddingException;
   public final Object getObject(java.security.Key key, String provider)
      throws java.io.IOException, ClassNotFoundException, java.security.NoSuchAlgorithmException,
      java.security.NoSuchProviderException, java.security.InvalidKeyException;
// Защищенные поля экземпляра
   protected byte[] encodedParams;
}
```

Передается методам: SealedObject.SealedObject()

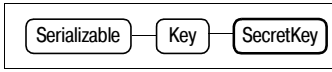
SecretKey

Java 1.4

javax.crypto ***сериализуемый***

Этот интерфейс представляет секретный ключ, который применяется в симметричных криптографических алгоритмах, основанных на том, что обе стороны – получатель и отправитель – владеют общей секретной информацией. SecretKey расширяет интерфейс java.security.Key, но не добавляет в него новых методов. Этот интерфейс предназначен для того, чтобы отличать секретные ключи от закрытых и открытых ключей, используемых в ассиметричных криптографических алгоритмах (алгоритмах открытого ключа). См. также java.security.PublicKey и java.security.PrivateKey.

Секретный ключ представляет собой массив байтовых значений и не использует специального формата кодировки. Тем не менее метод getFormat() класса, реализующего этот интерфейс, должен возвращать значение «RAW», а метод getEncoded() – последовательность байтов ключа (эти два метода определены в интерфейсе java.security.Key, который расширяет SecretKey).




```
public interface SecretKey extends java.security.Key {
}
```

Реализации: javax.crypto.interfaces.PBEKey, javax.crypto.spec.SecretKeySpec, javax.security.auth.kerberos.KerberosKey

Передается методам: SecretKeyFactory.{getKeySpec(), translateKey()}, SecretKeyFactorySpi.{engineGetKeySpec(), engineTranslateKey()}

Возвращается методами: KeyAgreement.generateSecret(), KeyAgreementSpi.engineGenerateSecret(), KeyGenerator.generateKey(), KeyGeneratorSpi.engineGenerateKey(), SecretKeyFactory.{generateSecret(), translateKey()}, SecretKeyFactorySpi.{engineGenerateSecret(), engineTranslateKey()}, javax.security.auth.kerberos.KerberosTicket.getSessionKey()

SecretKeyFactory Java 1.4

javax.crypto

Этот класс определяет API для преобразования секретного ключа из объекта SecretKey **в объект** javax.crypto.SecretKeySpec **(последний является более «прозрачным» представлением ключа). Этот класс во многом похож на** java.security.KeyFactory**, но он работает с секретными (или симметричными) ключами, а не с закрытым и открытым ключом (ассиметричным). Класс** SecretKeyFactory **не зависит от алгоритма. Он основан на системе поставщиков, поэтому создать объект этого класса можно одним из статических методов-фабрик** getInstance()**, указав имя требуемого алгоритма и (необязательно) имя выбранного поставщика. Поставщик «SunJCE» предоставляет реализации класса** SecretKeyFactory **для алгоритмов «DES», «DESede» и «PBEWithMD5AndDES».**

После создания объекта SecretKeyFactory **можно использовать метод** generateSecret() **для создания ключа** SecretKey **из объекта** java.security.spec.KeySpec **(или из его потомка** javax.crypto.spec.SecretKeySpec**) либо вызвать метод** getKeySpec()**, чтобы получить** KeySpec **для объекта** Key**. Так как подходящих типов ключей** KeySpec **может быть несколько, методу** getKeySpec() **требуется указать тип этого ключа в виде объекта** Class**. См. также классы** DESKeySpec**,** DESedeKeySpec **и** PBEKeySpec **пакета** javax.crypto.spec**.**

```
public class SecretKeyFactory {
// Защищенные конструкторы
   protected SecretKeyFactory(SecretKeyFactorySpi keyFacSpi, java.security.Provider provider,
String algorithm);
// Открытые методы класса
   public static final SecretKeyFactory getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static final SecretKeyFactory getInstance(String algorithm,
        java.security.Provider provider) throws java.security.NoSuchAlgorithmException;
   public static final SecretKeyFactory getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
// Открытые методы экземпляра
   public final SecretKey generateSecret(java.security.spec.KeySpec keySpec)
        throws java.security.spec.InvalidKeySpecException;
   public final String getAlgorithm();
   public final java.security.spec.KeySpec getKeySpec(SecretKey key, Class keySpec)
        throws java.security.spec.InvalidKeySpecException;
```

```
   public final java.security.Provider getProvider();
   public final SecretKey translateKey(SecretKey key) throws java.security.InvalidKeyException;
}
```

Возвращается методами: SecretKeyFactory.getInstance()

SecretKeyFactorySpi Java 1.4

java.crypto

Этот абстрактный класс определяет интерфейс поставщика сервисов для объекта SecretKeyFactory. Криптографический поставщик должен содержать реализации отдельных потомков этого класса для каждого типа поддерживаемых секретных ключей. В приложениях никогда не приходится использовать или наследовать этот класс.

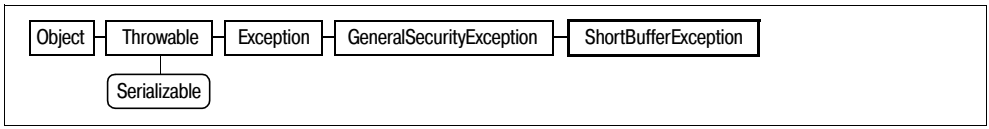
```
public abstract class SecretKeyFactorySpi {
// Открытые конструкторы
   public SecretKeyFactorySpi();
// Защищенные методы экземпляра
   protected abstract SecretKey engineGenerateSecret(java.security.spec.KeySpec keySpec)
        throws java.security.spec.InvalidKeySpecException;
   protected abstract java.security.spec.KeySpec engineGetKeySpec(SecretKey key, Class keySpec)
        throws java.security.spec.InvalidKeySpecException;
   protected abstract SecretKey engineTranslateKey(SecretKey key)
        throws java.security.InvalidKeyException;
}
```

Передается методам: SecretKeyFactory.SecretKeyFactory()

ShortBufferException Java 1.4

javax.crypto ***сериализуемое, проверяемое***

Это исключение сообщает о том, что выходной буфер слишком мал для хранения результатов операции.




```
public class ShortBufferException extends java.security.GeneralSecurityException {
// Открытые конструкторы
   public ShortBufferException();
   public ShortBufferException(String msg);
}
```

Генерируется методами: Cipher.{doFinal(), update()}, CipherSpi.{engineDoFinal(), engineUpdate()}, ExemptionMechanism.genExemptionBlob(), ExemptionMechanismSpi.engineGenExemptionBlob(), KeyAgreement.generateSecret(), KeyAgreementSpi.engineGenerateSecret(), Mac.doFinal()

Пакет javax.crypto.interfaces Java 1.4

Интерфейсы пакета javax.crypto.interfaces определяют открытые методы, которые должны поддерживаться в различных типах ключей шифрования. Интерфейсы «DH» представляют пары закрытых и открытых ключей, используемых в протоколе согласования ключей Диффи-Хеллмана. Интерфейс «PBE» предназначен для шифрования на основе паролей (Password-Based Encryption). Как правило, эти интерфейсы применяются только при создании реализации криптографического поставщика или при разработке собственной реализации алгоритмов криптографии. Для работы с ними требуется хотя бы поверхностное знание алгоритмов шифрования и математической теории, на которой они основаны. Необходимо также заметить, что пакет javax.crypto.spec содержит алгоритмо-зависимые классы, работающие с ключами шифрования.

Интерфейсы

```
public interface DHKey;
public interface DHPrivateKey extends DHKey, java.security.PrivateKey;
public interface DHPublicKey extends DHKey, java.security.PublicKey;
public interface PBEKey extends javax.crypto.SecretKey;
```

DHKey Java 1.4

javax.crypto.interfaces

Этот интерфейс представляет ключ Диффи-Хеллмана. Объект javax.crypto.spec.DHParameterSpec, возвращаемый методом getParams(), содержит параметры создания ключа; эти параметры определяют семейство ключей. См. описание интерфейсов-потомков DHPublicKey и DHPrivateKey, где представлены значения ключей.

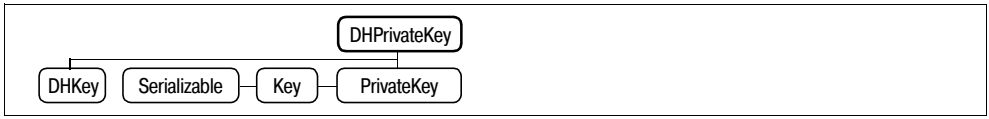
```
public interface DHKey {
// Открытые методы экземпляра
   public abstract javax.crypto.spec.DHParameterSpec getParams();
}
```

Реализации: DHPrivateKey, DHPublicKey

DHPrivateKey Java 1.4

javax.crypto.interfaces ***сериализуемый***

Этот интерфейс представляет закрытый ключ Диффи-Хеллмана. Обратите внимание на то, что он расширяет два интерфейса: DHKey и java.security.PrivateKey. Метод getX() возвращает значение закрытого ключа. Если вы работаете с объектом PrivateKey, который представляет ключ Диффи-Хеллмана, то этот объект можно привести к типу DHPrivateKey.



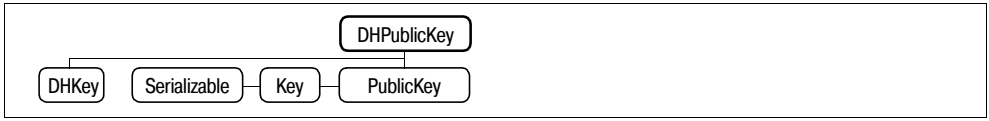

```
public interface DHPrivateKey extends DHKey, java.security.PrivateKey {
// Открытые методы экземпляра
   public abstract java.math.BigInteger getX();
}
```

DHPublicKey

Java 1.4

javax.crypto.interfaces *сериализуемый*

Этот интерфейс представляет открытый ключ Диффи-Хеллмана. Он расширяет два интерфейса: DHKey и java.security.PublicKey. Метод getY() возвращает значение открытого ключа. Если вы работаете с объектом PublicKey, который представляет ключ Диффи-Хеллмана, то этот объект можно привести к типу DHPublicKey.



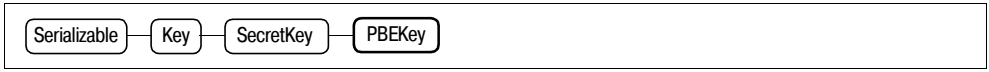

```
public interface DHPublicKey extends DHKey, java.security.PublicKey {
// Открытые методы экземпляра
   public abstract java.math.BigInteger getY();
}
```

PBEKey

Java 1.4

javax.crypto.interfaces *сериализуемый*

Этот интерфейс представляет ключ для шифрования на основе пароля. Если вы работаете с объектом SecretKey, который представляет ключ на основе пароля, этот объект можно привести к типу PBEKey.




```
public interface PBEKey extends javax.crypto.SecretKey {
// Открытые методы экземпляра
   public abstract int getIterationCount();
   public abstract char[] getPassword();
   public abstract byte[] getSalt();
}
```

Пакет javax.crypto.spec

Java 1.4

Пакет javax.crypto.spec содержит классы, определяющие «прозрачное» представление секретных, закрытых и открытых ключей Диффи-Хеллмана и параметров различных криптографических алгоритмов (в виде объектов java.security.spec.KeySpec и java.security.spec.AlgorithmParameterSpec). Классы этого пакета применяются вместе с java.security.KeyFactory, javax.crypto.SecretKeyFactory и java.security.AlgorithmParameters для преобразования объектов Key и AlgorithmParameters в их «прозрачные» представления и для обратного преобразования. Для успешного применения этого пакета нужно знать спецификации различных криптографических алгоритмов и их математическую основу.

Классы

```
public class DESedeKeySpec implements java.security.spec.KeySpec;
public class DESKeySpec implements java.security.spec.KeySpec;
public class DHGenParameterSpec implements java.security.spec.AlgorithmParameterSpec;
public class DHParameterSpec implements java.security.spec.AlgorithmParameterSpec;
public class DHPrivateKeySpec implements java.security.spec.KeySpec;
public class DHPublicKeySpec implements java.security.spec.KeySpec;
public class IvParameterSpec implements java.security.spec.AlgorithmParameterSpec;
public class PBEKeySpec implements java.security.spec.KeySpec;
public class PBEParameterSpec implements java.security.spec.AlgorithmParameterSpec;
public class RC2ParameterSpec implements java.security.spec.AlgorithmParameterSpec;
public class RC5ParameterSpec implements java.security.spec.AlgorithmParameterSpec;
public class SecretKeySpec implements java.security.spec.KeySpec, javax.crypto.SecretKey;
```

DESedeKeySpec Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением ключа DESede (тройной DES). Длина ключа – 24 байта.


```
public class DESedeKeySpec implements java.security.spec.KeySpec {
// Открытые конструкторы
   public DESedeKeySpec(byte[] key) throws java.security.InvalidKeyException;
   public DESedeKeySpec(byte[] key, int offset) throws java.security.InvalidKeyException;
// Открытые константы
   public static final int DES_EDE_KEY_LEN;                                    // =24
// Открытые методы класса
   public static boolean isParityAdjusted(byte[] key, int offset)
       throws java.security.InvalidKeyException;
// Открытые методы экземпляра
   public byte[] getKey();
}
```

DESKeySpec Java 1.4

java.crypto.spec

Этот класс является «прозрачным» представлением ключа типа DES. Ключ имеет длину 8 байт.

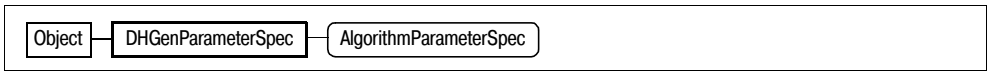

```
public class DESKeySpec implements java.security.spec.KeySpec {
// Открытые конструкторы
   public DESKeySpec(byte[] key) throws java.security.InvalidKeyException;
   public DESKeySpec(byte[] key, int offset) throws java.security.InvalidKeyException;
// Открытые константы
   public static final int DES_KEY_LEN;                                        // =8
// Открытые методы класса
```

```
   public static boolean isParityAdjusted(byte[] key, int offset)
        throws java.security.InvalidKeyException;
   public static boolean isWeak(byte[] key, int offset) throws java.security.InvalidKeyException;
// Открытые методы экземпляра
   public byte[] getKey();
}
```

DHGenParameterSpec Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением значений, которые используются при создании набора параметров алгоритма Диффи-Хеллмана (см. DHParameterSpec). Экземпляр этого класса можно передать методу init() объекта java.security.AlgorithmParameterGenerator, который будет вычислять параметры Диффи-Хеллмана.




```
public class DHGenParameterSpec implements java.security.spec.AlgorithmParameterSpec {
// Открытые конструкторы
   public DHGenParameterSpec(int primeSize, int exponentSize);
// Открытые методы экземпляра
   public int getExponentSize();
   public int getPrimeSize();
}
```

DHParameterSpec Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением для набора параметров алгоритма согласования ключей Диффи-Хеллмана. Все стороны, участвующие в согласовании, должны использовать общие параметры при создании закрытого и открытого ключей Диффи-Хеллмана.


```
public class DHParameterSpec implements java.security.spec.AlgorithmParameterSpec {
// Открытые конструкторы
   public DHParameterSpec(java.math.BigInteger p, java.math.BigInteger g);
   public DHParameterSpec(java.math.BigInteger p, java.math.BigInteger g, int l);
// Открытые методы экземпляра
   public java.math.BigInteger getG();
   public int getL();
   public java.math.BigInteger getP();
}
```

Возвращается методами: javax.crypto.interfaces.DHKey.getParams()

DHPrivateKeySpec Java 1.4

javax.crypto.spec

Этот класс типа java.security.spec.KeySpec является «прозрачным» представлением закрытого ключа Диффи-Хеллмана.


```
public class DHPrivateKeySpec implements java.security.spec.KeySpec {
// Открытые конструкторы
   public DHPrivateKeySpec(java.math.BigInteger x, java.math.BigInteger p, java.math.BigInteger g);
// Открытые методы экземпляра
   public java.math.BigInteger getG();
   public java.math.BigInteger getP();
   public java.math.BigInteger getX();
}
```

DHPublicKeySpec Java 1.4

javax.crypto.spec

Этот класс типа java.security.spec.KeySpec является «прозрачным» представлением открытого ключа Диффи-Хеллмана.


```
public class DHPublicKeySpec implements java.security.spec.KeySpec {
// Открытые конструкторы
   public DHPublicKeySpec(java.math.BigInteger y, java.math.BigInteger p, java.math.BigInteger g);
// Открытые методы экземпляра
   public java.math.BigInteger getG();
   public java.math.BigInteger getP();
   public java.math.BigInteger getY();
}
```

IvParameterSpec Java 1.4

javax.crypto.spec

Этот класс типа java.security.spec.AlgorithmParameterSpec является «прозрачным» представлением вектора инициализации (ВИ). ВИ нужен для создания блоков шифра, которые применяются в режиме обратной связи, например при использовании DES в режиме CBC.


```
public class IvParameterSpec implements java.security.spec.AlgorithmParameterSpec {
// Открытые конструкторы
   public IvParameterSpec(byte[] iv);
   public IvParameterSpec(byte[] iv, int offset, int len);
```

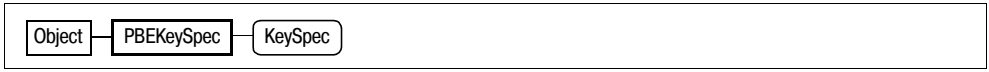
```
// Открытые методы экземпляра
   public byte[] getIV();
}
```

PBEKeySpec

Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением пароля, используемого в шифровании на основе пароля (PBE). Пароль хранится в массиве символов, а не в объекте `String`, поэтому в целях безопасности символы пароля можно заменить другими, когда пароль больше не используется.




```
public class PBEKeySpec implements java.security.spec.KeySpec {
// Открытые конструкторы
   public PBEKeySpec(char[] password);
   public PBEKeySpec(char[] password, byte[] salt, int iterationCount);
   public PBEKeySpec(char[] password, byte[] salt, int iterationCount, int keyLength);
// Открытые методы экземпляра
   public final void clearPassword();
   public final int getIterationCount();
   public final int getKeyLength();
   public final char[] getPassword();
   public final byte[] getSalt();
}
```

PBEParameterSpec

Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением параметров, используемых в алгоритме шифрования на основе пароля, определенном стандартом PKCS#5.


```
public class PBEParameterSpec implements java.security.spec.AlgorithmParameterSpec {
// Открытые конструкторы
   public PBEParameterSpec(byte[] salt, int iterationCount);
// Открытые методы экземпляра
   public int getIterationCount();
   public byte[] getSalt();
}
```

RC2ParameterSpec

Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением параметров алгоритма шифрования RC2. Объект этого класса инициализирует объект `Cipher`, который реализует ал-

горитм RC2. Обратите внимание, что поставщик «SunJCE» фирмы Sun не содержит реализации этого алгоритма.


```
public class RC2ParameterSpec implements java.security.spec.AlgorithmParameterSpec {
// Открытые конструкторы
   public RC2ParameterSpec(int effectiveKeyBits);
   public RC2ParameterSpec(int effectiveKeyBits, byte[] iv);
   public RC2ParameterSpec(int effectiveKeyBits, byte[] iv, int offset);
// Открытые методы экземпляра
   public int getEffectiveKeyBits();
   public byte[] getIV();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

RC5ParameterSpec

Java 1.4

javax.crypto.spec

Этот класс является «прозрачным» представлением параметров алгоритма шифрования RC5. Объект этого класса инициализирует объект `Cipher`, который реализует алгоритм RC5. Обратите внимание на то, что поставщик «SunJCE» фирмы Sun не содержит реализации этого алгоритма.


```
public class RC5ParameterSpec implements java.security.spec.AlgorithmParameterSpec {
// Открытые конструкторы
   public RC5ParameterSpec(int version, int rounds, int wordSize);
   public RC5ParameterSpec(int version, int rounds, int wordSize, byte[] iv);
   public RC5ParameterSpec(int version, int rounds, int wordSize, byte[] iv, int offset);
// Открытые методы экземпляра
   public byte[] getIV();
   public int getRounds();
   public int getVersion();
   public int getWordSize();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

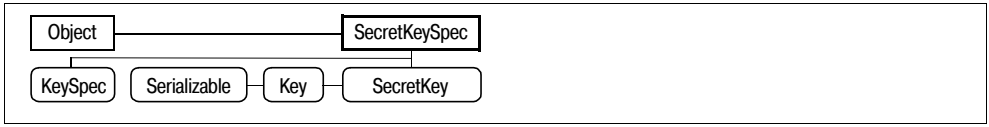
SecretKeySpec

Java 1.4

javax.crypto.spec ***сериализуемый***

Этот класс является «прозрачным» алгоритмо-независимым представлением секретного ключа. Данный класс применяется только с алгоритмами шифрования, секретные ключи которых могут быть представлены в виде массивов байтовых значений и не требуют дополнительных параметров (например, алгоритмы DES и DESede).

Заметьте, что `SecretKeySpec` **реализует интерфейс** `javax.crypto.SecretKey` **непосредственно, поэтому не требуется использовать дополнительный объект** `javax.crypto.SecretKeyFactory`**.**




```
public class SecretKeySpec implements java.security.spec.KeySpec, javax.crypto.SecretKey {
// Открытые конструкторы
   public SecretKeySpec(byte[] key, String algorithm);
   public SecretKeySpec(byte[] key, int offset, int len, String algorithm);
// Методы, реализующие Key
   public String getAlgorithm();
   public byte[] getEncoded();
   public String getFormat();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

Глава 19

javax.net и javax.net.ssl

В этой главе описывается пакет javax.net и его подпакет javax.net.ssl. Первоначально эти пакеты входили в расширение Java Secure Sockets Extension (JSSE), а впоследствии они были интегрированы в Java 1.4, поэтому их названия начинаются с «javax».

javax.net – это небольшой пакет, в котором определены лишь абстрактные классы-фабрики для создания сетевых и серверных сокетов. Пакет javax.net.ssl содержит потомки этих классов, имеющие более специализированное назначение. Они поддерживают безопасное сетевое соединение посредством протокола SSL и тесно связанного с ним протокола TLS.

Пакет javax.net

Java 1.4

В этом пакете определены классы-фабрики для создания сокетов и серверных сокетов. Эти классы можно применять для создания обычных сокетов типа java.net.Socket и java.net.ServerSocket. Однако более важно то, что можно создавать потомки этих классов-фабрик для сокетов других типов, например сокетов с поддержкой SSL из пакета javax.net.ssl.

Классы

```
public abstract class ServerSocketFactory;
public abstract class SocketFactory;
```

ServerSocketFactory

Java 1.4

javax.net

В этом абстрактном классе определены методы-фабрики для создания объектов серверных сокетов. Используйте статический метод getDefault(), чтобы получить определенный по умолчанию объект ServerSocketFactory, с помощью которого можно создавать обычные сокеты java.net.ServerSocket. После этого нужно вызвать один из методов createServerSocket(), чтобы создать новый сокет и, при необходимости, связать его с локальным портом, указав максимально допустимую задержку соединений при постановке их в очередь. См. также класс javax.net.ssl.SSLServerSocketFactory, предназначенный для создания объектов javax.net.ssl.SSLServerSocket, обеспечивающих безопасное соединение.

```
public abstract class ServerSocketFactory {
// Защищенные конструкторы
   protected ServerSocketFactory();
// Открытые методы класса
   public static ServerSocketFactory getDefault();
// Открытые методы экземпляра
   public java.net.ServerSocket createServerSocket() throws java.io.IOException;
   public abstract java.net.ServerSocket createServerSocket(int port)
        throws java.io.IOException;
   public abstract java.net.ServerSocket createServerSocket(int port, int backlog)
        throws java.io.IOException;
   public abstract java.net.ServerSocket createServerSocket(int port, int backlog,
        java.net.InetAddress ifAddress) throws java.io.IOException;
}
```

Подклассы: javax.net.ssl.SSLServerSocketFactory

Возвращается методами: ServerSocketFactory.getDefault(), javax.net.ssl.SSLServerSocketFactory.getDefault()

SocketFactory Java 1.4

javax.net

Этот абстрактный класс содержит методы-фабрики для создания объектов сокетов. С помощью статического метода getDefault() можно получить определенный по умолчанию объект SocketFactory, который создает обыкновенные сокеты типа java.net.Socket. (Этот объект SocketFactory применяется конструктором Socket() и облегчает создание сокетов.) После создания объекта SocketFactory нужно вызвать один из методов createSocket(). Этот метод возвратит новый сокет, который может быть связан с локальным адресом и портом. См. также описание класса-фабрики javax.net.ssl.SSLSocketFactory, который создает объекты javax.net.ssl.SSLSocket, поддерживающие безопасное соединение.

```
public abstract class SocketFactory {
// Защищенные конструкторы
   protected SocketFactory();
// Открытые методы класса
   public static SocketFactory getDefault();
// Открытые методы экземпляра
   public java.net.Socket createSocket() throws java.io.IOException;
   public abstract java.net.Socket createSocket(String host, int port)
        throws java.io.IOException, java.net.UnknownHostException;
   public abstract java.net.Socket createSocket(java.net.InetAddress host, int port)
        throws java.io.IOException;
   public abstract java.net.Socket createSocket(java.net.InetAddress address, int port,
        java.net.InetAddress localAddress, int localPort) throws java.io.IOException;
   public abstract java.net.Socket createSocket(String host, int port, java.net.InetAddress
        localHost, int localPort) throws java.io.IOException, java.net.UnknownHostException;
}
```

Подклассы: javax.net.ssl.SSLSocketFactory

Возвращается методами: SocketFactory.getDefault(), javax.net.ssl.SSLSocketFactory.getDefault()

Пакет javax.net.ssl Java 1.4

В этом пакете содержится API для работы с защищенными сетевыми сокетами, использующими протокол защищенных сокетов (Secure Socket Layer, SSL) или тесно связанный с ним безопасный транспортный протокол (Transport Layer Security, TLS). Здесь также определены классы SSLSocket и SSLServerSocket, наследующие классы обыкновенных и серверных сокетов пакета java.net. Клиенты, которые должны выполнять простые сетевые задачи с использованием SSL, могут создавать объект SSLSocket следующим образом:

```
SSLSocketFactory factory = SSLSocketFactory.getDefault();
SSLSocket securesock =(SSLSocket)factory.getSocket(hostname,443);     // порт https
```

После создания объекта SSLSocket его можно использовать точно так же, как обычный сокет java.net.Socket. После того как с помощью SSLSocket установлено соединение, можно вызвать метод getSession(), чтобы получить объект SSLSession, предоставляющий информацию о соединении. Следует заметить, что этот пакет поддерживает не только протокол SSL (на что указывает имя этого пакета и его ключевых классов), но и протокол TLS. (В реализации фирмы Sun предустановленный поставщик поддерживает SSL 3.0 и TLS 1.0.) Поскольку протокол TLS очень похож на SSL, в дальнейшем мы будем применять только термин SSL.

Класс SSLSocket позволяет выполнять различные сетевые задачи на основе соединения двух систем с использованием протокола SSL. В настоящее время SSL чаще всего применяется вместе с веб-протоколом https:. В платформе Java протокол https: поддерживается не только этим пакетом, но и классом java.net.URL, который позволяет безопасно передавать данные по сети без прямого использования пакета javax.net.ssl. Объект URLConnection, возвращаемый методом openConnection() объекта https:URL, можно привести к типу HttpsURLConnection. В последнем определены методы, учитывающие особенности SSL. Дополнительную информацию по применению URL можно найти в описании классов java.net.URL и java.net.URLConnection.

Этот пакет намного сложнее, чем может показаться из приведенного выше примера: он предоставляет возможность гибкой настройки SSL. Кроме того, этот пакет основан на системе поставщиков и не привязан к конкретным алгоритмам, как и все остальные пакеты системы безопасности Java. Знакомство с этим пакетом лучше всего начать не с классов сокетов, классов-фабрик или класса HttpsURLConnection, а с SSLContext. Этот класс является «фабрикой фабрик» сокетов. Он считается главным классом данного пакета. Чтобы настроить соединение на основе SSL, нужно создать объект SSLContext; при этом можно указать необходимого поставщика реализации. Затем нужно проинициализировать SSLContext, предоставив объект KeyManager – источник информации об аутентификации, которая при необходимости будет передана удаленному хосту. Кроме того, нужно проинициализировать объект TrustManager – верификатор информации об аутентификации, полученной от удаленного хоста. Также нужно предоставить объект java.security.SecureRandom для генерации случайных чисел. После инициализации объекта SSLContext его можно применять для создания фабрик SSLSocketFactory и SSLServerSocketFactory, использующих предоставленные объекты KeyManager и TrustManager.

Содержимое этого пакета также называется расширением защищенных сокетов Java (Java Secure Sockets Extension), или JSSE. До того как JSSE было встроено в Java 1.4, оно являлось стандартным расширением. JSSE 1.0.2 для Java 1.2 и 1.3 можно загрузить отдельно. Однако в JSSE, встроенном в Java 1.4, внесены некоторые изменения.

Интерфейсы

```
public interface HostnameVerifier;
public interface KeyManager;
public interface ManagerFactoryParameters;
public interface SSLSession;
public interface SSLSessionContext;
public interface TrustManager;
public interface X509KeyManager extends KeyManager;
public interface X509TrustManager extends TrustManager;
```

События

```
public class HandshakeCompletedEvent extends java.util.EventObject;
public class SSLSessionBindingEvent extends java.util.EventObject;
```

Слушатели событий

```
public interface HandshakeCompletedListener extends java.util.EventListener;
public interface SSLSessionBindingListener extends java.util.EventListener;
```

Другие классы

```
public abstract class HttpsURLConnection extends java.net.HttpURLConnection;
public class KeyManagerFactory;
public abstract class KeyManagerFactorySpi;
public class SSLContext;
public abstract class SSLContextSpi;
public final class SSLPermission extends java.security.BasicPermission;
public abstract class SSLServerSocket extends java.net.ServerSocket;
public abstract class SSLServerSocketFactory extends javax.net.ServerSocketFactory;
public abstract class SSLSocket extends java.net.Socket;
public abstract class SSLSocketFactory extends javax.net.SocketFactory;
public class TrustManagerFactory;
public abstract class TrustManagerFactorySpi;
```

Исключения

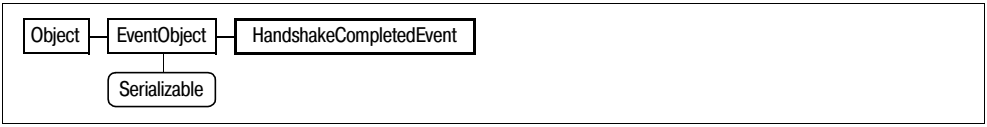
```
public class SSLException extends java.io.IOException;
   └ public class SSLHandshakeException extends SSLException;
   └ public class SSLKeyException extends SSLException;
   └ public class SSLPeerUnverifiedException extends SSLException;
   └ public class SSLProtocolException extends SSLException;
```

HandshakeCompletedEvent Java 1.4

javax.net.ssl ***сериализуемый, событие***

Сокет `SSLSocket` передает экземпляр этого класса методам `handshakeCompleted()` всех зарегистрированных объектов `HandshakeCompletedListener` после завершения этапа «рукопожатия» процедуры установления связи. Методы класса `HandshakeCompletedEvent` возвращают различного рода информацию, полученную на этапе «рукопожатия» (например, имя используемого набора шифров и цепочку сертификатов удаленного хоста).

Обратите внимание, что метод `getPeerCertificateChain()` возвращает объект пакета `javax.security.cert`, который в данной книге не описан. Указанные метод и пакет служат для обратной совместимости с ранними версиями JSSE API и считаются устаревшими (deprecated). Вместо них нужно применять метод `getPeerCertificates()`, работающий с пакетом `java.security.cert`.




```
public class HandshakeCompletedEvent extends java.util.EventObject {
// Открытые конструкторы
   public HandshakeCompletedEvent(SSLSocket sock, SSLSession s);
// Методы доступа к свойствам (по имени свойства)
   public String getCipherSuite();
   public java.security.cert.Certificate[] getLocalCertificates();
   public javax.security.cert.X509Certificate[] getPeerCertificateChain()
        throws SSLPeerUnverifiedException;
   public java.security.cert.Certificate[] getPeerCertificates() throws
SSLPeerUnverifiedException;
   public SSLSession getSession();
   public SSLSocket getSocket();
}
```

Передается методам: `HandshakeCompletedListener.handshakeCompleted()`

HandshakeCompletedListener

Java 1.4

javax.net.ssl ***слушатель событий***

Этот интерфейс реализуется всеми классами, которые с помощью метода `handshakeCompleted()` должны получать уведомления о завершении этапа «рукопожатия» соединения SSL. Можно зарегистрировать слушателя `HandshakeCompletedListener` для сокета `SSLSocket`, передав его методу `addHandshakeCompletedListener()` данного сокета. По завершении этапа «рукопожатия» сокет вызовет методы `handshakeCompleted()` всех зарегистрированных слушателей, передав им в качестве аргумента объект `HandshakeCompletedEvent`.


```
public interface HandshakeCompletedListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void handshakeCompleted(HandshakeCompletedEvent event);
}
```

Передается методам: `SSLSocket.{addHandshakeCompletedListener(), removeHandshakeCompletedListener()}`

HostnameVerifier

Java 1.4

javax.net.ssl

Объект, реализующий этот интерфейс, может применяться вместе с `HttpsURLConnection` для обработки ситуации, в которой имя хоста, содержащегося в URL, не совпадает с именем хоста, полученным на этапе «рукопожатия» SSL с сервером. Например, эта ситуация возникает, когда веб-сайт использует защищенный сертификат компании, предоставляющей хостинг. В этом случае вызывается метод `verify()` объекта `HostnameVerifier`, чтобы определить, продолжать ли дальше соединение или нет. Метод `verify()`

должен возвращать true для продолжения соединения и false в случае, если соединение нужно разорвать. Аргумент hostname этого метода определяет имя хоста, содержащееся в URL, а session – имя хоста, полученного при аутентификации сервера. Если объектом HttpsURLConnection не зарегистрирован ни один HostnameVerifier, то при несоответствии имен хостов соединение всегда будет завершаться неудачей. В приложениях, управляемых пользователем (например, в веб-броузере), объект HostnameVerifier можно применять для запроса подтверждения у пользователя о создании соединения.

```
public interface HostnameVerifier {
// Открытые методы экземпляра
   public abstract boolean verify(String hostname, SSLSession session);
}
```

Передается методам: javax.naming.ldap.StartTlsResponse.setHostnameVerifier(), HttpsURLConnection.{setDefaultHostnameVerifier(), setHostnameVerifier()}

Возвращается методами: HttpsURLConnection.{getDefaultHostnameVerifier(), getHostnameVerifier()}

Экземпляры: HttpsURLConnection.hostnameVerifier

HttpsURLConnection Java 1.4

javax.net.ssl

Этот класс является потомком java.net.URLConnection и предназначен для работы с URL с использованием протокола https:. Он расширяет java.net.HttpURLConnection. Помимо унаследованных методов в этом классе определены собственные методы для указания объектов SSLSocketFactory и HostnameVerifier, которые применяются при установлении соединения. Эти методы в статическом варианте позволяют определить объект-фабрику и объект-верификатор по умолчанию, которые будут использоваться со всеми объектами HttpsURLConnection. После установления соединения можно с помощью других методов этого класса получить информацию о соединении (например, о наборе шифров и сертификатах сервера).

Объект HttpsURLConnection можно получить, вызвав метод openConnection() объекта URL, использующего спецификатор протокола https://. Возвращаемое значение нужно привести к типу HttpsURLConnection. Соединение для объекта HttpsURLConnection в этот момент не установлено, поэтому для настройки соединения можно задействовать методы setHostnameVerifier() и setSSLSocketFactory(). Если не указан объект HostnameVerifier для данного экземпляра класса или не определен объект HostnameVerifier по умолчанию, то при несовпадении имен хостов соединение будет завершаться неудачей. Если не определен объект SSLSocketFactory для данного экземпляра, то будет использоваться объект по умолчанию. Для создания соединения нужно вызвать унаследованный метод connect(), а затем с помощью унаследованного метода getContent() можно получить содержимое URL или же вызвать унаследованный метод getInputStream(), возвращающий поток java.io.InputStream, с помощью которого можно прочитать содержимое URL.


```
public abstract class HttpsURLConnection extends java.net.HttpURLConnection {
// Защищенные конструкторы
```

```
   protected HttpsURLConnection(java.net.URL url) throws java.io.IOException;
// Открытые методы класса
   public static HostnameVerifier getDefaultHostnameVerifier();
   public static SSLSocketFactory getDefaultSSLSocketFactory();
   public static void setDefaultHostnameVerifier(HostnameVerifier v);
   public static void setDefaultSSLSocketFactory(SSLSocketFactory sf);
// Методы доступа к свойствам (по имени свойства)
   public abstract String getCipherSuite();
   public HostnameVerifier getHostnameVerifier();
   public void setHostnameVerifier(HostnameVerifier v);
   public abstract java.security.cert.Certificate[] getLocalCertificates();
   public abstract java.security.cert.Certificate[] getServerCertificates()
        throws SSLPeerUnverifiedException;
   public SSLSocketFactory getSSLSocketFactory();
   public void setSSLSocketFactory(SSLSocketFactory sf);
// Защищенные поля экземпляра
   protected HostnameVerifier hostnameVerifier;
}
```

KeyManager

Java 1.4

javax.net.ssl

Этот интерфейс служит для идентификации менеджеров ключей. Менеджер ключей отвечает за получение данных и управление данными (например, цепочкой сертификатов и связанным с ней закрытым ключом), которые могут применяться при аутентификации локального хоста удаленным хостом. Этот объект обычно используется сервером соединения SSL, но его может задействовать и клиентская сторона.

Для получения объектов `KeyManager` применяется `KeyManagerFactory`. Созданные таким образом объекты `KeyManager` всегда можно привести к типу интерфейса-потомка, работающего с конкретными верительными данными. См. `X509KeyManager`.

```
public interface KeyManager {
}
```

Реализации: `X509KeyManager`

Передается методам: `SSLContext.init()`, `SSLContextSpi.engineInit()`

Возвращается методами: `KeyManagerFactory.getKeyManagers()`, `KeyManagerFactorySpi.engineGetKeyManagers()`

KeyManagerFactory

Java 1.4

javax.net.ssl

Класс `KeyManagerFactory` отвечает за создание объектов `KeyManager`, использующих заданный алгоритм управления ключами. Получить объект `KeyManagerFactory` можно с помощью одного из методов `getInstance()`. При этом нужно обязательно указать требуемый алгоритм. Кроме того, можно указать поставщика. В Java 1.4 «SunX509» является единственным алгоритмом, поддерживаемым поставщиком по умолчанию «SunJSSE». После вызова метода `getInstance()` нужно проинициализировать объект-фабрику с помощью метода `init()`. Для алгоритма «SunX509» следует всегда использовать метод `init()` с двумя аргументами. Первым аргументом должен быть объект `KeyStore`, содержащий закрытые ключи и сертификаты, которые необходимы объектам `X509KeyManager`, а вторым – пароль, служащий для защиты закрытых ключей объекта `KeyStore`. После создания и инициализации объекта `KeyManagerFactory` нужно вы-

звать метод getKeyManagers() для создания объекта KeyManager. Этот метод возвращает массив объектов KeyManager, потому что некоторые алгоритмы управления ключами могут работать с несколькими типами ключей. Алгоритм «SunX509» работает только с ключами X509 и всегда возвращает массив с единственным элементом – объектом X509KeyManager. Как правило, этот массив передается методу init() объекта SSLContext.

Если при использовании алгоритма «SunX509» методу init() объекта-фабрики KeyManagerFactory не были переданы хранилище ключей KeyStore и пароль, то этот объект производит попытку чтения хранилища ключей KeyStore из файла, указанного в системном свойстве javax.net.ssl.keyStore, и использует пароль из свойства javax.net.ssl.keyStorePassword. Тип хранилища определен в javax.net.ssl.keyStoreType.

```
public class KeyManagerFactory {
// Защищенные конструкторы
   protected KeyManagerFactory(KeyManagerFactorySpi factorySpi, java.security.Provider provider,
                                                                String algorithm);
// Открытые методы класса
   public static final String getDefaultAlgorithm();
   public static final KeyManagerFactory getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static final KeyManagerFactory getInstance(String algorithm,
        java.security.Provider provider) throws java.security.NoSuchAlgorithmException;
   public static final KeyManagerFactory getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException,java.security.NoSuchProviderException;
// Открытые методы экземпляра
   public final String getAlgorithm();
   public final KeyManager[] getKeyManagers();
   public final java.security.Provider getProvider();
   public final void init(ManagerFactoryParameters spec)
        throws java.security.InvalidAlgorithmParameterException;
   public final void init(java.security.KeyStore ks, char[] password)
        throws java.security.KeyStoreException,
        java.security.NoSuchAlgorithmException, java.security.UnrecoverableKeyException;
}
```

Возвращается методами: KeyManagerFactory.getInstance()

KeyManagerFactorySpi Java 1.4

javax.net.ssl

Этот абстрактный класс определяет интерфейс поставщика сервисов (Service Provider Interface) для объекта KeyManagerFactory. Данный интерфейс необходимо реализовать при создании поставщиков сервисов, а в обычных приложениях использовать его не следует.

```
public abstract class KeyManagerFactorySpi {
// Открытые конструкторы
   public KeyManagerFactorySpi();
// Защищенные методы экземпляра
   protected abstract KeyManager[] engineGetKeyManagers();
   protected abstract void engineInit(ManagerFactoryParameters spec)
        throws java.security.InvalidAlgorithmParameterException;
   protected abstract void engineInit(java.security.KeyStore ks, char[] password)
        throws java.security.KeyStoreException,
        java.security.NoSuchAlgorithmException, java.security.UnrecoverableKeyException;
}
```

Передается методам: KeyManagerFactory.KeyManagerFactory()

ManagerFactoryParameters Java 1.4

javax.net.ssl

Этот интерфейс-маркер определяет объекты, которые предоставляют параметры инициализации объектов KeyManagerFactory и TrustManagerFactory для определенного алгоритма или поставщика. Для этих классов-фабрик предустановленный поставщик «SunJSSE» фирмы Sun поддерживает единственный тип алгоритма, «SunX509». Такие классы должны быть инициализированы с помощью объекта KeyStore. Они не требуют специальных параметров в виде объекта ManagerFactoryParameters. Тем не менее в пакете javax.net.ssl не определены потомки этого интерфейса, и он не используется вместе с предустановленным поставщиком. Однако в будущем он может применяться сторонними производителями поставщиков.

```
public interface ManagerFactoryParameters {
}
```

Передается методам: KeyManagerFactory.init(), KeyManagerFactorySpi.engineInit(), TrustManagerFactory.init(), TrustManagerFactorySpi.engineInit()

SSLContext Java 1.4

javax.net.ssl

Этот класс-фабрика предназначен для создания фабрик сокетов и серверных сокетов. Это главный класс пакета javax.net.ssl, однако его не требуется напрямую использовать в приложениях. В большинстве приложений применяются объекты SSLSocketFactory и SSLServerSocketFactory, возвращаемые статическими методами getDefault() этих классов. Приложения, которые в сетевых задачах с применением SSL будут использовать другую реализацию поставщика системы защиты, а не реализацию, установленную по умолчанию, должны применять собственные классы-фабрики сокетов, созданные объектом SSLContext. То же самое относится к приложениям, которые будут настраивать управление ключами и сертификатами соединения SSL.

Создать объект SSLContext можно с помощью метода getInstance(), передав ему в качестве аргументов имя защищенного протокола и (необязательно) имя поставщика. Предустановленный поставщик «SunJSSE» поддерживает следующие имена протоколов: «SSL», «SSLv2», «SSLv3», «TLS» и «TLSv1». После создания объекта SSLContext нужно вызвать его метод init() и передать ему объекты KeyManager, TrustManager и SecureRandom. Если один из аргументов равен null, то используется соответствующее значение по умолчанию. Наконец, с помощью методов getSocketFactory() и getServerSocketFactory() можно получить объекты SSLSocketFactory и SSLServerSocketFactory.

```
public class SSLContext {
// Защищенные конструкторы
   protected SSLContext(SSLContextSpi contextSpi, java.security.Provider provider, String protocol);
// Открытые методы класса
   public static SSLContext getInstance(String protocol) throws java.security.NoSuchAlgorithmException;
   public static SSLContext getInstance(String protocol, java.security.Provider provider)
        throws java.security.NoSuchAlgorithmException;
   public static SSLContext getInstance(String protocol, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
// Методы доступа к свойствам (по имени свойства)
```

```
   public final SSLSessionContext getClientSessionContext();
   public final String getProtocol();
   public final java.security.Provider getProvider();
   public final SSLSessionContext getServerSessionContext();
   public final SSLServerSocketFactory getServerSocketFactory();
   public final SSLSocketFactory getSocketFactory();
// Открытые методы экземпляра
   public final void init(KeyManager[] km, TrustManager[] tm, java.security.SecureRandom random)
        throws java.security.KeyManagementException;
}
```

Возвращается методами: `SSLContext.getInstance()`

SSLContextSpi Java 1.4

javax.net.ssl

Этот абстрактный класс определяет интерфейс поставщика сервисов для класса `SSLContext`. Данный интерфейс нужно реализовать при создании поставщика сервисов. В обычных приложениях этот класс не используется.

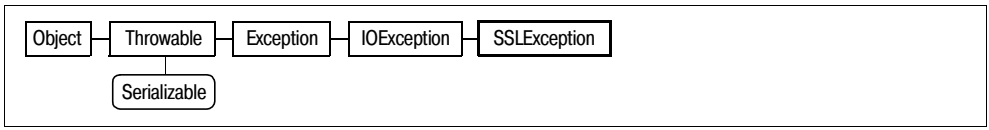
```
public abstract class SSLContextSpi {
// Открытые конструкторы
   public SSLContextSpi();
// Защищенные методы экземпляра
   protected abstract SSLSessionContext engineGetClientSessionContext();
   protected abstract SSLSessionContext engineGetServerSessionContext();
   protected abstract SSLServerSocketFactory engineGetServerSocketFactory();
   protected abstract SSLSocketFactory engineGetSocketFactory();
   protected abstract void engineInit(KeyManager[] km, TrustManager[] tm,
       java.security.SecureRandom sr) throws java.security.KeyManagementException;
}
```

Передается методам: `SSLContext.SSLContext()`

SSLException Java 1.4

javax.net.ssl ***сериализуемое, проверяемое***

Это исключение сообщает о возникновении особой ситуации, связанной с SSL. Данный класс является родительским классом для более специфичных типов исключений SSL.



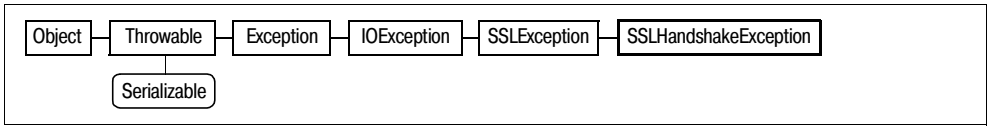

```
public class SSLException extends java.io.IOException {
// Открытые конструкторы
   public SSLException(String reason);
}
```

Подклассы: `SSLHandshakeException`, `SSLKeyException`, `SSLPeerUnverifiedException`, `SSLProtocolException`

SSLHandshakeException Java 1.4

javax.net.ssl ***сериализуемое, проверяемое***

Это исключение возникает, если этап «рукопожатия» при установлении соединения SSL завершился неудачей по причинам, не связанным с аутентификацией (см. SSLPeerUnverifiedException). Например, оно может генерироваться в том случае, когда клиент и сервер не смогли договориться о взаимоприемлемом наборе шифров. Когда возникает исключение этого типа, объект SSLSocket больше не пригоден.

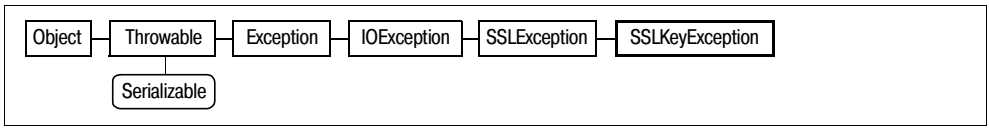



```
public class SSLHandshakeException extends SSLException {
// Открытые конструкторы
   public SSLHandshakeException(String reason);
}
```

SSLKeyException Java 1.4

javax.net.ssl ***сериализуемое, проверяемое***

Это исключение сообщает о проблемах, связанных с сертификатом открытого ключа и закрытым ключом, которые применяются сервером (клиентом) для аутентификации.

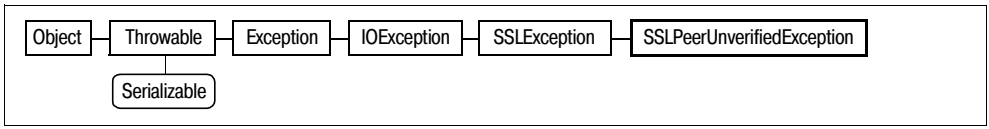



```
public class SSLKeyException extends SSLException {
// Открытые конструкторы
   public SSLKeyException(String reason);
}
```

SSLPeerUnverifiedException Java 1.4

java.net.ssl ***сериализуемое, проверяемое***

Это исключение сообщает о том, что аутентификация удаленного хоста завершилась неудачей.




```
public class SSLPeerUnverifiedException extends SSLException {
// Открытые конструкторы
   public SSLPeerUnverifiedException(String reason);
}
```

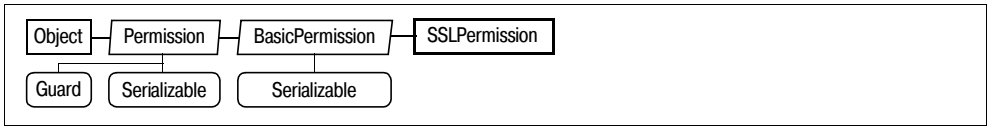
Генерируется методами: `HandshakeCompletedEvent.{getPeerCertificateChain(), getPeerCertificates()}`, `HttpsURLConnection.getServerCertificates()`, `SSLSession.{getPeerCertificateChain(), getPeerCertificates()`

SSLPermission

Java 1.4

javax.net.ssl ***сериализуемый***

Этот класс отвечает за доступ к уязвимым методам пакета `javax.net.ssl`. В нем определены два целевых объекта – «setHostnameVerifier» и «getSSLSessionContext». Первый требуется для разрешения на вызов методов `HttpURLConnection.setHostnameVerifier()` и `HttpURLConnection.setDefaultHostnameVerifier()`, а второй – для вызова метода `SSLSession.getSessionContext()`.



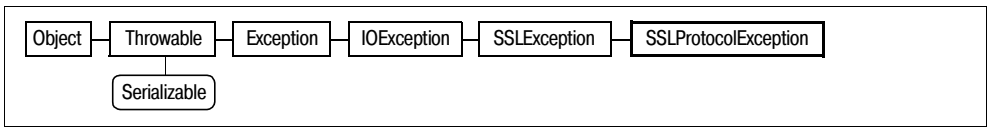

```
public final class SSLPermission extends java.security.BasicPermission {
// Открытые конструкторы
   public SSLPermission(String name);
   public SSLPermission(String name, String actions);
}
```

SSLProtocolException

Java 1.4

javax.net.ssl ***сериализуемое, проверяемое***

Это исключение сообщает об ошибке на уровне протокола SSL. Исключение этого типа обычно свидетельствует о наличии ошибки в реализации SSL, используемой на локальном или удаленном хосте.




```
public class SSLProtocolException extends SSLException {
// Открытые конструкторы
   public SSLProtocolException(String reason);
}
```

SSLServerSocket

Java 1.4

javax.net.ssl

Этот класс является сокетом типа `java.net.ServerSocket` с поддержкой SSL. Он применяется для ожидания и принятия запросов на соединение от клиентов, а также при создании объектов `SSLSocket` для связи с этими клиентами. Создать объект `SSLServerSocket` и связать его с локальным портом можно с помощью методов `createServerSocket()` класса `SSLServerSocketFactory`. После создания `SSLServerSocket` с ним можно работать точно так же, как с обычным сокетом `ServerSocket`: принять запрошенное клиен-

том соединение с помощью унаследованного метода accept(), возвращающего объект Socket. Этот объект Socket, а также SSLServerSocket могут быть приведены к типу SSLSocket.

В классе SSLServerSocket определены методы для настройки доступных протоколов и наборов шифров, а также для запроса полного набора поддерживаемых протоколов и шифров. См. описание класса SSLSocket, содержащего одноименные методы. Если требуется аутентификация сервера клиентами, нужно вызвать метод setWantClientAuth() или setNeedClientAuth(). После этого объекты SSLSocket, возвращенные методом accept(), будут запрашивать или требовать аутентификацию.

Как правило, в сетевых соединениях с использованием SSL клиент требует от сервера информацию об аутентификации. При создании объекта SSLServerSocket с помощью SSLServerSocketFactory, определенного по умолчанию, требуемая информация – это сертификат открытого ключа X.509 и соответствующий закрытый ключ. Для получения этой информации объект SSLServerSocketFactory, заданный по умолчанию, использует объект X509KeyManager. Предопределенный объект X509KeyManager пытается прочитать эту информацию из файла java.security.KeyStore, заданного в системном свойстве javax.net.ssl.keyStore. В качестве пароля он использует системное свойство javax.net.ssl.keyStorePassword, а значение, указанное в свойстве javax.net.ssl.keyStoreType, задает тип хранилища ключей. Хранилище ключей должно содержать только допустимые ключи и цепочки сертификатов, идентифицирующие сервер; объект X509KeyManager автоматически выбирает подходящий ключ и цепочку сертификатов.

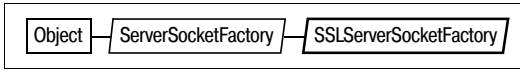

```
public abstract class SSLServerSocket extends java.net.ServerSocket {
// Защищенные конструкторы
   protected SSLServerSocket() throws java.io.IOException;
   protected SSLServerSocket(int port) throws java.io.IOException;
   protected SSLServerSocket(int port, int backlog) throws java.io.IOException;
   protected SSLServerSocket(int port, int backlog, java.net.InetAddress address)
        throws java.io.IOException;
// Методы доступа к свойствам (по имени свойства)
   public abstract String[] getEnabledCipherSuites();
   public abstract void setEnabledCipherSuites(String[] suites);
   public abstract String[] getEnabledProtocols();
   public abstract void setEnabledProtocols(String[] protocols);
   public abstract boolean getEnableSessionCreation();
   public abstract void setEnableSessionCreation(boolean flag);
   public abstract boolean getNeedClientAuth();
   public abstract void setNeedClientAuth(boolean flag);
   public abstract String[] getSupportedCipherSuites();
   public abstract String[] getSupportedProtocols();
   public abstract boolean getUseClientMode();
   public abstract void setUseClientMode(boolean flag);
   public abstract boolean getWantClientAuth();
   public abstract void setWantClientAuth(boolean flag);
}
```

SSLServerSocketFactory Java 1.4

javax.net.ssl

Этот класс является потомком javax.net.ServerSocketFactory. Он предназначен для создания объектов SSLServerSocket. Обычно в приложениях используется определенный по умолчанию объект SSLServerSocketFactory, возвращаемый статическим методом getDefault(). После получения объекта SSLServerSocketFactory можно вызывать один из его унаследованных методов createServerSocket(), которые создают объект SSLServerSocket и могут связать его с локальным портом. Возвращаемое значение методов createServerSocket() – это объект java.net.ServerSocket, который можно безопасно приводить к типу SSLServerSocket.

Если в приложении нужно изменить конфигурацию SSL, но нельзя воспользоваться предопределенным SSLServerSocketFactory, то можно создать собственный объект этого типа с помощью «фабрики фабрик» сокетов SSLContext. Подробности представлены в описании SSLContext.




```
public abstract class SSLServerSocketFactory extends javax.net.ServerSocketFactory {
// Защищенные конструкторы
   protected SSLServerSocketFactory();
// Открытые методы класса
   public static javax.net.ServerSocketFactory getDefault();                    // синхронизирован
// Открытые методы экземпляра
   public abstract String[] getDefaultCipherSuites();
   public abstract String[] getSupportedCipherSuites();
}
```

Возвращается методами: SSLContext.getServerSocketFactory(), SSLContextSpi.engineGetServerSocketFactory()

SSLSession Java 1.4

javax.net.ssl

Объект SSLSession содержит информацию о соединении SSL на основе SSLSocket. Чтобы получить объект SSLSession для сокета SSLSocket, используйте метод getSession() этого сокета. Методы объекта SSLSession возвращают информацию, полученную на этапе «рукопожатия» при установлении соединения. Метод getProtocol() возвращает используемую версию протокола SSL или TLS. Метод getCipherSuite() возвращает имя набора шифров для данного соединения. getPeerHost() возвращает имя удаленного хоста, а getPeerCertificates() – цепочку сертификатов (если она существует), полученную от удаленного хоста во время процедуры аутентификации.

Метод invalidate() завершает сессию. Он не влияет на текущие соединения, но для установления новых соединений и изменения существующих потребуется создание нового объекта SSLSession.

Если между двумя хостами установлено несколько соединений SSL, то эти соединения могут использовать общий объект SSLSession, если применяются протоколы одной версии и один и тот же набор шифров. Получить список объектов SSLSocket, совместно использующих сессию, невозможно, но сокеты могут обмениваться между

собой информацией с помощью метода putValue(). Этот метод связывает совместно используемый объект с некоторым определенным заранее именем, а с помощью метода getValue() другие сокеты могут найти этот объект. Метод removeValue() удаляет такую связь, а getValueNames() возвращает для данной сессии массив всех имен, связанных с объектами. Эти объекты могут реализовать интерфейс SSLSessionBindingListener, позволяющий им получать уведомления о том, что они присоединены к сессии или удалены из нее.

Заметьте, что возвращаемое значение метода getPeerCertificateChain() – это объект пакета javax.security.cert, который не описан в данной книге. Указанные метод и пакет служат для обратной совместимости с JSSE API и считаются устаревшими (deprecated). Вместо них следует применять метод getPeerCertificates(), работающий с пакетом java.security.cert.

```
public interface SSLSession {
// Методы доступа к свойствам (по имени свойства)
   public abstract String getCipherSuite();
   public abstract long getCreationTime();
   public abstract byte[] getId();
   public abstract long getLastAccessedTime();
   public abstract java.security.cert.Certificate[] getLocalCertificates();
   public abstract javax.security.cert.X509Certificate[] getPeerCertificateChain()
        throws SSLPeerUnverifiedException;
   public abstract java.security.cert.Certificate[] getPeerCertificates()
        throws SSLPeerUnverifiedException;
   public abstract String getPeerHost();
   public abstract String getProtocol();
   public abstract SSLSessionContext getSessionContext();
   public abstract String[] getValueNames();
// Открытые методы экземпляра
   public abstract Object getValue(String name);
   public abstract void invalidate();
   public abstract void putValue(String name, Object value);
   public abstract void removeValue(String name);
}
```

Передается методам: HandshakeCompletedEvent.HandshakeCompletedEvent(), HostnameVerifier.verify(), SSLSessionBindingEvent.SSLSessionBindingEvent()

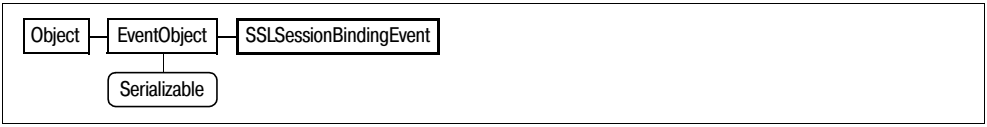
Возвращается методами: javax.naming.ldap.StartTlsResponse.negotiate(), HandshakeCompletedEvent.getSession(), SSLSessionBindingEvent.getSession(), SSLSessionContext.getSession(), SSLSocket.getSession()

SSLSessionBindingEvent

Java 1.4

javax.net.ssl ***сериализуемый, событие***

Объект этого типа передается методам valueBound() и valueUnbound() объекта, реализующего интерфейс SSLSessionBindingListener, когда метод putValue() (removeValue()) класса SSLSession связывает (удаляет) объект SSLSessionBindingListener в данной сессии. Метод getName() возвращает имя, которое связывается с объектом или отсоединяется от него, а getSession() возвращает объект SSLSession, в котором была создана (удалена) эта связь.




```
public class SSLSessionBindingEvent extends java.util.EventObject {
// Открытые конструкторы
   public SSLSessionBindingEvent(SSLSession session, String name);
// Открытые методы экземпляра
   public String getName();
   public SSLSession getSession();
}
```

Передается методам: SSLSessionBindingListener.{valueBound(), valueUnbound()}

SSLSessionBindingListener Java 1.4

javax.net.ssl ***слушатель события***

Этот интерфейс реализуется объектом, который должен получать уведомления, когда в сессии SSLSession с ним создается связь или такая связь удаляется. Если объект, который был передан в качестве аргумента методу putValue() объекта SSLSession, реализует этот интерфейс, то при вызове метода putValue() будет вызван метод valueBound() этого аргумента. Его метод valueUnbound() будет вызван, когда этот объект удаляется из SSLSession с помощью метода removeValue() или заменяется на другой объект методом putValue(). Аргументом обоих методов интерфейса является SSLSessionBindingEvent. Этот аргумент определяет имя, с которым был связан данный объект, и соответствующую сессию SSLSession.


```
public interface SSLSessionBindingListener extends java.util.EventListener {
// Открытые методы экземпляра
   public abstract void valueBound(SSLSessionBindingEvent event);
   public abstract void valueUnbound(SSLSessionBindingEvent event);
}
```

SSLSessionContext Java 1.4

javax.net.ssl

SSLSessionContext группирует объекты SSLSession и управляет ими. Это низкоуровневый интерфейс, который обычно не применяется в приложениях. Метод getIds() возвращает перечисление Enumeration идентификаторов сессий, а getSession() возвращает объект SSLSession, связанный с указанным идентификатором. Метод setSessionCacheSize() указывает максимальное разрешенное количество сессий в группе, а setSessionTimeout() устанавливает время ожидания этих сессий. Объект SSLSessionContext может служить кэшем для объектов SSLSession и позволяет повторно использовать эти объекты для одновременного установления нескольких соединений между двумя хостами.

От поставщиков не требуется поддерживать этот интерфейс. Если он реализован, то метод getSessionContext() класса SSLSession возвращает объект, реализующий этот ин-

терфейс. Этот объект также возвращают методы getClientSessionContext() и getServerSessionContext(), обеспечивая раздельное управление клиентским и серверным соединением SSL.

```
public interface SSLSessionContext {
// Открытые методы экземпляра
   public abstract java.util.Enumeration getIds();
   public abstract SSLSession getSession(byte[] sessionId);
   public abstract int getSessionCacheSize();
   public abstract int getSessionTimeout();
   public abstract void setSessionCacheSize(int size) throws IllegalArgumentException;
   public abstract void setSessionTimeout(int seconds) throws IllegalArgumentException;
}
```

Возвращается методами: SSLContext.{getClientSessionContext(), getServerSessionContext()}, SSLContextSpi.{engineGetClientSessionContext(), engineGetServerSessionContext()}, SSLSession.getSessionContext()

SSLSocket

Java 1.4

javax.net.ssl

Класс SSLSocket — это защищенный подкласс java.net.Socket, реализующий протоколы SSL или TLS, которые применяются для аутентификации сервера клиентом и шифрования передаваемых данных. Создать объект SSLSocket для соединения с сервером SSL можно с помощью одного из методов createSocket() объекта SSLSocketFactory (см. SSLSocketFactory). В серверном приложении получить объект SSLSocket для связи с клиентом SSL можно с помощью метода accept() класса SSLServerSocket (см. также SSLServerSocket).

Класс SSLSocket наследует все методы обычного сокета. Его можно использовать точно так же, как java.net.Socket. Кроме того, здесь определены методы для управления созданием безопасного соединения. Данные методы можно вызвать до того, как выполнится процедура установки соединения SSL («рукопожатие»). «Рукопожатие» не происходит сразу после создания сокета при первом соединении, поэтому можно устанавливать различные параметры SSL, управляющие этой процедурой. Вызов методов startHandshake(), getSession() или чтение/запись данных через сокет вызывают «рукопожатие», поэтому настроить конфигурацию сокета необходимо до выполнения перечисленных действий. Если при выполнении «рукопожатия» необходимо получить уведомление, то нужно вызвать метод addHandshakeCompletedListener() для регистрации соответствующего объекта-слушателя.

Метод getSupportedProtocols() возвращает список защищенных протоколов, поддерживаемых в данной реализации сокета. Метод setEnabledProtocols() позволяет указать имя (имена) поддерживаемых протоколов, которые будут применяться в данном сокете. Метод getSupportedCipherSuites() возвращает полный список наборов шифров, поддерживаемых данным поставщиком системы защиты. setEnabledCipherSuites() устанавливает список наборов шифров, которые применяются при соединении. Обратите внимание, что по умолчанию доступны не все поддерживаемые наборы шифров, а только те из них, которые непосредственно выполняют шифрование и требуют аутентификации сервера. Если вы хотите разрешить серверу остаться анонимным, вызовите метод setEnabledCipherSuites(), чтобы разрешить применение набора, не требующего аутентификации. Специфические протоколы и наборы шифров здесь не описаны, поскольку их корректное использование требует глубокого понимания

криптографии, изложение которой выходит за рамки данной книги. Чаще всего приложения используют набор протоколов и шифров, определенный по умолчанию.

Если в серверном приложении объект SSLSocket получен при принятии соединения сокетом SSLServerSocket, то можно вызвать метод setWantClientAuth(), запрашивающий аутентификацию клиента на сервере, или setNeedClientAuth(), который требует аутентификации клиента во время «рукопожатия». Но необходимо заметить, что обычно эффективнее вызывать эти методы из серверного сокета, чем из полученного объекта SSLSocket.

Вышеописанные методы для конфигурации сокета нужно вызывать до «рукопожатия» SSL. С помощью метода getSession() можно получить объект SSLSession, у которого затем можно запросить различную информацию о «рукопожатии» (например, об используемом протоколе, наборе шифров и результатах опознания сервера). Заметьте, что метод getSession() вызовет процедуру «рукопожатия», если она до этого не выполнялась, поэтому данный метод можно вызывать в любое время.


```
public abstract class SSLSocket extends java.net.Socket {
// Защищенные конструкторы
   protected SSLSocket();
   protected SSLSocket(String host, int port) throws java.io.IOException,
       java.net.UnknownHostException;
   protected SSLSocket(java.net.InetAddress address, int port)
       throws java.io.IOException, java.net.UnknownHostException;
   protected SSLSocket(String host, int port, java.net.InetAddress clientAddress, int clientPort)
       throws java.io.IOException, java.net.UnknownHostException;
   protected SSLSocket(java.net.InetAddress address, int port, java.net.InetAddress
       clientAddress, int clientPort) throws java.io.IOException, java.net.UnknownHostException;
// Методы регистрации событий (по имени события)
   public abstract void addHandshakeCompletedListener(HandshakeCompletedListener listener);
   public abstract void removeHandshakeCompletedListener(HandshakeCompletedListener listener);
// Методы доступа к свойствам (по имени свойства)
   public abstract String[] getEnabledCipherSuites();
   public abstract void setEnabledCipherSuites(String[] suites);
   public abstract String[] getEnabledProtocols();
   public abstract void setEnabledProtocols(String[] protocols);
   public abstract boolean getEnableSessionCreation();
   public abstract void setEnableSessionCreation(boolean flag);
   public abstract boolean getNeedClientAuth();
   public abstract void setNeedClientAuth(boolean need);
   public abstract SSLSession getSession();
   public abstract String[] getSupportedCipherSuites();
   public abstract String[] getSupportedProtocols();
   public abstract boolean getUseClientMode();
   public abstract void setUseClientMode(boolean mode);
   public abstract boolean getWantClientAuth();
   public abstract void setWantClientAuth(boolean want);
// Открытые методы экземпляра
   public abstract void startHandshake() throws java.io.IOException;
}
```

Передается методам: HandshakeCompletedEvent.HandshakeCompletedEvent()

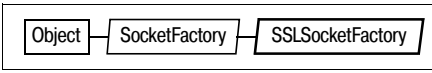
Возвращается методами: HandshakeCompletedEvent.getSocket()

SSLSocketFactory Java 1.4

javax.net.ssl

Этот класс является потомком javax.net.SocketFactory и предназначен для создания объектов SSLSocket. Как правило, в приложениях используется предопределенный объект SSLSocketFactory, возвращаемый статическим методом getDefault(). После получения объекта SSLSocketFactory вызывается унаследованный метод createSocket(), создающий объект SSLSocket и (необязательно) связывающий его с локальным адресом. Возвращаемое значение метода createSocket() – объект типа java.net.Socket, но при необходимости его можно привести к типу SSLSocket. В классе SSLSocketFactory определен еще один метод createSocket(), дополняющий одноименные унаследованные методы. Этот метод создает SSLSocket на основе уже существующего объекта Socket.

Если в приложении требуется изменить конфигурацию SSL и нельзя применять предопределенную фабрику сокетов, то можно получить объект-фабрику SSLSocketFactory из объекта SSLContext. Для получения подробной информации обратитесь к описанию SSLContext.




```
public abstract class SSLSocketFactory extends javax.net.SocketFactory {
// Открытые конструкторы
   public SSLSocketFactory();
// Открытые методы класса
   public static javax.net.SocketFactory getDefault();                    // синхронизирован
// Открытые методы экземпляра
   public abstract java.net.Socket createSocket(java.net.Socket s, String host, int port,
        boolean autoClose) throws java.io.IOException;
   public abstract String[] getDefaultCipherSuites();
   public abstract String[] getSupportedCipherSuites();
}
```

Передается методам: javax.naming.ldap.StartTlsResponse.negotiate(), HttpsURLConnection.{setDefaultSSLSocketFactory(), setSSLSocketFactory()}

Возвращается методами: HttpsURLConnection.{getDefaultSSLSocketFactory(), getSSLSocketFactory()}, SSLContext.getSocketFactory(), SSLContextSpi.engineGetSocketFactory()

TrustManager Java 1.4

javax.net.ssl

Этот интерфейс-маркер служит индикатором объекта-менеджера доверенностей, который отвечает за проверку верительных данных аутентификации (например, цепочки сертификатов), предоставленных удаленным хостом, и принимает решение о доверии этим данным и о принятии их. Объект TrustManager обычно применяется клиентами SSL для аутентификации сервера, но также может использоваться и серверами, требующими аутентификации клиентов.

Для получения объектов TrustManager нужно задействовать TrustManagerFactory. Объекты TrustManager, создаваемые фабрикой TrustManagerFactory, можно привести к типу интерфейсов-потомков, предназначенных для конкретных типов ключей. (См., например, X509TrustManager.)

```
public interface TrustManager {
}
```

Реализации: X509TrustManager

Передается методам: SSLContext.init(), SSLContextSpi.engineInit()

Возвращается методами: TrustManagerFactory.getTrustManagers(), TrustManagerFactorySpi.engineGetTrustManagers()

TrustManagerFactory Java 1.4

javax.net.ssl

Объект TrustManagerFactory отвечает за создание объектов TrustManager для конкретного типа алгоритма. Получить объект TrustManagerFactory можно с помощью одного из методов getInstance(), указав алгоритм и (необязательно) поставщика. В Java 1.4 «SunX509» является единственным алгоритмом, который поддерживает предустановленный поставщик «SunJSSE». После вызова метода getInstance() нужно инициализировать объект-фабрику с помощью метода init(). Если используется алгоритм «SunJSSE», то методу init() нужно передать объект KeyStore. Этот объект должен содержать открытые ключи надежных центров сертификации. После создания и инициализации объекта TrustManagerFactory его можно применять для создания объектов TrustManager с помощью метода getTrustManagers(). Этот метод возвращает массив объектов TrustManager, поскольку в некоторых алгоритмах управления доверенностями могут обрабатываться сертификаты ключей нескольких типов. Алгоритм «SunX509» работает только с ключами X.509 и всегда возвращает массив с одним элементом – объектом X509TrustManager. Затем, как правило, этот массив передается методу init() объекта SSLContext.

Если при использовании алгоритма «SunX509» методу init() объекта TrustManagerFactory не был передан объект KeyStore, то применяется хранилище ключей KeyStore, созданное на основе файла, имя которого определяется значением системного свойства javax.net.ssl.trustStore, если оно существует. (Тип хранилища ключей и пароль берутся из значений системных свойств javax.net.ssl.trustStoreType и javax.net.ssl.trustStorePassword.) В противном случае используется файл *jre/lib/security/jssecacerts* из дистрибутива Java, если такой файл существует. Если нет, то применяется файл *jre/lib/security/cacerts*, являющийся частью дистрибутива Java фирмы Sun. Вместе с Java фирмы Sun поставляется файл cacerts, содержащий сертификаты известных и надежных центров сертификации. Редактировать файл *cacerts* можно с помощью программы *keytool* (пароль по умолчанию – «changeit»).

```
public class TrustManagerFactory {
// Защищенные конструкторы
   protected TrustManagerFactory(TrustManagerFactorySpi factorySpi,
        java.security.Provider provider, String algorithm);
// Открытые методы класса
   public static final String getDefaultAlgorithm();
   public static final TrustManagerFactory getInstance(String algorithm)
        throws java.security.NoSuchAlgorithmException;
   public static final TrustManagerFactory getInstance(String algorithm,
        java.security.Provider provider) throws java.security.NoSuchAlgorithmException;
   public static final TrustManagerFactory getInstance(String algorithm, String provider)
        throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException;
// Открытые методы экземпляра
   public final String getAlgorithm();
```

```
    public final java.security.Provider getProvider();
    public final TrustManager[] getTrustManagers();
    public final void init(ManagerFactoryParameters spec)
        throws java.security.InvalidAlgorithmParameterException;
    public final void init(java.security.KeyStore ks) throws java.security.KeyStoreException;
}
```

Возвращается методами: TrustManagerFactory.getInstance()

TrustManagerFactorySpi Java 1.4

javax.net.ssl

Этот абстрактный класс определяет интерфейс поставщика сервисов для класса TrustManagerFactory. Его нужно реализовать в поставщиках системы безопасности, но в обычных приложениях он не требуется.

```
public abstract class TrustManagerFactorySpi {
// Открытые конструкторы
    public TrustManagerFactorySpi();
// Защищенные методы экземпляра
    protected abstract TrustManager[] engineGetTrustManagers();
    protected abstract void engineInit(ManagerFactoryParameters spec)
        throws java.security.InvalidAlgorithmParameterException;
    protected abstract void engineInit(java.security.KeyStore ks) throws java.security.KeyStoreException;
}
```

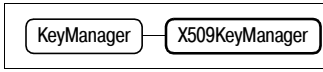
Передается методам: TrustManagerFactory.TrustManagerFactory()s

X509KeyManager Java 1.4

javax.net.ssl

Этот интерфейс является потомком KeyManager и предназначен для работы с сертификатами типа X.509. Объект X509KeyManager применяется во время процедуры «рукопожатия» SSL стороной, предоставляющей сертификат X.509 удаленному хосту для аутентификации. Обычно это выполняет сервер соединения SSL, однако в редких случаях данное действие может выполнить клиент. Для получения объекта X509KeyManager нужно реализовать этот интерфейс в собственном классе либо применить объект-фабрику KeyManagerFactory, созданный с указанием алгоритма «SunX509». В приложениях методы объекта KeyManagerFactory не вызываются явно. Вместо этого соответствующий объект X509KeyManager передается контексту SSLContext, отвечающему за установление соединений SSL. Когда системе нужно предоставить информацию аутентификации во время «рукопожатия» SSL, применяются различные методы менеджера ключей.

Объект X509KeyManager берет ключи и цепочки сертификатов из объекта KeyStore, который был передан методу init() фабрики KeyManagerFactory, создавшей данный объект X509KeyManager. Методы getPrivateKey() и getCertificateChain() возвращают закрытый ключ и цепочку сертификатов указанного псевдонима. Другие методы получают список всех псевдонимов хранилища ключей или выбирают псевдоним, соответствующий указанному типу ключей и центру сертификации. Таким образом, объект X509KeyManager может выбирать цепочку сертификатов (и соответствующий ключ) на основе тех типов ключей и центров сертификации, которые может принимать удаленный хост.



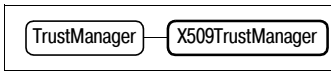

```
public interface X509KeyManager extends KeyManager {
// Открытые методы экземпляра
   public abstract String chooseClientAlias(String[] keyType, java.security.Principal[] issuers,
                                                              java.net.Socket socket);
   public abstract String chooseServerAlias(String keyType, java.security.Principal[] issuers,
                                                              java.net.Socket socket);
   public abstract java.security.cert.X509Certificate[] getCertificateChain(String alias);
   public abstract String[] getClientAliases(String keyType, java.security.Principal[] issuers);
   public abstract java.security.PrivateKey getPrivateKey(String alias);
   public abstract String[] getServerAliases(String keyType, java.security.Principal[] issuers);
}
```

X509TrustManager

Java 1.4

javax.net.ssl

Этот интерфейс является потомком TrustManager и предназначен для работы с сертификатами X.509. Менеджеры доверенностей применяются на этапе «рукопожатия» SSL для определения надежности верительных данных, предоставленных удаленным хостом. Эта проверка обычно выполняется клиентом соединения SSL, но может выполняться и на сервере. Для получения объекта X509TrustManager нужно реализовать его в собственном классе либо применить фабрику TrustManagerFactory, созданную с указанием алгоритма «SunX509». В приложениях методы этого интерфейса явно не вызываются; вместо этого объект X509TrustManager передается объекту SSLContext, отвечающему за установление соединений SSL. Если системе нужно определить, являются ли верительные данные, предоставленные удаленным хостом, надежными, то вызываются методы этого менеджера доверенностей.




```
public interface X509TrustManager extends TrustManager {
// Открытые методы экземпляра
   public abstract void checkClientTrusted(java.security.cert.X509Certificate[] chain,
        String authType) throws java.security.cert.CertificateException;
   public abstract void checkServerTrusted(java.security.cert.X509Certificate[] chain,
        String authType) throws java.security.cert.CertificateException;
   public abstract java.security.cert.X509Certificate[] getAcceptedIssuers();
}
```

Глава 20

javax.security.auth и подпакеты

В этой главе описывается пакет `javax.security.auth` и его подпакеты, формирующие службу аутентификации и авторизации Java – JAAS (Java Authentication and Authorization Service). До того как JAAS была интегрирована в Java 1.4, она являлась стандартным расширением, поэтому в названиях пакетов стоит префикс «javax». В книге рассматриваются следующие пакеты:

`javax.security.auth`

Пакет верхнего уровня, в котором определен класс `Subject` – главный класс JAAS.

`javax.security.auth.callback`

В этом пакете определены API обратного вызова для связи между низкоуровневым модулем входа в систему (login module) и конечным пользователем (например, передача имени пользователя и пароля).

`javax.security.auth.kerberos`

В этом пакете содержатся классы JAAS, связанные с протоколом сетевой аутентификации Kerberos.

`javax.security.auth.login`

В этом пакете определен класс `LoginContext` и другие классы, используемые в приложениях для организации процедуры входа JAAS.

`javax.security.auth.spi`

В этом пакете определен интерфейс поставщика сервисов для JAAS.

`javax.security.auth.x500`

В этом пакете определены классы JAAS принципалов (principals) X.500.

Пакет javax.security.auth

Java 1.4

Это пакет верхнего уровня службы аутентификации и авторизации Java (JAAS). Главный класс данного пакета – `Subject` – представляет пользователя, прошедшего аутентификацию. В этом классе определены статические методы, позволяющие запускать Java-программы от имени указанного в объекте `Subject` пользователя (то есть используя права этого пользователя). Остальные классы и интерфейсы этого пакета также являются важной частью инфраструктуры JAAS, но не используются в обычных приложениях. Как правило, в приложениях объект `Subject` не создается явно, а

получается из объекта javax.security.auth.login.LoginContext, созданного с помощью javax.security.auth.callback.Callback-Handler.

Интерфейсы

```
public interface Destroyable;
public interface Refreshable;
```

Классы

```
public final class AuthPermission extends java.security.BasicPermission;
public abstract class Policy;
public final class PrivateCredentialPermission extends java.security.Permission;
public final class Subject implements Serializable;
public class SubjectDomainCombiner implements java.security.DomainCombiner;
```

Исключения

```
public class DestroyFailedException extends Exception;
public class RefreshFailedException extends Exception;
```

AuthPermission Java 1.4

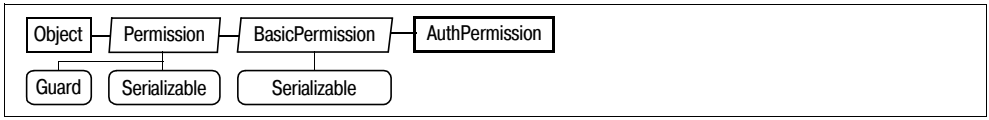
javax.security.auth *сериализуемый*

Этот потомок класса java.security.Permission контролирует использование различных методов данного пакета и подпакетов. Имя целевого объекта этого разрешения определяет, какие методы разрешено вызывать; права AuthPermission не имеют списков действий. Прикладным программистам никогда не приходится явно использовать этот класс. Системным программистам и администраторам следует ознакомиться со следующей таблицей целевых объектов и соответствующих прав:

Имя целевого объекта	Дает право на
DoAs	Вызов методов Subject.doAs()
DoAsPrivileged	Вызов методов Subject.doAsPrivileged()
GetSubject	Вызов метода Subject.getSubject()
getSubjectFromDomainCombiner	Вызов метода SubjectDomainCombiner.getSubject()
setReadOnly	Вызов метода Subject.setReadOnly()
modifyPrincipals	Изменение множества (Set) принципалов, связанных с объектом Subject
modifyPublicCredentials	Изменение множества (Set) открытых (public) верительных данных, связанных с объектом Subject
modifyPrivateCredentials	Изменение множества (Set) закрытых верительных данных, связанных с объектом Subject
refreshCredential	Вызов метода refresh() класса, представляющего верительные данные и реализующего интерфейс Refreshable
destroyCredential	Вызов метода destroy() класса, представляющего верительные данные и реализующего интерфейс Destroyable
createLoginContext.name	Создание экземпляра класса LoginContext с указанным именем. Если вместо имени стоит «*», то разрешается создавать LoginContext с любым именем

(продолжение)

Имя целевого объекта	Дает право на
getLoginConfiguration	Вызов метода getConfiguration() класса javax.security.auth.login.Configuration
setLoginConfiguration	Вызов метода setConfiguration() класса javax.security.auth.login.Configuration
refreshLoginConfiguration	Вызов метода refresh() класса javax.security.auth.login.Configuration




```
public final class AuthPermission extends java.security.BasicPermission {
// Открытые конструкторы
   public AuthPermission(String name);
   public AuthPermission(String name, String actions);
}
```

Destroyable

Java 1.4

javax.security.auth

Классы, содержащие конфиденциальную информацию, например верительные данные, могут реализовать этот интерфейс, содержащий методы для разрушения или удаления данной информации. Метод destroy() удаляет эту информацию. Он может вызвать исключение DestroyFailedException, если по какой-нибудь причине информация не может быть удалена. Этот метод может сгенерировать исключение SecurityException, если объект, вызывавший этот метод, не имеет соответствующих прав. После вызова метода destroy() метод isDestroyed() возвращает true, а все методы данного объекта будут вызывать исключение IllegalStateException.

```
public interface Destroyable {
// Открытые методы экземпляра
   public abstract void destroy() throws DestroyFailedException;
   public abstract boolean isDestroyed();
}
```

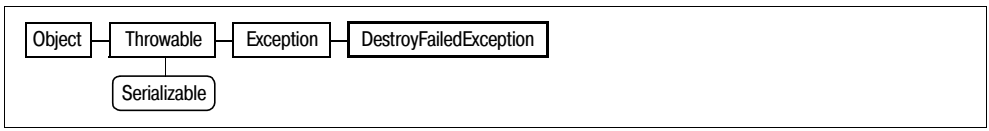
Реализации: javax.security.auth.kerberos.KerberosKey, javax.security.auth.kerberos.KerberosTicket, javax.security.auth.x500.X500PrivateCredential

DestroyFailedException

Java 1.4

javax.security.auth ***сериализуемое, проверяемое***

Это исключение сообщает о том, что метод destroy() объекта Destroyable не завершился успешно.



```
public class DestroyFailedException extends Exception {
// Открытые конструкторы
   public DestroyFailedException();
   public DestroyFailedException(String msg);
}
```

Генерируется методами: Destroyable.destroy(), javax.security.auth.kerberos.KerberosKey.destroy(), javax.security.auth.kerberos.KerberosTicket.destroy()

Policy

Java 1.4; устарел в Java 1.4

javax.security.auth

Этот устаревший (deprecated) класс представляет политику безопасности на основе Subject. Раньше службы JAAS API (данный пакет и его подпакеты) были расширением платформы Java, а этот класс дополнял класс java.security.Policy, который до версии 1.4 не поддерживал авторизацию на основе Subject. В Java 1.4 java.security.Policy поддерживает политики безопасности, основанные на понятиях источника кода (code origin), подписчиков кода (code signer) и субъектов (subjects). Поэтому данный класс больше не нужен и признан устаревшим.

```
public abstract class Policy {
// Защищенные конструкторы
   protected Policy();
// Открытые методы класса
   public static javax.security.auth.Policy getPolicy();
   public static void setPolicy(javax.security.auth.Policy policy);
// Открытые методы экземпляра
   public abstract java.security.PermissionCollection getPermissions(Subject subject,
        java.security.CodeSource cs);
   public abstract void refresh();
}
```

Передается методам: javax.security.auth.Policy.setPolicy()

Возвращается методами: javax.security.auth.Policy.getPolicy()

PrivateCredentialPermission

Java 1.4

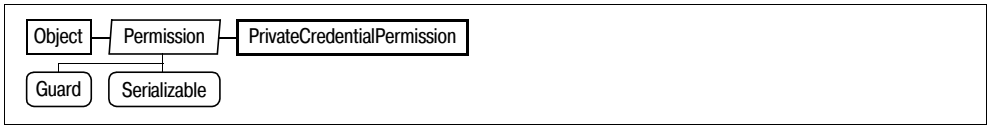
javax.security.auth ***сериализуемый***

Этот потомок класса Permission управляет доступом к частным верительным данным, которые принадлежат объекту Subject (указаны в виде множества объектов Principal). В приложениях этот класс используется редко. Этот класс может потребоваться системным программистам при реализации новых классов, представляющих верительные данные. Системные администраторы, которые настраивают системные политики, должны ознакомиться с этим классом.

Единственное действие права PrivateCredentialPermission — это «read». Имя целевого объекта для этого права имеет сложный синтаксис и определяет имя класса верительных данных и список принципалов. Последние указываются в виде имени класса Principal, за которым следует имя принципала в одинарных кавычках. Например, следующие строки, помещенные в файл политики безопасности, дают разрешение на чтение частных верительных данных KerberosKey принципала KerberosPrincipal с именем «david»:

```
    permission javax.security.auth.PrivateCredentialPermission
       "javax.security.auth.kerberos.KerberosKey javax.security.auth.kerberos.KerberosPrincipal
       \e"david\e"", "read";
```

Синтаксис имени целевого объекта позволяет использовать групповой символ «*» вместо имени класса `Principal` и имени принципала.




```
public final class PrivateCredentialPermission extends java.security.Permission {
// Открытые конструкторы
   public PrivateCredentialPermission(String name, String actions);
// Открытые методы экземпляра
   public String getCredentialClass();
   public String[][] getPrincipals();
// Открытые методы, замещающие Permission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
   public boolean implies(java.security.Permission p);
   public java.security.PermissionCollection newPermissionCollection();              // константа
}
```

Refreshable

Java 1.4

java.security.auth

Если класс реализует этот интерфейс, то его экземпляры имеют ограниченный срок действия (как у некоторых типов верительных данных), и их нужно периодически «обновлять» для продления срока действия. Метод `isCurrent()` возвращает `true`, если объект действителен в данный момент, и `false`, если срок действия истек, а объект требуется обновить. Метод `refresh()` предпринимает попытку сделать объект действительным или продлить срок его действия. Если эта попытка не удастся, метод вызовет исключение `RefreshFailedException` (может также возникнуть исключение `SecurityException`, если объект, вызывающий этот метод, не имеет необходимых прав).

```
public interface Refreshable {
// Открытые методы экземпляра
   public abstract boolean isCurrent();
   public abstract void refresh() throws RefreshFailedException;
}
```

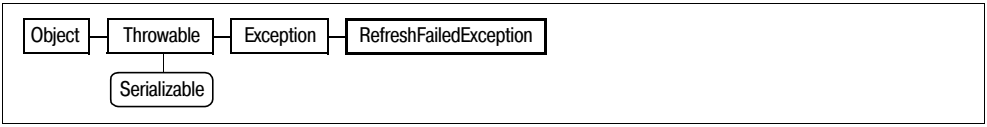
Реализации: `javax.security.auth.kerberos.KerberosTicket`

RefreshFailedException

Java 1.4

javax.security.auth ***сериализуемое, проверяемое***

Это исключение сообщает о том, что метод `refresh()` объекта `Refreshable` не завершился успешно.




```
public class RefreshFailedException extends Exception {
// Открытые конструкторы
   public RefreshFailedException();
   public RefreshFailedException(String msg);
}
```

Генерируется методами: `Refreshable.refresh()`, `javax.security.auth.kerberos.KerberosTicket.refresh()`

Subject

Java 1.4

javax.security.auth ***сериализуемый***

Класс `Subject` представляет главное понятие JAAS API – субъект. Субъект имеет следующие составляющие:

- Множество `java.util.Set` объектов `Principal`, определяющих личность (личности) субъекта `Subject` (identities).
- Множество `Set` объектов, определяющих публичные верительные данные, например публичные сертификаты ключей субъекта.
- Множество `Set` объектов, определяющих частные верительные данные, например закрытые ключи или билеты `Kerberos` субъекта `Subject`.

В классе `Subject` определены методы, позволяющие получить все три множества или их подмножества, которые содержат объекты только указанного класса. Хотя объект `Subject` создается только для чтения, все три множества можно изменять методами класса `java.util.Set`. После вызова метода `setReadOnly()` эти множества становятся неизменяемыми.

Как правило, в обычных приложениях объекты `Subject` не создаются. Вместо этого с помощью методов `login()` и `getSubject()` объекта `javax.security.auth.login.LoginContext` приложения получают объект `Subject`, представляющий аутентифицированного пользователя.

После того как `Subject` получен из объекта `LoginContext`, можно вызвать метод `doAs()` для выполнения программного кода с использованием прав, данных этому субъекту (`Subject`), и прав, которые имеет сам код. Метод `doAs()` выполняет код, определенный в методе `run()` объекта `PrivilegedAction` или `PrivilegedExceptionAction`. Метод `doAsPrivileged()` аналогичен `doAs()`, но выполняет указанный метод `run()` с использованием прав субъекта. При этом на него не налагаются ограничения непривилегированного кода из стека вызовов.

Обратите внимание, что многие методы этого класса генерируют исключение `SecurityException`, если объект, вызывающий метод, не получил необходимое разрешение `AuthPermission`.

```
public final class Subject implements Serializable {
// Открытые конструкторы
   public Subject();
   public Subject(boolean readOnly, java.util.Set principals, java.util.Set pubCredentials,
                                                              java.util.Set privCredentials);
// Открытые методы класса
   public static Object doAs(Subject subject, java.security.PrivilegedExceptionAction action)
        throws java.security.PrivilegedActionException;
   public static Object doAs(Subject subject, java.security.PrivilegedAction action);
   public static Object doAsPrivileged(Subject subject,
        java.security.PrivilegedExceptionAction action, java.security.AccessControlContext acc)
        throws java.security.PrivilegedActionException;
   public static Object doAsPrivileged(Subject subject, java.security.PrivilegedAction action,
                                                        java.security.AccessControlContext acc);
   public static Subject getSubject(java.security.AccessControlContext acc);
// Методы доступа к свойствам (по имени свойства)
   public java.util.Set getPrincipals();
   public java.util.Set getPrincipals(Class c);
   public java.util.Set getPrivateCredentials();
   public java.util.Set getPrivateCredentials(Class c);
   public java.util.Set getPublicCredentials();
   public java.util.Set getPublicCredentials(Class c);
   public boolean isReadOnly();                                          // по умолчанию: false
// Открытые методы экземпляра
   public void setReadOnly();
// Открытые методы, замещающие Object
   public boolean equals(Object o);
   public int hashCode();
   public String toString();
}
```

Передается методам: javax.security.auth.Policy.getPermissions(), Subject.{doAs(), doAsPrivileged()}, SubjectDomainCombiner.SubjectDomainCombiner(), javax.security.auth.login.LoginContext.LoginContext(), javax.security.auth.spi.LoginModule.initialize()

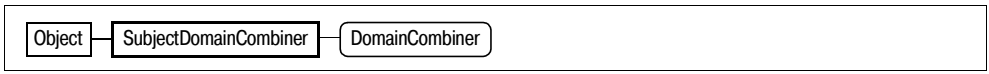
Возвращается методами: Subject.getSubject(), SubjectDomainCombiner.getSubject(), javax.security.auth.login.LoginContext.getSubject()

SubjectDomainCombiner

Java 1.4

javax.security.auth

Этот класс реализует интерфейс DomainCombiner и применяется для объединения прав источника программного кода и его подписчиков с правами, данными указанному объекту Subject. SubjectDomainCombiner создается методами Subject.doAs() и Subject.doAsPrivileged() объекта AccessControlContext и используется этим объектом.




```
public class SubjectDomainCombiner implements java.security.DomainCombiner {
// Открытые конструкторы
   public SubjectDomainCombiner(Subject subject);
// Открытые методы экземпляра
   public Subject getSubject();
```

```
// Методы, реализующие DomainCombiner
   public java.security.ProtectionDomain[] combine(java.security.ProtectionDomain[] currentDomains,
                                                   java.security.ProtectionDomain[] assignedDomains);
}
```

Пакет javax.security.auth.callback Java 1.4

Этот пакет содержит механизм, позволяющий низкоуровневому объекту javax.security.auth.spi.LoginModule взаимодействовать с конечным пользователем приложения с целью получения имени, пароля и другой информации, необходимой для аутентификации. Объект LoginModule посылает сообщения и запросы информации в виде объектов, реализующих интерфейс Callback. Приложение, которое аутентифицирует пользователя, должно предоставить (через javax.security.auth.login.LoginContext) объект CallbackHandler, создающий на основе объектов Callback текстовое сообщение или окно GUI для диалога с пользователем. Чтобы приложение могло предоставить альтернативный интерфейс входа (login interface), нужно создать собственный класс, реализующий CallbackHandler. Интерфейс CallbackHandler содержит только один метод, но реализация этого метода может быть достаточно объемной. Указания по реализации этого интерфейса представлены в описании классов Callback.

В комплект J2SE SDK для Java 1.4 фирмы Sun входят две реализации интерфейса CallbackHandler. Обе находятся в пакете com.sun.security.auth.callback, хотя не гарантируется, что эти классы содержатся во всех дистрибутивах Java. В приложениях, работающих в текстовом режиме, можно использовать TextCallbackHandler, а в приложениях с графическим интерфейсом – DialogCallbackHandler. Программистам, пишущим собственные реализации CallbackHandler, полезно изучить исходный код этих реализаций.

Интерфейсы

```
public interface Callback;
public interface CallbackHandler;
```

Классы

```
public class ChoiceCallback implements Callback, Serializable;
public class ConfirmationCallback implements Callback, Serializable;
public class LanguageCallback implements Callback, Serializable;
public class NameCallback implements Callback, Serializable;
public class PasswordCallback implements Callback, Serializable;
public class TextInputCallback implements Callback, Serializable;
public class TextOutputCallback implements Callback, Serializable;
```

Исключение

```
public class UnsupportedCallbackException extends Exception;
```

Callback Java 1.4

javax.security.auth.callback

Этот интерфейс не содержит методов, а служит интерфейсом-маркером, определяющим объекты, которые можно передать методу handle() объекта CallbackHandler. За исключением UnsupportedCallbackException, все классы данного пакета реализуют этот интерфейс.

```
public interface Callback {
}
```

Реализации: ChoiceCallback, ConfirmationCallback, LanguageCallback, NameCallback, PasswordCallback, TextInputCallback, TextOutputCallback

Передается методам: CallbackHandler.handle(), UnsupportedCallbackException.UnsupportedCallbackException()

Возвращается методами: UnsupportedCallbackException.getCallback()

CallbackHandler

Java 1.4

javax.security.auth.callback

CallbackHandler отвечает за диалог между конечным пользователем и объектом javax.security.auth.spi.LoginModule, который производит аутентификацию пользователя в контексте javax.security.auth.login.LoginContext приложения. Когда нужно аутентифицировать пользователя, создается контекст LoginContext и указывается объект CallbackHandler для этого контекста. Соответствующий объект LoginModule использует CallbackHandler для диалога с конечным пользователем – например, предлагая ему ввести имя и пароль.

LoginModule передает массив объектов, реализующих интерфейс Callback, методу handle() объекта CallbackHandler. Этот метод должен определить тип объекта Callback, отобразить содержащуюся в нем информацию и приглашение пользователю. Различные классы Callback имеют разное назначение и, соответственно, должны обрабатываться по-разному. Чаще всего применяются NameCallback и PasswordCallback: они представляют соответственно имя пользователя и пароль. Кроме того, часто используется TextOutputCallback, представляющий запрос на отображение сообщения пользователю (например, «Аутентификация не выполнена»). О том, как обрабатывать конкретные реализации Callback, рассказано в их описании. От классов, реализующих CallbackHandler, не требуется поддержка всех типов Callback, поэтому они могут вызывать исключение UnsupportedCallbackException, если переданный объект Callback не поддерживается.

Методу handle() передается массив объектов Callback. В данный момент времени обработчик CallbackHandler (например, консольный) может обрабатывать один объект Callback, последовательно выводя различную информацию, приглашения пользователю и возвращая введенные им данные. Или он может обработать все объекты Callback и вывести соответствующую информацию в одном диалоговом окне входа (при использовании графического интерфейса). Конечно, в реализациях LoginModule можно вызывать метод handle() несколько раз. Наконец, необходимо заметить, что если объекту CallbackHandler известно о пользователе из какого-нибудь источника, он может обрабатывать некоторые объекты Callback автоматически (например, автоматически передавать имя пользователя в NameCallback, если оно известно заранее).

В некоторых реализациях Java можно зарегистрировать объект CallbackHandler по умолчанию, указав в свойстве безопасности auth.login.defaultCallbackHandler имя соответствующей реализации. По умолчанию в политиках безопасности Java 1.4 фирмы Sun это свойство не определено. В Java 1.4 SDK фирмы Sun включены реализации CallbackHandler, осуществляющие диалог с пользователем в консольном режиме и в графической среде – классы TextCallbackHandler и DialogCallbackHandler пакета com.sun.security.auth.callback. Однако нужно отметить, что эти классы входят только в реализацию фирмы Sun и не являются частью спецификации Java; они могут отсутствовать в некоторых версиях.

```
public interface CallbackHandler {
// Открытые методы экземпляра
   public abstract void handle(Callback[] callbacks) throws java.io.IOException,
      UnsupportedCallbackException;
}
```

Передается методам: javax.security.auth.login.LoginContext.LoginContext(), javax.security.auth.spi.LoginModule.initialize()

ChoiceCallback Java 1.4

javax.security.auth.callback ***сериализуемый***

Этот класс, реализующий Callback, представляет собой запрос на отображение списка опций в текстовом режиме и позволяет пользователю выбрать несколько опций. CallbackHandler должен отображать текст приглашения пользователю, возвращаемый методом getPrompt(), а также строки, возвращаемые методами getChoices(). Если метод allowMultipleSelections() возвращает true, то пользователь может выбрать одну или несколько опций или не выбрать ничего; в противном случае пользователю разрешается выбрать только одну опцию. В любом случае объект CallbackHandler может вызвать метод getDefaultChoice() и назначить опцию с полученным индексом опцией по умолчанию. После того как пользователь сделал выбор, CallbackHandler должен передать индекс единственной выбранной опции методу setSelectedIndex() или индексы нескольких выбранных опций методу setSelectedIndexes().


```
public class ChoiceCallback implements Callback, Serializable {
// Открытые конструкторы
   public ChoiceCallback(String prompt, String[] choices, int defaultChoice,
        boolean multipleSelectionsAllowed);
// Методы доступа к свойствам (по имени свойства)
   public String[] getChoices();
   public int getDefaultChoice();
   public String getPrompt();
   public int[] getSelectedIndexes();
   public void setSelectedIndexes(int[] selections);
// Открытые методы экземпляра
   public boolean allowMultipleSelections();
   public void setSelectedIndex(int selection);
}
```

ConfirmationCallback Java 1.4

javax.security.auth.callback ***сериализуемый***

Этот класс типа Callback представляет собой вопрос пользователю типа «да/нет» или вопрос, предполагающий несколько вариантов ответа. Сначала обработчик CallbackHandler должен вызвать метод getPrompt(), чтобы получить текст вопроса, а затем вызвать метод getMessageType(), чтобы определить тип сообщения (INFORMATION, WARNING или ERROR) и представить запрос пользователю в соответствии с этим типом.

Далее обработчик должен определить соответствующий набор ответов на этот вопрос с помощью метода `getOptionType()`. Могут возвращаться следующие значения:

`YES_NO_OPTION`

Возможные ответы: «yes» и «no» (или соответствующие локализованные строки).

`YES_NO_CANCEL_OPTION`

Возможные ответы: «yes», «no» и «cancel» (или локализованные эквиваленты).

`OK_CANCEL_OPTION`

`CallbackHandler` должен предоставить возможность выбора между «ok» и «cancel» (или соответствующих локализованных строк).

`UNSPECIFIED_OPTION`

`CallbackHandler` должен вызвать метод `getOptions()` и использовать в качестве вариантов ответов возвращаемые строки.

Во всех этих случаях метод `CallbackHandler` должен вызвать `getDefaultOption()`, чтобы определить вариант, который будет помечен как ответ по умолчанию. Если `getOptionType()` возвращает `UNSPECIFIED_OPTION`, тогда метод `getDefaultOption()` возвратит индекс массива опций, возвращаемого методом `getOptions()`. В противном случае `getDefaultOption()` возвращает одну из констант `YES`, `NO`, `OK` или `CANCEL`.

Выбранный пользователем ответ на вопрос `CallbackHandler` должен передать методу `setSelectedIndex()`. Возвращаемое значение должно быть одной из констант `YES`, `NO`, `OK` или `CANCEL` либо индексом массива опций, возвращаемого методом `getOptions()`.


```
public class ConfirmationCallback implements Callback, Serializable {
// Открытые конструкторы
   public ConfirmationCallback(int messageType, String[] options, int defaultOption);
   public ConfirmationCallback(int messageType, int optionType, int defaultOption);
   public ConfirmationCallback(String prompt, int messageType, String[] options, int defaultOption);
   public ConfirmationCallback(String prompt, int messageType, int optionType, int defaultOption);
// Открытые константы
   public static final int CANCEL;                                                    // =2
   public static final int ERROR;                                                     // =2
   public static final int INFORMATION;                                               // =0
   public static final int NO;                                                        // =1
   public static final int OK;                                                        // =3
   public static final int OK_CANCEL_OPTION;                                          // =2
   public static final int UNSPECIFIED_OPTION;                                        // =-1
   public static final int WARNING;                                                   // =1
   public static final int YES;                                                       // =0
   public static final int YES_NO_CANCEL_OPTION;                                      // =1
   public static final int YES_NO_OPTION;                                             // =0
// Методы доступа к свойствам (по имени свойства)
   public int getDefaultOption();
   public int getMessageType();
   public String[] getOptions();
   public int getOptionType();
   public String getPrompt();
   public int getSelectedIndex();
```

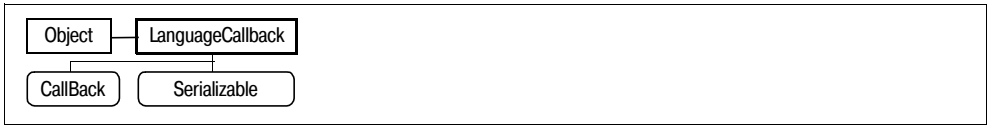
```
   public void setSelectedIndex(int selection);
}
```

LanguageCallback

Java 1.4

javax.security.auth.callback ***сериализуемый***

Этот класс, реализующий Callback, представляет выбор языка пользователя (в виде объекта Locale), который может применяться объектом LoginModule для локализации текстов приглашения пользователю и сообщений об ошибках из других объектов Callback. Если обработчику CallbackHandler уже известно, какой язык нужно использовать, он может просто передать соответствующий объект Locale методу setLocale().



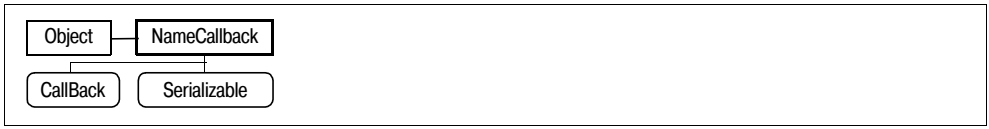

```
public class LanguageCallback implements Callback, Serializable {
// Открытые конструкторы
   public LanguageCallback();
// Открытые методы экземпляра
   public java.util.Locale getLocale();                          // по умолчанию:null
   public void setLocale(java.util.Locale locale);
}
```

NameCallback

Java 1.4

javax.security.auth.callback ***сериализуемый***

Этот класс типа Callback представляет запрос имени пользователя или другой текстовой информации для аутентификации пользователя. Интерактивный обработчик CallbackHandler должен вызвать методы getPrompt() и getDefaultName() и вывести возвращенный ими текст приглашения и (необязательно) имя пользователя по умолчанию. После того как пользователь ввел имя (или подтвердил имя по умолчанию), обработчик должен передать это имя методу setName().



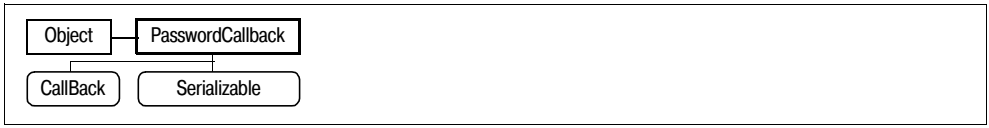

```
public class NameCallback implements Callback, Serializable {
// Открытые конструкторы
   public NameCallback(String prompt);
   public NameCallback(String prompt, String defaultName);
// Открытые методы экземпляра
   public String getDefaultName();
   public String getName();
   public String getPrompt();
   public void setName(String name);
}
```

PasswordCallback

Java 1.4

javax.security.auth.callback ***сериализуемый***

Этот класс типа `Callback` представляет запрос пароля. Объект `CallbackHandler` должен вывести приглашение пользователю, возвращаемое методом `getPrompt()`, а затем предоставить возможность ввести пароль. После того как пароль введен, его нужно передать методу `setPassword()`. Если `isEchoOn()` возвращает `true`, то обработчик должен отображать пароль, вводимый пользователем.



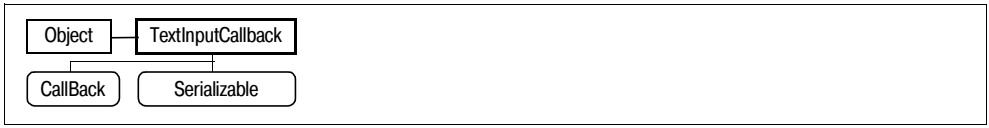

```
public class PasswordCallback implements Callback, Serializable {
// Открытые конструкторы
   public PasswordCallback(String prompt, boolean echoOn);
// Открытые методы экземпляра
   public void clearPassword();
   public char[] getPassword();
   public String getPrompt();
   public boolean isEchoOn();
   public void setPassword(char[] password);
}
```

TextInputCallback

Java 1.4

javax.security.auth.callback ***сериализуемый***

Данный класс, реализующий `Callback`, представляет приглашение пользователю ввести текст; это обобщенная версия класса `NameCallback`. Обработчик `CallbackHandler` должен вызвать метод `getPrompt()` и отобразить возвращенный текст приглашения пользователю. Затем он должен предоставить пользователю возможность ввести текст или выбрать текст по умолчанию, возвращаемый методом `getDefaultText()`. После того как пользователь ввел текст (или выбрал текст по умолчанию), обработчик должен передать этот текст методу `setText()`.




```
public class TextInputCallback implements Callback, Serializable {
// Открытые конструкторы
   public TextInputCallback(String prompt);
   public TextInputCallback(String prompt, String defaultText);
// Открытые методы экземпляра
   public String getDefaultText();
   public String getPrompt();
   public String getText();
   public void setText(String text);
}
```

TextOutputCallback Java 1.4

javax.security.auth.callback *сериализуемый*

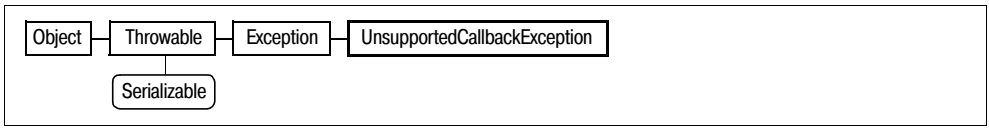
Этот тип объекта Callback представляет запрос на вывод текста. Обработчик запросов CallbackHandler должен вызвать метод getMessage() и отобразить возвращенную строку. Кроме того, он должен вызвать метод getMessageType() и использовать возвращенное значение (одну из констант, определенных в этом классе) для указания типа отображаемой информации.


```
public class TextOutputCallback implements Callback, Serializable {
// Открытые конструкторы
   public TextOutputCallback(int messageType, String message);
// Открытые константы
   public static final int ERROR;                                    // =2
   public static final int INFORMATION;                              // =0
   public static final int WARNING;                                  // =1
// Открытые методы экземпляра
   public String getMessage();
   public int getMessageType();
}
```

UnsupportedCallbackException Java 1.4

javax.security.auth.callback *сериализуемое, проверяемое*

В реализациях интерфейса CallbackHandler метод handle() может вызывать исключения данного типа, если объект Callback, переданный этому методу, не поддерживается или не распознается. Заметьте, что такой объект Callback нужно передать конструктору данного исключения.




```
public class UnsupportedCallbackException extends Exception {
// Открытые конструкторы
   public UnsupportedCallbackException(Callback callback);
   public UnsupportedCallbackException(Callback callback, String msg);
// Открытые методы экземпляра
   public Callback getCallback();
}
```

Генерируется методами: CallbackHandler.handle()

Пакет javax.security.auth.kerberos Java 1.4

В данном пакете определены классы, поддерживающие защищенный протокол сетевой аутентификации Kerberos. Эти классы представляют интерес для системных программистов, реализующих интерфейс `javax.security.auth.spi.LoginModule` на основе Kerberos. При разработке приложений с поддежкой Kerberos следует применять пакет `org.ietf.jgss`. Полное описание протокола Kerberos выходит за рамки данной книги. В дальнейшем предполагается, что читатель знаком с этим протоколом.

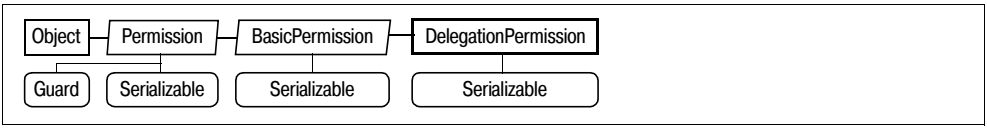
Классы

```
public final class DelegationPermission extends java.security.BasicPermission implements Serializable;
public class KerberosKey implements javax.security.auth.Destroyable, javax.crypto.SecretKey;
public final class KerberosPrincipal implements java.security.Principal, Serializable;
public class KerberosTicket implements javax.security.auth.Destroyable,
                                       javax.security.auth.Refreshable, Serializable;
public final class ServicePermission extends java.security.Permission implements Serializable;
```

DelegationPermission Java 1.4

javax.security.auth.kerberos ***сериализуемый***

Это класс, наследующий `java.security.Permission`, управляет передачей билетов Kerberos от принципала к сервису Kerberos от имени оригинального принципала. Имя целевого объекта в разрешении `DelegationPermission` состоит из имен принципалов двух сервисов Kerberos. Первое имя определяет сервис, которому билет передается, а второе – сервис, который будет использоваться первым сервисом от имени оригинального принципала Kerberos.



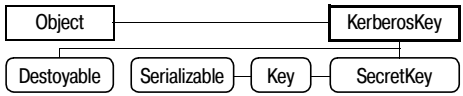

```
public final class DelegationPermission extends java.security.BasicPermission implements Serializable {
// Открытые конструкторы
   public DelegationPermission(String principals);
   public DelegationPermission(String principals, String actions);
// Открытые методы, замещающие BasicPermission
   public boolean equals(Object obj);
   public int hashCode();
   public boolean implies(java.security.Permission p);
   public java.security.PermissionCollection newPermissionCollection();
}
```

KerberosKey Java 1.4

javax.security.auth.kerberos ***сериализуемый***

Этот класс является реализацией интерфейса `javax.crypto.SecretKey` и представляет секретный ключ принципала Kerberos. Объект, реализующий `javax.security.auth.spi.LoginModule` на основе Kerberos, создает экземпляр класса `KerberosKey` и сохра-

няет его во множестве частных верительных данных аутентифицированного субъекта (объект Subject), созданного этим объектом.



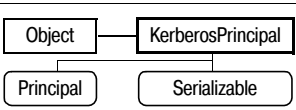

```
public class KerberosKey implements javax.security.auth.Destroyable,javax.crypto.SecretKey {
// Открытые конструкторы
   public KerberosKey(KerberosPrincipal principal, char[] password, String algorithm);
   public KerberosKey(KerberosPrincipal principal, byte[] keyBytes, int keyType, int versionNum);
// Открытые методы экземпляра
   public final int getKeyType();
   public final KerberosPrincipal getPrincipal();
   public final int getVersionNumber();
// Методы, реализующие Destroyable
   public void destroy() throws javax.security.auth.DestroyFailedException;
   public boolean isDestroyed();
// Методы, реализующие Key
   public final String getAlgorithm();
   public final byte[] getEncoded();
   public final String getFormat();
// Открытые методы, замещающие Object
   public String toString();
}
```

KerberosPrincipal

Java 1.4

javax.security.auth.kerberos ***сериализуемый***

Этот класс представляет принципал Kerberos с заданным именем. Кроме того, это имя может содержать область (realm). Если область не указана, применяется область по умолчанию, указанная в конфигурационном файле *krb5.conf* или в системном свойстве java.security.krb5.realm.




```
public final class KerberosPrincipal implements java.security.Principal, Serializable {
// Открытые конструкторы
   public KerberosPrincipal(String name);
   public KerberosPrincipal(String name, int nameType);
// Открытые константы
   public static final int KRB_NT_PRINCIPAL;                                    // =1
   public static final int KRB_NT_SRV_HST;                                      // =3
   public static final int KRB_NT_SRV_INST;                                     // =2
   public static final int KRB_NT_SRV_XHST;                                     // =4
   public static final int KRB_NT_UID;                                          // =5
   public static final int KRB_NT_UNKNOWN;                                      // =0
// Открытые методы экземпляра
   public int getNameType();
```

```
   public String getRealm();
// Методы, реализующие Principal
   public boolean equals(Object other);
   public String getName();
   public int hashCode();
   public String toString();
}
```

Передается методам: KerberosKey.KerberosKey(), KerberosTicket.KerberosTicket()

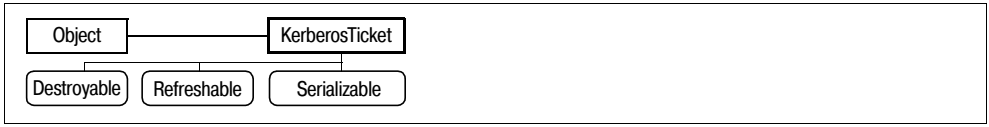
Возвращается методами: KerberosKey.getPrincipal(), KerberosTicket.{getClient(), getServer()}

KerberosTicket

Java 1.4

javax.security.auth.kerberos ***сериализуемый***

Этот класс представляет билет Kerberos – верительные данные, применяемые для аутентификации принципала Kerberos в сетевом сервисе. Объект, реализующий интерфейс javax.security.auth.spi.LoginModule, создает экземпляр класса KerberosTicket и сохраняет его во множестве частных верительных данных аутентифицированного субъекта (объект Subject), созданного этим объектом.




```
public class KerberosTicket implements javax.security.auth.Destroyable,
                                       javax.security.auth.Refreshable, Serializable {
// Открытые конструкторы
   public KerberosTicket(byte[] asn1Encoding, KerberosPrincipal client, KerberosPrincipal server,
      byte[] sessionKey, int keyType, boolean[] flags, java.util.Date authTime, java.util.Date
      startTime, java.util.Date endTime, java.util.Date renewTill, java.net.InetAddress[] clientAddresses);
// Методы доступа к свойствам (по имени свойства)
   public final java.util.Date getAuthTime();
   public final KerberosPrincipal getClient();
   public final java.net.InetAddress[] getClientAddresses();
   public boolean isCurrent();                                          // Реализует:Refreshable
   public boolean isDestroyed();                                        // Реализует:Destroyable
   public final byte[] getEncoded();
   public final java.util.Date getEndTime();
   public final boolean[] getFlags();
   public final boolean isForwardable();
   public final boolean isForwarded();
   public final boolean isInitial();
   public final boolean isPostdated();
   public final boolean isProxiable();
   public final boolean isProxy();
   public final boolean isRenewable();
   public final java.util.Date getRenewTill();
   public final KerberosPrincipal getServer();
   public final javax.crypto.SecretKey getSessionKey();
   public final int getSessionKeyType();
   public final java.util.Date getStartTime();
// Методы, реализующие Destroyable
```

```
   public void destroy() throws javax.security.auth.DestroyFailedException;
   public boolean isDestroyed();
// Методы, реализующие Refreshable
   public boolean isCurrent();
   public void refresh() throws javax.security.auth.RefreshFailedException;
// Открытые методы, замещающие Object
   public String toString();
}
```

ServicePermission

Java 1.4

javax.security.auth.kerberos ***сериализуемый***

Этот класс, наследующий `java.security.Permission`, защищает доступ к билетам Kerberos, используемым при получении указанного сервиса. Именем целевого объекта в разрешении `ServicePermission` является имя принципала данного сервиса. Действием для `ServicePermission` может быть «initiate» (для клиентов) или «accept» (для серверов).


```
public final class ServicePermission extends java.security.Permission implements Serializable {
// Открытые конструкторы
   public ServicePermission(String servicePrinicipal, String action);
// Открытые методы, замещающие Permission
   public boolean equals(Object obj);
   public String getActions();
   public int hashCode();
   public boolean implies(java.security.Permission p);
   public java.security.PermissionCollection newPermissionCollection();
}
```

Пакет javax.security.auth.login

Java 1.4

В этом пакете определен класс `LoginContext`, который является одним из основных классов JAAS, используемых в приложениях. Для аутентификации пользователя в приложении нужно создать объект `LoginContext`, указав имя приложения (оно применяется при поиске типа аутентификации, который указан в объекте `Configuration` этого приложения). Обычно указывается объект `javax.security.auth.callback.CallbackHandler`, который является связующим звеном между пользователем и модулем входа. Затем приложение вызывает метод `login()` объекта `LoginContext` для осуществления самой процедуры входа. Если данный метод выполняет возврат, не вызывая исключения `LoginException`, то это означает, что пользователь успешно прошел аутентификацию. В таком случае метод `getSubject()` объекта `LoginContext` возвращает объект `javax.security.auth.Subject`, представляющий пользователя. Например:

```
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
```

```
// Получить определенный по умолчанию CallbackHandler с графическим интерфейсом
CallbackHandler h = new com.sun.security.auth.callback.DialogCallbackHandler();

// Пытаемся создать контекст LoginContext для данного приложения
LoginContext context;
try {
  context = new LoginContext("MyAppName", h);
}
catch(LoginException e) {
  System.err.println("LoginContext configuration error: " + e.getMessage());


  System.exit(-1);
}

// Теперь используем этот контекст для аутентификации пользователя
try {
  context.login();
}
catch(LoginException e) {
  System.err.println("Authentication failed: " + e.getMessage());
  System.exit(-1);  // Или можно попробовать еще раз
}

// Если мы попали сюда, значит, аутентификация пользователя прошла успешно. Получим объект
// Subject для этого пользователя.
Subject subject = context.getSubject();
```

Чтобы процедура аутентификации работала корректно, система безопасности должна быть настроена с помощью файлов настройки из каталога *jre/lib/security* (в каталоге Java) и из других каталогов. В частности, в файле настройки процедуры входа необходимо указать, какие модули входа требуются для аутентификации пользователей определенного приложения (некоторым приложениям нужны несколько модулей). Описание этого процесса выходит за рамки данной книги. О том, как предоставить конфигурационную информацию во время исполнения, рассказано в описании класса `Configuration`.

Классы

```
public class AppConfigurationEntry;
public static class AppConfigurationEntry.LoginModuleControlFlag;
public abstract class Configuration;
public class LoginContext;
```

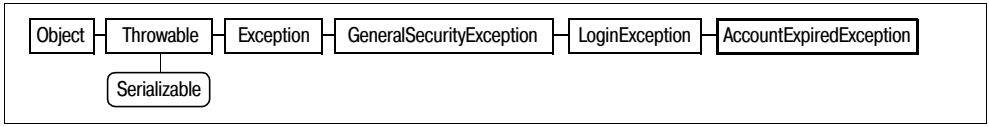
Исключения

```
public class LoginException extends java.security.GeneralSecurityException;
   └ public class AccountExpiredException extends LoginException;
   └ public class CredentialExpiredException extends LoginException;
   └ public class FailedLoginException extends LoginException;
```

AccountExpiredException Java 1.4

javax.security.auth.login ***сериализуемое, проверяемое***

Это исключение сообщает следующее: вход в программу не был выполнен из-за того, что срок действия учетной записи пользователя истек.




```
public class AccountExpiredException extends LoginException {
// Открытые конструкторы
   public AccountExpiredException();
   public AccountExpiredException(String msg);
}
```

AppConfigurationEntry Java 1.4

javax.security.auth.login

Экземпляр этого класса представляет модуль процедуры входа, который должен использоваться определенным приложением. Этот класс инкапсулирует следующую информацию: имя реализации `javax.security.auth.spi.LoginModule`, «контрольный флаг», определяющий, требуется ли обязательная аутентификация посредством этого модуля, и объект `java.util.Map`, содержащий параметры модуля в виде пар «имя/значение».

```
public class AppConfigurationEntry {
// Открытые конструкторы
   public AppConfigurationEntry(String loginModuleName,
          AppConfigurationEntry.LoginModuleControlFlag controlFlag, java.util.Map options);
// Внутренние классы
   public static class LoginModuleControlFlag;
// Открытые методы экземпляра
   public AppConfigurationEntry.LoginModuleControlFlag getControlFlag();
   public String getLoginModuleName();
   public java.util.Map getOptions();
}
```

Возвращается методами: `Configuration.getAppConfigurationEntry()`

AppConfigurationEntry.LoginModuleControlFlag Java 1.4

javax.security.auth.login

Этот внутренний класс содержит «контрольный флаг» и четыре константы для данного флага, которые определяют необходимость использования модуля входа в программу:

`REQUIRED`

Процедура аутентификации посредством данного модуля должна завершиться успешно; в противном случае весь процесс входа в программу завершится неудачно. Но если процесс аутентификации при участии данного модуля не смог завершиться, то объект `LoginContext` предпринимает попытку применять для аутентификации другие модули из списка. (В случае атаки такая мера помогает скрыть от атакующего источник возникновения ошибки при аутентификации.)

`REQUISITE`

Процедура аутентификации посредством данного модуля должна завершиться успешно; в противном случае весь процесс входа в программу завершится неудач-

но. Если процесс аутентификации при участии данного модуля не смог завершиться, то объект LoginContext не будет предпринимать попытки применять для этого другие модули.

SUFFICIENT

Процедура аутентификации с помощью этого модуля необязательна; процедура входа в программу завершится успешно, если остальные модули с флагами REQUIRED и REQUISITE успешно проведут аутентификацию пользователя. Если аутентификация посредством данного модуля выполнится успешно, то после этого объект LoginContext сразу же завершит процедуру входа в систему.

OPTIONAL

Аутентификация посредством этого модуля необязательна. В любом случае объект LoginContext будет продолжать процесс входа в систему, используя все остальные модули из списка.

```
public static class AppConfigurationEntry.LoginModuleControlFlag {
// Конструктор отсутствует
// Открытые константы
   public static final AppConfigurationEntry.LoginModuleControlFlag OPTIONAL;
   public static final AppConfigurationEntry.LoginModuleControlFlag REQUIRED;
   public static final AppConfigurationEntry.LoginModuleControlFlag REQUISITE;
   public static final AppConfigurationEntry.LoginModuleControlFlag SUFFICIENT;
// Открытые методы, замещающие Object
   public String toString();
}
```

Передается методам: AppConfigurationEntry.AppConfigurationEntry()

Возвращается методами: AppConfigurationEntry.getControlFlag()

Экземпляры:

AppConfigurationEntry.LoginModuleControlFlag.{OPTIONAL,REQUIRED, REQUISITE, SUFFICIENT}

Configuration Java 1.4

javax.security.auth.login

Этот абстрактный класс представляет файлы конфигурации системной и пользовательской процедур входа. Статический метод getConfiguration() возвращает глобальный объект Configuration, а статический метод setConfiguration() заменяет этот объект другим объектом. Вызов метода экземпляра refresh() влечет за собой повторное чтение файлов конфигурации. getAppConfigurationEntry() – ключевой метод данного класса. Он возвращает массив объектов AppConfigurationEntry, представляющих модули входа для приложения с указанным именем. Объект LoginContext использует этот класс для аутентификации пользователей указанного приложения. Как правило, при разработке приложений данный класс не требуется. Описание синтаксиса соответствующих файлов конфигурации можно найти в документации к конкретным реализациям Java.

```
public abstract class Configuration {
// Защищенные конструкторы
   protected Configuration();
// Открытые методы класса
   public static Configuration getConfiguration();                    // синхронизирован
   public static void setConfiguration(Configuration configuration);
// Открытые методы экземпляра
```

```
    public abstract AppConfigurationEntry[] getAppConfigurationEntry(String applicationName);
    public abstract void refresh();
}
```

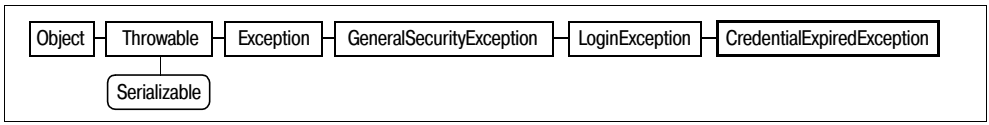
Передается методам: `Configuration.setConfiguration()`

Возвращается методами: `Configuration.getConfiguration()`

CredentialExpiredException Java 1.4

javax.security.auth.login ***сериализуемое, проверяемое***

Это исключение сообщает о том, что процедура входа не смогла завершиться из-за истечения срока действия верительных данных (например, пароля), которые больше не действительны.

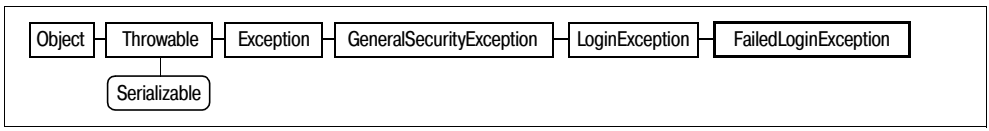



```
public class CredentialExpiredException extends LoginException {
// Открытые конструкторы
    public CredentialExpiredException();
    public CredentialExpiredException(String msg);
}
```

FailedLoginException Java 1.4

javax.security.auth.login ***сериализуемое, проверяемое***

Это исключение возникает, если процедура входа не смогла завершиться. Как правило, причиной является неверный пароль, имя пользователя или другая предоставленная информация. Модули входа в приложение, которые вызывают это исключение, могут предоставить соответствующую информацию в понятном пользователю виде с помощью метода `getMessage()`.




```
public class FailedLoginException extends LoginException {
// Открытые конструкторы
    public FailedLoginException();
    public FailedLoginException(String msg);
}
```

LoginContext Java 1.4

javax.security.auth.login

В JAAS API это один из самых важных классов для разработчиков приложений: в нем определен метод `login()` (и соответствующий метод `logout()`), с помощью которого приложение идентифицирует пользователя. Создать объект `LoginContext` можно с помо-

щью одного из открытых конструкторов. Им передается обязательный аргумент – имя приложения. Можно также передать объект javax.security.auth.Subject, который должен пройти аутентификацию, и обработчик javax.security.auth.callback.CallbackHandler, который дает возможность модулю (модулям) входа общаться с пользователем. Если объект Subject не указан, LoginContext создаст новый объект Subject. Если Subject указан, то LoginContext добавляет новую запись во множество принципалов и верительных данных. Если обработчик CallbackHandler не задан, объект LoginContext предпринимает попытку создать новый экземпляр класса-обработчика, имя которого указано в системном свойстве auth.login.defaultCallbackHandler из файла свойств системы безопасности.

После создания объекта LoginContext его можно использовать для аутентификации пользователя. Для этого нужно вызвать метод login(), а затем метод getSubject(), который возвращает объект Subject, представляющий аутентифицированного пользователя. После завершения работы с Subject нужно с помощью метода logout() выполнить процедуру выхода.

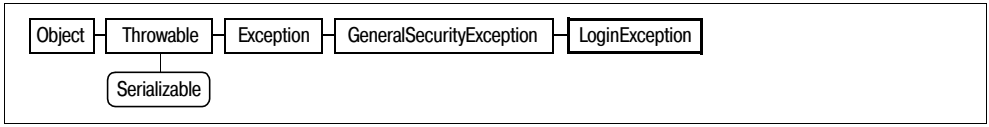
```
public class LoginContext {
// Открытые конструкторы
   public LoginContext(String name) throws LoginException;
   public LoginContext(String name, javax.security.auth.Subject subject) throws LoginException;
   public LoginContext(String name, javax.security.auth.callback.CallbackHandler callbackHandler)
        throws LoginException;
   public LoginContext(String name, javax.security.auth.Subject subject,
        javax.security.auth.callback.CallbackHandler callbackHandler) throws LoginException;
// Открытые методы экземпляра
   public javax.security.auth.Subject getSubject();
   public void login() throws LoginException;
   public void logout() throws LoginException;
}
```

LoginException

Java 1.4

javax.security.auth.login ***сериализуемое, проверяемое***

Это исключение сообщает о неполадках при создании объекта LoginContext или во время процедуры входа или выхода. Подклассы конкретизируют это исключение.




```
public class LoginException extends java.security.GeneralSecurityException {
// Открытые конструкторы
   public LoginException();
   public LoginException(String msg);
}
```

Подклассы: AccountExpiredException, CredentialExpiredException, FailedLoginException

Генерируется методами: LoginContext.{login(), LoginContext(), logout()}, javax.security.auth.spi.LoginModule.{abort(), commit(), login(), logout()}

Пакет javax.security.auth.spi Java 1.4

В пакете определен интерфейс поставщика сервисов для JAAS. Здесь определен единственный интерфейс LoginModule, который должен быть реализован в модулях входа.

Интерфейс

```
public interface LoginModule;
```

LoginModule Java 1.4

javax.security.auth.spi

Этот интерфейс нужно реализовать при разработке модулей входа в систему с использованием JAAS API. Поскольку он обычно не используется разработчиками приложений, методы этого интерфейса здесь не описаны.

```
public interface LoginModule {
// Открытые методы экземпляра
   public abstract boolean abort() throws javax.security.auth.login.LoginException;
   public abstract boolean commit() throws javax.security.auth.login.LoginException;
   public abstract void initialize(javax.security.auth.Subject subject,
        javax.security.auth.callback.CallbackHandler callbackHandler,
        java.util.Map sharedState, java.util.Map options);
   public abstract boolean login() throws javax.security.auth.login.LoginException;
   public abstract boolean logout() throws javax.security.auth.login.LoginException;
}
```

Пакет javax.security.auth.x500 Java 1.4

Этот пакет содержит классы, необходимые при аутентификации принципалов с использованием протокола X.500. Экземпляры класса могут храниться во множествах Subject принципалов и верительных данных. Не исключено, что класс X500Principal может представлять интерес для разработчиков приложений, однако он рассчитан на системных программистов, разрабатывающих реализации javax.security.auth.spi.LoginModule интерфейса на основе протокола X.500. См. также описание пакета java.security.cert, который содержит класс, представляющий сертификат X.509.

Классы

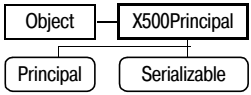
```
public final class X500Principal implements java.security.Principal, Serializable;
public final class X500PrivateCredential implements javax.security.auth.Destroyable;
```

X500Principal Java 1.4

javax.security.auth.x500 ***сериализуемый***

Этот класс реализует интерфейс java.security.Principal и поддерживает отмеченные (distinguished) имена X.500 (например, «CN=David,O=davidflanagan.com,C=US»). Конструкторы данного класса могут принимать эти имена в виде строки или в двоичной форме с использованием формата, указанного константой (одной из трех). Метод getName() без аргументов, определенный в интерфейсе Principal, возвращает отмечен-

ное имя, форматированное согласно спецификации RFC 2253. Наконец, getEncoded() возвращает имя в двоичной форме.



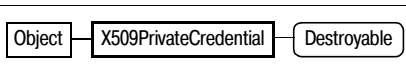

```
public final class X500Principal implements java.security.Principal, Serializable {
// Открытые конструкторы
   public X500Principal(java.io.InputStream is);
   public X500Principal(String name);
   public X500Principal(byte[] name);
// Открытые константы
   public static final String CANONICAL;                                    // ="CANONICAL"
   public static final String RFC1779;                                      // ="RFC1779"
   public static final String RFC2253;                                      // ="RFC2253"
// Открытые методы экземпляра
   public byte[] getEncoded();
   public String getName(String format);
// Методы, реализующие Principal
   public boolean equals(Object o);
   public String getName();
   public int hashCode();
   public String toString();
}
```

Возвращается методами: java.security.cert.X509Certificate.{getIssuerX500Principal(), getSubjectX500Principal()}, java.security.cert.X509CRL.getIssuerX500Principal()

X500PrivateCredential Java 1.4

javax.security.auth.x500

Этот класс связывает сертификат java.security.cert.X509Certificate с его закрытым ключом java.security.PrivateKey. Кроме того, он может связывать его с псевдонимом, который используется при извлечении сертификата из хранилища ключей java.security.KeyStore. Данный класс определяет методы, осуществляющие поиск сертификата, ключа и псевдонима, и методы интерфейса javax.security.cert.Destroyable.




```
public final class X500PrivateCredential implements javax.security.auth.Destroyable {
// Открытые конструкторы
   public X500PrivateCredential(java.security.cert.X509Certificate cert, java.security.PrivateKey key);
   public X500PrivateCredential(java.security.cert.X509Certificate cert,
                                java.security.PrivateKey key, String alias);
// Открытые методы экземпляра
   public String getAlias();
   public java.security.cert.X509Certificate getCertificate();
   public java.security.PrivateKey getPrivateKey();
// Методы, реализующие Destroyable
   public void destroy();
   public boolean isDestroyed();
}
```

Глава 21

javax.xml.parsers, java.xml.transform и подпакеты

Этот раздел предоставляет документацию по пакету `javax.xml.parser` и пакету `javax.xml.transform` и его подпакетам. Эти пакеты составляют API Java для работы с XML, или JAXP. Ранее (до интеграции в Java 1.4) JAXP существовал как стандартное расширение. Этим объясняется приставка «javax» в именах пакета. В разделе рассмотрены следующие пакеты:

`javax.xml.parsers`

Данный пакет определяет классы парсера (parser), которые выступают в качестве обертки (wrapper) вокруг базовых парсеров DOM и SAX XML. Кроме того, он определяет классы-фабрики, которые применяются для получения экземпляров этих классов парсера.

`javax.xml.transform`

Этот пакет определяет классы и интерфейсы для преобразования представления и содержимого XML-документа с помощью XSLT. Он определяет интерфейсы `Source` и `Result` для представления исходного и конечного документов. Подпакеты обеспечивают реализации этих классов, которые представляют документы различными способами.

`javax.xml.transform.dom`

Данный пакет реализует интерфейсы `Source` и `Result`, которые представляют документы как деревья, состоящие из DOM-документов.

`javax.xml.transform.sax`

Этот пакет реализует интерфейсы `Source` и `Result`, которые представляют документы как последовательности, состоящие из событий SAX-парсера. Кроме того, он определяет другие классы преобразования, связанные с SAX.

`javax.xml.transform.stream`

Этот пакет реализует интерфейсы `Source` и `Result`, которые представляют документы в виде потоков текста.

Пакет javax.xml.parsers Java 1.4

Данный пакет определяет классы, которые представляют XML-парсеры и классы-фабрики для получения экземпляров этих классов парсера. DocumentBuilder является XML-парсером, который основан на DOM и создан при помощи DocumentBuilderFactory. SAXParser является XML-парсером, который основан на SAX и создан из SAXParserFactory. Этот пакет не включает в себя реализации парсера. Напротив, он является слоем, не зависящим от реализации, который поддерживает подключаемые XML-парсеры. Более того, этот пакет не определяет DOM или SAX API, которые необходимы для работы с XML-документами. DOM API определяется в org.w3c.dom, а SAX API – в org.xml.sax и его подпакетах.

Классы

```
public abstract class DocumentBuilder;
public abstract class DocumentBuilderFactory;
public abstract class SAXParser;
public abstract class SAXParserFactory;
```

Исключение

```
public class ParserConfigurationException extends Exception;
```

Ошибки

```
public class FactoryConfigurationError extends Error;
```

DocumentBuilder Java 1.4

javax.xml.parsers

Этот класс определяет высокоуровневый API для базовой реализации DOM-парсера. Получите DocumentBuilder из DocumentBuilderFactory. После получения DocumentBuilder можно подготовить объекты org.xml.sax.ErrorHandler и org.xml.sax.EntityResolver. (Эти классы определяются в SAX API, но они полезны и для DOM-парсеров.) Методы isNamespaceAware() и isValidating() позволяют удостоствериться в том, что парсер сконфигурирован с теми функциональными особенностями, которые требуются вашему приложению. И наконец, методы parse() применяются для чтения XML-документа из потока, файла, URL или объекта org.xml.sax.InputSource, а также для анализа этого документа и его преобразования в дерево org.w3c.dom.Document. Объекты DocumentBuilder обычно не являются потокобезопасными (thread-safe).

Если вам необходимо получить пустой объект Document (например, для построения дерева документа с самого начала), вызывайте newDocument(). Или используйте getDOMImplementation() для получения объекта org.w3c.dom.DOMImplementation базовой реализации DOM, из которой вы также можете создать пустой Document.

В описании пакета org.w3c.dom представлена информация о возможных операциях с объектом Document после того, как вы использовали DocumentBuilder для его создания.

```
public abstract class DocumentBuilder {
// Защищенные конструкторы
   protected DocumentBuilder();
// Методы доступа к свойствам (по имени свойства)
   public abstract org.w3c.dom.DOMImplementation getDOMImplementation();
   public abstract boolean isNamespaceAware();
```

```
   public abstract boolean isValidating();
// Открытые методы экземпляра
   public abstract org.w3c.dom.Document newDocument();
   public abstract org.w3c.dom.Document parse(org.xml.sax.InputSource is)
        throws org.xml.sax.SAXException, java.io.IOException;
   public org.w3c.dom.Document parse(java.io.InputStream is)
        throws org.xml.sax.SAXException, java.io.IOException;
  public org.w3c.dom.Document parse(String uri) throws org.xml.sax.SAXException, java.io.IOException;
   public org.w3c.dom.Document parse(java.io.File f) throws org.xml.sax.SAXException,
        java.io.IOException;
   public org.w3c.dom.Document parse(java.io.InputStream is, String systemId)
        throws org.xml.sax.SAXException, java.io.IOException;
   public abstract void setEntityResolver(org.xml.sax.EntityResolver er);
   public abstract void setErrorHandler(org.xml.sax.ErrorHandler eh);
}
```

Возвращается методами: `DocumentBuilderFactory.newDocumentBuilder()`

DocumentBuilderFactory

Java 1.4

java.xml.parsers

Вы можете получить `DocumentBuilderFactory` путем создания экземпляра подкласса, специфичного для конкретной реализации, который предоставляется поставщиком парсера. Однако можно просто вызвать `newInstance()` для получения экземпляра фабрики, сконфигурированной для системы по умолчанию. После получения объекта фабрики вы можете использовать различные `set`-методы для конфигурирования свойств объектов `DocumentBuilder`, которые он создаст. Эти методы позволяют указать, будут ли парсеры , созданные фабрикой, выполнять следующие функции:

- Производить слияние разделов `CDATA` с примыкающими текстовыми узлами
- Развертывать ссылки на объекты или оставлять их неразвернутыми в дереве документа
- Пропускать XML-комментарии из дерева документа
- Пропускать пробельные символы из дерева документа
- Правильно обрабатывать пространства имен XML
- Подтверждать XML-документы в соответствии с DTD или по другой схеме

В дополнение к различным `set`-методам, которые не зависят от реализации, вы также можете использовать `setAttribute()` для передачи именованного атрибута, зависящего от реализации, базовой реализации парсера. После завершения конфигурирования объекта фабрики просто вызывайте `newDocumentBuilder()` для создания объекта `DocumentBuilder` со всеми указанными атрибутами. Объекты `DocumentBuilderFactory` обычно не являются потокобезопасными.

Пакет `javax.xml.parsers` позволяет «встраивать» реализации парсера. Встраиваемость обеспечивается методом `newInstance()`, который придерживается следующих правил при определении того, какую именно реализацию `DocumentBuilderFactory` следует использовать:

- Если определено системное свойство `javax.xml.parsers.DocumentBuilderFactory`, будет применяться класс, заданный этим свойством.

- В других случаях, если файл jre/lib/jaxp.properties существует в дистрибутиве Java и содержит определение для свойства javax.xml.parsers.DocumentBuilderFactory, будет применяться класс, заданный этим свойством.
- В других случаях, если какой-либо из JAR-файлов на пути к классам включает файл META-INF/services/javax.xml.parsers.DocumentBuilderFactory, применяется класс, указанный в этом файле.
- В других случаях используется реализация по умолчанию, предоставляемая реализацией Java.

```
public abstract class DocumentBuilderFactory {
// Защищенные конструкторы
   protected DocumentBuilderFactory();
// Открытые методы класса
   public static DocumentBuilderFactory newInstance() throws FactoryConfigurationError;
// Методы доступа к свойствам (по имени свойства)
   public boolean isCoalescing();
   public void setCoalescing(boolean coalescing);
   public boolean isExpandEntityReferences();
   public void setExpandEntityReferences(boolean expandEntityRef);
   public boolean isIgnoringComments();
   public void setIgnoringComments(boolean ignoreComments);
   public boolean isIgnoringElementContentWhitespace();
   public void setIgnoringElementContentWhitespace(boolean whitespace);
   public boolean isNamespaceAware();
   public void setNamespaceAware(boolean awareness);
   public boolean isValidating();
   public void setValidating(boolean validating);
// Открытые методы экземпляра
   public abstract Object getAttribute(String name) throws IllegalArgumentException;
   public abstract DocumentBuilder newDocumentBuilder() throws ParserConfigurationException;
   public abstract void setAttribute(String name, Object value) throws IllegalArgumentException;
}
```

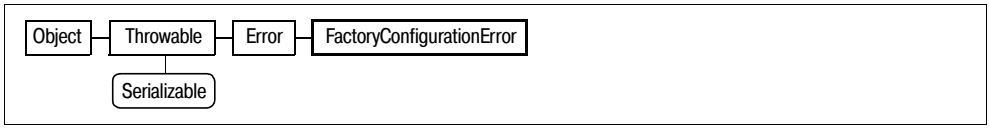
Возвращается методами: DocumentBuilderFactory.newInstance()

FactoryConfigurationError

Java 1.4

javax.xml.parsers ***сериализуемый, ошибка***

Свидетельствует о непоправимой проблеме, возникшей при создании экземпляра парсер-фабрики. Обычно это означает, что реализация парсера была встроена неправильно, а метод newInstance() не может установить месторасположение указанного класса реализации фабрики.




```
public class FactoryConfigurationError extends Error {
// Открытые конструкторы
   public FactoryConfigurationError();
   public FactoryConfigurationError(Exception e);
   public FactoryConfigurationError(String msg);
```

```
   public FactoryConfigurationError(Exception e, String msg);
// Открытые методы экземпляра
   public Exception getException();                                    // по умолчанию: null
// Открытые методы, замещающие Throwable
   public String getMessage();                                         // по умолчанию: null
}
```

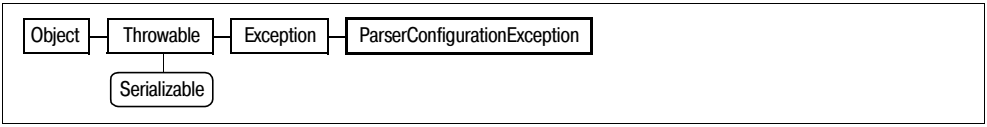
Генерируется методами: `DocumentBuilderFactory.newInstance()`, `SAXParserFactory.newInstance()`

ParserConfigurationException

Java 1.4

javax.xml.parsers ***сериализуемое, проверяемое***

Это исключение свидетельствует о проблеме с конфигурацией парсера, которая не позволяет объекту парсер-фабрики создать объект парсера.




```
public class ParserConfigurationException extends Exception {
// Открытые конструкторы
   public ParserConfigurationException();
   public ParserConfigurationException(String msg);
}
```

Генерируется методами: `DocumentBuilderFactory.newDocumentBuilder()`, `SAXParserFactory.{getFeature(), newSAXParser(), setFeature()}`

SAXParser

Java 1.4

javax.xml.parsers

Класс `SAXParser` является оберткой вокруг класса `org.xml.sax.XMLReader`. Он используется для анализа XML-документов с применением SAX версии 2 API. Получите `SAXParser` из `SAXParserFactory`. При необходимости вызовите `setProperty()`, чтобы установить свойство в базовом парсере. На сайте *http://www.saxproject.org* представлено описание стандартных свойств SAX и их значений. Наконец, вызовите один из методов `parse()` для анализа XML-документа из потока, файла, URL или `org.xml.sax.InputSource`. SAX API управляется событиями. SAX-парсер не строит дерево документа для описания XML-документа так, как это делает DOM-парсер. Напротив, он описывает XML-документ вашему приложению путем вызова методов объекта, который предоставляется приложением. Для этой цели служит объект `org.xml.sax.helpers.DefaultHandler`, который передается методу `parse()`: вы создаете его подкласс для реализации необходимых методов, а парсер вызовет эти методы, когда это будет нужно. Например, когда парсер сталкивается с XML-тегом в документе, то он анализирует тег и вызывает метод `startElement()` для того, чтобы сообщить вам об этом событии. Когда он находит отрывок текста, то он передает такой текст методу `characters()`.

Вместо методов `parse()` этого класса можно вызвать `getXMLReader()` для получения базового объекта `XMLReader` и работать с ним напрямую, чтобы обработать желаемый документ. Объекты `SAXParser` обычно не являются потокобезопасными.

Метод getParser(), как и методы parse(), берущие объект org.xml.sax.HandlerBas, основывается на SAX версии 1 API. Его применения следует избегать.

```
public abstract class SAXParser {
// Защищенные конструкторы
   protected SAXParser();
// Методы доступа к свойствам (по имени свойства)
   public abstract boolean isNamespaceAware();
   public abstract org.xml.sax.Parser getParser() throws org.xml.sax.SAXException;
   public abstract boolean isValidating();
   public abstract org.xml.sax.XMLReader getXMLReader() throws org.xml.sax.SAXException;
// Открытые методы экземпляра
   public abstract Object getProperty(String name)
        throws org.xml.sax.SAXNotRecognizedException,org.xml.sax.SAXNotSupportedException;
   public void parse(java.io.File f, org.xml.sax.helpers.DefaultHandler dh)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(org.xml.sax.InputSource is, org.xml.sax.HandlerBase hb)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(org.xml.sax.InputSource is, org.xml.sax.helpers.DefaultHandler dh)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(java.io.InputStream is, org.xml.sax.helpers.DefaultHandler dh)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(java.io.InputStream is, org.xml.sax.HandlerBase hb)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(String uri, org.xml.sax.HandlerBase hb)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(String uri, org.xml.sax.helpers.DefaultHandler dh)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(java.io.File f, org.xml.sax.HandlerBase hb)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(java.io.InputStream is, org.xml.sax.HandlerBase hb, String systemId)
        throws org.xml.sax.SAXException, java.io.IOException;
   public void parse(java.io.InputStream is, org.xml.sax.helpers.DefaultHandler dh,
        String systemId) throws org.xml.sax.SAXException, java.io.IOException;
   public abstract void setProperty(String name, Object value)
        throws org.xml.sax.SAXNotRecognizedException,org.xml.sax.SAXNotSupportedException;
}
```

Возвращается методами: SAXParserFactory.newSAXParser()

SAXParserFactory

Java 1.4

javax.xml.parsers

Этот класс является фабрикой для объектов SAXParser. Получите SAXParserFactory, вызвав метод newInstance(), который создает экземпляр по умолчанию подкласса SAXParserFactory или экземпляр какого-либо другого, «встроенного» SAXParserFactory.

После получения объекта SAXParserFactory можно использовать методы setValidating() и setNamespaceAware() для указания того, будут ли создаваемые парсеры парсерами с подтверждением и будут ли они знать, как обрабатывать пространства имен XML. Кроме того, можно вызвать setFeature() для того, чтобы задать функциональную особенность базовой реализации парсера. На сайте *http://www.saxproject.org* представлены имена стандартных функциональных особенностей парсера, которые могут быть активированы или отключены с помощью этого метода.

Как только вы создали и сконфигурировали объект фабрики, просто вызовите newSAXParser() для создания объекта SAXParser. Реализации SAXParserFactory обычно не являются потокобезопасными.

Пакет javax.xml.parsers позволяет «встраивать» реализации парсера. Встраиваемость обеспечивается методом newInstance(), который придерживается следующих правил при определении подкласса SAXParserFactory:

- Если определено системное свойство javax.xml.parsers.SAXParserFactory, будет задействован класс, указанный этим свойством.
- В противном случае, если файл jre/lib/jaxp.properties существует в дистрибутиве Java и содержит определение для свойства javax.xml.parsers.SAXParserFactory, будет применяться класс, указанный этим свойством.
- В противном случае, если какой-либо из JAR-файлов в пути к классам включает файл, названный META-INF/services/javax.xml.parsers.SAXParserFactory, применяется класс, указанный в этом файле.
- В противном случае применяется реализация по умолчанию, предоставляемая платформой Java.

```
public abstract class SAXParserFactory {
// Защищенные конструкторы
   protected SAXParserFactory();
// Открытые методы класса
   public static SAXParserFactory newInstance() throws FactoryConfigurationError;
// Открытые методы экземпляра
   public abstract boolean getFeature(String name) throws ParserConfigurationException,
      org.xml.sax.SAXNotRecognizedException,org.xml.sax.SAXNotSupportedException;
   public boolean isNamespaceAware();
   public boolean isValidating();
   public abstract SAXParser newSAXParser() throws ParserConfigurationException,
      org.xml.sax.SAXException;
   public abstract void setFeature(String name, boolean value)
      throws ParserConfigurationException, org.xml.sax.SAXNotRecognizedException,
      org.xml.sax.SAXNotSupportedException;
   public void setNamespaceAware(boolean awareness);
   public void setValidating(boolean validating);
}
```

Возвращается методами: SAXParserFactory.newInstance()

Пакет javax.xml.transform

Java 1.4

Этот пакет определяет высокоуровневый, не зависящий от реализации API для применения XSLT-машины или другой системы преобразования документа в целях преобразования содержимого XML-документа. Кроме того, данный API применяется для преобразования XML-документов из одной формы (например, поток текста в файле) в другую (например, дерево DOM-узлов). Интерфейс Source – это очень общее описание источника документа. В подпакетах данного пакета определены три конкретные реализации, которые представляют документы в текстовой форме как DOM-деревья и как последовательности событий SAX-парсера. Подобным образом интерфейс Result является высокоуровневым описанием того, в какую именно форму следует преобразовывать исходный документ. Три подпакета определяют три реализа-

ции Result, которые представляют XML-документы как потоки или файлы, как DOM-деревья и как последовательности событий SAX-парсера.

Класс TransformerFactory представляет машину для преобразования документа. Его реализация обеспечивает фабрику по умолчанию, которая представляет XSLT-машину. TransformerFactory можно применять для создания объектов Templates, которые представляют скомпилированные таблицы стилей XSL (или другие формы правил преобразования, зависящие от реализации). На самом деле документы преобразовываются из Source в Result с помощью объекта Transformer, который получают из объекта Templates или напрямую из TransformerFactory.

Интерфейсы

```
public interface ErrorListener;
public interface Result;
public interface Source;
public interface SourceLocator;
public interface Templates;
public interface URIResolver;
```

Классы

```
public class OutputKeys;
public abstract class Transformer;
public abstract class TransformerFactory;
```

Исключения

```
public class TransformerException extends Exception;
   └ public class TransformerConfigurationException extends TransformerException;
```

Ошибки

```
public class TransformerFactoryConfigurationError extends Error;
```

ErrorListener Java 1.4

javax.xml.transform

Этот интерфейс определяет методы, которые применяются объектами Transformer и TransformerFactory для передачи приложению предупреждений и сообщений об ошибках и неисправимых ошибках. Для использования ErrorListener приложение должно реализовать этот интерфейс и передать объект методу setErrorListener(), принадлежащему Transformer или TransformerFactory. Аргументом каждого метода этого интерфейса является объект TransformerException. Реализация этих методов может генерировать такое исключение или же просто записать каким-либо образом предупреждение или ошибку, а затем выполнить возврат. Transformer или TransformerFactory не должны продолжать обработку после сообщения о непоправимой ошибке при помощи вызова метода fatalError().

Если вы знакомы с SAX API для разбора XML-документов, то вы обнаружите, что этот интерфейс очень похож на org.xml.sax.ErrorHandler.

```
public interface ErrorListener {
// Открытые методы экземпляра
   public abstract void error(TransformerException exception) throws TransformerException;
   public abstract void fatalError(TransformerException exception) throws TransformerException;
   public abstract void warning(TransformerException exception) throws TransformerException;
}
```

Передается методам: Transformer.setErrorListener(), TransformerFactory.setErrorListener()

Возвращается методами: Transformer.getErrorListener(), TransformerFactory.getErrorListener()

OutputKeys Java 1.4

javax.xml.transform

Этот класс определяет строковые константы, которые вмещают в себя имена атрибутов тега <xsl:output> в таблице стилей XSLT. Кроме того, они являются разрешенными ключевыми значениями для объекта Properties, который возвращается методом Templates.getOutputProperties() и передается методу Transformer.setOutputProperties().

```
public class OutputKeys {
// Конструктор отсутствует
// Открытые константы
   public static final String CDATA_SECTION_ELEMENTS;          // ="cdata-section-elements"
   public static final String DOCTYPE_PUBLIC;                  // ="doctype-public"
   public static final String DOCTYPE_SYSTEM;                  // ="doctype-system"
   public static final String ENCODING;                        // ="encoding"
   public static final String INDENT;                          // ="indent"
   public static final String MEDIA_TYPE;                      // ="media-type"
   public static final String METHOD;                          // ="method"
   public static final String OMIT_XML_DECLARATION;            // ="omit-xml-declaration"
   public static final String STANDALONE;                      // ="standalone"
   public static final String VERSION;                         // ="version"
}
```

Result Java 1.4

javax.xml.transform

Этот интерфейс представляет в общем виде результат XML-преобразования. Метод setSystemId() задает системный идентификатор результата в виде URL. Это полезно, если результат должен быть записан как файл. Кроме того, это полезно для сообщения об ошибках и для процесса определения относительных URL, даже если объект Result не представляет файл. Все остальные методы, имеющие отношение к результату, зависят от конкретной реализации этого интерфейса. См. описание реализаций DOMResult, SAXResult и StreamResult в трех подпакетах этого пакета.

```
public interface Result {
// Открытые константы
   public static final String PI_DISABLE_OUTPUT_ESCAPING;  // ="javax.xml.transform.disable-output-escaping"
   public static final String PI_ENABLE_OUTPUT_ESCAPING;   // ="javax.xml.transform.enable-output-escaping"
// Открытые методы экземпляра
   public abstract String getSystemId();
   public abstract void setSystemId(String systemId);
}
```

Реализации: javax.xml.transform.dom.DOMResult, javax.xml.transform.sax.SAXResult, javax.xml.transform.stream.StreamResult

Передается методам: Transformer.transform(), javax.xml.transform.sax.TransformerHandler.setResult()

Source

Java 1.4

javax.xml.transform

Этот интерфейс представляет в очень общем виде источник XML-документа. Метод `setSystemId()` задает системный идентификатор документа в форме URL. Это полезно для процесса определения относительных URL и для сообщений об ошибках, даже если документ не считывается напрямую из URL. Все остальные методы, имеющие отношение к источнику документа, зависят от конкретной реализации этого интерфейса. См. описание реализаций `DOMSource`, `SAXSource` и `StreamSource` в трех подпакетах данного пакета.

```
public interface Source {
// Открытые методы экземпляра
   public abstract String getSystemId();
   public abstract void setSystemId(String systemId);
}
```

Реализации: `javax.xml.transform.dom.DOMSource`, `javax.xml.transform.sax.SAXSource`, `javax.xml.transform.stream.StreamSource`

Передается методам: `Transformer.transform()`, `TransformerFactory.{getAssociatedStylesheet(), newTemplates(), newTransformer()}`, `javax.xml.transform.sax.SAXSource.sourceToInputSource()`, `javax.xml.transform.sax.SAXTransformerFactory.{newTransformerHandler(), newXMLFilter()}`

Возвращается методами: `TransformerFactory.getAssociatedStylesheet()`, `URIResolver.resolve()`

SourceLocator

Java 1.4

javax.xml.transform

Этот интерфейс определяет методы, которые возвращают системные и открытые идентификаторы XML-документа, а также номер строки и колонки в пределах этого документа. Объекты `SourceLocator` используются с объектами `TransformerException` и `TransformerConfigurationException` для указания места в XML-файле, в котором произошло исключение. Тем не менее системные и открытые идентификаторы не всегда имеются для того или иного документа, поэтому `getSystemId()` и `getPublicId()` могут вернуть `null`. Кроме того, `Transformer` не должен точно отслеживать номера строк и колонок, поэтому `getLineNumber()` и `getColumnNumber()` могут вернуть –1, сообщая о том, что информация о номере строки и колонки отсутствует. Если они вернут любое значение, кроме –1, то его надо рассматривать как приближенное значение фактической величины. Строки и колонки в пределах документа нумеруются начиная с 1, а не с 0.

Если вы знакомы с SAX API для разбора XML, то вы обнаружите, что этот интерфейс – переименованная версия `org.xml.sax.Locator`.

```
public interface SourceLocator {
// Открытые методы экземпляра
   public abstract int getColumnNumber();
   public abstract int getLineNumber();
   public abstract String getPublicId();
   public abstract String getSystemId();
}
```

Реализации: `javax.xml.transform.dom.DOMLocator`

Передается методам: TransformerConfigurationException.TransformerConfigurationException(), TransformerException.{setLocator(), TransformerException()}

Возвращается методами: TransformerException.getLocator()

Templates

Java 1.4

javax.xml.transform

Этот интерфейс представляет набор правил, описывающих преобразование документа Source в документ Result. Теоретически пакет javax.xml.transform не зависит от типа преобразования, но на практике объект этого типа всегда представляет скомпилированную форму таблицы стилей XSLT. Объект Templates можно получить из объекта TransformerFactory. Кроме того, можно задействовать javax.xml.transform.sax.TemplatesHandler. После получения объекта Templates можно использовать метод newTransformer() для создания объекта Transformer. С его помощью можно применить шаблоны к Source, чтобы сгенерировать документ Result.

Метод getOutputProperties() возвращает объект java.util.Properties, который определяет пары «имя/значение», описывающие детали генерации текстовой версии документа Result. Эти свойства задаются в таблице стилей XSLT с помощью элемента <xsl:output>. Константы, определяемые OutputKeys, являются разрешенными именами выходных свойств. Возвращенный объект Properties содержит четко выраженные свойства, а также значения по умолчанию в родительском объекте Properties. Если запросить значение свойства с помощью getProperty(), то будет получено значение по умолчанию. С другой стороны, если запросить свойство с помощью метода get(), который унаследован Properties из его родительского класса, то будет получено значение свойства, если оно явно указано в таблице стилей, или null, если оно не было указано. Возвращаемый объект Properties является копией внутреннего значения, поэтому вы можете изменять его, не влияя на объект Templates (например, перед тем как передать объект Properties методу setOutputProperties() объекта Transformer).

Реализации Templates должны быть потокобезопасными. Объект Templates можно применять для создания любого количества объектов Transformer.

```
public interface Templates {
// Открытые методы экземпляра
   public abstract java.util.Properties getOutputProperties();
   public abstract Transformer newTransformer() throws TransformerConfigurationException;
}
```

Передается методам:
javax.xml.transform.sax.SAXTransformerFactory.{newTransformerHandler(), newXMLFilter()}

Возвращается методами: TransformerFactory.newTemplates(), javax.xml.transform.sax.TemplatesHandler.getTemplates()

Transformer

Java 1.4

javax.xml.transform

Объекты этого типа используются для преобразования документа Source в документ Result. Объект Transformer можно получить из объекта TransformerFactory, из объекта Templates, созданного TransformerFactory, или из объекта TransformerHandler, созданного SAXTransformerFactory (эти последние два типа представлены в пакете javax.xml.transform.sax).

После получения объекта Transformer у вас может возникнуть необходимость изменить его конфигурацию, прежде чем использовать его для преобразования документов. Методы setErrorListener() и setURIResolver() позволяют задать объекты ErrorListener и URLResolver для Transformer. С помощью методов setOutputProperty() и setOutputProperties() можно задать пары «имя/значение», влияющие на форматирование текста документа Result (если этот документ выполнен в текстовом формате). OutputKeys определяет константы, которые представляют набор стандартных имен выходных свойств. Выходные свойства, которые вы указываете при помощи этих методов, замещают любые выходные свойства, указанные в объекте Templates (при помощи тега <xsl:output>). Метод setParameter() позволяет указывать значения для любых параметров наивысшего уровня, определенных в таблице стилей (при помощи тегов <xsl:param>). Если имя любого такого параметра является составным именем, то оно появится в таблице стилей с приставкой пространства имен. Приставку нельзя применять с методом setParameter(); вместо этого нужно указать имя параметра, используя универсальный идентификатор ресурса (URI) пространства имен. URI находится в фигурных скобках, за которыми следует локальное имя. Если никакое пространство имен не задействовано, то можно использовать только имя параметра без фигурных скобок или URI.

После создания и конфигурирования объекта Transformer можно применять метод transform() для преобразования документа. Этот метод преобразовывает указанный документ Source и создает документ, заданный объектом Result.

Реализации Transformer обычно не являются потокобезопасными. Вы можете повторно использовать объект Transformer и вызывать transform() любое количество раз (только не одновременно). Выходные свойства и параметры, которые вы указываете, не меняются при вызове метода transform() и могут быть использованы повторно.

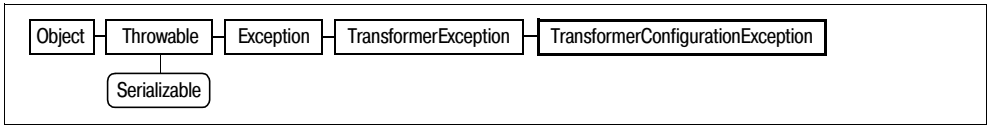
```
public abstract class Transformer {
// Защищенные конструкторы
   protected Transformer();
// Методы доступа к свойствам (по имени свойства)
   public abstract ErrorListener getErrorListener();
   public abstract void setErrorListener(ErrorListener listener) throws IllegalArgumentException;
   public abstract java.util.Properties getOutputProperties();
   public abstract void setOutputProperties(java.util.Properties oformat)
        throws IllegalArgumentException;
   public abstract URIResolver getURIResolver();
   public abstract void setURIResolver(URIResolver resolver);
// Открытые методы экземпляра
   public abstract void clearParameters();
   public abstract String getOutputProperty(String name) throws IllegalArgumentException;
   public abstract Object getParameter(String name);
   public abstract void setOutputProperty(String name, String value) throws IllegalArgumentException;
   public abstract void setParameter(String name, Object value);
   public abstract void transform(Source xmlSource, Result outputTarget) throws TransformerException;
}
```

Возвращается методами: Templates.newTransformer(), TransformerFactory.newTransformer(), javax.xml.transform.sax.TransformerHandler.getTransformer()

TransformerConfigurationException Java 1.4

javax.xml.transform ***сериализуемое, проверяемое***

Это исключение свидетельствует о неполадках, возникших при создании объекта `Transformer`. Например, они могут произойти при наличии синтаксической ошибки в таблице стилей XSL, которая содержит правила преобразования. Унаследованный метод `getLocator()` применяется для получения `SourceLocator`, описывающего место в документе, в котором произошло исключение.



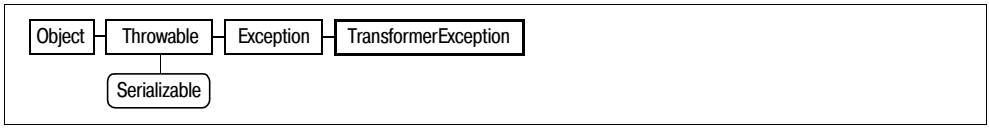

```
public class TransformerConfigurationException extends TransformerException {
// Открытые конструкторы
   public TransformerConfigurationException();
   public TransformerConfigurationException(Throwable e);
   public TransformerConfigurationException(String msg);
   public TransformerConfigurationException(String message, SourceLocator locator);
   public TransformerConfigurationException(String msg, Throwable e);
   public TransformerConfigurationException(String message, SourceLocator locator, Throwable e);
}
```

Генерируется методами: `Templates.newTransformer()`, `TransformerFactory.{getAssociatedStylesheet(), newTemplates(), newTransformer()}`, `javax.xml.transform.sax.SAXTransformerFactory.{newTemplatesHandler(), newTransformerHandler(), newXMLFilter()}`

TransformerException Java 1.4

javax.xml.transform ***сериализуемое, проверяемое***

Это исключение свидетельствует о неполадках, возникших при чтении или преобразовании документа. Вызовите `getLocator()` для получения объекта `SourceLocator`, описывающего место в документе, в котором произошло исключение.




```
public class TransformerException extends Exception {
// Открытые конструкторы
   public TransformerException(String message);
   public TransformerException(Throwable e);
   public TransformerException(String message, Throwable e);
   public TransformerException(String message, SourceLocator locator);
   public TransformerException(String message, SourceLocator locator, Throwable e);
// Открытые методы экземпляра
   public Throwable getException();
   public String getLocationAsString();
   public SourceLocator getLocator();
```

```
    public String getMessageAndLocation();
    public void setLocator(SourceLocator location);
// Открытые методы, замещающие Throwable
    public Throwable getCause();
    public Throwable initCause(Throwable cause);  // синхронизирован
    public void printStackTrace();
    public void printStackTrace(java.io.PrintStream s);
    public void printStackTrace(java.io.PrintWriter s);
}
```

Подклассы: TransformerConfigurationException

Передается методам: ErrorListener.{error(), fatalError(), warning()}

Генерируется методами: ErrorListener.{error(), fatalError(), warning()}, Transformer.transform(), URIResolver.resolve()

TransformerFactory

Java 1.4

javax.xml.transform

Экземпляр этого абстрактного класса представляет «машину для трансформации документа», например процессор XSLT. TransformerFactory используется для создания объектов Transformer. Эти объекты выполняют преобразование документов и могут быть использованы для трансформации правил преобразования (например, таблиц стилей XSLT) в скомпилированные объекты Templates.

Получите экземпляр TransformerFactory путем вызова статического метода newInstance(). Метод newInstance() возвращает экземпляр реализации по умолчанию для вашей инсталляции Java. Если же задано системное свойство javax.xml.transform.TransformerFactory, то он возвращает экземпляр класса реализации, заданного этим свойством. Реализация TransformerFactory по умолчанию, предоставляемая дистрибутивом Java, преобразует XML-документы, используя таблицы стилей XSL.

Вы можете конфигурировать экземпляр TransformerFactory путем вызова setErrorListener() и setURIResolver(), чтобы указать объект ErrorListener и объект URIResolver, которые фабрика должна будет задействовать при чтении и анализе таблиц стилей XSL. Методы setAttribute() и getAttribute() могут быть использованы для того, чтобы задавать (и получать) атрибуты машины преобразования, зависящие от реализации. Машина по умолчанию, которая поставляется Sun, не определяет каких-либо атрибутов. Метод getFeature() применяется для проверки того, поддерживает ли фабрика данную функцию. Для обеспечения уникальности имена функций выражены в виде URI; каждая реализация Source и Result, определенная в трех подпакетах данного пакета, определяет константу FEATURE. Эта константа задает URL, который можно применять для выяснения того, поддерживает ли TransformerFactory данный тип Source или Result.

После получения и конфигурирования объекта TransformerFactory его можно применять несколькими способами. Если вызвать метод newTransformer(), который не принимает аргументов, то будет получен объект Transformer, который преобразовывает формат или представление XML-документа без преобразования его содержимого. Например, Transformer, созданный таким образом, можно применять для преобразования DOM-дерева, представленного объектом javax.xml.transform.dom.DOMSource, в поток XML-текста, хранящегося в файле, определяемом javax.xml.transform.stream.StreamResult.

Вот другой способ использования TransformerFactory: вызовите метод newTemplates(), передав объект Source, который представляет таблицу стилей XSL. Это приведет к появлению объекта Templates, который можно использовать для получения объекта Transformer, применяющего таблицу стилей для преобразования содержимого документа. Если вы планируете создать только один объект Transformer из объекта Templates, то можно скомбинировать эти этапы и просто передать объект Source, представляющий таблицу стилей, версии метода newTransformer() с одним аргументом.

XML-документы могут включать в себя ссылки на таблицы стилей XSL в виде команды обработки xml-таблицы стилей. Метод getAssociatedStylesheet() читает XML-документ, представленный объектом Source, и возвращает новый объект Source, представляющий таблицу стилей (или объединение всех таблиц стилей), содержащихся в этом документе. Эти таблицы соответствуют ограничениям по носителям, названию и набору символов, которые определены тремя другими параметрами (они могут иметь значение null). Если нужно обработать XML-документ с использованием таблицы стилей, которую он сам определяет, применяйте этот метод для получения объекта Source. Этот объект можно передать newTransformer() для создания объекта Transformer, который можно использовать для преобразования документа.

Реализации TransformerFactory обычно не являются потокобезопасными.

```
public abstract class TransformerFactory {
// Защищенные конструкторы
   protected TransformerFactory();
// Открытые методы класса
   public static TransformerFactory newInstance() throws TransformerFactoryConfigurationError;
// Открытые методы экземпляра
   public abstract Source getAssociatedStylesheet(Source source, String media, String title,
        String charset) throws TransformerConfigurationException;
   public abstract Object getAttribute(String name) throws IllegalArgumentException;
   public abstract ErrorListener getErrorListener();
   public abstract boolean getFeature(String name);
   public abstract URIResolver getURIResolver();
   public abstract Templates newTemplates(Source source) throws TransformerConfigurationException;
   public abstract Transformer newTransformer() throws TransformerConfigurationException;
   public abstract Transformer newTransformer(Source source) throws TransformerConfigurationException;
   public abstract void setAttribute(String name, Object value) throws IllegalArgumentException;
   public abstract void setErrorListener(ErrorListener listener) throws IllegalArgumentException;
   public abstract void setURIResolver(URIResolver resolver);
}
```

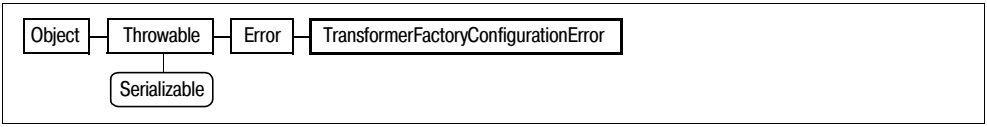
Подклассы: javax.xml.transform.sax.SAXTransformerFactory

Возвращается методами: TransformerFactory.newInstance()

TransformerFactoryConfigurationError Java 1.4

javax.xml.transform ***сериализуемый, ошибка***

Этот класс ошибок свидетельствует о возникновении непоправимой ошибки при создании TransformerFactory. Обычно это означает неполадки с конфигурацией: например, системное свойство javax.xml.transform.TransformerFactory имеет значение, которое не является допустимым именем класса, или же путь к классу не содержит указанный класс реализации фабрики.




```
public class TransformerFactoryConfigurationError extends Error {
// Открытые конструкторы
   public TransformerFactoryConfigurationError();
   public TransformerFactoryConfigurationError(String msg);
   public TransformerFactoryConfigurationError(Exception e);
   public TransformerFactoryConfigurationError(Exception e, String msg);
// Открытые методы экземпляра
   public Exception getException();                                    // по умолчанию:null
// Открытые методы, замещающие Throwable
   public String getMessage();                                         // по умолчанию:null
}
```

Генерируется методами: `TransformerFactory.newInstance()`

URIResolver

Java 1.4

javax.xml.transform

С помощью этого интерфейса приложение может объяснить объекту `Transformer`, как воспринимать URI, которые появляются в таблице стилей XSLT. Если `URIResolver` передается методу `setURIResolver()`, принадлежащему `Transformer` или `TransformerFactory`, а затем `Transformer` или `TransformerFactory` получит URI, то он передаст этот URI вместе с базовым URI методу `resolve()`, принадлежащему `URIResolver`. Если `resolve()` возвращает объект `Source`, то `Transformer` задействует этот `Source`. Если `Transformer` или `TransformerFactory` не имеют зарегистрированного `URIResolver` или метод `resolve()` возвращает `null`, то преобразователь или фабрика будут пытаться самостоятельно обработать данный URI.

```
public interface URIResolver {
// Открытые методы экземпляра
   public abstract Source resolve(String href, String base) throws TransformerException;
}
```

Передается методам: `Transformer.setURIResolver()`, `TransformerFactory.setURIResolver()`

Возвращается методами: `Transformer.getURIResolver()`, `TransformerFactory.getURIResolver()`

Пакет javax.xml.transform.dom

Java 1.4

Этот пакет содержит реализации `Source` и `Result`, которые работают с деревьями документов DOM и поддеревьями.

Интерфейсы

```
public interface DOMLocator extends javax.xml.transform.SourceLocator;
```

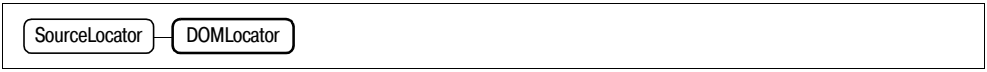
Классы

```
public class DOMResult implements javax.xml.transform.Result;
public class DOMSource implements javax.xml.transform.Source;
```

DOMLocator Java 1.4

javax.xml.transform.dom

Этот класс расширяет SourceLocator с целью определения метода, необходимого для возвращения объекта DOM Node, который обычно применяется для обозначения источника ошибки в процессе преобразования. См. SourceLocator и TransformerException.




```
public interface DOMLocator extends javax.xml.transform.SourceLocator {
// Открытые методы экземпляра
   public abstract org.w3c.dom.Node getOriginatingNode();
}
```

DOMResult Java 1.4

javax.xml.transform.dom

Этот класс является реализацией Result, которая записывает XML-содержимое путем генерирования DOM-дерева для представления этого содержимого. Если передать org.w3c.dom.Node конструктору или setNode(), то DOMResult создаст дерево-результат как дочерний элемент узла (обычно это узел Document или Element). Если не указывать узел, то DOMResult создаст новый узел Document при создании дерева-результата. Извлечь этот Document можно с помощью getNode().


```
public class DOMResult implements javax.xml.transform.Result {
// Открытые конструкторы
   public DOMResult();
   public DOMResult(org.w3c.dom.Node node);
   public DOMResult(org.w3c.dom.Node node, String systemID);
// Открытые константы
   public static final String FEATURE;   // ="http://javax.xml.transform.dom.DOMResult/feature"
// Открытые методы экземпляра
   public org.w3c.dom.Node getNode();                                    // по умолчанию:null
   public void setNode(org.w3c.dom.Node node);
// Методы, реализующие Result
   public String getSystemId();                                          // по умолчанию:null
   public void setSystemId(String systemId);
}
```

DOMSource Java 1.4

javax.xml.transform.dom

Этот класс является реализацией Source, которая считывает XML-документ из дерева или поддерева документа DOM. Передайте объект org.w3c.dom.Node, который представляет корень дерева или поддерева, конструктору или setNode(). Кроме того, полезно предоставить идентификатор системы (имя файла или URL) для использова-

ния в сообщениях об ошибках и для обработки относительных URL, которые содержатся в документе.


```
public class DOMSource implements javax.xml.transform.Source {
// Открытые конструкторы
   public DOMSource();
   public DOMSource(org.w3c.dom.Node n);
   public DOMSource(org.w3c.dom.Node node, String systemID);
// Открытые константы
   public static final String FEATURE;     // ="http://javax.xml.transform.dom.DOMSource/feature"
// Открытые методы экземпляра
   public org.w3c.dom.Node getNode();                                    // по умолчанию:null
   public void setNode(org.w3c.dom.Node node);
// Методы, реализующие Source
   public String getSystemId();                                          // по умолчанию:null
   public void setSystemId(String baseID);
}
```

Пакет javax.xml.transform.sax Java 1.4

Этот пакет определяет реализации Source и Result, которые работают с SAX-событиями. Помимо этого, он включает в себя расширение класса TransformerFactory, который имеет дополнительные методы для возвращения объектов TemplatesHandler и TransformerHandler. Эти объекты реализуют интерфейсы SAX-обработчика и способны работать с объектом SAX-парсера, чтобы превращать серию событий SAX-анализа в объект Templates или документ Result. SAXSource и SAXResult адаптируют каркас org.xml.sax для использования в каркасе javax.xml.transform. Наоборот, SAXTransformerFactory, TemplatesHandler и TransformerHandler адаптируют каркас javax.xml.transform для использования внутри каркаса org.xml.sax.

Интерфейсы

```
public interface TemplatesHandler extends org.xml.sax.ContentHandler;
public interface TransformerHandler
      extends org.xml.sax.ContentHandler, org.xml.sax.DTDHandler, org.xml.sax.ext.LexicalHandler;
```

Классы

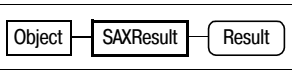
```
public class SAXResult implements javax.xml.transform.Result;
public class SAXSource implements javax.xml.transform.Source;
public abstract class SAXTransformerFactory extends javax.xml.transform.TransformerFactory;
```

SAXResult Java 1.4

javax.xml.transform.sax

Этот класс является реализацией Result, которая описывает содержимое преобразованного документа путем запуска методов указанного ContentHandler. То есть SAXResult действует как объект org.xml.sax.XMLReader, вызывая методы указанного объекта org.xml.sax.ContentHandler при анализе преобразуемого документа. Кроме того, вы мо-

жете предоставить объект org.xml.sax.ext.LexicalHandler, чьи методы будут вызываться SAXResult с помощью вызова setLexicalHandler() или предоставления объекта ContentHandler, который также реализует интерфейс LexicalHandler.




```
public class SAXResult implements javax.xml.transform.Result {
// Открытые конструкторы
   public SAXResult();
   public SAXResult(org.xml.sax.ContentHandler handler);
// Открытые константы
   public static final String FEATURE;    // ="http://javax.xml.transform.sax.SAXResult/feature"
// Открытые методы экземпляра
   public org.xml.sax.ContentHandler getHandler();                          // по умолчанию:null
   public org.xml.sax.ext.LexicalHandler getLexicalHandler();               // по умолчанию:null
   public void setHandler(org.xml.sax.ContentHandler handler);
   public void setLexicalHandler(org.xml.sax.ext.LexicalHandler handler);
// Методы, реализующие Result
   public String getSystemId();                                             // по умолчанию:null
   public void setSystemId(String systemId);
}
```

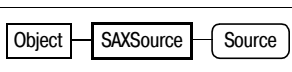
SAXSource

Java 1.4

javax.xml.transform.sax

Этот класс – реализация Source, которая описывает документ, представленный как серия вызовов методов SAX-событий. SAXSource нуждается в объекте org.xml.sax.InputSource, который описывает поток, подлежащий синтаксическому разбору. При необходимости для него можно задать объект org.xml.sax.XMLReader или org.xml.sax.XMLFilter, который генерирует SAX-события. (Если XMLReader или XMLFilter не указан, то объект Transformer создаст XMLReader по умолчанию.) Поскольку в этом случае требуется InputSource, поведение SAXSource не особенно отличается от поведения StreamSource, если только не применяется XMLFilter.

SAXSource также имеет один статический метод, sourceToInputSource(). Этот метод возвращает метод SAX InputSource из заданного объекта Source или null, если указанный Source не может быть преобразован в InputSource.




```
public class SAXSource implements javax.xml.transform.Source {
// Открытые конструкторы
   public SAXSource();
   public SAXSource(org.xml.sax.InputSource inputSource);
   public SAXSource(org.xml.sax.XMLReader reader, org.xml.sax.InputSource inputSource);
// Открытые константы
   public static final String FEATURE;      // ="http://javax.xml.transform.sax.SAXSource/feature"
// Открытые методы класса
   public static org.xml.sax.InputSource sourceToInputSource(javax.xml.transform.Source source);
// Открытые методы экземпляра
   public org.xml.sax.InputSource getInputSource();                         // по умолчанию:null
```

```
    public org.xml.sax.XMLReader getXMLReader();                    // по умолчанию:null
    public void setInputSource(org.xml.sax.InputSource inputSource);
    public void setXMLReader(org.xml.sax.XMLReader reader);
// Методы, реализующие Source
    public String getSystemId();                                    // по умолчанию:null
    public void setSystemId(String systemId);
}
```

SAXTransformerFactory Java 1.4

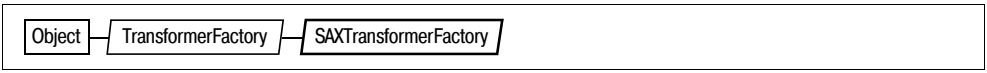
javax.xml.transform.sax

Этот класс расширяет TransformerFactory с целью определения дополнительных методов фабрики, которые полезны при работе с документами, представленными как последовательности SAX-событий. Передайте константу FEATURE методу getFeature() вашего объекта TransformerFactory, чтобы определить, поддерживаются ли методы newTemplatesHandler() и newTransformerHandler() и безопасно ли приводить объект TransformerFactory к SAXTransformerFactory. Используйте константу FEATURE_XMLFILTER с getFeature(), чтобы определить, поддерживаются ли методы newXMLFilter().

Метод newTemplatesHandler() возвращает объект TemplatesHandler, который можно использовать как объект org.xml.sax.ContentHandler для получения SAX-событий, сгенерированных SAX-парсером, и преобразования этих событий в объект Templates.

Методы newTransformerHandler() похожи: они возвращают объект TransformerHandler, который может принимать SAX-события, представлять исходный документ и преобразовывать их в документ Result. Версия newTransformerHandler() без аргументов создает TransformerHandler, который просто модифицирует форму документа, не применяя таблицу стилей к его содержимому. Две другие версии newTransformerHandler() используют таблицу стилей, представленную как объект Source или как Templates.

Если поддерживаются методы newXMLFilter(), то они возвращают объект org.xml.sax.XMLFilter, который может выступать как приемник и как источник SAX-событий и фильтрует эти события, применяя правила преобразования, задаваемые объектами Templates или Source.




```
public abstract class SAXTransformerFactory extends javax.xml.transform.TransformerFactory {
// Защищенные конструкторы
    protected SAXTransformerFactory();
// Открытые константы
    public static final String FEATURE; // ="http://javax.xml.transform.sax.SAXTransformerFactory/feature"
    public static final String FEATURE_XMLFILTER;
                    // ="http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter"
// Открытые методы экземпляра
    public abstract TemplatesHandler newTemplatesHandler()
        throws javax.xml.transform.TransformerConfigurationException;
    public abstract TransformerHandler newTransformerHandler()
        throws javax.xml.transform.TransformerConfigurationException;
    public abstract TransformerHandler newTransformerHandler(javax.xml.transform.Source src)
        throws javax.xml.transform.TransformerConfigurationException;
    public abstract TransformerHandler newTransformerHandler(javax.xml.transform.Templates templates)
        throws javax.xml.transform.TransformerConfigurationException;
    public abstract org.xml.sax.XMLFilter newXMLFilter(javax.xml.transform.Source src)
```

```
        throws javax.xml.transform.TransformerConfigurationException;
    public abstract org.xml.sax.XMLFilter newXMLFilter(javax.xml.transform.Templates templates)
        throws javax.xml.transform.TransformerConfigurationException;
}
```

TemplatesHandler Java 1.4

javax.xml.transform.sax

Этот интерфейс расширяет org.xml.sax.ContentHandler и добавляет метод getTemplates(). Объект, который реализует данный интерфейс, может быть использован для вызовов методов от какого-либо источника SAX-событий и для преобразования этих событий (в качестве таблицы стилей XSL) в объект Templates. Получите TemplatesHandler из SAXTransformerFactory. Зарегистрируйте его при помощи метода setContentHandler(), который принадлежит org.xml.sax.XMLReader, и вызовите метод parse() программы чтения. Когда parse() выполнит возврат, вызовите метод getTemplates(), чтобы получить объект Templates.


```
public interface TemplatesHandler extends org.xml.sax.ContentHandler {
// Открытые методы экземпляра
    public abstract String getSystemId();
    public abstract javax.xml.transform.Templates getTemplates();
    public abstract void setSystemId(String systemID);
}
```

Возвращается методами: SAXTransformerFactory.newTemplatesHandler()

TransformerHandler Java 1.4

javax.xml.transform.sax

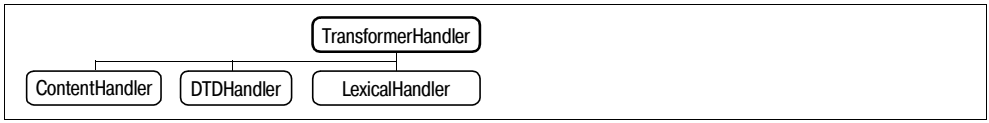
Этот интерфейс расширяет org.xml.sax.ContentHandler и родственные интерфейсы, чтобы он мог принимать SAX-события, сгенерированные org.xml.sax.SAXReader или org.xml.sax.XMLFilter. Создайте TransformerHandler путем вызова одного из методов newTransformerHandler(), принадлежащего SAXTransformerFactory.

Далее вызовите метод setResult(), чтобы указать объект Result, описывающий конечный документ, который вы желаете получить в результате преобразования. Вы также можете вызвать getTransformer() для получения объекта Transformer, ассоциированного с этим TransformerHandler, если вам необходимо задать выходные свойства или значения параметров для преобразования.

Теперь зарегистрируйте TransformerHandler в объекте XMLReader или XMLFilter путем вызова setContentHandler(), setDTDHandler() и setProperty(). Используйте имя свойства «http://www.xml.org/sax/properties/lexical-handler» в вызове, направленном к setProperties(), для регистрации TransformerHandler в качестве org.xml.sax.ext.LexicalHandler для парсера или фильтра.

И наконец, вызовите один из методов parse() для объекта XMLReader или XMLFilter. Это заставит программу чтения или фильтр начать анализ исходного документа и преобразовать его в вызовы методов TransformerHandler. TransformerHandler преобразует эти запросы в соответствии с указаниями в объекте Templates или Source, который пере-

дан при первоначальном вызове метода newTransformerHandler(), и сгенерирует конечный документ согласно объекту Result, который был передан setResult().




```
public interface TransformerHandler extends org.xml.sax.ContentHandler,
      org.xml.sax.DTDHandler, org.xml.sax.ext.LexicalHandler {
// Открытые методы экземпляра
   public abstract String getSystemId();
   public abstract javax.xml.transform.Transformer getTransformer();
   public abstract void setResult(javax.xml.transform.Result result) throws IllegalArgumentException;
   public abstract void setSystemId(String systemID);
}
```

Возвращается методами: SAXTransformerFactory.newTransformerHandler()

Пакет javax.xml.transform.stream Java 1.4

Данный пакет содержит реализации Source и Result, которые работают с файлами и потоками.

Классы

```
public class StreamResult implements javax.xml.transform.Result;
public class StreamSource implements javax.xml.transform.Source;
```

StreamResult Java 1.4

javax.xml.transform.stream

Этот класс является реализацией Result, которая записывает текстовое представление преобразованного документа в поток или файл. XML-документы определяют свою собственную кодировку, поэтому обычно предпочтительнее конструировать StreamResult с помощью File или OutputStream и отказаться от Writer, основанного на символах, который может использовать кодировку, отличную от указанной в документе.


```
public class StreamResult implements javax.xml.transform.Result {
// Открытые конструкторы
   public StreamResult();
   public StreamResult(java.io.File f);
   public StreamResult(String systemId);
   public StreamResult(java.io.Writer writer);
   public StreamResult(java.io.OutputStream outputStream);
// Открытые константы
   public static final String FEATURE;  // ="http://javax.xml.transform.stream.StreamResult/feature"
// Методы доступа к свойствам (по имени свойства)
   public java.io.OutputStream getOutputStream();                          // по умолчанию:null
   public void setOutputStream(java.io.OutputStream outputStream);
```

```
   public String getSystemId();                          // Реализует:Result; по умолчанию:null
   public void setSystemId(java.io.File f);
   public void setSystemId(String systemId);                                  // Реализует:Result
   public java.io.Writer getWriter();                                        // по умолчанию:null
   public void setWriter(java.io.Writer writer);
// Методы, реализующие Result
   public String getSystemId();                                              // по умолчанию:null
   public void setSystemId(String systemId);
}
```

StreamSource Java 1.4

javax.xml.transform.stream

Этот класс является реализацией Source, которая считывает текстовый формат XML-документа из файла, потока байтов или потока символов. XML-документы объявляют свою собственную кодировку, поэтому предпочтительнее создать объекта StreamSource из InputStream, а не из Reader, чтобы XML-процессор мог правильно обрабатывать объявленную кодировку. При создании StreamSource из потока байтов или потока символов вам следует предоставить идентификатор системы (то есть имя файла или URL). Для этого применяется один из конструкторов с двумя аргументами или вызов setSystemId(). Идентификатор (ID) системы требуется в том случае, если XML-файл, подлежащий обработке, включает в себя относительные URL, которые необходимо обработать.


```
public class StreamSource implements javax.xml.transform.Source {
// Открытые конструкторы
   public StreamSource();
   public StreamSource(java.io.InputStream inputStream);
   public StreamSource(java.io.Reader reader);
   public StreamSource(java.io.File f);
   public StreamSource(String systemId);
   public StreamSource(java.io.Reader reader, String systemId);
   public StreamSource(java.io.InputStream inputStream, String systemId);
// Открытые константы
   public static final String FEATURE; // ="http://javax.xml.transform.stream.StreamSource/feature"
// Методы доступа к свойствам (по имени свойства)
   public java.io.InputStream getInputStream();                              // по умолчанию:null
   public void setInputStream(java.io.InputStream inputStream);
   public String getPublicId();                                              // по умолчанию:null
   public void setPublicId(String publicId);
   public java.io.Reader getReader();                                        // по умолчанию:null
   public void setReader(java.io.Reader reader);
   public String getSystemId();                             // Реализует:Source; по умолчанию:null
   public void setSystemId(java.io.File f);
   public void setSystemId(String systemId);                                  // Реализует:Source
// Методы, реализующие Source
   public String getSystemId();                                              // по умолчанию:null
   public void setSystemId(String systemId);
}
```

Глава 22

org.ietf.jgss

Этот пакет в Java связывает Общие службы защиты данных API (Generic Security Services, GSS), определяемые Рабочей группой по проектированию Интернета (IETF). GSS представляет собой API, позволяющий двум равноправным пользователям сети проводить аутентификацию друг друга и обеспечивать целостность и конфиденциальность передаваемых данных. GSS является «общим» API, поскольку он может работать поверх любого «механизма защиты», поддерживающего аутентификацию, кодирование и защиту целостности данных. Kerberos является единственным механизмом, который в настоящее время поддерживается реализацией JGSS компании Sun. Реализация Sun интегрирована с JAAS API, определяемым пакетом `javax.security.auth` и его подпакетами. Чтобы воспользоваться JGSS, клиенты и серверы должны сначала войти в систему JAAS, чтобы обеспечить доступ реализации JGSS к необходимым сертификатам Kerberos.

Начинать изучение данного пакета следует с `GSSManager`, который является основным классом-фабрикой API. Чтобы понять и использовать этот пакет, вы должны обладать глубокими знаниями по GSS API и Kerberos. Мы не пытались полностью описать классы этого пакета, поскольку данный вопрос не относится к теме этого справочника. Более подробную информацию можно найти в следующих документах RFC, опубликованных IETF:

- RFC 2743 определяет GSS API независимо от языка: *http://www.ietf.org/rfc/rfc2743.txt*.
- RFC 2853 определяет связывание GSS API в Java: *http://www.ietf.org/rfc/rfc2853.txt*.
- RFC 1510 определяет Службу аутентификации в сети Kerberos: *http://www.ietf.org/rfc/rfc1510.txt*.
- RFC 1964 определяет механизм защиты Kerberos для GSS: *http://www.ietf.org/rfc/rfc1964.txt*.

Помимо этих основных источников, существует документация JGSS с учебными пособиями для версии Java 1.4, выпущенными компанией Sun. Их можно найти по адресу *http://java.sun.com/j2se/1.4/docs/guide/security/jgss/tutorials/index.html*.

Интерфейсы

```
public interface GSSContext;
public interface GSSCredential extends Cloneable;
public interface GSSName;
```

Классы

```
public class ChannelBinding;
public abstract class GSSManager;
public class MessageProp;
public class Oid;
```

Исключение

```
public class GSSException extends Exception;
```

ChannelBinding Java 1.4

org.left.jgss

Объект ChannelBinding обеспечивает дополнительные ограничения в контексте устанавливаемой защиты, а именно адреса клиента и сервера.

```
public class ChannelBinding {
// Открытые конструкторы
   public ChannelBinding(byte[] appData);
   public ChannelBinding(java.net.InetAddress initAddr, java.net.InetAddress acceptAddr,
                                                        byte[] appData);
// Открытые методы экземпляра
   public java.net.InetAddress getAcceptorAddress();
   public byte[] getApplicationData();
   public java.net.InetAddress getInitiatorAddress();
// Открытые методы, замещающие Object
   public boolean equals(Object obj);
   public int hashCode();
}
```

Передается методам: GSSContext.setChannelBinding()

GSSContext Java 1.4

org.ietf.jgss

Данный интерфейс представляет «контекст защиты», в который входит аутентификация, кодирование и сохранение целостности данных. Объект GSSContext получают из фабрики GSSManager. Прежде чем использовать GSSContext для обмена данными с равноправным пользователем сети, следует «установить» контекст защиты (обычно это подразумевает взаимную аутентификацию) в «цикл установки контекста». В этом цикле клиент многократно вызывает initSecContext(), а сервер многократно вызывает acceptSecContext() до тех пор, пока метод isEstablished() не возвратит true. Как только контекст установлен, можно с помощью wrap() применить службы защиты (например, кодирование) к массиву байтов или к потоку и выполнить обратную процедуру с помощью unwrap(). Методы getMIC() и verifyMIC() можно использовать для создания и проверки «кода целостности сообщения», чтобы защитить целостность передаваемых незашифрованных данных.

```
public interface GSSContext {
// Открытые константы
   public static final int DEFAULT_LIFETIME;                    // =0
   public static final int INDEFINITE_LIFETIME;                 // =2147483647
// Методы доступа к свойствам (по имени свойства)
   public abstract boolean getAnonymityState();
```

```
   public abstract boolean getConfState();
   public abstract boolean getCredDelegState();
   public abstract GSSCredential getDelegCred() throws GSSException;
   public abstract boolean isEstablished();
   public abstract boolean isInitiator() throws GSSException;
   public abstract boolean getIntegState();
   public abstract int getLifetime();
   public abstract Oid getMech() throws GSSException;
   public abstract boolean getMutualAuthState();
   public abstract boolean isProtReady();
   public abstract boolean getReplayDetState();
   public abstract boolean getSequenceDetState();
   public abstract GSSName getSrcName() throws GSSException;
   public abstract GSSName getTargName() throws GSSException;
   public abstract boolean isTransferable() throws GSSException;
// Открытые методы экземпляра
   public abstract void acceptSecContext(java.io.InputStream inStream,
        java.io.OutputStream outStream) throws GSSException;
   public abstract byte[] acceptSecContext(byte[] inToken, int offset, int len)
        throws GSSException;
   public abstract void dispose() throws GSSException;
   public abstract byte[] export() throws GSSException;
   public abstract void getMIC(java.io.InputStream inStream, java.io.OutputStream outStream,
                                              MessageProp msgProp) throws GSSException;
   public abstract byte[] getMIC(byte[] inMsg, int offset, int len, MessageProp msgProp)
        throws GSSException;
   public abstract int getWrapSizeLimit(int qop, boolean confReq, int maxTokenSize)
        throws GSSException;
   public abstract int initSecContext(java.io.InputStream inStream,
        java.io.OutputStream outStream) throws GSSException;
   public abstract byte[] initSecContext(byte[] inputBuf, int offset, int len) throws GSSException;
   public abstract void requestAnonymity(boolean state) throws GSSException;
   public abstract void requestConf(boolean state) throws GSSException;
   public abstract void requestCredDeleg(boolean state) throws GSSException;
   public abstract void requestInteg(boolean state) throws GSSException;
   public abstract void requestLifetime(int lifetime) throws GSSException;
   public abstract void requestMutualAuth(boolean state) throws GSSException;
   public abstract void requestReplayDet(boolean state) throws GSSException;
   public abstract void requestSequenceDet(boolean state) throws GSSException;
   public abstract void setChannelBinding(ChannelBinding cb) throws GSSException;
   public abstract void unwrap(java.io.InputStream inStream, java.io.OutputStream outStream,
                                              MessageProp msgProp) throws GSSException;
   public abstract byte[] unwrap(byte[] inBuf, int offset, int len, MessageProp msgProp)
                                                        throws GSSException;
   public abstract void verifyMIC(java.io.InputStream tokStream, java.io.InputStream msgStream,
                                               MessageProp msgProp) throws GSSException;
   public abstract void verifyMIC(byte[] inToken, int tokOffset, int tokLen, byte[] inMsg,
          int msgOffset, int msgLen, MessageProp msgProp) throws GSSException;
   public abstract void wrap(java.io.InputStream inStream, java.io.OutputStream outStream,
                                           MessageProp msgProp) throws GSSException;
   public abstract byte[] wrap(byte[] inBuf, int offset, int len, MessageProp msgProp)
                                                          throws GSSException;
}
```

Возвращается методами: GSSManager.createContext()

GSSCredential Java 1.4

org.ietf.jgss *клонируемый*

Этот интерфейс определяет API высокого уровня для общих защитных сертификатов, таких как паспорт или ключ Kerberos.


```
public interface GSSCredential extends Cloneable {
// Открытые константы
   public static final int ACCEPT_ONLY;                                    // =2
   public static final int DEFAULT_LIFETIME;                               // =0
   public static final int INDEFINITE_LIFETIME;                            // =2147483647
   public static final int INITIATE_AND_ACCEPT;                            // =0
   public static final int INITIATE_ONLY;                                  // =1
// Открытые методы экземпляра
   public abstract void add(GSSName name, int initLifetime, int acceptLifetime, Oid mech,
                  int usage) throws GSSException;
   public abstract void dispose() throws GSSException;
   public abstract boolean equals(Object another);
   public abstract Oid[] getMechs() throws GSSException;
   public abstract GSSName getName() throws GSSException;
   public abstract GSSName getName(Oid mech) throws GSSException;
   public abstract int getRemainingAcceptLifetime(Oid mech) throws GSSException;
   public abstract int getRemainingInitLifetime(Oid mech) throws GSSException;
   public abstract int getRemainingLifetime() throws GSSException;
   public abstract int getUsage() throws GSSException;
   public abstract int getUsage(Oid mech) throws GSSException;
   public abstract int hashCode();
}
```

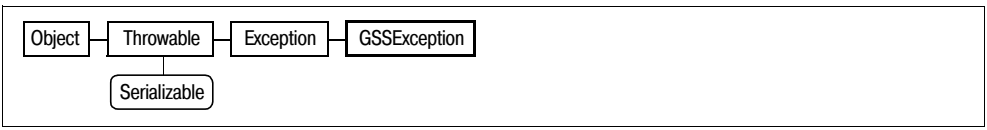
Передается методам: `GSSManager.createContext()`

Возвращается методами: `GSSContext.getDelegCred()`, `GSSManager.createCredential()`

GSSException Java 1.4

org.ietf.jgss *сериализуемое, проверяемое*

Данный класс исключений сигнализирует обо всех исключениях пакета `org.ietf.jgss`. Метод `getMajor()` возвращает код ошибки GSS, который должен быть одной из целочисленных констант, определяемых этим классом. Метод `getMajorString()` возвращает сообщение об ошибке, соответствующее этому коду ошибки. Методы `getMinor()` и `getMinorString()` возвращают код ошибки и строку ошибки, характерные для данного базового механизма защиты.




```
public class GSSException extends Exception {
// Открытые конструкторы
```

```
   public GSSException(int majorCode);
   public GSSException(int majorCode, int minorCode, String minorString);
// Открытые константы
   public static final int BAD_BINDINGS;                                    // =1
   public static final int BAD_MECH;                                        // =2
   public static final int BAD_MIC;                                         // =6
   public static final int BAD_NAME;                                        // =3
   public static final int BAD_NAMETYPE;                                    // =4
   public static final int BAD_QOP;                                         // =14
   public static final int BAD_STATUS;                                      // =5
   public static final int CONTEXT_EXPIRED;                                 // =7
   public static final int CREDENTIALS_EXPIRED;                             // =8
   public static final int DEFECTIVE_CREDENTIAL;                            // =9
   public static final int DEFECTIVE_TOKEN;                                 // =10
   public static final int DUPLICATE_ELEMENT;                               // =17
   public static final int DUPLICATE_TOKEN;                                 // =19
   public static final int FAILURE;                                         // =11
   public static final int GAP_TOKEN;                                       // =22
   public static final int NAME_NOT_MN;                                     // =18
   public static final int NO_CONTEXT;                                      // =12
   public static final int NO_CRED;                                         // =13
   public static final int OLD_TOKEN;                                       // =20
   public static final int UNAUTHORIZED;                                    // =15
   public static final int UNAVAILABLE;                                     // =16
   public static final int UNSEQ_TOKEN;                                     // =21
// Открытые методы экземпляра
   public int getMajor();
   public String getMajorString();
   public int getMinor();
   public String getMinorString();
   public void setMinor(int minorCode, String message);
// Открытые методы, замещающие Throwable
   public String getMessage();
   public String toString();
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

GSSManager Java 1.4

org.ietf.jgss

Это центральный класс-фабрика пакета `org.ietf.jgss`. Предопределенный `GSSManager` можно получить с помощью вызова статического метода `getInstance()` или создания экземпляра какого-либо подкласса от поставщика. С помощью метода `getMechs()` можно выяснить, какие механизмы защиты поддерживаются менеджером. (По умолчанию механизм `GSSManager` компании Sun поддерживает только механизм Kerberos.) Чтобы создать `GSSContext`, вызовите `createContext()`. Для создания `GSSCredential` вызовите `createCredential()`. Чтобы создать `GSSName`, вызовите `createName()`.

```
public abstract class GSSManager {
// Открытые конструкторы
   public GSSManager();
// Открытые методы класса
   public static GSSManager getInstance();
// Открытые методы экземпляра
```

```
    public abstract void addProviderAtEnd(java.security.Provider p, Oid mech) throws GSSException;
    public abstract void addProviderAtFront(java.security.Provider p, Oid mech) throws GSSException;
    public abstract GSSContext createContext(GSSCredential myCred) throws GSSException;
    public abstract GSSContext createContext(byte[] interProcessToken) throws GSSException;
    public abstract GSSContext createContext(GSSName peer, Oid mech, GSSCredential myCred,
        int lifetime) throws GSSException;
    public abstract GSSCredential createCredential(int usage) throws GSSException;
    public abstract GSSCredential createCredential(GSSName name, int lifetime, Oid mech, int usage)
        throws GSSException;
    public abstract GSSCredential createCredential(GSSName name, int lifetime, Oid[] mechs,
        int usage) throws GSSException;
    public abstract GSSName createName(String nameStr, Oid nameType) throws GSSException;
    public abstract GSSName createName(byte[] name, Oid nameType) throws GSSException;
    public abstract GSSName createName(String nameStr, Oid nameType, Oid mech) throws GSSException;
    public abstract GSSName createName(byte[] name, Oid nameType, Oid mech) throws GSSException;
    public abstract Oid[] getMechs();
    public abstract Oid[] getMechsForName(Oid nameType);
    public abstract Oid[] getNamesForMech(Oid mech) throws GSSException;
}
```

Возвращается методами: GSSManager.getInstance()

GSSName

Java 1.4

org.ietf.jgss

GSSName представляет участника сеанса GSS, такого как пользователь или сетевой сервер. Константы, определяемые этим классом, представляют допустимое множество «типов имен».

```
public interface GSSName {
// Открытые константы
    public static final Oid NT_ANONYMOUS;
    public static final Oid NT_EXPORT_NAME;
    public static final Oid NT_HOSTBASED_SERVICE;
    public static final Oid NT_MACHINE_UID_NAME;
    public static final Oid NT_STRING_UID_NAME;
    public static final Oid NT_USER_NAME;
// Открытые методы экземпляра
    public abstract GSSName canonicalize(Oid mech) throws GSSException;
    public abstract boolean equals(Object another);
    public abstract boolean equals(GSSName another) throws GSSException;
    public abstract byte[] export() throws GSSException;
    public abstract Oid getStringNameType() throws GSSException;
    public abstract int hashCode();
    public abstract boolean isAnonymous();
    public abstract boolean isMN();
    public abstract String toString();
}
```

Передается методам: GSSCredential.add(), GSSManager.{createContext(), createCredential()}, GSSName.equals()

Возвращается методами: GSSContext.{getSrcName(), getTargName()}, GSSCredential.getName(), GSSManager.createName(), GSSName.canonicalize()

MessageProp Java 1.4

org.ietf.jgss

Данный вспомогательный класс, используемый с методами `wrap()`, `unwrap()`, `getMIC()` и `verifyMIC()` контекста `GSSContext`, передает и возвращает свойства сообщения или код целостности сообщения.

```
public class MessageProp {
// Открытые конструкторы
   public MessageProp(boolean privState);
   public MessageProp(int qop, boolean privState);
// Методы доступа к свойствам (по имени свойства)
   public boolean isDuplicateToken();
   public boolean isGapToken();
   public int getMinorStatus();
   public String getMinorString();
   public boolean isOldToken();
   public boolean getPrivacy();
   public void setPrivacy(boolean privState);
   public int getQOP();
   public void setQOP(int qop);
   public boolean isUnseqToken();
// Открытые методы экземпляра
   public void setSupplementaryStates(boolean duplicate, boolean old, boolean unseq, boolean gap,
                                      int minorStatus, String minorString);
}
```

Передается методам: `GSSContext.{getMIC(), unwrap(), verifyMIC(), wrap()}`

Oid Java 1.4

org.ietf.jgss

`Oid` является универсальным «идентификатором объекта» на базе иерархической системы нумерации. Класс `Oid` в данном пакете идентифицирует механизмы защиты и имена типов. `Oid` механизма защиты Kerberos – 1.2.840.113554.1.2.2.

```
public class Oid {
// Открытые конструкторы
   public Oid(String strOid) throws GSSException;
   public Oid(java.io.InputStream derOid) throws GSSException;
   public Oid(byte[] data) throws GSSException;
// Открытые методы экземпляра
   public boolean containedIn(Oid[] oids);
   public byte[] getDER() throws GSSException;
// Открытые методы, замещающие Object
   public boolean equals(Object other);
   public int hashCode();
   public String toString();
}
```

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: `GSSContext.getMech()`, `GSSCredential.getMechs()`, `GSSManager.{getMechs(), getMechsForName(), getNamesForMech()}`, `GSSName.getStringNameType()`

Экземпляры: `GSSName.{NT_ANONYMOUS, NT_EXPORT_NAME, NT_HOSTBASED_SERVICE, NT_MACHINE_UID_NAME, NT_STRING_UID_NAME, NT_USER_NAME}`

Глава 23

org.w3c.dom

Пакет org.w3c.dom

Java 1.4

Данный пакет определяет связывание Java с ядром и модулями XML для API стандарта DOM второго уровня. Этот API определяется Консорциумом производителей программного обеспечения для WWW (World Wide Web Consortium, W3C). DOM означает объектную модель документа (Document Object Model), а API стандарта DOM задает способ представления документа XML в виде дерева с узлами. Сюда входят методы, позволяющие обходить, изучать, изменять и строить с нуля деревья документов. Узел (`Node`) является центральным интерфейсом пакета. Данный интерфейс определяет основные методы обхода и изменения дерева с узлами. Он реализуется всеми узлами дерева документа. Большинство других интерфейсов пакета являются расширением узла `Node`, представляющего специальные типы содержимого XML. Наиболее важные и широко используемые подынтерфейсы (subinterfaces): `Document`, `Element` и `Text`. Объект `Document` служит корнем дерева документа и определяет методы поиска в дереве элементов с установленным именем тега или атрибутом ID. Интерфейс `Element` представляет элемент или тег XML, содержащий методы манипулирования атрибутами элемента. Интерфейс `Text` представляет фрагмент простого текста внутри `Element` и содержит методы запроса и изменения данного текста. `NodeList` и `DOMImplementation` тоже являются важными интерфейсами, хотя и не расширяют `Node`.

Интерфейсы

```
public interface Attr extends Node;
public interface CDATASection extends Text;
public interface CharacterData extends Node;
public interface Comment extends CharacterData;
public interface Document extends Node;
public interface DocumentFragment extends Node;
public interface DocumentType extends Node;
public interface DOMImplementation;
public interface Element extends Node;
public interface Entity extends Node;
public interface EntityReference extends Node;
public interface NamedNodeMap;
public interface Node;
public interface NodeList;
public interface Notation extends Node;
```

```
public interface ProcessingInstruction extends Node;
public interface Text extends CharacterData;
```

Исключение

```
public class DOMException extends RuntimeException;
```

Attr Java 1.4

org.w3c.dom

Объект Attr представляет атрибут узла Element. Объекты Attr связаны с узлами Element, но они не являются частью дерева документа: метод getParentNode() объекта Attr всегда возвращает null. Чтобы определить, к какому Element относится Attr, используйте getOwnerElement(). Объект Attr можно получить, вызвав метод getAttributeNode() интерфейса Element, а NamedNodeMap всех объектов Attr для элемента можно получить с помощью метода Node getAttributes().

Метод getName() возвращает имя атрибута, а getValue() – значение атрибута в виде строки. Метод getSpecified() возвращает true, если атрибут был явно определен в исходном документе посредством обращения к setValue(), и возвращает false, если атрибут был получен по умолчанию из DTD или какой-либо другой структуры.

XML позволяет атрибутам содержать ссылки на текст и сущность (entity). Метод getValue() возвращает значение атрибута в виде отдельной строки. Однако чтобы узнать точную композицию атрибута, следует изучить потомков узла Attr: они могут содержать узлы Text и/или EntityReference.

В большинстве случаев удобнее работать с атрибутами при помощи таких методов интерфейса Element, как getAttribute() и setAttribute(). Эти методы не используют всю совокупность узлов Attr.


```
public interface Attr extends Node {
// Открытые методы экземпляра
   public abstract String getName();
   public abstract org.w3c.dom.Element getOwnerElement();
   public abstract boolean getSpecified();
   public abstract String getValue();
   public abstract void setValue(String value) throws DOMException;
}
```

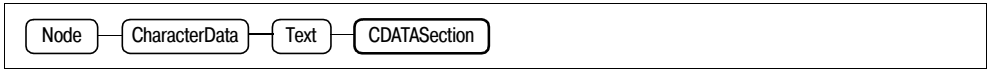
Передается методам: javax.imageio.metadata.IIOMetadataNode.{removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}, org.w3c.dom.Element.{removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}

Возвращается методами: javax.imageio.metadata.IIOMetadataNode.{getAttributeNode(), getAttributeNodeNS(), removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}, org.w3c.dom.Document.{createAttribute(), createAttributeNS()}, org.w3c.dom.Element.{getAttributeNode(), getAttributeNodeNS(), removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}

CDATASection Java 1.4

org.w3c.dom

Данный интерфейс представляет раздел CDATA документа XML. CDATASection является подынтерфейсом Text и не определяет никаких собственных методов. Доступ к содержимому раздела CDATA можно получить с помощью метода getNodeValue(), унаследованного от Node, или метода getData(), унаследованного от CharacterData. Отметим, что метод Node.normalize() не объединяет смежные разделы CDATA, хотя узлы CDATASection часто интерпретируют аналогично узлам Text.




```
public interface CDATASection extends Text {
}
```

Возвращается методами: org.w3c.dom.Document.createCDATASection()

CharacterData Java 1.4

org.w3c.dom

Это общий интерфейс, расширяемый интерфейсами Text, CDATASection (расширяет Text) и Comment. Любой узел дерева документа, реализующий CharacterData, также реализует один из более узких типов. Данный интерфейс предназначен для группирования методов работы со строками, которые являются общими для всех текстовых типов узлов.

Интерфейс CharacterData определяет изменяемую строку. Метод getData() возвращает «символьные данные» в виде объекта String, а setData() позволяет устанавливать эти данные из объекта String. Метод getLength() возвращает количество символьных данных, а substringData() — только заданную часть этих данных в виде строки. Методы appendData(), deleteData(), insertData() и replaceData() изменяют данные. Они позволяют присоединять строку к концу текста, удалять область, вводить строку в указанное место и заменять область на определенную строку.


```
public interface CharacterData extends Node {
// Открытые методы экземпляра
   public abstract void appendData(String arg) throws DOMException;
   public abstract void deleteData(int offset, int count) throws DOMException;
   public abstract String getData() throws DOMException;
   public abstract int getLength();
   public abstract void insertData(int offset, String arg) throws DOMException;
   public abstract void replaceData(int offset, int count, String arg)
                                                        throws DOMException;
   public abstract void setData(String data) throws DOMException;
   public abstract String substringData(int offset, int count) throws DOMException;
}
```

Реализации: Comment, Text

Comment

Java 1.4

org.w3c.dom

Узел Comment представляет комментарий документа XML. Доступ к содержимому комментария (то есть к тексту между <!- - и - ->) можно получить с помощью метода getData(), унаследованного от CharacterData, или метода getNodeValue(), унаследованного от Node. Этим содержимым можно управлять с помощью различных методов, унаследованных от CharacterData.


```
public interface Comment extends CharacterData {
}
```

Возвращается методами: org.w3c.dom.Document.createComment()

Document

Java 1.4

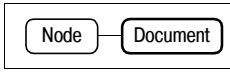
org.w3c.dom

Данный интерфейс представляет документ DOM, а объект, который реализует этот интерфейс, служит корнем дерева документа DOM. Большинство методов, определяемых интерфейсом Document, являются методами-фабриками, которые применяются для создания различных типов узлов, включаемых в этот документ. Существуют две версии методов для создания атрибутов и элементов. Методы, в имени которых присутствует «NS», зависят от пространства имен. Для них необходимо указывать имя атрибута или элемента в виде комбинации пространства имен URI (универсального идентификатора ресурса) и локального имени. Во всем API стандарта DOM методы, содержащие в имени «NS», зависят от пространства имен. Далее описаны другие важные методы.

Метод getElementsByTagName() и его вариант getElementsByTagNameNS(), зависящий от пространства имен, ищут в дереве документа узлы Element с указанным именем тега и возвращают NodeList с перечнем совпавших узлов. Интерфейс Element определяет методы по тем именам, которые участвуют в поиске внутри поддерева, определенного объектом Element.

getElementById() похож на предыдущий метод поиска. В дереве документа он ищет отдельный элемент с указанным уникальным значением атрибута ID. Этот метод пригодится, если вы используете атрибут ID для однозначной идентификации определенных тегов внутри документа XML. Данный метод не ищет атрибуты с именем «id» или «ID». Он ищет атрибуты с типом ID языка XML, как объявлено в DTD документа. Такие атрибуты часто называют «id», хотя это необязательно.

Документ XML должен содержать элемент с одним корнем. Метод getDocumentElement() возвращает объект Element. Это не означает, что узел Document имеет только один дочерний элемент. У него должен быть только один дочерний элемент Element, но могут быть и другие элементы, такие как узлы Comment и ProcessingInstruction. Метод getDoctype() возвращает объект DocumentType, который представляет DTD документа, или null, если объект отсутствует. Метод getImplementation() возвращает объект DOMImplementation, который представляет реализацию DOM, создавшую данное дерево документа.




```
public interface Document extends Node {
// Открытые методы экземпляра
   public abstract Attr createAttribute(String name) throws DOMException;
   public abstract Attr createAttributeNS(String namespaceURI, String qualifiedName)
        throws DOMException;
   public abstract CDATASection createCDATASection(String data) throws DOMException;
   public abstract Comment createComment(String data);
   public abstract DocumentFragment createDocumentFragment();
   public abstract org.w3c.dom.Element createElement(String tagName) throws DOMException;
   public abstract org.w3c.dom.Element createElementNS(String namespaceURI, String qualifiedName)
        throws DOMException;
   public abstract EntityReference createEntityReference(String name) throws DOMException;
   public abstract ProcessingInstruction createProcessingInstruction(String target, String data)
        throws DOMException;
   public abstract Text createTextNode(String data);
   public abstract DocumentType getDoctype();
   public abstract org.w3c.dom.Element getDocumentElement();
   public abstract org.w3c.dom.Element getElementById(String elementId);
   public abstract NodeList getElementsByTagName(String tagname);
   public abstract NodeList getElementsByTagNameNS(String namespaceURI, String localName);
   public abstract DOMImplementation getImplementation();
   public abstract Node importNode(Node importedNode, boolean deep) throws DOMException;
}
```

Реализации: org.w3c.dom.html.HTMLDocument

Возвращается методами: javax.imageio.metadata.IIOMetadataNode.getOwnerDocument(), javax.xml.parsers.DocumentBuilder.{newDocument(), parse()}, DOMImplementation.createDocument(), Node.getOwnerDocument(), org.w3c.dom.html.HTMLFrameElement.getContentDocument(), org.w3c.dom.html.HTMLIFrameElement.getContentDocument(), org.w3c.dom.html.HTMLObjectElement.getContentDocument()

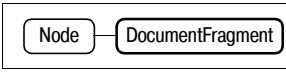
DocumentFragment Java 1.4

org.w3c.dom

Интерфейс DocumentFragment представляет часть – или фрагмент – документа. Точнее, он представляет один или несколько смежных узлов документа со всеми подэлементами. Узлы DocumentFragment не являются частью дерева документа, поэтому getParentNode() всегда возвращает null. Хотя DocumentFragment не имеет родителя, он может иметь дочерние элементы. С помощью унаследованных методов Node можно присоединять узлы к DocumentFragment, удалять или заменять их.

Особое поведение узлов DocumentFragment позволяет выгодно их использовать: если получен запрос о вводе DocumentFragment в дерево документа, то вместо самого узла DocumentFragment вводятся все его подэлементы. Здесь DocumentFragment играет роль временного заполнителя для последовательности узлов, которую в любой момент можно ввести в документ.

Для создания нового, пустого DocumentFragment необходимо вызвать метод createDocumentFragment() выбранного Document.




```
public interface DocumentFragment extends Node {
}
```

Возвращается методами: org.w3c.dom.Document.createDocumentFragment()

DocumentType

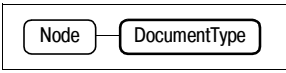
Java 1.4

org.w3c.dom

Данный интерфейс представляет DTD для документа. Поскольку DTD не входит в сам документ, объект DocumentType не является частью дерева документа DOM несмотря на то, что он расширяет интерфейс Node. Если Document содержит DTD, то представляющий его объект DocumentType можно получить вызовом метода getDoctype() объекта Document.

Методы getName(), getPublicId(), getSystemId() и getInternalSubset() возвращают строки, содержащие имя, идентификатор public, системный идентификатор и внутреннее подмножество типа документа. Вместо этого может быть возвращен null. Метод getEntities() возвращает NamedNodeMap, предназначенный только для чтения. Данный NamedNodeMap представляет отображение имен в значения для всех внутренних и внешних общих сущностей (entities), объявленных DTD. С помощью этого NamedNodeMap можно найти объект Entity по имени. Аналогично, метод getNotations() возвращает NamedNodeMap, предназначенный только для чтения, который позволяет отыскивать объект Notation, объявленный в DTD по имени.

DocumentType не предоставляет доступ к большей части DTD, которая обычно состоит из объявлений элементов и атрибутов. Будущие версии API стандарта DOM могут оказаться более содержательными.




```
public interface DocumentType extends Node {
// Открытые методы доступа к свойствам (по имени свойства)
   public abstract NamedNodeMap getEntities();
   public abstract String getInternalSubset();
   public abstract String getName();
   public abstract NamedNodeMap getNotations();
   public abstract String getPublicId();
   public abstract String getSystemId();
}
```

Передается методам: DOMImplementation.createDocument()

Возвращается методами: org.w3c.dom.Document.getDoctype(), DOMImplementation.createDocumentType()

DOMException

Java 1.4

org.w3c.dom ***сериализуемое, непроверяемое***

Каждый раз, когда API стандарта DOM генерирует исключение, создается экземпляр этого класса. В отличие от большинства API Java, API стандарта DOM не определяет

специализированные подклассы для определения различных категорий исключений. Вместо этого открытое поле `code` определяет более узкий тип исключения. Значением этого поля будет одна из констант, определяемых этим классом:

`INDEX_SIZE_ERR`

Указывает на ошибку выхода индекса за пределы массива или строки.

`DOMSTRING_SIZE_ERR`

Указывает на слишком большой для объекта `String` текст. Исключения такого типа предназначены для реализаций DOM в других языках и не должны возникать в Java.

`HIERARCHY_REQUEST_ERR`

Указывает на предпринятую попытку незаконно установить узел в древовидной иерархии документа.

`WRONG_DOCUMENT_ERR`

Указывает на предпринятую попытку использовать узел с документом, отличным от создавшего этот узел документа.

`INVALID_CHARACTER_ERR`

Указывает на использование запрещенного символа (например, в имени элемента).

`NO_DATA_ALLOWED_ERR`

В настоящее время не используется.

`NO_MODIFICATION_ALLOWED_ERR`

Указывает на предпринятую попытку модифицировать узел, предназначенный только для чтения и не допускающий модификацию. Узлы `Entity`, `EntityReference` и `Notation`, а также все их подэлементы предназначены только для чтения.

`NOT_FOUND_ERR`

Указывает на отсутствие узла в предполагаемом месте его нахождения.

`NOT_SUPPORTED_ERR`

Указывает, что данная реализация DOM не поддерживает метод или свойство.

`INUSE_ATTRIBUTE_ERR`

Указывает на предпринятую попытку связать узел `Attr` с узлом `Element`, если данный `Attr` уже связан с другим `Element`.

`INVALID_STATE_ERR`

Указывает на предпринятую попытку использовать объект в состоянии, не позволяющем такое использование.

`SYNTAX_ERR`

Указывает на синтаксическую ошибку в обозначенной строке. Описываемый здесь оперативный модуль API стандарта DOM не генерирует исключения такого типа.

`INVALID_MODIFICATION_ERR`

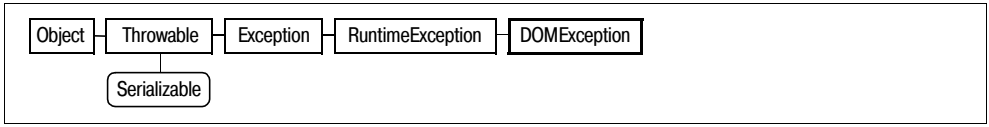
Описываемый здесь оперативный модуль API стандарта DOM не генерирует исключения такого типа.

`NAMESPACE_ERR`

Указывает на ошибку в пространстве имен элемента или атрибута.

`INVALID_ACCESS_ERR`

Указывает на предпринятую попытку получить доступ к объекту способом, который не поддерживается в данной реализации.




```
public class DOMException extends RuntimeException {
// Открытые конструкторы
   public DOMException(short code, String message);
// Открытые константы
   public static final short DOMSTRING_SIZE_ERR;                    // =2
   public static final short HIERARCHY_REQUEST_ERR;                 // =3
   public static final short INDEX_SIZE_ERR;                        // =1
   public static final short INUSE_ATTRIBUTE_ERR;                   // =10
   public static final short INVALID_ACCESS_ERR;                    // =15
   public static final short INVALID_CHARACTER_ERR;                 // =5
   public static final short INVALID_MODIFICATION_ERR;              // =13
   public static final short INVALID_STATE_ERR;                     // =11
   public static final short NAMESPACE_ERR;                         // =14
   public static final short NO_DATA_ALLOWED_ERR;                   // =6
   public static final short NO_MODIFICATION_ALLOWED_ERR;           // =7
   public static final short NOT_FOUND_ERR;                         // =8
   public static final short NOT_SUPPORTED_ERR;                     // =9
   public static final short SYNTAX_ERR;                            // =12
   public static final short WRONG_DOCUMENT_ERR;                    // =4
// Открытые поля экземпляра
   public short code;
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

DOMImplementation Java 1.4

org.w3c.dom

Данный интерфейс определяет методы, глобальные для реализации DOM, а не методы, характерные для отдельного объекта `Document`. Ссылку на объект `DOMImplementation`, представляющий вашу реализацию, можно получить с помощью вызова метода `getImplementation()` любого объекта `Document`. Метод `createDocument()` возвращает новый, пустой объект `Document`, который можно заполнить узлами, создаваемыми при помощи методов `create`, определяемых интерфейсом `Document`.

Метод `hasFeature()` позволяет проверить, поддерживает ли ваша реализация DOM указанную версию функции или модуля стандарта DOM. Данный метод должен возвращать `true` при передаче имени функции «core» и версии «1.0», а также при передаче имени функции «core» или «xml» и версии «2.0». Стандарт DOM содержит несколько необязательных модулей, однако в платформе Java не настроены подпакеты этого пакета, определяющие API этих необязательных модулей, поэтому маловероятно, что связанные реализации DOM и Java будут поддерживать такие модули.

С помощью класса `javax.xml.parsers.DocumentBuilder` можно получить объект `DOMImplementation`, вызвав `getDOMImplementation()`. Кроме того, данный класс определяет метод `newDocument()` для создания пустых объектов `Document`.

```
public interface DOMImplementation {
// Открытые методы экземпляра
   public abstract org.w3c.dom.Document createDocument(String namespaceURI, String qualifiedName,
```

```
                                DocumentType doctype) throws DOMException;
   public abstract DocumentType createDocumentType(String qualifiedName, String publicId,
                                String systemId) throws DOMException;
   public abstract boolean hasFeature(String feature, String version);
}
```

Реализации: org.w3c.dom.css.DOMImplementationCSS, org.w3c.dom.html.HTMLDOMImplementation

Возвращается методами: javax.xml.parsers.DocumentBuilder.getDOMImplementation(), org.w3c.dom.Document.getImplementation()

Element

Java 1.4

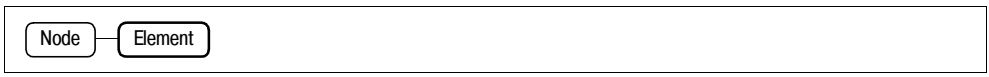
org.w3c.dom

Данный интерфейс представляет элемент (или тег) документа XML. Метод getTagName() возвращает имя тега элемента, включая префикс пространства имен, если такой существует. В работе с пространством имен лучше использовать методы, зависящие от пространства, которые определяются интерфейсом Node. Получить URI пространства имен для элемента можно с помощью метода getNamespaceURI(), а локальное имя элемента внутри этого пространства имен – методом getLocalName(). Метод getPrefix() позволяет получить префикс пространства имен, а метод setPrefix() – изменить его (при этом URI пространства имен не меняется).

Element определяет метод getElementsByTagName() и соответствующий ему метод getElementsByTagNameNS(), зависящий от пространства имен. Их поведение ничем не отличается от поведения методов с такими же именами в объекте Document, за исключением того, что описываемые методы ищут именованные элементы только внутри поддерева, расположенного в корне данного Element.

Оставшиеся методы интерфейса Element предназначены для получения и установки значений атрибутов, тестирования на наличие атрибута и удаления атрибута из Element. Существует большое количество методов выполнения четырех основных операций над атрибутами. Метод работы с атрибутом зависит от пространства имен, если его имя содержит «NS». Если имя метода содержит «Node», то метод работает с объектами Attr, а не с более простыми строковыми представлениями значения атрибута. Атрибуты в документах XML могут содержать ссылки на сущности (entity references). Если в значениях атрибутов представлены ссылки на сущности, то лучше использовать интерфейс Attr, поскольку расширение такой ссылки на сущность может привести к созданию для объекта Attr поддерева с узлами. Однако намного легче работать с методами, которые интерпретируют значения атрибутов как простые строки. Кроме методов атрибутов, определяемых интерфейсом Element, можно получить NamedNodeMap объектов Attr с помощью метода getAttributes() интерфейса Node.

Наконец, getAttribute() и подобные ему методы, а также hasAttribute() и подобные ему методы возвращают значение или проверяют наличие как явно заданных атрибутов, так и атрибутов со значениями по умолчанию, определяемыми в DTD документа. Чтобы выяснить, был ли атрибут явно указан в документе, нужно получить его объект Attr и вызвать метод getSpecified().




```
public interface Element extends Node {
// Открытые методы экземпляра
   public abstract String getAttribute(String name);
```

```
   public abstract Attr getAttributeNode(String name);
   public abstract Attr getAttributeNodeNS(String namespaceURI, String localName);
   public abstract String getAttributeNS(String namespaceURI, String localName);
   public abstract NodeList getElementsByTagName(String name);
   public abstract NodeList getElementsByTagNameNS(String namespaceURI, String localName);
   public abstract String getTagName();
   public abstract boolean hasAttribute(String name);
   public abstract boolean hasAttributeNS(String namespaceURI, String localName);
   public abstract void removeAttribute(String name) throws DOMException;
   public abstract Attr removeAttributeNode(Attr oldAttr) throws DOMException;
   public abstract void removeAttributeNS(String namespaceURI, String localName) throws DOMException;
   public abstract void setAttribute(String name, String value) throws DOMException;
   public abstract Attr setAttributeNode(Attr newAttr) throws DOMException;
   public abstract Attr setAttributeNodeNS(Attr newAttr) throws DOMException;
   public abstract void setAttributeNS(String namespaceURI, String qualifiedName, String value)
         throws DOMException;
}
```

Реализации: `javax.imageio.metadata.IIOMetadataNode`, `org.w3c.dom.html.HTMLElement`

Передается методам: `org.w3c.dom.css.DocumentCSS.getOverrideStyle()`, `org.w3c.dom.css.ViewCSS.getComputedStyle()`

Возвращается методами: `Attr.getOwnerElement()`, `org.w3c.dom.Document.{createElement(), createElementNS(), getDocumentElement(), getElementById()}`

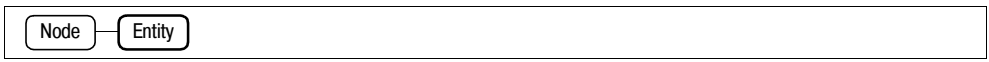
Entity Java 1.4

org.w3c.dom

Данный интерфейс представляет сущность, определяемую в документе DTD формата XML. Унаследованный из интерфейса `Node` метод `getNodeName()` определяет имя сущности. Узлы-потомки узла `Entity` представляют содержимое сущности. Методы, определяемые данным интерфейсом, возвращают открытый и системный идентификаторы для внешних сущностей, а также название нотации для сущностей, не подлежащих разбору. Узлы `Entity` и их подэлементы не входят в дерево документа (метод `Entity getParentNode()` всегда возвращает `null`). Вместо этого документ может содержать одну или несколько ссылок на сущность. См. интерфейс `EntityReference`.

В DTD документа сущности определяются как часть внешнего файла DTD или как часть «внутреннего подмножества», которое определяет локальные сущности, характерные для данного документа. Интерфейс `DocumentType` содержит метод `getEntities()`, который возвращает `NamedNodeMap`, отображая имена сущностей на узлы `Entity`. Это единственный способ получить объект `Entity`; узлы `Entity` никогда не появляются в самом дереве документа, поскольку они являются частью DTD.

Узлы `Entity` и все их подэлементы предназначены только для чтения; их нельзя редактировать или модифицировать.




```
public interface Entity extends Node {
// Открытые методы экземпляра
   public abstract String getNotationName();
   public abstract String getPublicId();
   public abstract String getSystemId();
}
```

EntityReference Java 1.4

org.w3c.dom

Данный интерфейс представляет ссылку из документа XML на сущность, определяемую в DTD документа. В документах XML символьные сущности и такие предопределенные сущности, как <, всегда расширяемы; они не создают узлы EntityReference. Некоторые парсеры XML расширяют все ссылки на сущности. Документы, создаваемые таким парсером, не содержат узлы EntityReference.

Данный интерфейс не определяет собственные методы. Метод getNodeName() интерфейса Node предоставляет имя сущности, на которую производится ссылка. Метод getEntities() интерфейса DocumentType позволяет отыскать объект Entity, связанный с этим именем. Однако DocumentType может не содержать Entity с указанным именем (например, если для разбора внешнего подмножества DTD не нужны парсеры XML, которые не выполняют подтверждение). В таком случае EntityReference является ссылкой на именованную сущность с неизвестным содержимым и не содержит подэлементов. С другой стороны, если DocumentType содержит узел Entity с указанным именем, то узлы EntityReference являются копиями узлов Entity и представляют расширение сущности. (Если расширение сущности включает префиксы пространства имен, не связанные с URI пространства имен, то потомки EntityReference могут не быть точными копиями потомков Entity.)

Подобно узлам Entity, узлы EntityReference и их подэлементы предназначены только для чтения; их нельзя редактировать и модифицировать.


```
public interface EntityReference extends Node {
}
```

Возвращается методами: org.w3c.dom.Document.createEntityReference()

NamedNodeMap Java 1.4

org.w3c.dom

Интерфейс NamedNodeMap определяет совокупность узлов, которые можно найти по имени или по URI пространства имен и локальному имени. Он не связан с интерфейсом java.util.Map. С помощью метода getNamedItem() можно найти и возвратить узел, метод getNodeName() которого возвращает указанное значение. С помощью getNamedItemNS() можно найти и возвратить узел, методы getNamespaceURI() и getLocalName() которого возвращают указанные значения. NamedNodeMap отображает имена на узлы и не устанавливает узлы в каком-либо определенном порядке. Тем не менее он устанавливает произвольный порядок для узлов и допускает их поиск по индексу. Метод getLength() позволяет найти количество узлов, содержащихся в NamedNodeMap, а с помощью метода item() можно получить объект Node по указанному индексу.

Если NamedNodeMap предназначен не только для чтения, то с помощью removeNamedItem() и removeNamedItemNS() можно удалить именованный узел из отображения. Метод setNamedItem() позволяет присоединить узел к отображению, отображая его имя или URI пространства имен и локального имени.

Объекты NamedNodeMap – «живые», поскольку они немедленно отображают любые изменения в дереве документа. Например, если получить NamedNodeMap, представляющий атрибуты элемента, а затем к этому элементу присоединить новый атрибут, то новый атрибут автоматически становится доступным через NamedNodeMap: не нужно обновлять NamedNodeMap, чтобы получить модифицированное множество атрибутов.

Лишь некоторые методы API стандарта DOM могут возвращать NamedNodeMap. Наиболее яркий пример использования – возвращаемое значение метода getAttributes() интерфейса Node. Однако с атрибутами обычно легче работать с помощью методов интерфейса Element. Два метода DocumentType возвращают объекты NamedNodeMap, предназначенные только для чтения.

```
public interface NamedNodeMap {
// Открытые методы экземпляра
   public abstract int getLength();
   public abstract Node getNamedItem(String name);
   public abstract Node getNamedItemNS(String namespaceURI, String localName);
   public abstract Node item(int index);
   public abstract Node removeNamedItem(String name) throws DOMException;
   public abstract Node removeNamedItemNS(String namespaceURI, String localName)
        throws DOMException;
   public abstract Node setNamedItem(Node arg) throws DOMException;
   public abstract Node setNamedItemNS(Node arg) throws DOMException;
}
```

Возвращается методами: javax.imageio.metadata.IIOMetadataNode.getAttributes(), DocumentType.{getEntities(), getNotations()}, Node.getAttributes()

Node

Java 1.4

org.w3c.dom

Все объекты дерева документа DOM (включая и сам объект Document) реализуют интерфейс Node, в котором предусмотрены основные методы навигации и манипулирования деревом.

Методы getParentNode() и getChildNodes() позволяют проходить вверх и вниз по дереву. Чтобы перечислить подэлементы данного узла, можно пройти по циклу элементов NodeList, возвращаемых getChildNodes(), или вернуться к началу цикла с помощью методов getFirstChild() и getNextSibling() (или getLastChild() и getPreviousSibling()). Чтобы проверить узел на наличие потомков, можно вызвать hasChildNodes(). Метод getOwnerDocument() возвращает узел Document, потомком которого является данный узел или с которым он связан. Это позволяет быстро переходить к корню дерева документа.

Несколько методов позволяют присоединять элементы к дереву или изменять список элементов. Метод appendChild() присоединяет узел к концу этого списка узлов. Метод insertChild() вставляет узел в этот список узлов перед указанным узлом. Метод removeChild() удаляет указанный узел из этого списка. Метод replaceChild() заменяет один подэлемент данного узла на другой. Для всех этих методов верно следующее: если присоединяемый или вводимый узел является частью дерева документа, то сначала его удаляют из текущего родителя. Копию данного узла можно вывести с помощью cloneNode(). Если вы хотите клонировать все подэлементы данного узла, следует передать true.

Помимо интерфейса Node, каждый объект дерева документа реализует один из специализированных подынтерфейсов, например Element или Text. Метод getNodeType() позволяет легко определить подынтерфейс, реализуемый узлом: возвращаемое значение является одной из констант _NODE, определяемых данным классом. Например, определить возможный способ обработки узла неизвестного типа можно на основе возвращаемого значения getNodeType().

Методы getNodeName() и getNodeValue() предоставляют дополнительную информацию об узле, однако в нижеприведенной таблице показано, что возвращаемые ими строки интерпретируются в зависимости от типа узла. Обычно для получения такой информации подынтерфейс определяет специализированные методы (такие как метод getTagName() интерфейса Element и метод getData() интерфейса Text). Если узел не предназначен только для чтения, то с помощью метода setNodeValue() можно изменить значение, связанное с узлом.

Тип узла	Название узла	Значение узла
ELEMENT_NODE	Имя тега элемента	null
ATTRIBUTE_NODE	Имя атрибута	Значение атрибута
TEXT_NODE	«#text»	Текст узла
CDATA_SECTION_NODE	«#cdata-section»	Текст узла
ENTITY_REFERENCE_NODE	Имя ссылочной сущности	null
ENTITY_NODE	Имя сущности	null
PROCESSING_INSTRUCTION_NODE	Целевой объект инструкции обработки	Остаток иструкции обработки
COMMENT_NODE	«#comment»	Текст комментария
DOCUMENT_NODE	«#document»	null
DOCUMENT_TYPE_NODE	Имя типа документа	null
DOCUMENT_FRAGMENT_NODE	«#document-fragment»	null
NOTATION_NODE	Название нотации	null

В документах с пространствами имен метод getNodeName() узла Element или Attr возвращает уточненное имя узла, которое может включать префикс пространства имен. В документах с пространствами имен можно использовать методы getNamespaceURI(), getLocalName() и getPrefix(), зависящие от пространства.

Узлы Element могут содержать список атрибутов, а интерфейс Element определяет несколько методов для работы с этими атрибутами. Помимо этого, Node определяет метод hasAttributes(), который определяет наличие атрибутов в узле. Если узел содержит атрибуты, их можно извлечь с помощью getAttributes().

Содержимое документа XML представлено узлами Text, которые содержат методы манипулирования данным текстом. Интерфейс Node определяет метод normalize(), который предназначен для нормализации всех подэлементов узла путем удаления пустых узлов Text и объединения смежных узлов Text в единый составной узел. Обычно деревья документа имеют нормализованную форму с самого начала, однако модификации дерева могут привести к денормализации документов.

Большинство других интерфейсов данного пакета расширяют Node. Чаще всего применяют Document, Element и Text.

```
public interface Node {
// Открытые константы
   public static final short ATTRIBUTE_NODE;                                    // =2
   public static final short CDATA_SECTION_NODE;                                // =4
   public static final short COMMENT_NODE;                                      // =8
   public static final short DOCUMENT_FRAGMENT_NODE;                            // =11
   public static final short DOCUMENT_NODE;                                     // =9
   public static final short DOCUMENT_TYPE_NODE;                                // =10
   public static final short ELEMENT_NODE;                                      // =1
   public static final short ENTITY_NODE;                                       // =6
   public static final short ENTITY_REFERENCE_NODE;                             // =5
   public static final short NOTATION_NODE;                                     // =12
   public static final short PROCESSING_INSTRUCTION_NODE;                       // =7
   public static final short TEXT_NODE;                                         // =3
// Открытые методы доступа к свойствам (по имени свойства)
   public abstract NamedNodeMap getAttributes();
   public abstract NodeList getChildNodes();
   public abstract Node getFirstChild();
   public abstract Node getLastChild();
   public abstract String getLocalName();
   public abstract String getNamespaceURI();
   public abstract Node getNextSibling();
   public abstract String getNodeName();
   public abstract short getNodeType();
   public abstract String getNodeValue() throws DOMException;
   public abstract void setNodeValue(String nodeValue) throws DOMException;
   public abstract org.w3c.dom.Document getOwnerDocument();
   public abstract Node getParentNode();
   public abstract String getPrefix();
   public abstract void setPrefix(String prefix) throws DOMException;
   public abstract Node getPreviousSibling();
// Открытые методы экземпляра
   public abstract Node appendChild(Node newChild) throws DOMException;
   public abstract Node cloneNode(boolean deep);
   public abstract boolean hasAttributes();
   public abstract boolean hasChildNodes();
   public abstract Node insertBefore(Node newChild, Node refChild) throws DOMException;
   public abstract boolean isSupported(String feature, String version);
   public abstract void normalize();
   public abstract Node removeChild(Node oldChild) throws DOMException;
   public abstract Node replaceChild(Node newChild, Node oldChild) throws DOMException;
}
```

Реализации: `Attr`, `CharacterData`, `org.w3c.dom.Document`, `DocumentFragment`, `DocumentType`, `org.w3c.dom.Element`, `org.w3c.dom.Entity`, `EntityReference`, `Notation`, `ProcessingInstruction`

Передается методам: Методов слишком много, чтобы их перечислить.

Возвращается методами: Методов слишком много, чтобы их перечислить.

Экземпляры: `javax.imageio.metadata.IIOInvalidTreeException.offendingNode`

NodeList Java 1.4

org.w3c.dom

Данный интерфейс представляет упорядоченное множество узлов, предназначенных только для чтения. Метод `getLength()` возвращает количество узлов в списке, а `item()` возвращает `Node` по указанному индексу (индекс первого узла равен 0). Элементы `NodeList` – это подтверждаемые объекты `Node`: `NodeList` никогда не содержит элементы `null`.

Заметим, что объекты `NodeList` – «живые», то есть они не являются статическими и немедленно отображают изменения в дереве документа. Например, если удалить одного их потомков узла, представленных в `NodeList`, то этот потомок будет удален и из `NodeList`. Будьте осторожны при выполнении итерации по элементам `NodeList`, поскольку изменения дерева документа, производимые в теле цикла (например, удаление узлов), могут отразиться на содержимом `NodeList`.

```
public interface NodeList {
// Открытые методы экземпляра
   public abstract int getLength();
   public abstract Node item(int index);
}
```

Реализации: `javax.imageio.metadata.IIOMetadataNode`

Возвращается методами: `javax.imageio.metadata.IIOMetadataNode.{getChildNodes(), getElementsByTagName(), getElementsByTagNameNS()}, org.w3c.dom.Document.{getElementsByTagName(), getElementsByTagNameNS()}, org.w3c.dom.Element.{getElementsByTagName(), getElementsByTagNameNS()}, Node.getChildNodes(), org.w3c.dom.html.HTMLDocument.getElementsByName()`

Notation Java 1.4

org.w3c.dom

Данный интерфейс представляет нотацию, объявляемую в DTD документа XML. В XML нотации предназначены для указания формата сущности, не подлежащей разбору, или для формального объявления целевого объекта обработки.

Метод `getNodeName()` интерфейса `Node` возвращает имя нотации. Методы `getSystemId()` и `getPublicId()` возвращают системный и открытый идентификаторы, указанные в объявлении нотации. Метод `getNotations()` интерфейса `DocumentType` возвращает `NamedNodeMap` объектов `Notation`, объявляемых в DTD, и позволяет отыскивать объекты `Notation` по имени нотации.

Узлы `Notation` не входят в дерево документа, а метод `getParentNode()` всегда возвращает `null`, поскольку нотации находятся в DTD, а не в самом документе. Аналогично, узел `Notation` не имеет подэлементов, а `getChildNodes()` всегда возвращает `null`, поскольку в объявлениях нотаций в XML отсутствует содержимое. Объекты `Notation` предназначены только для чтения и не могут быть модифицированы.


```
public interface Notation extends Node {
// Открытые методы экземпляра
   public abstract String getPublicId();
   public abstract String getSystemId();
}
```

ProcessingInstruction Java 1.4

org.w3c.dom

Данный интерфейс представляет команду обработки (processing instruction, или PI) в XML, которая определяет произвольную строку данных для указанного целевого процессора. Методы `getTarget()` и `getData()` возвращают блоки с данными PI. Кроме того, эти значения можно получить с помощью методов `getNodeName()` и `getNodeValue()` интерфейса `Node`. Блок данных PI можно изменить с помощью метода `setData()` или `setNodeValue()` интерфейса `Node`. Узлы `ProcessingInstruction` не имеют подэлементов.


```
public interface ProcessingInstruction extends Node {
// Открытые методы экземпляра
   public abstract String getData();
   public abstract String getTarget();
   public abstract void setData(String data) throws DOMException;
}
```

Возвращается методами: `org.w3c.dom.Document.createProcessingInstruction()`

Text Java 1.4

org.w3c.dom

Узел `Text` представляет фрагмент простого текста и не содержит разметки формата XML. Текст находится внутри элементов и атрибутов, а узлы `Text` обычно являются подэлементами узлов `Element` и `Attr`. Узлы `Text` наследуются от `CharacterData`, а доступ к текстовому содержимому узла `Text` можно получить с помощью метода `getData()`, унаследованного от `CharacterData`, или метода `getNodeValue()`, унаследованного от `Node`.

Управлять узлами `Text` можно посредством любых методов, унаследованных от `CharacterData`. Интерфейс `Text` определяет собственный метод: `splitText()` разбивает узел `Text` в указанной позиции символа. Метод изменяет исходный узел таким образом, что он будет содержать только текст до указанной позиции. Затем он создает новый узел `Text`, содержащий текст, начиная с указанной позиции, и вводит этот узел в дерево документа сразу же после исходного. Метод `Node.normalize()` выполняет обратную процедуру: он удаляет пустые узлы `Text` и объединяет смежные узлы `Text` в единый узел.

Узлы `Text` не могут иметь подэлементов.


```
public interface Text extends CharacterData {
// Открытые методы экземпляра
   public abstract Text splitText(int offset) throws DOMException;
}
```

Реализации: `CDATASection`

Возвращается методами: `org.w3c.dom.Document.createTextNode()`, `Text.splitText()`

Глава 24

org.xml.sax, org.xml.sax.ext и org.xml.sax.helpers

Данная глава описывает пакет `org.xml.sax` и его подпакеты. Пакет `org.xml.sax` определяет упрощенный API для XML (Simplified API for XML), или SAX, – фактический стандарт разбора документов XML. Пакет `org.xml.sax.ext` определяет необязательные расширения API стандарта SAX, а пакет `org.xml.sax.helpers` – вспомогательные классы, которые часто применяются с SAX. Данные пакеты включены в версию Java 1.4, однако поскольку их определяла не компания Sun, они имеют приставку «org.xml», а не «java».

Пакет org.xml.sax

Java 1.4

Это основной пакет SAX-разбора документов XML. SAX представляет собой API, который «управляется событиями»: SAX-парсер читает документ XML и генерирует поток «событий SAX», описывающий содержимое документа. Данные «события» на самом деле являются вызовами методов одного или нескольких объектов обработчиков, зарегистрированных приложением с помощью парсера. Интерфейс `XMLReader` определяет API, который должен быть реализован парсером SAX. Интерфейсы `ContentHandler`, `ErrorHandler`, `EntityResolver` и `DTDHandler` определяют объекты обработчиков. С помощью `XMLReader` приложение регистрирует объекты, реализующие один или несколько из этих интерфейсов.

Данный пакет определяет интерфейсы SAX1 и SAX2. Интерфейсы `AttributeList`, `DocumentHandler`, `Parser` и класс `HandlerBase` составляют часть API стандарта SAX1, но они признаны устаревшими, а предпочтение отдается `Attributes`, `ContentHandler`, `XMLReader` и `org.xml.sax.helpers.DefaultHandler`.

Интерфейсы

```
public interface AttributeList;
public interface Attributes;
public interface ContentHandler;
public interface DocumentHandler;
public interface DTDHandler;
public interface EntityResolver;
public interface ErrorHandler;
```

```
public interface Locator;
public interface Parser;
public interface XMLFilter extends XMLReader;
public interface XMLReader;
```

Классы

```
public class HandlerBase implements DocumentHandler, DTDHandler, EntityResolver, ErrorHandler;
public class InputSource;
```

Исключения

```
public class SAXException extends Exception;
   └ public class SAXNotRecognizedException extends SAXException;
   └ public class SAXNotSupportedException extends SAXException;
   └ public class SAXParseException extends SAXException;
```

AttributeList

Java 1.4; устарел в Java 1.4

org.xml.sax

Данный интерфейс входит в API стандарта SAX1, однако он признан устаревшим, а предпочтение вместо него отдается интерфейсу `Attributes` стандарта SAX2, поддерживающему пространства имен XML.

```
public interface AttributeList {
// Открытые методы экземпляра
   public abstract int getLength();
   public abstract String getName(int i);
   public abstract String getType(String name);
   public abstract String getType(int i);
   public abstract String getValue(String name);
   public abstract String getValue(int i);
}
```

Реализации: `org.xml.sax.helpers.AttributeListImpl`

Передается методам: `DocumentHandler.startElement()`, `HandlerBase.startElement()`, `org.xml.sax.helpers.AttributeListImpl.{AttributeListImpl(), setAttributeList()}`, `org.xml.sax.helpers.ParserAdapter.startElement()`

Attributes

Java 1.4

org.xml.sax

Данный интерфейс представляет список атрибутов элемента XML и содержит информацию об именах, типах и значениях атрибутов. Если SAX-парсер прочел DTD или схему документа, то список атрибутов будет содержать атрибуты, которые не определены в документе, но имеют значения по умолчанию, указанные в DTD или схеме.

Наиболее широко применяется метод `getValue()`, который возвращает значение указанного атрибута. (Существует еще один вариант этого метода, который возвращает значение пронумерованного атрибута; его мы обсудим позже.) Если SAX-парсер не обрабатывает пространства имен, то можно использовать вариант `getValue()` с одним аргументом. В противном случае при помощи варианта с двумя аргументами следует указать URI, однозначно определяющий пространство имен, и «локальное имя» выбранного атрибута внутри этого пространства. Методы `getType()` похожи на предыдущие, за исключением того, что они возвращают не значение, а тип указанного атри-

бута. Метод getType() может возвращать полезную информацию, если парсер прочел DTD или схему документа и знает тип каждого атрибута.

В документах XML порядок атрибутов тега может быть любым. Объекты атрибутов не стремятся сохранить исходный порядок тегов документа. Тем не менее определенный порядок, позволяющий обрабатывать атрибуты в цикле, все-таки существует. Метод getName() возвращает количество элементов в списке. Версии методов getValue() и getType() возвращают значение и тип атрибута в указанной позиции списка. Кроме того, можно получить имя атрибута в указанной позиции, но способ запроса будет зависеть от способности или неспособности парсера обрабатывать пространства имен. Если он не обрабатывает пространства имен, то имя в указанной позиции можно получить методом getQName(). В противном случае, локальную пару имен для пронумерованного атрибута можно получить с помощью getURI() и getLocalName(). Метод getQName() возвращает пустую строку при включенной обработке пространства имен, а getLocalName() возвратит пустую строку, если обработка пространства имен выключена.

```
public interface Attributes {
// Открытые методы экземпляра
   public abstract int getIndex(String qName);
   public abstract int getIndex(String uri, String localPart);
   public abstract int getLength();
   public abstract String getLocalName(int index);
   public abstract String getQName(int index);
   public abstract String getType(String qName);
   public abstract String getType(int index);
   public abstract String getType(String uri, String localName);
   public abstract String getURI(int index);
   public abstract String getValue(String qName);
   public abstract String getValue(int index);
   public abstract String getValue(String uri, String localName);
}
```

Реализации: org.xml.sax.helpers.AttributesImpl

Передается методам: org.xml.sax.ContentHandler.startElement(), org.xml.sax.helpers.AttributesImpl.{AttributesImpl(), setAttributes()}, org.xml.sax.helpers.DefaultHandler.startElement(), org.xml.sax.helpers.XMLFilterImpl.startElement(), org.xml.sax.helpers.XMLReaderAdapter.startElement()

ContentHandler Java 1.4

org.xml.sax

Данный интерфейс является ключевым для разбора XML с API стандарта SAX. Вызывая различные методы интерфейса ContentHandler, XMLReader сообщает приложению о содержимом разбираемого документа XML. Для разбора документа с SAX нужно реализовать этот интерфейс, чтобы определить методы, при необходимости вызываемые парсером. Эти методы рассмотрены отдельно, поскольку данный интерфейс играет решающую роль в API стандарта SAX:

setDocumentLocator()

Обычно (но необязательно) перед вызовом других методов парсер вызывает данный метод для передачи объекта Locator методу ContentHandler. Locator определяет методы возврата номера текущей строки и столбца разбираемого документа. Кроме того, он обеспечивает возврат подтверждаемых значений при последующих

вызовах ContentHandler, если парсер предоставляет объект Locator. Например, ContentHandler может вызывать методы данного объекта во время вывода сообщений об ошибке.

`startDocument(), endDocument()`

Парсер вызывает эти методы только один раз – соответственно, в начале и в конце разбора. Метод `startDocument()` – это первый вызываемый метод, если не считать необязательного `setDocumentLocator()`, а последним вызываемым методом ContentHandler всегда будет `endDocument()`.

`startElement(), endElement()`

Парсер вызывает эти методы для каждого начального и конечного тега. Этим методам передается по три аргумента, описывающих тег: два первых аргумента обоих методов возвращают URI, который однозначно идентифицирует пространство имен, и локальное имя тега из этого пространства имен. Если в данный момент парсер не разбирает пространства имен, то третий аргумент предоставляет полное имя тега. Кроме аргументов имени тега, `startElement()` также получает объект Attributes, описывающий атрибуты тега.

`characters()`

При вызове данный метод сообщает приложению о строке текста в документе XML, найденной парсером. Текст содержится внутри заданного массива символов в определенной стартовой позиции и содержит определенное количество символов.

`ignorableWhitespace()`

Этот метод похож на `characters()`, но с его помощью парсеры могут сообщать приложению об «игнорируемом пробеле» в содержимом элемента XML.

`processingInstruction()`

Парсер вызывает этот метод, чтобы сообщить приложению о встреченной им инструкции обработки XML (XML Processing Instruction, или PI) с указанными целевой строкой и строками данных.

`skippedEntity()`

Если парсер XML встречает в документе сущность, но не расширяет и не разбирает ее содержимое, то он сообщает об этом приложению посредством передачи имени сущности данному методу.

`startPrefixMapping(), endPrefixMapping()`

Данные методы сообщают приложению об отображении пространств имен из определенного префикса на указанный URI пространства имен.

Интерфейс DTDHandler подобен ContentHandler. Приложение может реализовать этот интерфейс для получения уведомления от парсера об относящихся к DTD событиях. Аналогично, пакет `org.xml.sax.ext` определяет два интерфейса «расширения». Если парсер поддерживает такие расширения, то с их помощью можно получить еще больше информации о документе (например, комментарии и разделы CDATA) и о DTD (включая полное множество объявлений элемента, атрибута и сущности). Полезным классом является `org.xml.sax.helpers.DefaultHandler`. Он реализует ContentHandler и три других интерфейса, которые обычно применяются с классом XMLReader, и обеспечивает пустую реализацию для всех методов. Приложения могут создать подкласс DefaultHandler, только подменив нужные им методы. Обычно это удобнее, чем прямая реализация интерфейсов.

```
public interface ContentHandler {
// Открытые методы экземпляра
```

```
   public abstract void characters(char[] ch, int start, int length) throws SAXException;
   public abstract void endDocument() throws SAXException;
   public abstract void endElement(String namespaceURI, String localName, String qName)
                                   throws SAXException;
   public abstract void endPrefixMapping(String prefix) throws SAXException;
   public abstract void ignorableWhitespace(char[] ch, int start, int length)
                                   throws SAXException;
   public abstract void processingInstruction(String target, String data) throws SAXException;
   public abstract void setDocumentLocator(Locator locator);
   public abstract void skippedEntity(String name) throws SAXException;
   public abstract void startDocument() throws SAXException;
   public abstract void startElement(String namespaceURI, String localName, String qName,
                                   org.xml.sax.Attributes atts) throws SAXException;
   public abstract void startPrefixMapping(String prefix, String uri) throws SAXException;
}
```

Реализации: javax.xml.transform.sax.TemplatesHandler, javax.xml.transform.sax.TransformerHandler, org.xml.sax.helpers.DefaultHandler, org.xml.sax.helpers.XMLFilterImpl, org.xml.sax.helpers.XMLReaderAdapter

Передается методам: javax.xml.transform.sax.SAXResult.{SAXResult(), setHandler()}, XMLReader.setContentHandler(), org.xml.sax.helpers.ParserAdapter.setContentHandler(), org.xml.sax.helpers.XMLFilterImpl.setContentHandler()

Возвращается методами: javax.xml.transform.sax.SAXResult.getHandler(), XMLReader.getContentHandler(), org.xml.sax.helpers.ParserAdapter.getContentHandler(), org.xml.sax.helpers.XMLFilterImpl.getContentHandler()

DocumentHandler

Java 1.4; устарел в Java 1.4

org.xml.sax

Данный интерфейс входит в API стандарта SAX1, однако он признан устаревшим, а предпочтение вместо него отдается интерфейсу ContentHandler **стандарта SAX2, поддерживающему пространства имен XML.**

```
public interface DocumentHandler {
// Открытые методы экземпляра
   public abstract void characters(char[] ch, int start, int length) throws SAXException;
   public abstract void endDocument() throws SAXException;
   public abstract void endElement(String name) throws SAXException;
   public abstract void ignorableWhitespace(char[] ch, int start, int length)
        throws SAXException;
   public abstract void processingInstruction(String target, String data) throws SAXException;
   public abstract void setDocumentLocator(Locator locator);
   public abstract void startDocument() throws SAXException;
   public abstract void startElement(String name, org.xml.sax.AttributeList atts)
        throws SAXException;
}
```

Реализации: HandlerBase, org.xml.sax.helpers.ParserAdapter

Передается методам: org.xml.sax.Parser.setDocumentHandler(), org.xml.sax.helpers.XMLReaderAdapter.setDocumentHandler()

DTDHandler

Java 1.4

org.xml.sax

Данный интерфейс определяет методы, которые приложение может реализовать для получения от XMLReader уведомления о нотации и необрабатываемых объявлениях сущности в DTD документа XML. Нотации и необрабатываемые сущности – это малоизвестные особенности (features) XML, которые применяются нечасто (как и этот интерфейс). Чтобы использовать DTDHandler, определите класс, реализующий данный интерфейс, или просто подкласс вспомогательного класса org.xml.sax.helpers.DefaultHandler и передайте экземпляр данного класса методу setDTDHandler() интерфейса XMLReader. Затем, если в DTD документа парсер встретит какое-либо объявление нотации или необрабатываемой сущности, он вызовет метод notationDecl() или unparsedEntityDecl(). Далее в документе неразбираемые значения могут встретиться в качестве значений атрибутов, поэтому если приложение заинтересовано в их дальнейшем использовании, оно должно запомнить имя сущности и ID системы.

```
public interface DTDHandler {
// Открытые методы экземпляра
   public abstract void notationDecl(String name, String publicId, String systemId)
                                        throws SAXException;
   public abstract void unparsedEntityDecl(String name, String publicId, String systemId,
                                        String notationName) throws SAXException;
}
```

Реализации: javax.xml.transform.sax.TransformerHandler, HandlerBase, org.xml.sax.helpers.DefaultHandler, org.xml.sax.helpers.XMLFilterImpl

Передается методам: org.xml.sax.Parser.setDTDHandler(), XMLReader.setDTDHandler(), org.xml.sax.helpers.ParserAdapter.setDTDHandler(), org.xml.sax.helpers.XMLFilterImpl.setDTDHandler(), org.xml.sax.helpers.XMLReaderAdapter.setDTDHandler()

Возвращается методами: XMLReader.getDTDHandler(), org.xml.sax.helpers.ParserAdapter.getDTDHandler(), org.xml.sax.helpers.XMLFilterImpl.getDTDHandler()

EntityResolver

Java 1.4

org.xml.sax

Приложение может реализовать данный интерфейс, если необходимо помочь парсеру разрешить (resolve) внешние сущности. Если передать экземпляр EntityResolver методу setEntityResolver() интерфейса XMLReader, то парсер будет вызывать метод resolveEntity() каждый раз, когда необходимо прочитать внешнюю сущность. С помощью открытого или системного идентификатора этот метод должен возвращать InputSource, который может применяться парсером для прочтения содержимого внешней сущности. Если эта сущность содержит действительный системный идентификатор, то парсер может напрямую читать внешнюю сущность, не прибегая к помощи EntityResolver. В то же время данный интерфейс полезен для отображения сетевых URL на копии, кэшируемые локально, или для отображения открытых идентификаторов на локальные файлы. Вспомогательный класс org.xml.sax.helpers.DefaultHandler включает базовую реализацию данного интерфейса, поэтому метод resolveEntity() можно подменить, если создан подкласс DefaultHandler.

```
public interface EntityResolver {
// Открытые методы экземпляра
   public abstract InputSource resolveEntity(String publicId, String systemId)
        throws SAXException,java.io.IOException;
}
```

Реализации: HandlerBase, org.xml.sax.helpers.DefaultHandler, org.xml.sax.helpers.XMLFilterImpl

Передается методам: javax.xml.parsers.DocumentBuilder.setEntityResolver(), org.xml.sax.Parser.setEntityResolver(), XMLReader.setEntityResolver(), org.xml.sax.helpers.ParserAdapter.setEntityResolver(), org.xml.sax.helpers.XMLFilterImpl.setEntityResolver(), org.xml.sax.helpers.XMLReaderAdapter.setEntityResolver()

Возвращается методами: XMLReader.getEntityResolver(), org.xml.sax.helpers.ParserAdapter.getEntityResolver(), org.xml.sax.helpers.XMLFilterImpl.getEntityResolver()

ErrorHandler Java 1.4

org.xml.sax

Перед разбором документа XML приложение должно предоставить XMLReader **реализацию этого интерфейса, вызвав его метод** setErrorHandler()**. Если понадобится сгенерировать предупреждение, сообщение об ошибке или о неисправимой ошибке, то программа считывания вызовет соответствующий метод** ErrorHandler**. Метод** error() **сообщает о таких исправимых ошибках, как сомнительная достоверность документа. После вызова** error() **парсер продолжает разбор. Метод** fatalError() **сообщает о таких неисправимых ошибках, как сомнительная формальная правильность. После вызова** fatalError() **парсер не может продолжить разбор. Объект** ErrorHandler **может любым образом отвечать на предупреждения, ошибки, фатальные ошибки и генерировать исключения от этих методов.**

Можно создать подкласс вспомогательного класса org.xml.sax.helpers.DefaultHandler **и подменить предоставляемые интерфейсом методы, сообщающие об ошибках, а не реализовывать этот интерфейс напрямую. Методы** warning() **и** error() **из** DefaultHandler **не выполняют никаких действий, а метод** fatalError() **выдает переданный ему объект** SAXParseException**.**

```
public interface ErrorHandler {
// Открытые методы экземпляра
   public abstract void error(SAXParseException exception) throws SAXException;
   public abstract void fatalError(SAXParseException exception) throws SAXException;
   public abstract void warning(SAXParseException exception) throws SAXException;
}
```

Реализации: HandlerBase, org.xml.sax.helpers.DefaultHandler, org.xml.sax.helpers.XMLFilterImpl

Передается методам: javax.xml.parsers.DocumentBuilder.setErrorHandler(), org.xml.sax.Parser.setErrorHandler(), XMLReader.setErrorHandler(), org.xml.sax.helpers.ParserAdapter.setErrorHandler(), org.xml.sax.helpers.XMLFilterImpl.setErrorHandler(), org.xml.sax.helpers.XMLReaderAdapter.setErrorHandler()

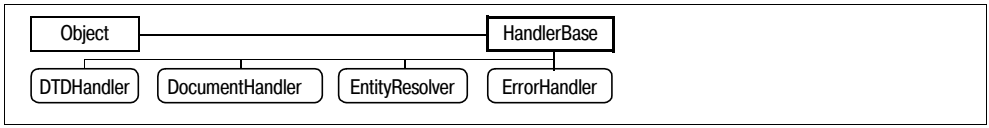
Возвращается методами: XMLReader.getErrorHandler(), org.xml.sax.helpers.ParserAdapter.getErrorHandler(), org.xml.sax.helpers.XMLFilterImpl.getErrorHandler()

HandlerBase

Java 1.4; устарел в Java 1.4

org.xml.sax

Данный класс входит в API стандарта SAX1, однако он устарел, а вместо него предпочтение отдается классу `org.xml.sax.helpers.DefaultHandler` стандарта SAX2.




```
public class HandlerBase implements DocumentHandler, DTDHandler, EntityResolver, ErrorHandler {
// Открытые конструкторы
   public HandlerBase();
// Методы, реализующие DocumentHandler
   public void characters(char[] ch, int start, int length) throws SAXException;        // пустой
   public void endDocument() throws SAXException;                                        // пустой
   public void endElement(String name) throws SAXException;                              // пустой
   public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException; // пустой
   public void processingInstruction(String target, String data) throws SAXException;   // пустой
   public void setDocumentLocator(Locator locator);                                      // пустой
   public void startDocument() throws SAXException;                                      // пустой
   public void startElement(String name, org.xml.sax.AttributeList attributes)
                                                                  throws SAXException;  // пустой
// Методы, реализующие DTDHandler
   public void notationDecl(String name, String publicId, String systemId);             // пустой
   public void unparsedEntityDecl(String name, String publicId, String systemId,
                                                                 String notationName);  // пустой
// Методы, реализующие EntityResolver
   public InputSource resolveEntity(String publicId, String systemId) throws SAXException;    // константа
// Методы, реализующие ErrorHandler
   public void error(SAXParseException e) throws SAXException;                           // пустой
   public void fatalError(SAXParseException e) throws SAXException;
   public void warning(SAXParseException e) throws SAXException;                         // пустой
}
```

Передается методам: `javax.xml.parsers.SAXParser.parse()`

InputSource

Java 1.4

org.xml.sax

Этот простой класс описывает источник входных данных для `XMLReader`. Объект `InputSource`, который можно передать методу `parse()` программы `XMLReader`, также является возвращаемым значением метода `EntityResolver.resolveEntity()`.

При помощи одного из методов конструкторов создайте `InputSource()`, указав системный идентификатор (URL) разбираемого файла либо байтовый или символьный поток, из которого парсер должен считывать документ. Кроме конструктора вы можете вызвать метод `setSystemId()`, чтобы определить разбираемый документ, и/или метод `setPublicId()`, чтобы предоставить этому документу идентификаторы. Например, чтобы объект мог отобразить сообщение об ошибке, необходимо имя файла или URL. Если определить документ для разбора в виде URL или байтового потока, то с помощью `setEncoding()` можно задать кодирование символов документа. Парсер будет ис-

пользовать предоставленное значение кодировки, однако в объявлениях <?xml?> документов XML предусмотрено собственное описание кодировки, поэтому парсер сможет определить кодировку документа, даже если вы не вызывали setEncoding().

Данный класс позволяет определить несколько источников входных данных. Сначала XMLReader вызовет getCharacterStream() и задействует возвращаемый Reader, если таковой будет получен. Если метод возвращает null, то он вызывает getByteStream() и использует возвращаемый им InputStream. Наконец, если не найден ни один символьный или байтовый поток, парсер вызывает getSystemId() и стремится прочитать документ XML из возвращаемого URL.

XMLReader никогда не использует методы set() для модификации состояния объекта InputSource.

```
public class InputSource {
// Открытые конструкторы
   public InputSource();
   public InputSource(java.io.Reader characterStream);
   public InputSource(java.io.InputStream byteStream);
   public InputSource(String systemId);
// Методы доступа к свойствам (по имени свойства)
   public java.io.InputStream getByteStream();                    // по умолчанию:null
   public void setByteStream(java.io.InputStream byteStream);
   public java.io.Reader getCharacterStream();                    // по умолчанию:null
   public void setCharacterStream(java.io.Reader characterStream);
   public String getEncoding();                                   // по умолчанию:null
   public void setEncoding(String encoding);
   public String getPublicId();                                   // по умолчанию:null
   public void setPublicId(String publicId);
   public String getSystemId();                                   // по умолчанию:null
   public void setSystemId(String systemId);
}
```

Передается методам: javax.xml.parsers.DocumentBuilder.parse(), javax.xml.parsers.SAXParser.parse(), javax.xml.transform.sax.SAXSource.{SAXSource(), setInputSource()}, org.xml.sax.Parser.parse(), XMLReader.parse(), org.xml.sax.helpers.ParserAdapter.parse(), org.xml.sax.helpers.XMLFilterImpl.parse(), org.xml.sax.helpers.XMLReaderAdapter.parse()

Возвращается методами: javax.xml.transform.sax.SAXSource.{getInputSource(), sourceToInputSource()}, EntityResolver.resolveEntity(), HandlerBase.resolveEntity(), org.xml.sax.helpers.DefaultHandler.resolveEntity(), org.xml.sax.helpers.XMLFilterImpl.resolveEntity()

Locator

Java 1.4

org.xml.sax

Чтобы передать приложению объект, реализующий данный интерфейс, XMLReader должен вызвать метод setDocumentLocator() объекта ContentHandler приложения перед вызовом любых других методов этого ContentHandler. ContentHandler может использовать методы данного объекта Locator внутри любого другого метода, вызываемого парсером для определения разбираемого документа и номера строки и столбца. Эта информация особенно полезна при выдаче сообщений об ошибках или предупреждений. Методы getSystemId() и getPublicId() возвращают системные и открытые идентификаторы разбираемого документа или null, если такая информация недоступна

парсеру. Методы getLineNumber() и getColumnNumber() возвращают номера строки и столбца следующего символа, который будет прочитан парсером (нумерация строк и столбцов начинается с 1, а не с 0). Из этих методов парсер может возвращать приблизительное значение или значение –1, если не был отслежен номер строки и столбца.

```
public interface Locator {
// Открытые методы экземпляра
   public abstract int getColumnNumber();
   public abstract int getLineNumber();
   public abstract String getPublicId();
   public abstract String getSystemId();
}
```

Реализации: org.xml.sax.helpers.LocatorImpl

Передается методам: org.xml.sax.ContentHandler.setDocumentLocator(), DocumentHandler.setDocumentLocator(), HandlerBase.setDocumentLocator(), SAXParseException.SAXParseException(), org.xml.sax.helpers.DefaultHandler.setDocumentLocator(), org.xml.sax.helpers.LocatorImpl.LocatorImpl(), org.xml.sax.helpers.ParserAdapter.setDocumentLocator(), org.xml.sax.helpers.XMLFilterImpl.setDocumentLocator(), org.xml.sax.helpers.XMLReaderAdapter.setDocumentLocator()

Parser

Java 1.4; устарел в Java 1.4

org.xml.sax

Данный интерфейс входит в API стандарта SAX1, однако он устарел, а вместо него предпочтение отдается интерфейсу XMLReader стандарта SAX2, поддерживающему пространства имен XML.

```
public interface Parser {
// Открытые методы экземпляра
   public abstract void parse(InputSource source) throws SAXException, java.io.IOException;
   public abstract void parse(String systemId) throws SAXException, java.io.IOException;
   public abstract void setDocumentHandler(DocumentHandler handler);
   public abstract void setDTDHandler(DTDHandler handler);
   public abstract void setEntityResolver(EntityResolver resolver);
   public abstract void setErrorHandler(ErrorHandler handler);
   public abstract void setLocale(java.util.Locale locale) throws SAXException;
}
```

Реализации: org.xml.sax.helpers.XMLReaderAdapter

Передается методам: org.xml.sax.helpers.ParserAdapter.ParserAdapter()

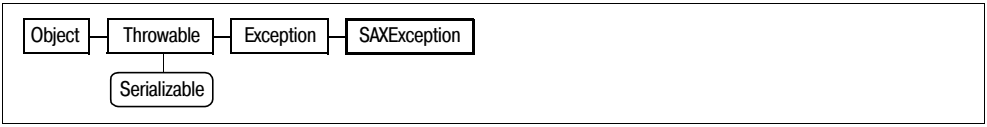
Возвращается методами: javax.xml.parsers.SAXParser.getParser(), org.xml.sax.helpers.ParserFactory.makeParser()

SAXException

Java 1.4

org.xml.sax ***сериализуемое, проверяемое***

Сигнализирует о неполадках, возникших при разборе документа XML. Данный класс служит общим родительским классом для более специфичных типов исключений SAX. В XMLReader такие исключения может генерировать метод parse(). Кроме того, приложение может сгенерировать SAXException от любого метода обработчика, вызванного парсером (например, ContentHandler и ErrorHandler).




```
public class SAXException extends Exception {
// Открытые конструкторы
   public SAXException(String message);
   public SAXException(Exception e);
   public SAXException(String message, Exception e);
// Открытые методы экземпляра
   public Exception getException();
// Открытые методы, замещающие Throwable
   public String getMessage();
   public String toString();
}
```

Подклассы: SAXNotRecognizedException, SAXNotSupportedException, SAXParseException

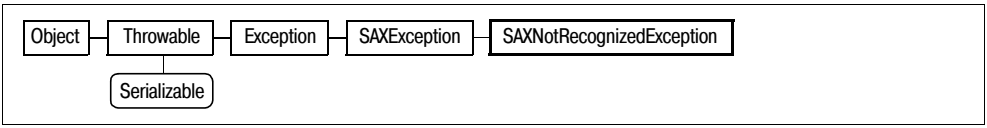
Генерируется методами: Методов слишком много, чтобы их перечислить.

SAXNotRecognizedException

Java 1.4

org.xml.sax ***сериализуемое, проверяемое***

Сигнализирует об имени свойства или особенности (feature), нераспознаваемых парсером. См. XMLReader, методы setFeature() и setProperty().




```
public class SAXNotRecognizedException extends SAXException {
// Открытые конструкторы
   public SAXNotRecognizedException(String message);
}
```

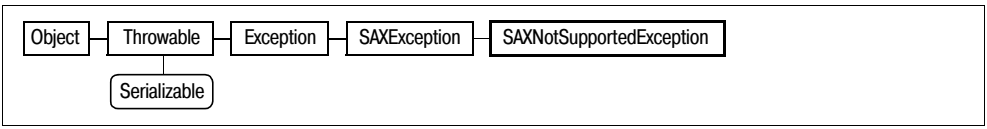
Генерируется методами: Методов слишком много, чтобы их перечислить.

SAXNotSupportedException

Java 1.4

org.xml.sax ***сериализуемое, проверяемое***

Сигнализирует о том, что парсер распознает, но не поддерживает указанное свойство или особенность. Свойство или функция могут совсем не поддерживаться или могут предназначаться только для чтения. В последнем случае данное исключение генерируется методом setFeature() или setProperty(), а не соответствующим методом getFeature() или getProperty() объекта XMLReader.



```
public class SAXNotSupportedException extends SAXException {
// Открытые конструкторы
   public SAXNotSupportedException(String message);
}
```

Генерируется методами: Методов слишком много, чтобы их перечислить.

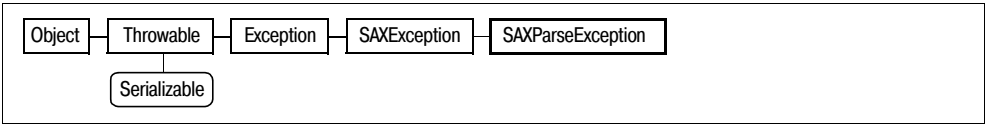
SAXParseException

Java 1.4

org.xml.sax ***сериализуемое, проверяемое***

Исключение данного типа сигнализирует о предупреждении или ошибке разбора XML. `SAXParseException` содержит методы возврата системных и открытых идентификаторов документа, в которых возникла ошибка, и методы возврата приблизительного номера строки и столбца, которые указывают место возникновения ошибки. Парсер не обязан добывать и отслеживать всю эту информацию, поэтому методы могут возвращать `null` или -1, если информация недоступна. (Дополнительная информация представлена в описании `Locator`.)

Обычно исключения данного типа генерируются приложением в методах интерфейса `ErrorHandler`. Парсер никогда не генерирует `SAXParseException` сам, а передает каждому методу `ErrorHandler` экземпляр данного класса, созданный соответствующим образом. Однако решение о генерации исключения принимает объект `ErrorHandler` приложения.




```
public class SAXParseException extends SAXException {
// Открытые конструкторы
   public SAXParseException(String message, Locator locator);
   public SAXParseException(String message, Locator locator, Exception e);
   public SAXParseException(String message, String publicId, String systemId, int lineNumber,
                                                                              int columnNumber);
   public SAXParseException(String message, String publicId, String systemId, int lineNumber,
                                                          int columnNumber, Exception e);
// Открытые методы экземпляра
   public int getColumnNumber();
   public int getLineNumber();
   public String getPublicId();
   public String getSystemId();
}
```

Передается методам: `ErrorHandler.{error(), fatalError(), warning()}`, `HandlerBase.{error(), fatalError(), warning()}`, `org.xml.sax.helpers.DefaultHandler.{error(), fatalError(), warning()}`, `org.xml.sax.helpers.XMLFilterImpl.{error(), fatalError(), warning()}`

XMLFilter

Java 1.4

org.xml.sax

`XMLFilter` расширяет `XMLReader` и ведет себя аналогично `XMLReader`, за исключением того, что он не разбирает сам документ, а фильтрует предоставляемые «родительским»

объектом XMLReader события SAX. Метод setParent() позволяет связать объект XMLReader с XMLFilter, который будет его фильтром.

XMLFilter является источником событий SAX и одновременно их получателем, поэтому реализация должна воплощать ContentHandler и связанные с ним интерфейсы, чтобы получать события от объекта-родителя, фильтровать их, а затем передавать фильтрованные события объекту ContentHandler, который зарегистрирован в программе фильтрации. Во вспомогательном классе org.xml.sax.helpers.XMLFilterImpl представлена простейшая реализация XMLFilter, которая формирует интерфейс XMLReader и ContentHandler, а также связанные с ним интерфейсы обработчиков. XMLFilterImpl ничего не фильтрует – он лишь пропускает (passes through) все свои вызовы методов. Вы можете создать его подкласс и подменить методы, требующие фильтрации.


```
public interface XMLFilter extends XMLReader {
// Открытые методы экземпляра
   public abstract XMLReader getParent();
   public abstract void setParent(XMLReader parent);
}
```

Реализации: org.xml.sax.helpers.XMLFilterImpl

Возвращается методами:

javax.xml.transform.sax.SAXTransformerFactory.newXMLFilter()

XMLReader

Java 1.4

org.xml.sax

Данный интерфейс определяет методы, которые должны быть реализованы парсером XML стандарта SAX2. XMLReader не может определять конструктор создания XMLReader, поскольку это интерфейс. Чтобы получить объект XMLReader, можно создать экземпляр какого-либо класса, реализующего данный интерфейс. Если воспользоваться классами SAXParserFactory и SAXParser пакета парсеров javax.xml, то можно не связывать код ни с какой конкретной реализацией. Интерфейс XMLReader никак не связан с классом java.io.Reader или с любым другим классом символьного потока.

После получения экземпляра XMLReader нужно зарегистрировать в нем объекты обработчиков, чтобы он мог вызывать методы, уведомляющие приложение о результатах разбора. Все приложения должны регистрировать ContentHandler и ErrorHandler с помощью методов setContentHandler() и setErrorHandler(). Кроме того, некоторые приложения могут зарегистрировать EntityResolver и/или DTDHandler. Приложения могут регистрировать объекты DeclHandler и LexicalHandler пакета org.xml.sax.ext, если реализация парсера поддерживает такие интерфейсы обработчиков расширения. Объекты DeclHandler и LexicalHandler регистрируются с помощью метода setProperty().

Помимо регистрации объектов обработчиков для XMLReader, вы можете конфигурировать поведение парсера с помощью setFeature() и setProperty(). Особенности (features) и свойства (properties) являются парами «имя/значение». Для сохранения однотипности имена особенностей и свойств выражены в виде URL, которые обычно не связаны ни с каким веб-содержимым: они просто являются уникальными идентификаторами. Особенности содержат булевы значения, а свойства – произвольные значения

объектов. Особенности и свойства составляют механизм расширения, который позволяет приложению установить детали, зависящие от реализации, которые определяют поведение парсера. Кроме того, существует несколько «стандартных» особенностей и свойств, поддерживаемых большинством парсеров SAX (если не всеми). Если парсер не распознает имя особенности или свойства, то методы `setFeature()` и `setProperty()`, а также соответствующие им методы запроса `getFeature()` и `getProperty()` генерируют `SAXNotRecognizedException`. И наоборот, методы генерируют `SAXNotSupportedException`, если парсер распознает имя особенности или свойства, но не поддерживает эту особенность или свойство. Методы `set` генерируют это исключение, если парсер допускает только запрос особенности или свойства, а не установку.

Существуют следующие стандартные особенности. Их имена имеют вид URL, который начинается с префикса «http://www.xml.org/sax/features/». В дальнейшем для краткости этот префикс не будет указываться. Все парсеры обязаны поддерживать две из этих особенностей. Другие особенности могут не поддерживаться.

`namespaces`

Если `true` (по умолчанию), парсер поддерживает пространства имен и предоставляет URI и локальное имя пространства имен для имен элемента и атрибута. Данная особенность должна поддерживаться каждой реализацией парсера.

`namespace-prefixes`

Если `true`, парсер предоставляет уточненнное имя (или «qName») для имен элемента и атрибута. `qName` состоит из префикса пространства имен, двоеточия и локального имени. Значением по умолчанию для данной особенности будет `false`; она также должна поддерживаться во всех реализациях парсера.

`validation`

Если `true`, парсер подтверждает документы XML и читает все внешние сущности.

`external-general-entities`

Если `true`, парсер обрабатывает общие внешние сущности. Значение всегда будет `true`, если особенность `validation` равна `true`.

`external-parameter-entities`

Если `true`, парсер обрабатывает внешние сущности параметра. Значение всегда будет `true`, если особенность `validation` равна `true`.

`lexical-handler/parameter-entities`

Если `true`, парсер сообщит начало и конец сущностей параметров интерфейсу расширения `LexicalHandler`.

`string-interning`

Если `true`, парсер использует метод `String.intern()` для всех возвращаемых им строк (имена элемента, атрибута, сущности и нотации, а также префиксы пространств имен и URI). Если так же ведет себя приложение, он может тестировать эти строки на равенство `==`, а не вызывать менее эффективный метод `equals()`.

Существуют следующие стандартные свойства. Подобно именам особенностей, их имена имеют вид URL, который начинается с префикса «http://www.xml.org/sax/properties/», опущенного в следующем списке. Все свойства могут не поддерживаться.

`declaration-handler`

Объект `org.xml.sax.ext.DeclHandler`, которому парсер сообщает содержимое DTD.

lexical-handler

Объект org.xml.sax.ext.LexicalHandler, в котором парсер вызывает методы для описания лексической структуры документа XML (то есть комментарии и разделы CDATA).

xml-string

Это свойство предназначено только для чтения. Его можно запросить только внутри метода обработчика, вызываемого парсером. Значением этого свойства является String. Здесь представлено содержимое документа, инициировавшего данный вызов обработчика.

dom-node

XMLReader, «разбирающий» не текстовую форму документа XML, а дерево DOM, использует значение данного свойства как объект org.w3c.dom.Node, с которого должен начинаться разбор.

Наконец, после получения объекта XMLReader, конфигурирования его особенностей и свойств, а также установки ContentHandler, ErrorHandler и других необходимых объектов обработчиков можно приступить к разбору документа XML. Для этого необходимо вызвать методы parse(), указав разбираемый документ либо как системный идентификатор (URL), либо как объект InputSource, который также позволяет использовать потоки.

```
public interface XMLReader {
// Открытые методы экземпляра
   public abstract org.xml.sax.ContentHandler getContentHandler();
   public abstract DTDHandler getDTDHandler();
   public abstract EntityResolver getEntityResolver();
   public abstract ErrorHandler getErrorHandler();
   public abstract boolean getFeature(String name) throws SAXNotRecognizedException,
                                                          SAXNotSupportedException;
   public abstract Object getProperty(String name) throws SAXNotRecognizedException,
                                                          SAXNotSupportedException;
   public abstract void parse(String systemId) throws java.io.IOException, SAXException;
   public abstract void parse(InputSource input) throws java.io.IOException, SAXException;
   public abstract void setContentHandler(org.xml.sax.ContentHandler handler);
   public abstract void setDTDHandler(DTDHandler handler);
   public abstract void setEntityResolver(EntityResolver resolver);
   public abstract void setErrorHandler(ErrorHandler handler);
   public abstract void setFeature(String name, boolean value)
        throws SAXNotRecognizedException,SAXNotSupportedException;
   public abstract void setProperty(String name, Object value)
        throws SAXNotRecognizedException,SAXNotSupportedException;
}
```

Реализации: XMLFilter, org.xml.sax.helpers.ParserAdapter

Передается методам: javax.xml.transform.sax.SAXSource.{SAXSource(), setXMLReader()}, XMLFilter.setParent(), org.xml.sax.helpers.XMLFilterImpl.{setParent(), XMLFilterImpl()}, org.xml.sax.helpers.XMLReaderAdapter.XMLReaderAdapter()

Возвращается методами: javax.xml.parsers.SAXParser.getXMLReader(), javax.xml.transform.sax.SAXSource.getXMLReader(), XMLFilter.getParent(), org.xml.sax.helpers.XMLFilterImpl.getParent(), org.xml.sax.helpers.XMLReaderFactory.createXMLReader()

Пакет org.xml.sax.ext

Java 1.4

Данный пакет определяет расширения для базового API стандарта SAX2. Такие расширения не требуют поддержки SAX-парсеров и приложений SAX. В то же время есть парсеры и приложения SAX, которые поддерживают эти расширения. Для них определены стандартные интерфейсы, которые позволяют парсерам обеспечивать приложение дополнительной информацией о документе XML. `DeclHandler` определяет методы сообщения содержимого DTD, а `LexicalHandler` определяет методы сообщения лексической структуры документа XML.

На момент написания данной книги готовится к выпуску версия «Расширения 1.1 SAX2». В текущей бета-версии к данному пакету добавлено три новых интерфейса: `Attributes2`, `EntityResolver2` и `Locator2`. Эти расширения здесь не описаны, поскольку они еще не выпущены. Для получения более подробной информации загляните на сайт *http://www.saxproject.org*.

Интерфейсы

```
public interface DeclHandler;
public interface LexicalHandler;
```

DeclHandler

Java 1.4

org.xml.sax.ext

Данный интерфейс расширения определяет методы, вызываемые SAX-парсером для сообщения приложению об объявляемых в DTD элементах, атрибутах и сущностях. Если приложение требует эту информацию о DTD, то объект, реализующий данный интерфейс, следует передать методу `setProperty()` программы `XMLReader`, указав имя свойства «http://www.xml.org/sax/properties/declaration-handler». Поскольку это обработчик расширения, он не требует поддержки SAX-парсеров, а попытка зарегистрировать `DeclHandler` может привести к генерации `SAXNotRecognizedException` или `SAXNotSupportedException`.

```
public interface DeclHandler {
// Открытые методы экземпляра
   public abstract void attributeDecl(String eName, String aName, String type,
       String valueDefault, String value) throws org.xml.sax.SAXException;
   public abstract void elementDecl(String name, String model) throws org.xml.sax.SAXException;
   public abstract void externalEntityDecl(String name, String publicId, String systemId)
                                                              throws org.xml.sax.SAXException;
   public abstract void internalEntityDecl(String name, String value)
                                                              throws org.xml.sax.SAXException;
}
```

LexicalHandler

Java 1.4

org.xml.sax.ext

Данный интерфейс расширения определяет методы, вызываемые SAX-парсером для сообщения приложению о лексической структуре документа XML. Если приложение требует такую информацию (например, для создания нового документа со структурой, аналогичной считываемой), то объект, реализующий данный интерфейс, следу-

ет передать методу setProperty() программы XMLReader, указав имя свойства «http://www.xml.org/sax/properties/lexical-handler». Поскольку это обработчик расширения, он не требует поддержки SAX-парсером, а попытка зарегистрировать DeclHandler может привести к генерации SAXNotRecognizedException или SAXNotSupportedException.

Если LexicalHandler будет успешно зарегистрирован в объекте XMLReader, то парсер вызовет startDTD() и endDTD(), чтобы сообщить о начале и конце DTD документа. Чтобы сообщить о начале и конце раздела CDATA, он вызовет startCDATA() и endCDATA(). Содержимое раздела CDATA будет сообщено с помощью метода characters() интерфейса ContentHandler. Для расширения сущности парсер сначала вызывает startEntity(), чтобы указать имя этой сущности, а после того как расширение сущности завершено, вызывает endEntity(). Наконец, парсер вызывает метод comment() каждый раз, когда он встречает комментарий XML.

```
public interface LexicalHandler {
// Открытые методы экземпляра
   public abstract void comment(char[] ch, int start, int length)
                   throws org.xml.sax.SAXException;
   public abstract void endCDATA() throws org.xml.sax.SAXException;
   public abstract void endDTD() throws org.xml.sax.SAXException;
   public abstract void endEntity(String name) throws org.xml.sax.SAXException;
   public abstract void startCDATA() throws org.xml.sax.SAXException;
   public abstract void startDTD(String name, String publicId, String systemId)
                   throws org.xml.sax.SAXException;
   public abstract void startEntity(String name) throws org.xml.sax.SAXException;
}
```

Реализации: javax.xml.transform.sax.TransformerHandler

Передается методам: javax.xml.transform.sax.SAXResult.setLexicalHandler()

Возвращается методами: javax.xml.transform.sax.SAXResult.getLexicalHandler()

Пакет org.xml.sax.helpers Java 1.4

Этот пакет содержит вспомогательные классы, необходимые программистам при работе с SAX-парсерами. Наиболее часто используется DefaultHandler: это реализация по умолчанию четырех стандартных интерфейсов обработчиков, позволяющих приложению легко перехватывать и обрабатывать сообщения. XMLReaderFactory обеспечивает независимость от реализации, позволяя приложению задействовать реализацию XMLReader, указанную в системном свойстве. XMLFilterImpl является «пустой» реализацией интерфейса XMLFilter, который реализует различные интерфейсы обработчиков, необходимые для соединения фильтра с «породившим» его XMLReader. Сам он ничего не фильтрует, но вы можете легко создать его подкласс и добавить фильтрацию. Для работы со старыми API, которые ожидают или возвращают объекты Parser стандарта SAX1, можно применить ParserAdapter, чтобы заставить объект Parser вести себя как объект XMLReader стандарта SAX2, или использовать XMLReaderAdapter, чтобы заставить XMLReader вести себя как Parser.

Классы

```
public class AttributeListImpl implements org.xml.sax.AttributeList;
public class AttributesImpl implements org.xml.sax.Attributes;
public class DefaultHandler implements org.xml.sax.ContentHandler, org.xml.sax.DTDHandler,
       org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler;
```

```
public class LocatorImpl implements org.xml.sax.Locator;
public class NamespaceSupport;
public class ParserAdapter implements org.xml.sax.DocumentHandler, org.xml.sax.XMLReader;
public class ParserFactory;
public class XMLFilterImpl implements org.xml.sax.ContentHandler, org.xml.sax.DTDHandler,
       org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler, org.xml.sax.XMLFilter;
public class XMLReaderAdapter implements org.xml.sax.ContentHandler, org.xml.sax.Parser;
public final class XMLReaderFactory;
```

AttributeListImpl

Java 1.4; устарел в Java 1.4

org.xml.sax.helpers

Этот устаревший класс представляет собой реализацию устаревшего интерфейса org.xml.sax.AttributeList стандарта SAX1. Вместо них предпочтение отдается реализации AttributesImpl интерфейса org.xml.sax.Attributes стандарта SAX2.

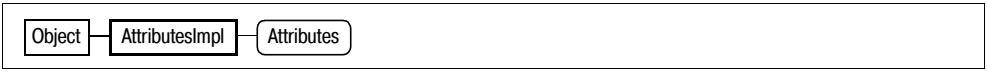

```
public class AttributeListImpl implements org.xml.sax.AttributeList {
// Открытые конструкторы
   public AttributeListImpl();
   public AttributeListImpl(org.xml.sax.AttributeList atts);
// Открытые методы экземпляра
   public void addAttribute(String name, String type, String value);
   public void clear();
   public void removeAttribute(String name);
   public void setAttributeList(org.xml.sax.AttributeList atts);
// Методы, реализующие AttributeList
   public int getLength();                                          // по умолчанию:0
   public String getName(int i);
   public String getType(int i);
   public String getType(String name);
   public String getValue(String name);
   public String getValue(int i);
}
```

AttributesImpl

Java 1.4

org.xml.sax.helpers

Данный вспомогательный класс является универсальной реализацией интерфейса Attributes. В дополнение к реализации всех методов Attributes он определяет различные set-методы для установки имен, значений и типов атрибутов. Кроме того, он определяет метод addAttribute() для присоединения нового атрибута к концу списка, метод removeAttribute() для удаления атрибута из списка и метод clear() для удаления всех атрибутов. Помимо этого, существует конструктор AttributesImpl(), инициализирующий новый объект AttributesImpl с помощью копии заданного объекта Attributes. Данный класс необходим в реализациях XMLFilter, применяемых для фильтрации атрибутов элемента, или в реализациях ContentHandler, применяемых для создания и сохранения копии объекта Attributes.




```
public class AttributesImpl implements org.xml.sax.Attributes {
// Открытые конструкторы
   public AttributesImpl();
   public AttributesImpl(org.xml.sax.Attributes atts);
// Открытые методы экземпляра
   public void addAttribute(String uri, String localName, String qName, String type, String value);
   public void clear();
   public void removeAttribute(int index);
   public void setAttribute(int index, String uri, String localName, String qName, String type,
                                                                                    String value);
   public void setAttributes(org.xml.sax.Attributes atts);
   public void setLocalName(int index, String localName);
   public void setQName(int index, String qName);
   public void setType(int index, String type);
   public void setURI(int index, String uri);
   public void setValue(int index, String value);
// Методы, реализующие Attributes
   public int getIndex(String qName);
   public int getIndex(String uri, String localName);
   public int getLength();      // по умолчанию:0
   public String getLocalName(int index);
   public String getQName(int index);
   public String getType(String qName);
   public String getType(int index);
   public String getType(String uri, String localName);
   public String getURI(int index);
   public String getValue(int index);
   public String getValue(String qName);
   public String getValue(String uri, String localName);
}
```

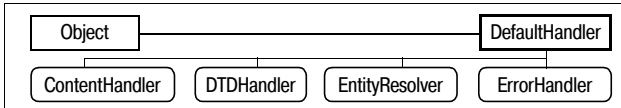
DefaultHandler

Java .14

org.xml.sax.helpers

Данный вспомогательный класс реализует четыре широко используемых интерфейса обработчика SAX из пакета `org.xml.sax` и определяет базовые реализации для всех их методов. Обычно удобнее создать подкласс `DefaultHandler` и подменить требуемые методы, нежели реализовывать все интерфейсы (и все их методы) с нуля. `DefaultHandler` формирует `ContentHandler`, `ErrorHandler`, `EntityResolver` и `DTDHandler`, поэтому экземпляр этого класса (или определяемого вами подкласса) можно передать методам `setContentHandler()`, `setErrorHandler()`, `setEntityResolver()` и `setDTDHandler()` объекта `XMLReader`. Кроме того, экземпляр подкласса `DefaultHandler` можно напрямую передать одному из методов `parse()` парсера `javax.xml.parsers.SAXParser`. `SAXParser` вызовет четыре соответствующих метода внутреннего `XMLReader`.

Все методы `DefaultHandler` (кроме двух) имеют пустые тела и ничего не делают. Исключения составляют методы `resolveEntity()`, возвращающий `null` и предписывающий парсеру определить сущность, и `fatalError()`, выдающий передаваемый ему объект `SAXParseException`.




```
public class DefaultHandler implements org.xml.sax.ContentHandler,
      org.xml.sax.DTDHandler,  org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler {
// Открытые конструкторы
   public DefaultHandler();
// Методы, реализующие ContentHandler
   public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXException; // пустой
   public void endDocument() throws org.xml.sax.SAXException;                                // пустой
   public void endElement(String uri, String localName, String qName)
                          throws org.xml.sax.SAXException;                                    // пустой
   public void endPrefixMapping(String prefix) throws org.xml.sax.SAXException;              // пустой
   public void ignorableWhitespace(char[] ch, int start, int length)
                          throws org.xml.sax.SAXException;                                    // пустой
   public void processingInstruction(String target, String data) throws org.xml.sax.SAXException;
                                                                                              // пустой
   public void setDocumentLocator(org.xml.sax.Locator locator);                              // пустой
   public void skippedEntity(String name) throws org.xml.sax.SAXException;                   // пустой
   public void startDocument() throws org.xml.sax.SAXException;                              // пустой
   public void startElement(String uri, String localName, String qName, org.xml.sax.Attributes
               attributes)  throws org.xml.sax.SAXException;                                 // пустой
   public void startPrefixMapping(String prefix, String uri) throws org.xml.sax.SAXException;
                                                                                              // пустой
// Методы, реализующие DTDHandler
   public void notationDecl(String name, String publicId, String systemId)
               throws org.xml.sax.SAXException;                                           // пустой
   public void unparsedEntityDecl(String name, String publicId, String systemId,
               String notationName) throws org.xml.sax.SAXException;                      // пустой
// Методы, реализующие EntityResolver
   public org.xml.sax.InputSource resolveEntity(String publicId, String systemId)
              throws org.xml.sax.SAXException;                                        // константа
// Методы, реализующие ErrorHandler
   public void error(org.xml.sax.SAXParseException e) throws org.xml.sax.SAXException;   // пустой
   public void fatalError(org.xml.sax.SAXParseException e) throws org.xml.sax.SAXException;
   public void warning(org.xml.sax.SAXParseException e) throws org.xml.sax.SAXException; // пустой
}
```

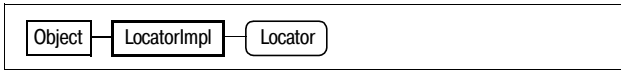
Передается методам: javax.xml.parsers.SAXParser.parse()

LocatorImpl

Java 1.4

org.xml.sax.helpers

Данный вспомогательный класс является очень простой реализацией интерфейса Locator. Он определяет конструктор копирования, который создает новый объект LocatorImpl, копирующий состояние указанного объекта Locator. Ценность данного конструктора состоит в том, что он позволяет приложениям копировать состояние Locator и сохранять его для последующего использования.



```
public class LocatorImpl implements org.xml.sax.Locator {
// Открытые конструкторы
   public LocatorImpl();
   public LocatorImpl(org.xml.sax.Locator locator);
// Методы доступа к свойствам (по имени свойства)
   public int getColumnNumber();                          // Реализует:Locator, по умолчанию:0
   public void setColumnNumber(int columnNumber);
   public int getLineNumber();                            // Реализует:Locator, по умолчанию:0
   public void setLineNumber(int lineNumber);
   public String getPublicId();                           // Реализует:Locator, по умолчанию:null
   public void setPublicId(String publicId);
   public String getSystemId();                           // Реализует:Locator, по умолчанию:null
   public void setSystemId(String systemId);
// Методы, реализующие Locator
   public int getColumnNumber();                          // по умолчанию:0
   public int getLineNumber();                            // по умолчанию:0
   public String getPublicId();                           // по умолчанию:null
   public String getSystemId();                           // по умолчанию:null
}
```

NamespaceSupport

Java 1.4

org.xml.sax.helpers

Данный вспомогательный класс помогает разработчикам парсера SAX обрабатывать пространства имен XML. Он редко используется приложениями SAX.

```
public class NamespaceSupport {
// Открытые конструкторы
   public NamespaceSupport();
// Открытые константы
   public static final String XMLNS;                    // ="http://www.w3.org/XML/1998/namespace"
// Открытые методы экземпляра
   public boolean declarePrefix(String prefix, String uri);
   public java.util.Enumeration getDeclaredPrefixes();
   public String getPrefix(String uri);
   public java.util.Enumeration getPrefixes();
   public java.util.Enumeration getPrefixes(String uri);
   public String getURI(String prefix);
   public void popContext();
   public String[] processName(String qName, String[] parts, boolean isAttribute);
   public void pushContext();
   public void reset();
}
```

ParserAdapter

Java 1.4

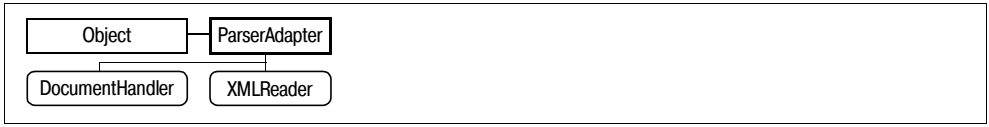
org.xml.sax.helpers

Данный адаптерный класс (adapter class) ведет себя аналогично объекту `XMLReader` стандарта SAX2, за исключением того, что он берет входные данные из объекта `Parser` стандарта SAX1. Для этого он реализует устаревший интерфейс `DocumentHandler` стандарта SAX1, который позволяет получать события из `Parser`. `ParserAdapter` предоставляет собственный слой обработки пространства имен, чтобы преобразовать независящий от пространства имен `Parser` в зависящий от пространства имен `XMLReader`.

Этот класс необходим, если вы работаете со старым API, который предоставляет объект Parser стандарта SAX1, а хотите работать с тем же парсером, но использовать API XMLReader стандарта SAX2: нужно просто передать объект Parser конструктору ParserAdapter() и использовать получившийся в результате объект так же, как и любой другой объект XMLReader.

API стандарта SAX1 и стандарта SAX2 не совпадают друг с другом, а Parser не может быть полностью адаптирован к XMLReader. В частности, ParserAdapter никогда не вызывает метод обработчика skippedEntity(), поскольку API Parser стандарта SAX1 не посылает уведомления о пропущенных сущностях. Кроме того, он не стремится установить, действительно ли два атрибута элемента с пространством имен в префиксе определяют один и тот же атрибут.

Для получения дополнительной информации обратитесь к описанию адаптера XMLReaderAdapter, который работает в обратном направлении, заставляя SAX2-парсер вести себя аналогично SAX1-парсеру.




```
public class ParserAdapter implements org.xml.sax.DocumentHandler, org.xml.sax.XMLReader {
// Открытые конструкторы
   public ParserAdapter() throws org.xml.sax.SAXException;
   public ParserAdapter(org.xml.sax.Parser parser);
// Методы, реализующие DocumentHandler
   public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXException;
   public void endDocument() throws org.xml.sax.SAXException;
   public void endElement(String qName) throws org.xml.sax.SAXException;
   public void ignorableWhitespace(char[] ch, int start, int length) throws org.xml.sax.SAXException;
   public void processingInstruction(String target, String data) throws org.xml.sax.SAXException;
   public void setDocumentLocator(org.xml.sax.Locator locator);
   public void startDocument() throws org.xml.sax.SAXException;
   public void startElement(String qName, org.xml.sax.AttributeList qAtts) throws org.xml.sax.SAXException;
// Методы, реализующие XMLReader
   public org.xml.sax.ContentHandler getContentHandler();
   public org.xml.sax.DTDHandler getDTDHandler();
   public org.xml.sax.EntityResolver getEntityResolver();
   public org.xml.sax.ErrorHandler getErrorHandler();
   public boolean getFeature(String name) throws org.xml.sax.SAXNotRecognizedException,
                                                 org.xml.sax.SAXNotSupportedException;
   public Object getProperty(String name) throws org.xml.sax.SAXNotRecognizedException,
                                                 org.xml.sax.SAXNotSupportedException;
   public void parse(String systemId) throws java.io.IOException, org.xml.sax.SAXException;
   public void parse(org.xml.sax.InputSource input) throws java.io.IOException,
                                                           org.xml.sax.SAXException;
   public void setContentHandler(org.xml.sax.ContentHandler handler);
   public void setDTDHandler(org.xml.sax.DTDHandler handler);
   public void setEntityResolver(org.xml.sax.EntityResolver resolver);
   public void setErrorHandler(org.xml.sax.ErrorHandler handler);
   public void setFeature(String name, boolean state)
        throws org.xml.sax.SAXNotRecognizedException,org.xml.sax.SAXNotSupportedException;
   public void setProperty(String name, Object value)
        throws org.xml.sax.SAXNotRecognizedException,org.xml.sax.SAXNotSupportedException;
}
```

ParserFactory

Java 1.4; устарел в Java 1.4

org.xml.sax.helpers

Данный устаревший класс SAX1 является фабрикой устаревших объектов `Parser` стандарта SAX1. Новые приложения должны использовать `XMLReaderFactory` стандарта SAX2 в качестве фабрики объектов `XMLReader` стандарта SAX2.

```
public class ParserFactory {
// Конструктор отсутствует
// Открытые методы класса
   public static org.xml.sax.Parser makeParser() throws ClassNotFoundException,
      IllegalAccessException,InstantiationException,NullPointerException,ClassCastException;
   public static org.xml.sax.Parser makeParser(String className) throws ClassNotFoundException,
      IllegalAccessException, InstantiationException, ClassCastException;
}
```

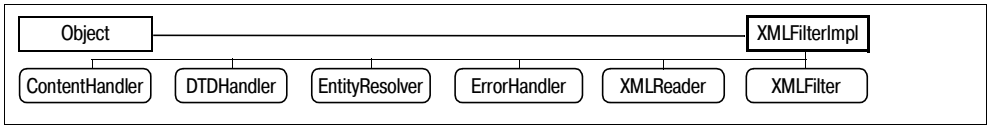
XMLFilterImpl

Java 1.4

org.xml.sax.helpers

Данный класс реализует `XMLFilter`, который ничего не фильтрует. Для замещения методов, которые необходимы при фильтрации, можно создать его подкласс.

`XMLFilterImpl` реализует `ContentHandler`, `ErrorHandler`, `EntityResolver` и `DTDHandler`, чтобы получать события SAX от «родительского» объекта `XMLReader`. Кроме того, он реализует интерфейс `XMLFilter`, который является расширением `XMLReader`. Таким образом, он действует как `XMLReader` и может посылать события SAX зарегистрированным в нем объектам обработчиков. Каждый метод обработчика данного класса просто вызывает соответствующий метод соответствующего обработчика, зарегистрированного в фильтре. Методы `XMLReader` для получения и установки особенностей и свойств просто вызывают соответствующий метод родительского объекта `XMLReader`. Подобным образом действуют методы `parse()`: чтобы начать разбор, они передают свой аргумент соответствующему методу `parse()` родительского объекта.




```
public class XMLFilterImpl implements org.xml.sax.ContentHandler, org.xml.sax.DTDHandler,
                org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler, org.xml.sax.XMLFilter {
// Открытые конструкторы
   public XMLFilterImpl();
   public XMLFilterImpl(org.xml.sax.XMLReader parent);
// Методы, реализующие ContentHandler
   public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXException;
   public void endDocument() throws org.xml.sax.SAXException;
   public void endElement(String uri, String localName, String qName)
                                                        throws org.xml.sax.SAXException;
   public void endPrefixMapping(String prefix) throws org.xml.sax.SAXException;
   public void ignorableWhitespace(char[] ch, int start, int length)
                                                        throws org.xml.sax.SAXException;
   public void processingInstruction(String target, String data) throws org.xml.sax.SAXException;
   public void setDocumentLocator(org.xml.sax.Locator locator);
```

```
    public void skippedEntity(String name) throws org.xml.sax.SAXException;
    public void startDocument() throws org.xml.sax.SAXException;
    public void startElement(String uri, String localName, String qName,
          org.xml.sax.Attributes atts) throws org.xml.sax.SAXException;
    public void startPrefixMapping(String prefix, String uri) throws org.xml.sax.SAXException;
// Методы, реализующие DTDHandler
    public void notationDecl(String name, String publicId, String systemId)
                                                          throws org.xml.sax.SAXException;
    public void unparsedEntityDecl(String name, String publicId, String systemId,
          String notationName) throws org.xml.sax.SAXException;
// Методы, реализующие EntityResolver
    public org.xml.sax.InputSource resolveEntity(String publicId, String systemId)
          throws org.xml.sax.SAXException, java.io.IOException;
// Методы, реализующие ErrorHandler
    public void error(org.xml.sax.SAXParseException e) throws org.xml.sax.SAXException;
    public void fatalError(org.xml.sax.SAXParseException e) throws org.xml.sax.SAXException;
    public void warning(org.xml.sax.SAXParseException e) throws org.xml.sax.SAXException;
// Методы, реализующие XMLFilter
    public org.xml.sax.XMLReader getParent();                              // по умолчанию:null
    public void setParent(org.xml.sax.XMLReader parent);
// Методы, реализующие XMLReader
    public org.xml.sax.ContentHandler getContentHandler();                 // по умолчанию:null
    public org.xml.sax.DTDHandler getDTDHandler();                         // по умолчанию:null
    public org.xml.sax.EntityResolver getEntityResolver();                 // по умолчанию:null
    public org.xml.sax.ErrorHandler getErrorHandler();                     // по умолчанию:null
    public boolean getFeature(String name) throws org.xml.sax.SAXNotRecognizedException,
                                          org.xml.sax.SAXNotSupportedException;
    public Object getProperty(String name) throws org.xml.sax.SAXNotRecognizedException,
                                          org.xml.sax.SAXNotSupportedException;
    public void parse(String systemId) throws org.xml.sax.SAXException, java.io.IOException;
    public void parse(org.xml.sax.InputSource input) throws org.xml.sax.SAXException,
                                          java.io.IOException;
    public void setContentHandler(org.xml.sax.ContentHandler handler);
    public void setDTDHandler(org.xml.sax.DTDHandler handler);
    public void setEntityResolver(org.xml.sax.EntityResolver resolver);
    public void setErrorHandler(org.xml.sax.ErrorHandler handler);
    public void setFeature(String name, boolean state)
         throws org.xml.sax.SAXNotRecognizedException, org.xml.sax.SAXNotSupportedException;
    public void setProperty(String name, Object value)
         throws org.xml.sax.SAXNotRecognizedException, org.xml.sax.SAXNotSupportedException;
}
```

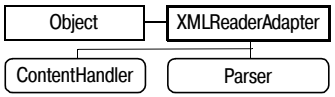
XMLReaderAdapter

Java 1.4

org.xml.sax.helpers

Данный адаптерный класс обертывает (wraps) объект `XMLReader` стандарта SAX2 и заставляет его вести себя аналогично объекту `Parser` стандарта SAX1. Это необходимо при работе со старым API, которому необходим устаревший объект `Parser`. Создайте `XMLReaderAdapter` путем передачи `XMLReader` конструктору `XMLReaderAdapter()`. Получившийся в результате объект можно применять так же, как и любой другой объект `Parser` стандарта SAX1. Этот класс реализует `ContentHandler`, что позволяет ему получать события SAX от `XMLReader`. Кроме того, он реализует интерфейс `Parser`, который позволяет ему зарегистрировать `DocumentHandler` стандарта SAX1. Методы `ContentHandler`

реализуются для вызова соответствующих методов зарегистрированного DocumentHandler.




```
public class XMLReaderAdapter implements org.xml.sax.ContentHandler, org.xml.sax.Parser {
// Открытые конструкторы
   public XMLReaderAdapter() throws org.xml.sax.SAXException;
   public XMLReaderAdapter(org.xml.sax.XMLReader xmlReader);
// Методы, реализующие ContentHandler
   public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXException;
   public void endDocument() throws org.xml.sax.SAXException;
   public void endElement(String uri, String localName, String qName)
                                                        throws org.xml.sax.SAXException;
   public void endPrefixMapping(String prefix);                                    // пустой
   public void ignorableWhitespace(char[] ch, int start, int length)
                                                        throws org.xml.sax.SAXException;
   public void processingInstruction(String target, String data)
                                                        throws org.xml.sax.SAXException;
   public void setDocumentLocator(org.xml.sax.Locator locator);
   public void skippedEntity(String name) throws org.xml.sax.SAXException;         // пустой
   public void startDocument() throws org.xml.sax.SAXException;
   public void startElement(String uri, String localName, String qName,
                            g.xml.sax.Attributes atts) throws org.xml.sax.SAXException;
   public void startPrefixMapping(String prefix, String uri);                      // пустой
// Методы, реализующие Parser
   public void parse(String systemId) throws java.io.IOException, org.xml.sax.SAXException;
   public void parse(org.xml.sax.InputSource input) throws java.io.IOException,
                                                           org.xml.sax.SAXException;
   public void setDocumentHandler(org.xml.sax.DocumentHandler handler);
   public void setDTDHandler(org.xml.sax.DTDHandler handler);
   public void setEntityResolver(org.xml.sax.EntityResolver resolver);
   public void setErrorHandler(org.xml.sax.ErrorHandler handler);
   public void setLocale(java.util.Locale locale) throws org.xml.sax.SAXException;
}
```

XMLReaderFactory

Java 1.4

org.xml.sax.helpers

Этот класс-фабрика определяет два статических метода-фабрики для создания объектов XMLReader. Первый метод принимает имя класса в качестве своего аргумента. Он динамически загружает и создает экземпляр класса, а затем приводит его к объекту XMLReader. Второй метод-фабрика не принимает никаких аргументов; он читает системное свойство «org.xml.sax.driver» и использует значение этого свойства в качестве имени класса реализации XMLReader для загрузки и создания экземпляра. С помощью метода XMLReaderFactory, вызываемого без аргументов, приложение, которое создает экземпляр SAX-парсера, получает независимость от базовой реализации парсера. Конечный пользователь или администратор системы, в которой работает такое приложение, может легко изменить реализацию парсера, установив системное свойство.

Пакет `javax.xml.parsers` **предоставляет подобный** `SAXParserFactory`**, который в какой-то мере более полезен.**

```
public final class XMLReaderFactory {
// Конструктор отсутствует
// Открытые методы класса
   public static org.xml.sax.XMLReader createXMLReader() throws org.xml.sax.SAXException;
   public static org.xml.sax.XMLReader createXMLReader(String className)
                                                        throws org.xml.sax.SAXException;
}
```

Глава 25

Указатель классов, методов и полей

A

B

C

D

E

engineGetCertificateAlias(): KeyStoreSpi
engineGetCertificateChain(): KeyStoreSpi
engineGetCertificates(): CertStoreSpi
engineGetCertPathEncodings(): CertificateFactorySpi
engineGetClientSessionContext(): SSLContextSpi
engineGetCreationDate(): KeyStoreSpi
engineGetCRLs(): CertStoreSpi
engineGetDigestLength(): MessageDigestSpi
engineGetEncoded(): AlgorithmParametersSpi
engineGetIV(): CipherSpi
engineGetKey(): KeyStoreSpi
engineGetKeyManagers(): KeyManagerFactorySpi
engineGetKeySize(): CipherSpi
engineGetKeySpec(): KeyFactorySpi, SecretKeyFactorySpi
engineGetMacLength(): MacSpi
engineGetOutputSize(): CipherSpi, ExemptionMechanismSpi
engineGetParameter(): SignatureSpi
engineGetParameters(): CipherSpi, SignatureSpi
engineGetParameterSpec(): AlgorithmParametersSpi
engineGetServerSessionContext(): SSLContextSpi
engineGetServerSocketFactory(): SSLContextSpi
engineGetSocketFactory(): SSLContextSpi
engineGetTrustManagers(): TrustManagerFactorySpi
engineInit(): AlgorithmParameterGeneratorSpi, AlgorithmParametersSpi, CipherSpi, ExemptionMechanismSpi, KeyAgreementSpi, KeyGeneratorSpi, KeyManagerFactorySpi, MacSpi, SSLContextSpi, TrustManagerFactorySpi
engineInitSign(): SignatureSpi
engineInitVerify(): SignatureSpi
engineIsCertificateEntry(): KeyStoreSpi
engineIsKeyEntry(): KeyStoreSpi
engineLoad(): KeyStoreSpi
engineNextBytes(): SecureRandomSpi
engineReset(): MacSpi, MessageDigestSpi
engineSetCertificateEntry(): KeyStoreSpi
engineSetKeyEntry(): KeyStoreSpi
engineSetMode(): CipherSpi
engineSetPadding(): CipherSpi
engineSetParameter(): SignatureSpi
engineSetSeed(): SecureRandomSpi
engineSign(): SignatureSpi
engineSize(): KeyStoreSpi
engineStore(): KeyStoreSpi
engineToString(): AlgorithmParametersSpi
engineTranslateKey(): KeyFactorySpi, SecretKeyFactorySpi
engineUnwrap(): CipherSpi
engineUpdate(): CipherSpi, MacSpi, MessageDigestSpi, SignatureSpi
engineValidate(): CertPathValidatorSpi
engineVerify(): SignatureSpi
engineWrap(): CipherSpi
ENGLISH: Locale
enqueue(): Reference
ensureCapacity(): ArrayList, StringBuffer, Vector
entering(): Logger
Entity: org.w3c.dom
ENTITY_NODE: Node
ENTITY_REFERENCE_NODE: Node
EntityReference: org.w3c.dom
EntityResolver: org.xml.sax
entries(): Acl, JarFile, ZipFile
Entry: java.util.Map
entrySet(): AbstractMap, Attributes, HashMap, Hashtable, IdentityHashMap, Map, Provider, TreeMap, WeakHashMap
enumerate(): Thread, ThreadGroup
Enumeration: java.util
enumeration(): Collections
eof: OptionalDataException
EOFException: java.io
eolIsSignificant(): StreamTokenizer
eos: GZIPInputStream
equals(): AbstractList, AbstractMap, AbstractSet, AccessControlContext, AllPermission, Arrays, Attribute, Attributes, BasicPermission, BigDecimal, BigInteger, BitSet, Boolean, Byte, ByteBuffer, Calendar, Certificate, CertPath, ChannelBinding, Character, CharBuffer, Charset, ChoiceFormat, CodeSource, CollationKey, Collator, Collection, Comparator, Constructor, Date, DateFormat, DateFormatSymbols, DecimalFormat, DecimalFormatSymbols, DelegationPermission, Double, DoubleBuffer, Entry, Field, FieldPosition, File, FilePermission, Float, FloatBuffer, GregorianCalendar, GSSCredential, GSSName, Hashtable, Identity, IdentityHashMap, IndexedPropertyDescriptor, Inet4Address, Inet6Address, InetAddress, InetSocketAddress, IntBuffer, Integer, KerberosPrincipal, Level, List, Locale, Long, LongBuffer, Manifest, Map, MessageFormat, Method, Name, NetworkInterface, NumberFormat, Object, Oid, ParsePosition, Permission, Principal, PrivateCredentialPermission, PropertyDescriptor, PropertyPermission,

F

G

getBuffer(): StringWriter
getBundle(): ResourceBundle
getByAddress(): InetAddress
getByInetAddress(): NetworkInterface
getByName(): InetAddress, NetworkInterface
getByte(): Array, Field
getByteArray(): AbstractPreferences, Preferences
getBytes(): String
getByteStream(): InputSource
getCalendar(): DateFormat
getCalendarField(): Field
getCallback(): UnsupportedCallbackException
getCAName(): TrustAnchor
getCanonicalFile(): File
getCanonicalHostName(): InetAddress
getCanonicalPath(): File
getCAPublicKey(): TrustAnchor
getCause(): CertPathBuilderException, CertPathValidatorException, CertStoreException, ClassNotFoundException, ExceptionInInitializerError, InvocationTargetException, PrivilegedActionException, Throwable, TransformerException, UndeclaredThrowableException, WriteAbortedException
getCertificate(): KeyStore, X500PrivateCredential, X509CertSelector
getCertificateAlias(): KeyStore
getCertificateChain(): KeyStore, X509KeyManager
getCertificateChecking(): X509CRLSelector
getCertificates(): CertPath, CertStore, CodeSource, JarEntry, JarURLConnection
getCertificateValid(): X509CertSelector
getCertPath(): CertPathBuilderResult, CertPathValidatorException, PKIXCertPathBuilderResult
getCertPathCheckers(): PKIXParameters
getCertPathEncodings(): CertificateFactory
getCertStoreParameters(): CertStore
getCertStores(): PKIXParameters
getChannel(): DatagramSocket, FileInputStream, FileOutputStream, RandomAccessFile, ServerSocket, Socket
getChar(): Array, ByteBuffer, Field
getCharacterInstance(): BreakIterator
getCharacterStream(): InputSource
getChars(): String, StringBuffer
getCharsetName(): IllegalCharsetNameException, UnsupportedCharsetException
getChecksum(): CheckedInputStream, CheckedOutputStream
getChild(): AbstractPreferences, NodeChangeEvent
getChildBeanContextChild(): BeanContextSupport
getChildBeanContextMembershipListener(): BeanContextSupport
getChildBeanContextServicesListener(): BeanContextServicesSupport
getChildNodes(): Node
getChildPropertyChangeListener(): BeanContextSupport
getChildren(): PolicyNode
getChildSerializable(): BeanContextSupport
getChildVetoableChangeListener(): BeanContextSupport
getChildVisibility(): BeanContextSupport
getChoices(): ChoiceCallback
getCipherSuite(): HandshakeCompletedEvent, HttpsURLConnection, SSLSession
getClass(): Object
getClassContext(): SecurityManager
getClasses(): Class
getClassLoader(): Class, ProtectionDomain
getClassName(): MissingResourceException, StackTraceElement
getClient(): KerberosTicket
getClientAddresses(): KerberosTicket
getClientAliases(): X509KeyManager
getClientSessionContext(): SSLContext
getCodeSource(): ProtectionDomain
getCollationElementIterator(): RuleBasedCollator
getCollationKey(): Collator, RuleBasedCollator
getCollection(): CollectionCertStoreParameters
getColumnNumber(): Locator, LocatorImpl, SAXParseException, SourceLocator
getComment(): ZipEntry
getComponent(): BeanContextChildComponentProxy
getComponentType(): Class
getCompressedSize(): ZipEntry
getConfiguration(): Configuration
getConfState(): GSSContext
getConstructor(): Class
getConstructors(): Class
getContainer(): BeanContextContainerProxy
getContent(): ContentHandler, URL, URLConnection
getContentEncoding(): URLConnection
getContentHandler(): ParserAdapter, XMLFilterImpl, XMLReader
getContentLength(): URLConnection
getContents(): ListResourceBundle
getContentType(): URLConnection
getContentTypeFor(): FileNameMap
getContext(): AccessController

getElementById(): Document
getElementsByTagName(): Document, Element
getElementsByTagNameNS(): Document, Element
getEnabledCipherSuites(): SSLServerSocket, SSLSocket
getEnabledProtocols(): SSLServerSocket, SSLSocket
getEnableSessionCreation(): SSLServerSocket, SSLSocket
getEncoded(): AlgorithmParameters, Certificate, CertPath, EncodedKeySpec, EncryptedPrivateKeyInfo, KerberosKey, KerberosTicket, Key, PKCS8EncodedKeySpec, PolicyQualifierInfo, SecretKeySpec, X500Principal, X509CRL, X509CRLEntry, X509EncodedKeySpec
getEncoding(): Handler, InputSource, InputStreamReader, OutputStreamWriter
getEncodings(): CertPath
getEncryptedData(): EncryptedPrivateKeyInfo
getEndIndex(): CharacterIterator, FieldPosition, StringCharacterIterator
getEndTime(): KerberosTicket
getEntities(): DocumentType
getEntityResolver(): ParserAdapter, XMLFilterImpl, XMLReader
getEntries(): Manifest
getEntry(): JarFile, ZipFile
getEntryName(): JarURLConnection
getenv(): System
getEras(): DateFormatSymbols
getErrorHandler(): ParserAdapter, XMLFilterImpl, XMLReader
getErrorIndex(): ParsePosition
getErrorListener(): Transformer, TransformerFactory
getErrorManager(): Handler
getErrorOffset(): ParseException
getErrorStream(): HttpURLConnection, Process
getEventPropertyName(): EventHandler
getEventSetDescriptors(): BeanInfo, SimpleBeanInfo
getException(): ClassNotFoundException, ExceptionInInitializerError, FactoryConfigurationError, PrivilegedActionException, SAXException, TransformerException, TransformerFactoryConfigurationError
getExceptionListener(): Encoder, XMLDecoder
getExceptionTypes(): Constructor, Method
getExemptionMechanism(): Cipher
getExpectedPolicies(): PolicyNode
getExpiration(): URLConnection
getExponent(): RSAOtherPrimeInfo
getExponentSize(): DHGenParameterSpec
getExtendedKeyUsage(): X509Certificate, X509CertSelector
getExtensionValue(): X509Extension
getExtra(): ZipEntry
getFD(): FileInputStream, FileOutputStream, RandomAccessFile
getFeature(): ParserAdapter, SAXParserFactory, TransformerFactory, XMLFilterImpl, XMLReader
GetField: java.io.ObjectInputStream
getField(): Class, FieldPosition, ObjectStreamClass
getFieldAttribute(): FieldPosition
getFields(): Class, ObjectStreamClass
getFile(): URL
getFileDescriptor(): DatagramSocketImpl, SocketImpl
getFileName(): StackTraceElement
getFileNameMap(): URLConnection
getFilePointer(): RandomAccessFile
getFilter(): Handler, Logger
getFirst(): LinkedList
getFirstChild(): Node
getFirstDayOfWeek(): Calendar
getFlags(): KerberosTicket
getFloat(): AbstractPreferences, Array, ByteBuffer, Field, Preferences
getFollowRedirects(): HttpURLConnection
getFormat(): Certificate, EncodedKeySpec, KerberosKey, Key, PKCS8EncodedKeySpec, SecretKeySpec, X509EncodedKeySpec
getFormats(): ChoiceFormat, MessageFormat
getFormatsByArgumentIndex(): MessageFormat
getFormatter(): Handler
getFragment(): URI
getG(): DHParameterSpec, DHPrivateKeySpec, DHPublicKeySpec, DSAParameterSpec, DSAParams, DSAPrivateKeySpec, DSAPublicKeySpec
getGetListenerMethod(): EventSetDescriptor
getGreatestMinimum(): Calendar, GregorianCalendar
getGregorianChange(): GregorianCalendar
getGroupingSeparator(): DecimalFormatSymbols
getGroupingSize(): DecimalFormat
getGuarantor(): Certificate
getHandler(): SAXResult
getHandlers(): Logger
getHead(): Formatter, XMLFormatter

getKeyUsage(): X509Certificate, X509CertSelector
getL(): DHParameterSpec
getLanguage(): Locale
getLast(): LinkedList
getLastAccessedTime(): SSLSession
getLastChild(): Node
getLastModified(): URLConnection
getLeastMaximum(): Calendar, GregorianCalendar
getLength(): Array, AttributeList, AttributeListImpl, Attributes, AttributesImpl, Bidi, CharacterData, DatagramPacket, NamedNodeMap, NodeList
getLevel(): Handler, Logger, LogRecord
getLevelAt(): Bidi
getLexicalHandler(): SAXResult
getLifetime(): GSSContext
getLimits(): ChoiceFormat
getLineInstance(): BreakIterator
getLineNumber(): LineNumberInputStream, LineNumberReader, Locator, LocatorImpl, SAXParseException, SourceLocator, StackTraceElement
getListener(): EventListenerProxy
getListenerMethodDescriptors(): EventSetDescriptor
getListenerMethodName(): EventHandler
getListenerMethods(): EventSetDescriptor
getListenerType(): EventSetDescriptor
getLocalAddress(): DatagramSocket, Socket
getLocalCertificates(): HandshakeCompletedEvent, HttpsURLConnection, SSLSession
getLocale(): BeanContextSupport, LanguageCallback, MessageFormat, ResourceBundle
getLocalHost(): InetAddress
getLocalizedInputStream(): Runtime
getLocalizedMessage(): Throwable
getLocalizedName(): Level
getLocalizedOutputStream(): Runtime
getLocalName(): Attributes, AttributesImpl, Node
getLocalPatternChars(): DateFormatSymbols
getLocalPort(): DatagramSocket, DatagramSocketImpl, ServerSocket, Socket, SocketImpl
getLocalSocketAddress(): DatagramSocket, ServerSocket, Socket
getLocation(): CodeSource
getLocationAsString(): TransformerException
getLocator(): TransformerException
getLogger(): Logger, LogManager
getLoggerName(): LogRecord
getLoggerNames(): LogManager
getLoginModuleName(): AppConfigurationEntry
getLogManager(): LogManager
getLong(): AbstractPreferences, Array, ByteBuffer, Field, Long, Preferences
getLoopbackMode(): MulticastSocket
getLowestSetBit(): BigInteger
getMacLength(): Mac
getMainAttributes(): JarURLConnection, Manifest
getMajor(): GSSException
getMajorString(): GSSException
getManifest(): JarFile, JarInputStream, JarURLConnection
getMatchAllSubjectAltNames(): X509CertSelector
getMaxCRL(): X509CRLSelector
getMaxExpansion(): CollationElementIterator
getMaximum(): Calendar, GregorianCalendar
getMaximumFractionDigits(): NumberFormat
getMaximumIntegerDigits(): NumberFormat
getMaxPathLength(): PKIXBuilderParameters
getMaxPriority(): ThreadGroup
getMech(): GSSContext
getMechs(): GSSCredential, GSSManager
getMechsForName(): GSSManager
getMessage(): CertPathBuilderException, CertPathValidatorException, CertStoreException, FactoryConfigurationError, GSSException, InvalidClassException, LogRecord, MalformedInputException, PatternSyntaxException, SAXException, TextOutputCallback, Throwable, TransformerFactoryConfigurationError, UnmappableCharacterException, URISyntaxException, WriteAbortedException
getMessageAndLocation(): TransformerException
getMessageDigest(): DigestInputStream, DigestOutputStream
getMessageType(): ConfirmationCallback, TextOutputCallback
getMethod(): Class, MethodDescriptor, ZipEntry
getMethodDescriptors(): BeanInfo, SimpleBeanInfo
getMethodName(): StackTraceElement, Statement
getMethods(): Class
getMIC(): GSSContext
getMillis(): LogRecord
getMinCRL(): X509CRLSelector
getMinimalDaysInFirstWeek(): Calendar

getPassword(): PasswordAuthentication, PasswordCallback, PBEKey, PBEKeySpec
getPasswordAuthentication(): Authenticator
getPath(): File, URI, URL
getPathToNames(): X509CertSelector
getPattern(): PatternSyntaxException
getPatternSeparator(): DecimalFormatSymbols
getPeerCertificateChain(): HandshakeCompletedEvent, SSLSession
getPeerCertificates(): HandshakeCompletedEvent, SSLSession
getPeerHost(): SSLSession
getPercent(): DecimalFormatSymbols
getPercentInstance(): NumberFormat
getPerMill(): DecimalFormatSymbols
getPermission(): AccessControlException, HttpURLConnection, URLConnection
getPermissions(): Acl, Policy, ProtectionDomain, SecureClassLoader, URLClassLoader
getPersistenceDelegate(): Encoder
getPolicy(): Policy, X509CertSelector
getPolicyQualifier(): PolicyQualifierInfo
getPolicyQualifierId(): PolicyQualifierInfo
getPolicyQualifiers(): PolicyNode
getPolicyQualifiersRejected(): PKIXParameters
getPolicyTree(): PKIXCertPathValidatorResult
getPort(): DatagramPacket, DatagramSocket, InetSocketAddress, LDAPCertStoreParameters, Socket, SocketImpl, URI, URL
getPositivePrefix(): DecimalFormat
getPositiveSuffix(): DecimalFormat
getPrefix(): NamespaceSupport, Node
getPrefixes(): NamespaceSupport
getPreviousSibling(): Node
getPrime(): RSAOtherPrimeInfo
getPrimeExponentP(): RSAMultiPrimePrivateCrtKey, RSAMultiPrimePrivateCrtKeySpec, RSAPrivateCrtKey, RSAPrivateCrtKeySpec
getPrimeExponentQ(): RSAMultiPrimePrivateCrtKey, RSAMultiPrimePrivateCrtKeySpec, RSAPrivateCrtKey, RSAPrivateCrtKeySpec
getPrimeP(): RSAMultiPrimePrivateCrtKey, RSAMultiPrimePrivateCrtKeySpec, RSAPrivateCrtKey, RSAPrivateCrtKeySpec
getPrimeQ(): RSAMultiPrimePrivateCrtKey, RSAMultiPrimePrivateCrtKeySpec, RSAPrivateCrtKey, RSAPrivateCrtKeySpec
getPrimeSize(): DHGenParameterSpec
getPrincipal(): AclEntry, Certificate, KerberosKey
getPrincipals(): PrivateCredentialPermission, ProtectionDomain, Subject
getPriority(): Thread
getPrivacy(): MessageProp
getPrivate(): KeyPair
getPrivateCredentials(): Subject
getPrivateExponent(): RSAPrivateKey, RSAPrivateKeySpec
getPrivateKey(): Signer, X500PrivateCredential, X509KeyManager
getPrivateKeyValid(): X509CertSelector
getPrompt(): ChoiceCallback, ConfirmationCallback, NameCallback, PasswordCallback, TextInputCallback
getPropagatedFrom(): BeanContextEvent
getPropagationId(): PropertyChangeEvent
getProperties(): System
getProperty(): LogManager, ParserAdapter, Properties, SAXParser, Security, System, XMLFilterImpl, XMLReader
getPropertyChangeEvent(): PropertyVetoException
getPropertyChangeListeners(): PropertyChangeSupport
getPropertyDescriptors(): BeanInfo, SimpleBeanInfo
getPropertyEditorClass(): PropertyDescriptor
getPropertyName(): PropertyChangeEvent, PropertyChangeListenerProxy, VetoableChangeListenerProxy
getPropertyType(): PropertyDescriptor
getProtectionDomain(): Class
getProtocol(): SSLContext, SSLSession, URL
getProvider(): AlgorithmParameterGenerator, AlgorithmParameters, CertificateFactory, CertPathBuilder, CertPathValidator, CertStore, Cipher, ExemptionMechanism, KeyAgreement, KeyFactory, KeyGenerator, KeyManagerFactory, KeyPairGenerator, KeyStore, Mac, MessageDigest, SecretKeyFactory, SecureRandom, Security, Signature, SSLContext, TrustManagerFactory
getProviders(): Security
getProxyClass(): Proxy
getPublic(): KeyPair
getPublicCredentials(): Subject
getPublicExponent(): RSAKeyGenParameterSpec, RSAMultiPrimePrivateCrtKey, RSAMultiPrimePrivateCrtKeySpec, RSAPrivateCrtKey, RSAPrivateCrtKeySpec, RSAPublicKey, RSAPublicKeySpec
getPublicId(): DocumentType, Entity, InputSource, Locator, LocatorImpl, Notation,

getSessionKey(): KerberosTicket
getSessionKeyType(): KerberosTicket
getSessionTimeout(): SSLSessionContext
getShort(): Array, ByteBuffer, Field
getShortDescription(): FeatureDescriptor
getShortMonths(): DateFormatSymbols
getShortWeekdays(): DateFormatSymbols
getSigAlgName(): X509Certificate, X509CRL
getSigAlgOID(): X509Certificate, X509CRL
getSigAlgParams(): X509Certificate, X509CRL
getSignature(): SignedObject, X509Certificate, X509CRL
getSigners(): Class
getSigProvider(): PKIXParameters
getSize(): ZipEntry
getSocket(): HandshakeCompletedEvent
getSocketAddress(): DatagramPacket
getSocketFactory(): SSLContext
getSoLinger(): Socket
getSoTimeout(): DatagramSocket, ServerSocket, Socket
getSource(): EventObject
getSourceAsBeanContextServices(): BeanContextServiceAvailableEvent, BeanContextServiceRevokedEvent
getSourceClassName(): LogRecord
getSourceMethodName(): LogRecord
getSourceString(): CollationKey
getSpecificationTitle(): Package
getSpecificationVendor(): Package
getSpecificationVersion(): Package
getSpecified(): Attr
getSpi(): AbstractPreferences
getSrcName(): GSSContext
getSSLSocketFactory(): HttpsURLConnection
getStackTrace(): Throwable
getStartTime(): KerberosTicket
getStrength(): Collator
getString(): ResourceBundle
getStringArray(): ResourceBundle
getStringNameType(): GSSName
getSubject(): LoginContext, Subject, SubjectDomainCombiner
getSubjectAlternativeNames(): X509Certificate, X509CertSelector
getSubjectAsBytes(): X509CertSelector
getSubjectAsString(): X509CertSelector
getSubjectDN(): X509Certificate
getSubjectKeyIdentifier(): X509CertSelector
getSubjectPublicKey(): X509CertSelector
getSubjectPublicKeyAlgID(): X509CertSelector
getSubjectUniqueID(): X509Certificate
getSubjectX500Principal(): X509Certificate
getSuperclass(): Class
getSupportedCipherSuites(): SSLServerSocket, SSLServerSocketFactory, SSLSocket, SSLSocketFactory
getSupportedExtensions(): PKIXCertPathChecker
getSupportedProtocols(): SSLServerSocket, SSLSocket
getSymbol(): Currency
getSystemClassLoader(): ClassLoader
getSystemId(): DocumentType, DOMResult, DOMSource, Entity, InputSource, Locator, LocatorImpl, Notation, Result, SAXParseException, SAXResult, SAXSource, Source, SourceLocator, StreamResult, StreamSource, TemplatesHandler, TransformerHandler
getSystemResource(): ClassLoader
getSystemResourceAsStream(): ClassLoader
getSystemResources(): ClassLoader
getSystemScope(): IdentityScope
getTagName(): Element
getTags(): PropertyEditor, PropertyEditorSupport
getTail(): Formatter, XMLFormatter
getTarget(): EventHandler, ProcessingInstruction, Statement
getTargetCertConstraints(): PKIXParameters
getTargetException(): InvocationTargetException
getTargName(): GSSContext
getTBSCertificate(): X509Certificate
getTBSCertList(): X509CRL
getTcpNoDelay(): Socket
getTemplates(): TemplatesHandler
getText(): BreakIterator, TextInputCallback
getThisUpdate(): X509CRL
getThreadGroup(): SecurityManager, Thread
getThreadID(): LogRecord
getThrown(): LogRecord
getTime(): Calendar, Date, ZipEntry
getTimeInMillis(): Calendar
getTimeInstance(): DateFormat
getTimeToLive(): DatagramSocketImpl, MulticastSocket
getTimeZone(): Calendar, DateFormat, TimeZone
getTimezoneOffset(): Date
getTotalIn(): Deflater, Inflater
getTotalOut(): Deflater, Inflater
getTrafficClass(): DatagramSocket, Socket
getTransformer(): TransformerHandler
getTrustAnchor(): PKIXCertPathValidatorResult

H

I

J

K

L

M

N

O

P

Q

R

registerEditor(): PropertyEditorManager
registerValidation(): ObjectInputStream
rehash(): Hashtable
rejectedSetBCOnce: BeanContextChildSupport
relativize(): URI
release(): FileLock
releaseBeanContextResources(): BeanContextChildSupport, BeanContextServicesSupport
releaseService(): BCSSProxyServiceProvider, BeanContextServiceProvider, BeanContextServices, BeanContextServicesSupport
remainder(): BigInteger
remaining(): Buffer
remove(): AbstractCollection, AbstractList, AbstractMap, AbstractPreferences, AbstractSequentialList, ArrayList, Attributes, BCSIterator, BeanContextSupport, Collection, Dictionary, Encoder, HashMap, HashSet, Hashtable, IdentityHashMap, Iterator, LinkedList, List, ListIterator, Map, Preferences, Provider, ReferenceQueue, Set, TreeMap, TreeSet, Vector, WeakHashMap
removeAll(): AbstractCollection, AbstractSet, BeanContextSupport, Collection, List, Set, Vector
removeAllElements(): Vector
removeAttribute(): AttributeListImpl, AttributesImpl, Element
removeAttributeNode(): Element
removeAttributeNS(): Element
removeBeanContextMembershipListener(): BeanContext, BeanContextSupport
removeBeanContextServicesListener(): BeanContextServices, BeanContextServicesSupport
removeCertificate(): Identity
removeChild(): Node
removeEldestEntry(): LinkedHashMap
removeElement(): Vector
removeElementAt(): Vector
removeEntry(): Acl
removeFirst(): LinkedList
removeHandler(): Logger
removeHandshakeCompletedListener(): SSLSocket
removeIdentity(): IdentityScope
removeLast(): LinkedList
removeMember(): Group
removeNamedItem(): NamedNodeMap
removeNamedItemNS(): NamedNodeMap
removeNode(): AbstractPreferences, Preferences
removeNodeChangeListener(): AbstractPreferences, Preferences
removeNodeSpi(): AbstractPreferences
removePermission(): AclEntry
removePreferenceChangeListener(): AbstractPreferences, Preferences
removePropertyChangeListener(): BeanContextChild, BeanContextChildSupport, Customizer, LogManager, PropertyChangeSupport, PropertyEditor, PropertyEditorSupport
removeProvider(): Security
removeRange(): AbstractList, ArrayList, Vector
removeShutdownHook(): Runtime
removeSpi(): AbstractPreferences
removeValue(): SSLSession
removeVetoableChangeListener(): BeanContextChild, BeanContextChildSupport, VetoableChangeSupport
renameTo(): File
reorderVisually(): Bidi
REPLACE: CodingErrorAction
replace(): String, StringBuffer
replaceAll(): Collections, Matcher, String
replaceChild(): Node
replaceData(): CharacterData
replaceFirst(): Matcher, String
replacement(): CharsetDecoder, CharsetEncoder
replaceObject(): ObjectOutputStream
replaceWith(): CharsetDecoder, CharsetEncoder
REPORT: CodingErrorAction
reportError(): Handler
requestAnonymity(): GSSContext
requestConf(): GSSContext
requestCredDeleg(): GSSContext
requestInteg(): GSSContext
requestLifetime(): GSSContext
requestMutualAuth(): GSSContext
requestPasswordAuthentication(): Authenticator
requestReplayDet(): GSSContext
requestSequenceDet(): GSSContext
REQUIRED: LoginModuleControlFlag
requiresBidi(): Bidi
REQUISITE: LoginModuleControlFlag
reset(): Adler32, Buffer, BufferedInputStream, BufferedReader, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, CharsetDecoder, CharsetEncoder, Checksum, CollationElementIterator, CRC32, Deflater, FilterInputStream, FilterReader, Inflater, InputStream, LineNumberInputStream,

S

setInputStream(): StreamSource
setInstanceFollowRedirects(): HttpURLConnection
setInt(): Array, Field
setInterface(): MulticastSocket
setInternationalCurrencySymbol(): DecimalFormatSymbols
setIssuer(): X509CertSelector
setIssuerNames(): X509CRLSelector
setKeepAlive(): Socket
setKeyEntry(): KeyStore
setKeyPair(): Signer
setKeyUsage(): X509CertSelector
setLastModified(): File
setLength(): DatagramPacket, RandomAccessFile, StringBuffer
setLenient(): Calendar, DateFormat
setLevel(): Deflater, Handler, Logger, LogRecord, ZipOutputStream
setLexicalHandler(): SAXResult
setLineNumber(): LineNumberInputStream, LineNumberReader, LocatorImpl
setLocale(): BeanContextSupport, LanguageCallback, MessageFormat, Parser, XMLReaderAdapter
setLocalName(): AttributesImpl
setLocalPatternChars(): DateFormatSymbols
setLocator(): TransformerException
setLoggerName(): LogRecord
setLong(): Array, Field
setLoopbackMode(): MulticastSocket
setMatchAllSubjectAltNames(): X509CertSelector
setMaxCRLNumber(): X509CRLSelector
setMaximumFractionDigits(): DecimalFormat, NumberFormat
setMaximumIntegerDigits(): DecimalFormat, NumberFormat
setMaxPathLength(): PKIXBuilderParameters
setMaxPriority(): ThreadGroup
setMessage(): LogRecord
setMessageDigest(): DigestInputStream, DigestOutputStream
setMethod(): ZipEntry, ZipOutputStream
setMillis(): LogRecord
setMinCRLNumber(): X509CRLSelector
setMinimalDaysInFirstWeek(): Calendar
setMinimumFractionDigits(): DecimalFormat, NumberFormat
setMinimumIntegerDigits(): DecimalFormat, NumberFormat
setMinor(): GSSException
setMinusSign(): DecimalFormatSymbols
setMinutes(): Date
setMonetaryDecimalSeparator(): DecimalFormatSymbols
setMonth(): Date
setMonths(): DateFormatSymbols
setMultiplier(): DecimalFormat
setName(): Acl, FeatureDescriptor, NameCallback, Thread
setNameConstraints(): X509CertSelector
setNamedItem(): NamedNodeMap
setNamedItemNS(): NamedNodeMap
setNamespaceAware(): DocumentBuilderFactory, SAXParserFactory
setNaN(): DecimalFormatSymbols
setNeedClientAuth(): SSLServerSocket, SSLSocket
setNegativePermissions(): AclEntry
setNegativePrefix(): DecimalFormat
setNegativeSuffix(): DecimalFormat
setNetworkInterface(): MulticastSocket
setNode(): DOMResult, DOMSource
setNodeValue(): Node
setNumberFormat(): DateFormat
setObject(): Customizer
setOffset(): CollationElementIterator, ObjectStreamField
setOOBInline(): Socket
setOption(): SocketOptions
setOut(): System
setOutputProperties(): Transformer
setOutputProperty(): Transformer
setOutputStream(): StreamHandler, StreamResult
setOwner(): XMLDecoder, XMLEncoder
setPackageAssertionStatus(): ClassLoader
setParameter(): Signature, Transformer
setParameters(): LogRecord
setParent(): Logger, ResourceBundle, XMLFilter, XMLFilterImpl
setParseIntegerOnly(): NumberFormat
setPassword(): PasswordCallback
setPathToNames(): X509CertSelector
setPatternSeparator(): DecimalFormatSymbols
setPercent(): DecimalFormatSymbols
setPerMill(): DecimalFormatSymbols
setPersistenceDelegate(): Encoder
setPolicy(): Policy, X509CertSelector
setPolicyMappingInhibited(): PKIXParameters
setPolicyQualifiersRejected(): PKIXParameters
setPort(): DatagramPacket
setPositivePrefix(): DecimalFormat
setPositiveSuffix(): DecimalFormat
setPreferred(): FeatureDescriptor
setPrefix(): Node

T

U

V

W

X

Y

Z

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

А

Б

B

Г

И

K

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ц

Ч

Ш

Э

Я