Томас М., Пател П., Хадсон А., Болл Д.

Секреты программирования для Internet на JAVA

3. Объектная ориентация в Java

4. Синтаксис и семантика

5. Апплет в работе

6. Интерфейс прикладного программирования

7. Пользовательский интерфейс

Часть III. Программирование на Java

8. Еще об интерфейсе пользователя

9. Графика и изображения

10. Структура программы

11. Многопотоковость

12. Программирование за рамками модели апплета

Часть IV. Java и Сеть

13. Работа с сетью на уровне сокетов и потоков

14. Связь по сети с помощью URL

15. Разработка серверов на Java

Часть V. Примеры приложений Интернет

16. Интерактивная анимация: рекламный апплет

17. Взаимодействие с CGI: Java-магазин

18. Взаимодействие с серверами других протоколов: шахматный клиент

19. Как написать свой собственный сервер: планировщик встреч

Часть VI. Приложения

Приложение A. О странице Online Companion

Приложение B. Диск CD-ROM

Введение

Java - это мощный современный язык программирования, разработанный фирмой Sun Microsystems. Поначалу его планировали применять в системах интерактивного телевидения, однако когда Sun выпустила HotJava, броузер World Wide Web, позволяющий "прокручивать" внутри себя небольшие программы, иначе называемые апплетами (от англ. applet, "приложеньице"), вызываемые из Web-страниц, Java серьезно заинтересовал сообщество глобальной компьютерной сети Интернет. Вскоре после этого возможность работы с апплетами была добавлена в самый распространенный Web-броузер - Netscape Navigator 2.0. На сегодняшний день встроенные в Web-страницы апплеты на языке Java стали обязательным атрибутом каждого Web-сервера, претендующего на применение "высокой технологии".

Достоинство языка Java, конечно, состоит не только в том, что программы на нем можно размещать на Web-страницах. Кроме этого, Java просто мощный и легкий в изучении объектно-ориентированный язык. С его помощью решаются многие из повседневных сложных проблем, с которыми приходится встречаться программистам, разрабатывающим устойчивые, хорошо работающие приложения. Java при помощи класса thread обеспечивает многопотоковость приложений, а также самостоятельно, в фоновом режиме, производит сборку мусора (garbage collection), освобождая ненужные области памяти. Интерфейс прикладного программирования Java (API), входящий в состав комплекта разработчика Java Developers Kit, созданного фирмой Sun, дает программисту независимый от операционной среды доступ к необходимым для создания сложных приложений Интернет средствам, таким как сетевые сокеты и графическая оконная система.

Идея независимости программы от платформы, на которой она исполняется, стала реальностью при помощи Java. Java-апплеты в состоянии работать на любом компьютере, на котором можно запустить Web-броузер, поддерживающий Java. Самостоятельные Java-приложения компилируются в машинонезависимый байтовый код, выполняющийся без изменений на любом компьютере с Java-интерпретатором. Таким образом, Java - первый язык программирования, претендующий на звание по-настоящему независимого от компьютерной платформы.

Об этой книге

За последние месяцы слово "Java" стало известно практически всем. Однако для многих программистов и разработчиков WWW Java по-прежнему остается тайной. Одной из причин этого является частое использование Java для разработки Web-серверов, в результате чего многие программисты считают, что Java - всего лишь новое средство для создания более сложных и умных страниц WWW.

Такое предположение имеет под собой все основания, однако эта книга ставит своей целью развеять представление о Java как языке для описания домашних Web-страниц. Мы надеемся, что, с одной стороны, она расширит возможности разработчиков Web-серверов, а с другой - поможет программистам превратить Web в платформу программирования.

Первое и самое главное в Java - его новизна. Первые четыре главы посвящены описанию структуры языка, его достоинствам по сравнению с другими языками, синтаксису и семантике. Далее мы сосредоточимся на написании апплетов и посвятим им всю оставшуюся часть книги. В четвертой части мы обсудим, каким образом апплеты общаются друг с другом по Сети. Показав, как апплет взаимодействует с существующими серверами Интернет и как создать свой собственный сервер, мы продемонстрируем создание на базе апплетов по-настоящему распределенных сетевых программ.

Книга заканчивается описанием четырех сравнительно больших проектов, разобранных нами с самого начала и до конца. Они были выдуманы с целью отразить потребности реального мира, с которыми вам, возможно, придется столкнуться. Кроме того, мы надеемся, что, исследуя проекты, вы сможете свести все полученные из книги знания воедино.

CD-ROM и Online Companion

К книге прилагается диск CD-ROM; кроме того, для вас доступна Web-страница под названием "Online Companion". На диске находится комплект разработчика Java (Java Developers Kit, JDK) для операционных систем Windows 95/NT и Macintosh. JDK для UNIX можно найти на странице Online Companion. Кроме того, на диске находятся все рассматриваемые в книге программы-примеры, а также различные дополнительные апплеты, программы и утилиты.

Web-страница Online Companion (http://ww.vmedia.com/java.html) обеспечит вас последними новостями из мира Java. Язык Java, можно сказать, до сих пор находится в младенческом возрасте, поэтому ни одна книга по Java не может считаться полной даже спустя всего месяц после выхода из печати. Поэтому и была создана Online Companion - здесь вы найдете списки последних изменений в языке, последние достижения в области компиляторов и средств разработки и просто новости из мира Java.

Требования к аппаратным и программным средствам

Апплеты и программы на Java можно разрабатывать на любом компьютере, оборудованном компилятором Java. Фирма Sun выпустила JDK для следующих платформ:

- Microsoft Windows 95 и Windows NT,
- Sun Solaris 2,
- Apple Macintosh.

В добавление к этим платформам существуют компиляторы и для других платформ, в основном вариантов UNIX, включая Linux - бесплатную, совместимую с UNIX операционную систему для процессоров Intelx86 и DEC Alpha.

Просматриваются апплеты в широко известном броузере Netscape Navigator 2.0, доступном для большинства компьютерных платформ. Во время написания книги броузер HotJava, распространявшийся с альфа-версией JDK, был несовместим с современной его версией. Поэтому для разработки апплетов мы его не использовали.

Содержимое книги

Ниже приведено описание каждой главы.

Часть I. Введение в Java

Глава 1, "World Wide Web и Java", познакомит вас с фундаментальными концепциями и понятиями, на которых базируется Java.

В главе 2, "Основы программирования на Java", вы установите комплект разработчика JDK и запустите ваш первый апплет и первое самостоятельное приложение.

Часть II. Апплеты

Глава 3, "Объектная ориентация в Java", представляет собой введение в правила написания программ на объектно-ориентированных языках и поясняет, как объектная ориентация реализована в Java. Читатели, знакомые с концепцией объектной ориентированности, могут пропустить этот материал и приступить к чтению глав, посвященных непосредственно Java.

В главе 4, "Синтаксис и семантика", эти понятия рассматриваются во всех подробностях. Синтаксис Java очень похож на синтаксис языка C, поэтому программисты, уже знакомые с C и C++, могут ограничиться беглым просмотром этой главы. В любом случае мы настоятельно рекомендуем вам обратить внимание на разделы "Массивы" и "Исключения".

Глава 5, "Апплет в работе", научит вас основам программирования интерактивных апплетов. В ней мы стремились дать вам возможность как можно скорее начать писать работающие приложения.

В главе 6, "Интерфейс прикладного программирования", рассматриваются многие полезные классы, встроенные в API, например Vector для работы с векторами или Hashtable для работы с хеш-таблицами.

В главе 7, "Пользовательский интерфейс", мы рассматриваем основные элементы раздела API под названием Abstract Windowing Toolkit (AWT) - подсистемы, дающей программисту возможность эффективно работать с оконными и графическими элементами интерфейса пользователя, например с меню выбора, кнопками, полосами прокрутки и списками. AWT здорово облегчает создание привлекательных и практичных пользовательских интерфейсов в Java-программах и апплетах.

Часть III. Программирование на Java

В главе 8, "Еще об интерфейсе пользователя", описываются более сложные элементы AWT, такие как диалоги, фреймы, меню и менеджеры размещения, входящие в состав JDK.

В главе 9, "Графика и изображения", мы выходим за пределы AWT и учимся самостоятельно рисовать картинки в Java на уровне пикселов или используя графические примитивы.

В главе 10, "Структура программы", описывается методика объединения классов и интерфейсов Java в пакеты так, чтобы их можно было использовать в дальнейших разработках, а также методика защиты кода программы при помощи механизма обработки ошибок.

В главе 11, "Многопотоковость", вы познакомитесь с механизмом многопотоковости в Java, а

также с некоторыми проблемами, возникающими в программе при одновременной работе нескольких потоков.

В главе 12, "Программирование за рамками модели апплета", вы изучите технику программирования самостоятельных Java-приложений. Самостоятельные Java-приложения совместно с использованием в программе машинозависимых процедур позволяют обойти некоторые свойственные апплетам функциональные ограничения.

Часть IV. Java и Сеть

В главе 13, "Работа с сетью на уровне сокетов и потоков", вы научитесь открывать и устанавливать соединения с другими сетевыми компьютерами и познакомитесь с классами Java, предназначенными для ввода-вывода данных.

В главе 14, "Связь по сети с помощью URL", описывается способ доступа к ресурсам Сети из Java-программы при помощи URL.

В главе 15, "Разработка серверов на Java", мы несколько отойдем от апплетов - главной темы книги - для того, чтобы познакомить вас с техникой конструирования самостоятельных приложений-серверов.

Часть V. Примеры приложений Интернет

В этой части содержатся четыре учебные главы.

В главе 16, "Интерактивная анимация: рекламный апплет", показано, как можно сконструировать интерактивный апплет-аниматор, обладающий гибкими возможностями по настройке и конфигурации.

В главе 17, "Взаимодействие с CGI: Java-магазин", мы создадим Java-апплет, предназначенный для работы в качестве виртуального магазина.

В главе 18, "Взаимодействие с серверами других протоколов: шахматный клиент", мы создадим апплет-клиент, предназначенный для игры в шахматы с системой Internet Chess Server, шахматным сервером Интернет.

Глава 19, "Как написать свой собственный сервер: планировщик встреч", заканчивает нашу книгу. В ней описывается процесс разработки собственного протокола для взаимодействия системы клиент-сервер на примере программы планировщика встреч.

Приложения

В приложении А, "О странице Online Companion", рассказывается о дополнительных источниках информации, которые читатель может найти на странице Online Companion.

В приложении Б, "Диск CD-ROM", объясняется, как пользоваться приложенным к книге диском, и описывается его содержимое.

Нумерация примеров

Как уже говорилось выше, на диске CD-ROM находятся тексты всех программ, рассматриваемых в книге. Если, читая книгу, вам вдруг захочется посмотреть вживую, как работает то или иное приложение, взгляните на номер листинга. Предположим, вас заинтересовал "Пример 2-7а". Это значит, что на диске текст этого примера находится в каталоге Chapter2/Example2-7 (глава 2, пример 2-7). Буква "а" в конце номера в данном случае не имеет значения - она просто облегчает ориентацию в тексте книги.

Приступаем

Теперь, когда вы уже знаете, чего можно, а чего нельзя ожидать от этой книги, настало время приступить к изложению самого материала. Мы искренне надеемся, что по мере все более глубоко проникновения в тайны языка Java вы будете получать от этого все больше и больше удовольствия.

Глава 1

World Wide Web и Java

Лишь немногие языки программирования удостаивались такого интереса, какой проявляется по отношению к Java. Вместе с тем лишь немногие языки, и Java в их числе, определяют заранее, чем является написанная на них программа и что программист вообще может сделать, пользуясь данными языковыми средствами. Программы, написанные на других языках, как правило, привязаны к конкретной операционной платформе, а программы на Java - нет. Обычно полученное из Интернет программное обеспечение проверяется на наличие вирусов (кроме случаев, когда вы полностью доверяете лицу или компании-изготовителю), Java же предлагает собственный способ безопасной загрузки и запуска программ из сети.

До появления языка Java программы продавались в коробках. Установленная на компьютере программа всегда занимала определенное место на его жестком диске. Сетевая программа, кроме того, устанавливалась на вашу локальную сеть в соответствии с лицензионным соглашением, а затем вам приходилось следить, следить и еще раз следить за появлением новых версий этой же самой программы. Java-программа появляется перед вами прямо из сети, передаваясь по проводам. Когда она больше не нужна, она исчезает. Вы гарантированно имеете дело только с самой последней версией программы, а программа не испытывает никаких сложностей в получении дополнительной информации по сети.

Кроме выдающихся возможностей работы в сети и независимости от компьютерной платформы, языку Java присуща объектная ориентация и многопотоковость. Эти языковые свойства позволяют точнее отражать и моделировать в Java-программе реалии окружающего мира и проблемы, которые нужно решить. Кроме того, Java - динамический язык: небольшие кусочки программы умеют собираться в целую программу прямо на стадии ее исполнения, а не как обычно, на стадии написания.

В наши дни в Интернет не утихает шум и гам по поводу Java, и многое из того, что говорится об этом языке, очень похоже на то, что вы только что прочитали. Однако все это - лишь вершина айсберга. Часть языка Java, несомненно, еще сырovата и существует только в черновых проектах, но мы верим, что чем больше вы узнаете о нем, тем привлекательнее он будет вам казаться.

Наибольшее волнение среди публики, несомненно, вызывается апплетами - небольшими программами, которые можно встраивать в Web-страницы. Но Java - это не просто очередное средство сделать Web-сервер "умнее". Главное и основное в Java - это его мощность, независимость от платформы и способность решать задачи самого общего характера. Вероятно, еще не дочитав эту книгу до конца, вы начнете писать собственные Java-приложения лишь по той причине, что с ними не нужно возиться, перенося с одного типа компьютера на другой. Кроме того, апплеты куда мощнее, чем любое из прочих средств разукрашивания Web-страниц.

Например, на рис. 1-1 изображен апплет, который мы будем разрабатывать в последней главе этой книги. Он представляет собой обычный раздел ежедневника, посвященный планированию встреч и снабженный некоторыми дополнительными интересными свойствами. Вы можете получить доступ к этому апплету через свой Web-броузер. При этом вам не обязательно физически присутствовать в локальной сети, в которой этот апплет установлен, работать на той же компьютерной платформе и вообще находиться в той же самой стране. Вам предъявляется единственное требование - иметь доступ к сети Интернет.

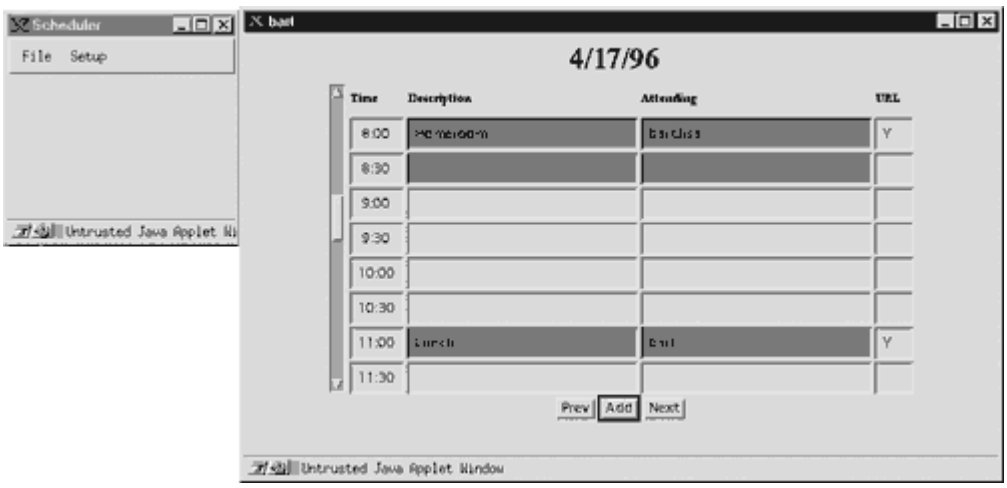

Рис. 1.1.

Web-броузеры, способные запускать апплеты

Во время написания этой книги существовал лишь один полностью приспособленный к работе с апплетами Web-броузер. Это всем известный броузер корпорации Netscape Communications под названием Netscape Navigator 2.0. Для тех читателей, у кого его нет, в главе 2 приведены подробные инструкции, как получить Netscape Navigator 2.0 прямо из Интернет, бесплатно для индивидуального пользования. Ознакомиться со списком Web-броузеров, способных запускать апплеты, можно на странице Online Companion по адресу http://www.vmedia.com/java.html. Список этот постоянно обновляется и в дальнейшем, мы надеемся, будет неуклонно расширяться.

Главной темой данной книги является написание приложений, способных распространяться по Интернет и пользующихся Web-сервером в качестве платформы программирования. Кроме апплета Планировщик встреч, мы создадим апплет-клиент, работающий вместе с шахматным сервером и обменивающийся информацией через Web-сервер, покажем, как основать собственный виртуальный магазин и создавать гибкие, удобно настраиваемые интерактивные апплеты. Попутно вы научитесь конструировать мощные Java-приложения и апплеты, о которых раньше приходилось только мечтать.

В этой главе мы начнем с изучения того, как работают Java-программы и в чем их преимущество при работе в сети. Далее мы остановимся на апплетах и том, каким образом они расширяют возможности системы Web. Чтобы оправдать наше высказывание о возможностях использования Java не только в области украшений Web-серверов, мы кратко рассмотрим историю этого языка. Кроме того, мы исследуем свойства Java, превращающие его в простой в использовании и изучении язык программирования общего назначения.

Как работает Java-программа

Язык Java является объектно-ориентированным, многопотоковым, динамическим и так далее, но вовсе не эти свойства превращают его в самый лучший язык для сетевого программирования. Главное здесь то, что Java-программы исполняются в виртуальной машине, размещенной внутри компьютера, на котором запущена программа.

Java-программа не имеет никакого контакта с настоящим, физическим компьютером; все, о чем она знает, - это виртуальная машина. Такой подход приводит нас к некоторым важным заключениям.

Во-первых, как уже отмечалось выше, Java-программы не зависят от компьютерной платформы, на которой они исполняются. Вам приходилось когда-нибудь разрабатывать приложения для нескольких операционных систем сразу? Если да, то скорее всего особого удовольствия от этого процесса вы не получили. Закончив разработку, вы наверняка узнали такую массу подробностей о той или иной операционной системе, о которой даже не задумывались ранее. Когда вы напишете и скомпилируете Java-программу, она будет работать без изменений на любой платформе, где есть виртуальная машина. Другими словами, Java-программа всегда пишется только для единственной платформы - виртуальной машины.

Переносимость языка или переносимость программы

Мы имеем полное право сказать, что язык Java машинонезависим, то есть переносим. Однако это будет лишь часть правды. Язык ANSI C, например, тоже не зависит от платформы, однако программы на нем не являются переносимыми - их необходимо каждый раз компилировать заново на каждой новой платформе. Кроме того, язык ANSI C оставляет такие вещи, как размеры

и форматы внутренних структур данных, на усмотрение разработчиков конкретной операционной среды - в Java же все они заранее строго определены и неизменны. И это всего лишь одно из преимуществ!

Во-вторых, виртуальная машина решает, что Java-программе позволено, а что делать нельзя. Программы на языках типа C или C++ запускаются напрямую операционной системой. Поэтому они получают прямой доступ к системным ресурсам компьютера, включая оперативную память и файловую систему.

Поскольку Java-программы запускаются виртуальной машиной, ее разработчики и решают, что можно, а чего нельзя позволять делать программе. Окружение, в котором работает Java-программа, называется оболочкой времени выполнения (runtime environment). Виртуальная машина играет роль бастиона на пути между Java-программой и компьютером, на котором та выполняется. Java-программа никогда не сможет получить прямой доступ к устройствам ввода-вывода, файловой системе и даже памяти. Вместо Java-программы все это делает виртуальная машина.

Когда загружается и запускается апплет, виртуальная машина полностью запрещает ему доступ к файловой системе. Виртуальная машина может дать только косвенный доступ к избранным системным ресурсам - вот почему мы доверяем апплетам и знаем, что они не способны уничтожать файлы или распространять вирусы.

Архитектура оболочки времени выполнения Java позволяет программе собираться по кусочкам прямо в процессе выполнения. Это практично, поскольку наиболее важные части программы можно постоянно хранить в памяти, а менее важные - загружать по мере необходимости. Java-программы умеют делать это, пользуясь механизмом "динамического связывания" (dynamic binding). Если все ваши программы загружаются с жесткого диска быстрого компьютера, это свойство не так уж важно. Все меняется, как только вы начинаете загружать программу из Интернет. Здесь вступает в силу ограниченная скорость сетевого соединения. В этом случае Java-программа способна сперва загрузить часть, необходимую для начала работы, запуститься, а уж затем постепенно подгрузить оставшуюся часть. Как мы увидим ниже, динамическое связывание, кроме всего прочего, облегчает сопровождение Java-программ.

Свойства виртуальной машины

Кроме функций бастиона между Java-программой и компьютером, виртуальная машина решает еще множество разнообразных задач. Например, она умеет манипулировать строковыми данными, содержит большое количество графических примитивов, функций по управлению пользовательским интерфейсом, основными структурами данных и математическими вычислениями. Чтобы все это использовать, вам необходимо познакомиться с интерфейсом прикладного программирования (API), который мы подробно рассмотрим в главе 6. Существование API приводит к тому, что размер даже самых сложных апплетов редко превышает после компиляции 100 килобайт.

Оболочка времени выполнения Java решает несколько основных проблем, встречающихся в области сетевого программирования. Поскольку Интернет основывается на колоссальном количестве различных комьютерных платформ, возможность писать действительно переносимые сетевые программы дает большие преимущества. Java-программы не могут сделать что-то, что не позволено виртуальной машиной. И наконец, оболочка времени выполнения позволяет создавать программы, способные загружаться по сети и оперативно запускаться.

Из чего состоит апплет

Любой Java-программе для работы необходима виртуальная машина. Java-программы специального типа, названные апплетами, запускаются в виртуальной машине, которая, в свою очередь, находится внутри Web-броузера, например Netscape Navigator 2.0. Виртуальная машина броузера сконструирована таким образом, что любой апплет лишен возможности сделать что-либо плохое компьютеру, на который он загрузился.

Обратите внимание, что апплет на рис. 1-2 работает на компьютере-клиенте. До появления Java большинство программ были вынуждены выполняться на Web-сервере. Выполнение апплета на компьютере-клиенте - один из самых значительных прорывов в области программирования для Web. До Java Web-страницы были статичны. Теперь, когда в Web-страницу можно встроить апплет, они стали интерактивными. Апплеты могут общаться с компьютером, с которого они были загружены, и быть частью больших систем.

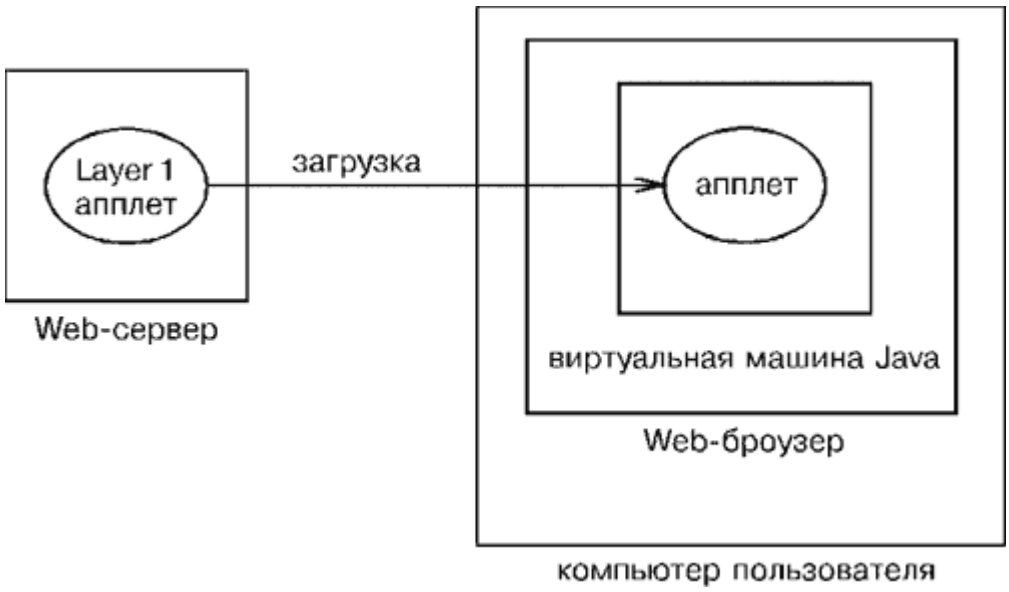

Рис. 1.2.

Переходя от главы к главе, вы будете знакомиться с новыми возможностями апплетов. Но сначала давайте рассмотрим работу апплетов в наиболее общих чертах.

Оболочка времени выполнения апплета

Как уже говорилось выше, виртуальная машина, в которой работает апплет, находится внутри Web-броузера. Она предназначена для запуска апплетов и только апплетов и служит бастионом между апплетом и компьютером. Так же, как и любая другая оболочка времени выполнения, она обслуживает запросы на доступ к памяти и управляет доступом к устройствам ввода-вывода.

Однако роль бастиона, выполняемая виртуальной машиной, в данном случае существенней, чем аналогичная роль в случае запуска самостоятельного Java-приложения. Например, никому не хочется, чтобы загруженный из сети апплет получал доступ к местной файловой системе. Сам по себе язык Java не запрещает обращаться к файловой системе. При желании вы можете написать программу, стирающую весь жесткий диск. И она, запустившись, сделает это в точности так же, как это сделала бы аналогичная программа на C или команда операционной системы. Да, мы можем написать программу, которая стирает жесткий диск, но мы определенно не хотим, чтобы это сделал полученный из Интернет апплет.

Итак, оболочка времени выполнения апплета запрещет выполнять операции, связанные с доступом к файловой системе. Мы рассмотрим, как это делается, а также вещи, которые апплет не может делать, в первую очередь. Защита файловой системы обеспечивает защиту от большинства повреждений, наносимых вирусами, но этого недостаточно. Нам по-прежнему нужна уверенность в том, что загружаемый апплет безопасен - и это мы обсудим вторым пунктом.

Чего не может апплет

Программист в состоянии написать корректную с точки зрения языка программу, уничтожающую содержимое всего жесткого диска. Тогда почему, спросите вы, хакер с дурными наклонностями не может встроить эту программу в апплет, апплет в Web-страницу и попросить потенциальную жертву "посмотреть, как это работает"? Да, все это он может. Но как только апплет, загруженный в Web-броузер, попытается запуститься, он прекратит работу, так и не дотронувшись до файлов пользователя. Причина проста - оболочка времени выполнения апплета понятия не имеет о том, что такое местная файловая система.

Безопасность апплетов

На следующих нескольких страницах будет обсуждено, почему теоретически апплет не может причинить вреда компьютеру, на котором он выполняется. Но теория есть теория, и хотя случаи, когда апплету, написанному хакером, удалось бы навредить пользовательскому компьютеру, пока не известны, тем не менее, фирма Sun уделяет пристальное внимание некоторым недочетам в системе безопасности оболочки времени выполнения апплетов. Самый свежий отчет по этой деятельности Sun можно найти на странице Online Companion по адресу http://www.vmedia.com/java.html.

Архитектура оболочки времени выполнения Java-приложений столь мощна потому, что за ее создателями всегда остается последнее слово на тему "что может и чего не может делать программа". Например, если бы создателям захотелось запретить программам печатать обидные слова на экране компьютера, они с легкостью сделали бы это. Менее осмотрительно, но более важно было бы потребовать от оболочки времени выполнения доступа лишь к заданным файлам или определенным устройствам ввода-вывода и вообще контроля за любым действием системы по отношению к компьютеру.

Так же, как и остальные оболочки времени выполнения Java, оболочка времени выполнения апплетов запрещает прямой доступ к памяти. Апплету доступна только та память, что отведена оболочкой времени выполнения; и даже с этой отведенной памятью апплет может делать только то, что ему позволено.

Безопасная загрузка апплетов по сети

Итак, мы выяснили, что апплетам не дозволено обращаться напрямую ни к системной памяти, ни к местной файловой системе. Тем не менее у хакера, желающего пробиться к ресурсам чужого компьютера, существует еще один способ преодолеть препятствия. Чтобы предотвратить нарушения в системе, к которым приводит работа некорректного апплета, оболочка времени выполнения использует несколько способов.

Во-первых, в процессе загрузки она проверяет файл с расширением .class. Файл .class содержит дополнительную информацию, позволяющую убедиться в том, что это действительно файл ожидаемого формата, соблюдающий правила Java. Оболочка времени выполнения тщательно проверяет наличие этой информации в файле .class. Если бы этого не делалось, хакер, досконально знающий особенности определенной операционной системы, теоретически мог бы получить доступ к памяти компьютера.

Во-вторых, Java-программа загружается в специально отведенное для нее место в памяти. Почему это так важно? Если бы этого не происходило, у хакера возникла бы возможность подменить часть кода оболочки времени выполнения собственным кодом совсем иного назначения! Далее он загружал бы класс, получающий доступ к файловой системе или делал бы другие не менее опасные и нежелательные вещи.

Немного истории

Формально язык Java появился на свет 23 мая 1995 года, когда компания Sun Microsystems заявила о выпуске его в свет в качестве языка программирования для Интернет. Тем не менее Java не является частью какого-либо долгосрочного плана по решению проблем программирования Интернет. Так же как и не является быстро разработанным средством, задуманным для обогащения фирмы в связи с растущей популярностью Web-серверов. Изначально Java являлся частью проекта по созданию бытовой техники нового поколения, начатом в 1990 году, за годы до того, как Интернет и Web завоевали массовое признание. Давайте поближе познакомимся с истоками Java и тем, какие уроки, в конце концов повлиявшие на язык, вынесли создатели из этого проекта.

История Java в подробностях

Разнообразные ссылки на общую информацию по Java, включая статьи о его прошлом и будущем, можно найти на странице Online Companion. Одна из таких ссылок - основной источник включенной в книгу информации о прошлом Java. Это статья, написанная Майклом О'Коннелом (Michael O'Connel) для сетевого журнала Sun World Online, расположенная по адресу http://www.sun.com/sunworldonline/swol-07-1995 /swol-07-java.html.
Замечательная статья о Java, написанная Джоржем Гилдером (George Gilder) для Forbes ASAP, называется "The Coming Software Shift" и расположена по адресу http://www.seas.upenn.edu/~gai1/ggindex.html. Мы рекомендуем иногда посещать это оглавление, чтобы следить за новинками, - Джордж Гилдер является замечательным обозревателем новейших компьютерных технологий.

В список задач, которые вышеназванный проект был призван решить, отнюдь не входила задача по созданию сетевого языка программирования. Главной целью проекта было объединить все те микрокомпьютеры, которые нас окружают, - они встроены в видеомагнитофоны, телевизоры и микроволновые печи - в единую сеть. В 1990 году этот рынок показался Sun весьма перспективным, и с целью его исследования внутри компании была создана специальная, почти полностью автономная группа.

В начале 80-х подобные подразделения создавались фирмой IBM для исследования рынка персональных компьютеров, в результате чего появился IBM PC. Группа Sun по исследованию

бытовой техники, получившая кодовое название "Green", так и не смогла революционизировать рынок бытовой техники наподобие того, как IBM коренным образом и навсегда поменяла представления о персональных компьютерах. Два года спустя после создания группа попыталась получить заказ на изготовление устройств для системы интерактивного телевидения, но потерпела фиаско и вскоре после этого практически прекратила существование. Приведем слова создателя Java, Джеймса Гослинга (James Gosling), процитированные из вышеназванной статьи журнала Sun World Online: "...мы обнаружили, что рынок бытовой техники не является чем-то реальным. Люди просто расставили акценты без какой бы то ни было причины".

Несмотря на это печальное открытие, группа все-таки выполнила некоторые из поставленных перед ней задач. Отделенные от корпоративной бюрократии и обладая всеми ресурсами, доступными в крупной компании, группа талантливых компьютерных профессионалов имела возможность интенсивно изучать все аспекты и проблемы функционирования сетевых приложений для встраиваемой аппаратуры. И ее деятельность оказалась успешной если не в коммерческом, так в техническом отношении.

Частью этого технического успеха явился язык программирования под названием Oak ("дуб"), предназначенный для использования во встраиваемой аппаратуре. В 1995 году Sun осознала, что этот язык дает неплохую возможность для решения проблем программирования Интернет. К сожалению, оказалось, что слово "Oak" уже кем-то зарегистрировано в качестве торговой марки, и было решено переименовать язык в "Java".

Уроки рынка бытовой электронной техники

Оказалось, что проблемы, решаемые в области бытовой электроники, сродни проблемам, сопровождающим быстрый рост World Wide Web. До того как Web приобрел популярность, Интернет в основном соединял учебные и научные учреждения. Несмотря на то, что Интернет - хорошо проверенная временем технология, он еще не был готов к широкому внедрению в бизнес и к оказанию услуг конечным потребителям. Давайте посмотрим, какое отношение проблемы бытовой электроники имеют к Интернет.

Разнообразие архитектур

И телевизор и кофеварка имеют встроенные микропроцессоры, но вряд ли одного и того же типа. Группа Green не могла разрабатывать систему, рассчитывая на какой-то заранее определенный тип архитектуры. С появлением Web в Интернет попали персональные компьютеры разных производителей - к примеру, PC и Macintosh. Перед разработчиками встали те же самые проблемы совместимости.

Переносимость программного обеспечения

Переносимостью должны обладать не только языки сетевого программирования, но и программы, на них написанные. До появления Java проблема решалась перекомпиляцией исходного текста для платформы, на которой программа должна была исполняться. Броузеры Mosaic, например, существуют для платформ Macintosh, Windows и различных вариантов UNIX.

Такой подход не годился группе Green по двум причинам. Во-первых, переносить код на каждую из десятков имеющихся платформ было бы слишком долгим занятием. Во-вторых, потребители просто не захотели бы связываться с проблемами языковой переносимости. Другими словами, если пользователь персонального компьютера воспринимает модификацию программы как нечто само собой разумеющееся, то пользователь кофеварки желает получить свой кофе - и не более того.

Каково же решение этих проблем? Все Java-программы работают внутри виртуальной машины Java, о которой мы уже говорили. Виртуальная машина является надстройкой над процессором, поэтому Java-программе нет никакого дела до конкретных особенностей той или иной платформы. Разумеется, все вышесказанное напрямую применимо к программированию в Интернет.

Простота и компактность

Кофеварка может быть оборудована даже мини-компьютером, но все равно - ее процессор не может быть Pentium или PowerPC с тоннами оперативной памяти. Для небольших систем, подобных кофеварке, чрезвычайно важно обеспечить функционирование программ в условиях ограниченности ресурсов. Группа Green решила и эту проблему, разработав виртуальную машину небольших размеров, в масштабах кило-, а не мегабайт. Кроме того, ее устройство было сделано простым, а это значит, что и сам язык должен оставаться простым. Несмотря на то, что

программное обеспечение, написанное на Java, может быть весьма сложным, сам язык при этом остается простым и очевидным. Грубо говоря, ядро языка Java отделено от его функциональности.

Встроенная способность к загрузке через сеть

Бытовые приборы почти наверняка не оснащены жесткими дисками. Программное обеспечение бытовой аппаратуры может храниться где угодно и загружаться в прибор только по мере необходимости, на ходу. Таким образом, Java является языком программирования сети, а не компьютера.

Сетевая безопасность

Если программа передается по сети, необходимо заботиться о безопасности данных. Группа Green разрабатывала среду для создания программного обеспечения, а не само программное обеспечение. Поскольку программное обеспечение, как планировалось, будет создаваться сторонними поставщиками, нужно было удостовериться, что никакой загруженный по сети код не сможет повредить или уничтожить аппаратуру, на которой он выполняется. То же самое относится к проблемам, возникающим при запуске загруженных из Интернет программ. Никто не хочет, чтобы компьютерный вирус вдруг уничтожил его жесткий диск!

До появления Java эта проблема решалась при помощи специальных антивирусных программ. Группа Green применила более элегантное решение - она запретила программам делать что-либо, что может привести к повреждению аппаратуры, на которой они исполняются.

Java попадает в сети

Случайно это получилось или нет - но группе Green удалось решить многие проблемы программирования в Интернет. Язык Java нейтрален к архитектуре, безопасен для сети и к тому времени, как появился Web, был вполне готов в функциональном отношении. В 1994 году Sun осознала, что прерванная попытка внедриться на рынок бытовой электроники привела к созданию замечательного продукта - Java.

В то время, когда Java еще только разрабатывался, в Интернет работали в основном суперкомпьютеры и рабочие станции. Интернет интересовал инженеров и ученых, и если вы хотели работать в Интернет, вам было не обойтись без солидных познаний в области UNIX. Для миллионов пользователей PC и Macintosh Интернет был чем-то отчужденным и непознанным. Так было до тех пор, пока в апреле 1993 года не появился первый Web-броузер под названием NCSA Mosaic 1.0.

Группа Green создала свой броузер - HotJava - и выпустила его в Интернет весной 1995 года. В качестве собственно броузера HotJava не доставало функциональности. Отсутствовала поддержка последних HTML-расширений, броузер был сравнительно медленным. Но HotJava обладал одним замечательным свойством - он умел загружать небольшие приложения, названные апплетами, и запускать их внутри себя. Апплеты исполнялись при помощи виртуальной машины в соответствии со сценарием, первоначально предназначавшимся для бытовой аппаратуры.

В течение нескольких недель со дня официального выхода языка Java ведущий производитель броузеров - корпорация Netscape Communications - заявила о поддержке языка Java и, соответственно, Java-апплетов. Netscape лицензировала Java у Sun и встроила виртуальную машину в очередную версию Netscape Navigator 2.0. Ранние версии этого броузера появились в октябре 1995, значительно расширив аудиторию пользователей Java, ранее ограниченную броузером HotJava. Между тем Sun продолжала совершенствовать язык в основном в направлении упрощения программирования апплетов. Версия Java 1.0 появилась в январе 1996 года.

С самого первого появления HotJava компания Sun опубликовала в Интернет подробности реализации языка, тем самым облегчив усилия по переносу Java на различные компьютерные платформы. Стратегия Sun даже натолкнулась на непонимание многих промышленных обозревателей. Казалось бы, зачем вкладывать миллионы долларов в технологию, которая становится всеобщим достоянием?

Билл Джой (Bill Joy), один из основателей Sun, отвечает в статье, процитированной нами ранее: "Кто бы ты ни был, большинство замечательных людей не работают на тебя. Нам нужна стратегия, которая бы позволила производить инновации везде, сразу по всему миру". Для того чтобы в разработке Java-приложений участвовали и мы с вами, спецификация Java и была передана общественности.

Что же надеется выиграть от всего этого фирма Sun? Sun, как ведущий разработчик Web-серверов, получает расширение рынка сбыта собственного товара. Естественно, у Sun, как и любого другого производителя, нет никаких гарантий. Компания по-прежнему должна

конкурировать и делать отличный продукт, чтобы получать прибыль. Применяемую Sun стратегию решаются использовать лишь немногие компьютерные компании.

Sun производит компьютеры с новой, прогрессивной архитектурой, способные непосредственно выполнять приложения Java. Вполне возможно, что Java, в конце концов, вернется к корням, откуда он ведет происхождение, - к бытовой электронике.

Все это значит, что приобретенный опыт программирования на Java наверняка сослужит вам хорошую службу в дальнейшем, какой бы оборот ни приняло развитие событий в информационной супермагистрали Интернет. Теперь давайте взглянем на Java пристальнее и рассмотрим те его свойства, за которые он может понравиться больше, чем любой другой язык программирования.

Почему вы полюбите Java

Надеемся, что наш исторический экскурс привел к лучшему пониманию возможного будущего Java. На следующих страницах мы постараемся не предаваться бездумному восхвалению языка - он, конечно, тоже не лишен недостатков, - но объективно взглянуть на вещи.

Лучшее всех времен и народов

Языки программирования, разработанные на заре компьютерной эволюции, были вынуждены следовать правилам программирования, которые на сегодняшний день уже называются примитивными. Разумеется, изобретение того или иного языка навсегда останется вехой в компьютерной истории человечества, однако нужно понимать, что вместе с нововведениями новые языки всегда приносили с собой букет различного рода ограничений, свойственных той архитектуре и области знаний, для которых они предназначались. Даже теперь, когда у нас есть скоростные современные процессоры и масса памяти для работы программ, эти языки-"привидения" по-прежнему продолжают отягощать труд программистов.

Существует требование обратной совместимости, но Java ему не подчиняется. Разработанный с нуля, этот язык вовсе не нуждается в том, чтобы программа на нем могла работать на компьютере - музейном экспонате. Некоторые свойства старинных языков можно сравнить с человеческим аппендиксом: он давно уже не выполняет своей функции, зато может воспалиться. Если уж мы взялись сотворить что-то действительно революционное, то мы можем позволить себе не следовать сложившимся традициям - что и было сделано при разработке Java.

Java - революционный язык в том смысле, что он не содержит свойств, необходимых только из соображений обратной совместимости. В то же время он вовсе не заставляет изучать массу новоизобретенных программистских концепций. Java представляет собой синтез нескольких языков программирования - его предшественников.

Коротко говоря, Java является упрощенной версией языка C++ с некоторыми добавками из других языков программирования. Из него убраны некоторые трудные для понимания и вряд ли кому-то по-настоящему нужные свойства. Давайте рассмотрим некоторые из этих свойств.

Нет препроцессора

Для тех, кто не знаком с C или C++, кратко поясним, что препроцессор осуществляет глобальные подстановки в исходном тексте до того, как передать его непосредственно компилятору. В результате компилятор получает совсем не то, что написал программист, и человек, ответственный за сопровождение программы, должен выяснять, что именно делает препроцессор. Разработчики Java сочли, что препроцессор совершает с кодом не совсем очевидные и понятные действия, поэтому из Java он был убран.

Не беспокойтесь о библиотеках и файле Makefile

Makefile - небольшой файл, в котором описана процедура компиляции программы: где, в каких файлах находится исходный текст с библиотеками и в какой последовательности они должны компилироваться. Одна из задач Makefile - сделать так, чтобы при изменении одного файла с исходными текстами не потребовалось бы заново компилировать весь проект. В Java необходимость в Makefile исчезает, потому что в языке нет всех тех проблем, из-за которых обычным языкам этот файл требуется. Java - динамический язык, а это значит, что части Java-программы соединяются между собой во время исполнения, а не при компиляции. Мы вкратце рассмотрим это свойство Java в разделе "Java - динамический язык".

Нет прямого доступа к памяти и арифметики указателей

В C и C++ указателем является целое число, представляющее собой адрес в памяти компьютера. Обращаясь к указателям в программах на C/C++, вы на самом деле просите компьютер обратиться по определенному адресу в памяти. По этому адресу обычно расположены данные, которые вас интересуют. Над указателями можно совершать арифметические действия.

Арифметика указателей - часть языка C, вытекающая из его способности быть системным языком программирования. Когда программист занимается низко-уровневым программированием на определенной платформе, арифметика указателей действительно необходима. Для высокоуровневого программирования использование указателей - наоборот, плохая практика, часто приводящая к тому, что исходный текст программы невозможно понять. И поскольку применение указателей часто приводило к сбоям и ошибкам в работе программы, они были полностью удалены из Java.

СОВЕТ Если вы - искушенный программист, у вас, возможно, возник-нет вопрос: насколько вообще может быть полезен язык, не обладающий способностью прямого доступа к памяти. Несмотря на невозможность прямого доступа, в Java имеются ссылочные переменные - аналоги указателей из C/C++. С их помощью можно построить связанный список или, например, стек. Разница состоит в том, что при помощи ссылочной переменной невозможно обратиться к памяти напрямую или привести значение-адрес в памяти к целому числу.

Нет подстановки операторов

Многие языки программирования позволяют определить, что будет делать конкретный оператор, например +, применительно к различным типам данных. Подстановка оператора записывается как процедура, действующая определенным образом в зависимости от типа операндов. Вот пример подстановки операторов, где SetA и SetB являются определенными в программе наборами целых чисел:

```
SetA=SetA+SetB;
```

Что программист пытается здесь сделать? Мы не можем сказать в точности, мы можем только спросить у автора процедуры и надеяться, что он заложил в этот оператор интуитивно понятный смысл, например поочередное сложение каждого члена одного вектора с каждым членом другого. Но в любом случае подстановка операторов вызывает множество вопросов. Непонятно, что произойдет, если длина одного вектора окажется больше длины другого. Чтобы ответить на этот вопрос, нам все равно придется заглянуть в исходный код соответствующей процедуры. Поскольку подстановка операторов приводит к ухудшению читаемости текста и усложняет язык, разработчики Java решили не включать это свойство в язык Java.

Нет множественного наследования

Для тех, кто не знаком с концепцией объектно-ориентированного программирования, смысл следующих абзацев станет яснее после прочтения главы 3. Сейчас мы рассмотрим пример, участниками которого будут обыкновенные дети. Предположим, что в некоем сообществе людей у каждой семьи есть один родитель. Дети в этом сообществе ведут себя так: если они сами знают, как делается что-либо, они делают это. Если не знают - спрашивают родителя. Если родитель не знает - он спрашивает своего родителя и т. д. Перед вами очень простая модель единичного наследования.

В случае множественного наследования ребенок может спрашивать у нескольких родителей. Несомненно, это усложняет жизнь нашего гипотетического ребенка. Разные родители могут и советовать по-разному, как сделать ту или иную вещь.

Когда мы пишем на объектно-ориентированном языке, модули нашей программы могут наследовать способы сделать что-либо в точности так, как это происходит в нашем примере. Если в языке допускается множественное наследование, он должен быть сложнее, так же как и проблемы, которые ему приходится решать. Почему в некоторых языках существует множественное наследование. Мы подробнее рассмотрим это в главе 3, а пока скажем, что программные модули зависят от параметров нескольких родительских модулей. Концепция, применяемая в Java, лишена необходимости применять множественное наследование.

Объектная ориентация

Как мы уже говорили, Java отчасти хорош уже только потому, что он новый язык. Однако одно из самых серьезных его преимуществ заключается в объектной ориентации. Давайте рассмотрим, что означает объектная ориентация и как она реализована в Java.

Для того чтобы понять принцип объектной ориентации, достаточно запомнить один факт: компьютер не мыслит так, как это делают люди. Он просто запоминает в своих регистрах нули и единицы. Антитезисом к слову "интуиция" является слово "язык ассемблера" - очень узкий угол зрения на решение вселенских проблем. Компьютеры живут в упрощенном мире, мы, наоборот, - в чрезмерно сложном. Простая механическая операция, занимающая у нас часы, занимает у компьютера лишь сотые доли секунды, тогда как ученые до сих пор не нашли способа обучить компьютер составлению элементарно грамотного предложения на естественном языке.

На начальной стадии компьютерной эволюции мы были вынуждены играть по их правилам. Потом были разработаны языки, позволяющие более удобно для нас, людей, объяснять компьютерам, что именно мы от них хотим. Первым настоящим прорывом было создание процедурных языков. Вместо того чтобы передавать компьютеру огромный список инструкций, программист теперь мог ограничиться небольшим списком, инструкции из которого могли повторяться многократно - это называется подпрограммами.

Объектная ориентация дала нам в руки новый уровень абстракции. Правда, этот подход по-прежнему далек от того, как мы привыкли приниматься за решение проблемы. Попробуйте визуализировать следующие инструкции: "пойди в магазин, купи молока, вернись домой". Мы предписали совершить три определенных действия, однако скорее всего вы начнете думать над задачей, ориентируясь не на глаголы, а на существительные. Например, вам не хочется, чтобы купленное молоко было скисшим, и не хочется идти в тот магазин, где молоко не продается. Если мы спросим, как вы будете решать эту задачу, вы ответите: "Начиная с той позиции, где я сейчас нахожусь, я пойду поищу какое-нибудь транспортное средство и проеду до того магазина, где продается молоко. Затем я куплю молоко, проверив, не скисшее ли оно, а затем вернусь в то место, где я живу".

Суть нашей молочной задачи состоит в том, что мы имеем дело со свойствами объектов - молока, магазина, места проживания. Процедурные языки программирования заставляют сконцентрироваться на действиях, которые необходимо совершить для решения проблемы. Объектно-ориентированные языки позволяют рассматривать ситуацию с точки зрения вовлеченных в нее объектов и их свойств. Вместо того чтобы проверять каждый пакет молока в поисках свежего, мы можем приписать объекту "молоко" свойство "скисшее". В табл. 1-1 проиллюстрированы все наши объекты, их свойства и вопросы, которые к ним применимы.

Таблица 1-1. Объекты, вовлеченные в нашу задачу

Объект	Данные	Вопросы
Транспортное средство	Дальность поездки	Какие магазины находятся в пределах дальности?
Молоко	Срок годности	
	Цена	Сколько времени молоко еще не скиснет?
		Насколько дорого молоко?
Магазин	Местонахождение	В этом магазине продается молоко?
Дом	Местонахождение	Доеду ли я до дома?
	Владелец	Это мой дом?

Программа на Java записывается в рамках представления объектов, вовлеченных в задачу. Как будет видно из главы 3, наши объекты будут весьма точно повторять подход, отраженный в табл. 1-1. Например, для того чтобы запрограммировать разобранную ситуацию на Java, необходимо написать процедуры, отвечающие на поставленные в третьем столбце вопросы и присваивающие значения переменным, указанным во втором столбце таблицы. Разница в том, что эти процедуры относятся исключительно к объектам. Это ключевой момент в объектно-ориентированном программировании - описывать задачу в тех терминах, в которых мы ее решаем сами.

Встроенная работа с сетью

С самого начала предполагалось, что Java будет сетевым языком программирования. Мы уже знакомы с преимуществами использования виртуальной машины Java. Она предотвращает возможность повреждения компьютера от действий загруженных по сети некорректных программ, позволяет программам загружаться быстро и не зависеть от архитектуры компьютера.

Все эти преимущества встроены в язык Java, и мы даже можем о них не думать, занимаясь повседневным программированием. Поскольку Java изначально предназначен для сети, в его интерфейс прикладного программирования, API, встроены механизмы взаимодействия с сетью. При помощи API мы можем работать с сетью как на высоком уровне абстракции, пользуясь услугами URL (Uniform Resource Locator), так и на самом низком уровне, просто перемещая пакеты данных туда и обратно.

У нас есть возможность писать апплеты, общающиеся с компьютером, с которого они были загружены. В главе 19 мы напишем именно такой апплет. Апплет будет загружаться при помощи стандартного протокола HTTP (HyperText Transfer Protocol). HTTP предназначен только для извлечения информации, а не для обмена данными, поэтому загруженный апплет будет пользоваться собственным протоколом для последующего взаимодействия с Web-сервером.

В этом примере мы не занимались шифрованием передаваемых данных, но если захотите, вы без труда сможете его реализовать. Если вдруг вы изобретете новый механизм, скажем, сжатия видеоданных, вам достаточно будет написать апплет, который знает этот механизм, и сервер, знающий, как передавать видео. Вместо того чтобы писать громадную клиентскую программу и уговаривать людей ее установить, вы просто пишете апплет, который, загрузившись, сам становится клиентом для приема видео. API дает вам способ самостоятельно определить методы и протоколы передачи данных по сети, в то время как независимость от компьютерной платформы гарантирует, что апплет будет работать везде.

Java - динамический язык

Скомпилированная программа на C/C++ представляет собой монолитный файл, наполненный машинными инструкциями. В случае больших программ его размер исчисляется мегабайтами. Когда вы пишете большой проект, нередко случается, что в нем используется уже когда-то написанный код. В таком случае помогают библиотеки. При компиляции ваш код объединяется с кодом из библиотеки, и все это вместе становится исполняемой программой. Если в библиотеке обнаружится ошибка, каждая скомпилированная с ее помощью программа будет нуждаться в повторной компиляции. У Java-программ эта проблема отсутствует. Дело в том, что модули Java-программы собираются только в момент ее исполнения, а до этого существуют отдельно друг от друга. Это значит, что модули Java-программы существуют независимо от программ, которым они принадлежат.

Java - многопотоковая среда

Вы хотели когда-нибудь быть одновременно в двух местах? Если да, то многопотоковое программирование - для вас. Так же как и объектная ориентация, многопотоковость придумана, чтобы облегчить вам жизнь как программисту. Ее задача - позволить описать набор действий, которые должны происходить одновременно.

Предположим, что вы пишете программу рисования на экране компьютера окружности, начиная с центра с постепенно возрастающим радиусом. Программу можно записать в псевдокоде следующим образом:

```
// псевдокод, а не Java!
set_center
set_the_color
radius=1
do {
        draw_circle(radius)
        radius=radius+1
}
while (radius  final_circle_size)
```

Вы показываете программу своему шефу, но он просит написать программу, рисующую одновременно две окружности. Назад, к рабочему столу! Результат вашего творчества напоминает следующее:

```
// псевдокод, а не Java!
set_center_for_Circle1
set_center_for_Circle2
set_color_for_Circle1
set_color_for_Circle2
Circle1_radius=1
Circle2_radius=1
do {
        if (Circle1_radius  final_circle1_size)
        draw_circle1(Circle1_radius)
        if (Circle2_radius  final_circle2_size)
        draw_circle2(Circle2_radius)
```

```
        if (Circle1_radius  final_circle1_size)
        Circle1_radius=Circle1_radius+1
        if (Circle2_radius  final_circle2_size)
                Circle2_radius=Circle2_radius+1
}
while (Circle1_radius  final_circle1_size
        AND
                Circle2_radius  final_circle2_size)
```

Все, что мы здесь делаем, - это последовательно повторяем каждую инструкцию для обеих окружностей. Механическое повторение хорошо поддается программированию, в особенности многопотоковым способом. Вместо того чтобы повторять одни и те же инструкции для обеих окружностей, мы можем записать инструкции для одной, оформив их в виде потока, а затем запустить этот поток два раза для двух различных окружностей.

Если вы видели работу апплетов на Web, в частности анимацию, вы, вероятно, уже знакомы с внешним проявлением многопотоковости языка Java.

Что дальше?

Теперь, когда вы получили представление о том, что такое Java и апплеты, начнется настоящая работа. В следующей главе мы установим комплект разработчика Java (JDK) и напишем несколько простых программ. Затем в главе 3 мы познакомимся с объектной ориентацией в Java, объясним синтаксис и семантику языка в главе 4. В главе 5 мы напишем несколько простых апплетов. Далее мы изучим Java API, напишем массу профессиональных апплетов с использованием графических интерфейсов и апплетов, обменивающихся данными по сети.

Глава 2

Основы программирования на Java

Появление языка Java вызвало в компьютерном мире большой фурор, и вам, конечно, хочется поскорее узнать, чем же этот язык столь замечателен. К концу этой главы мы с вами напишем первые несколько программ на языке Java. Наша цель сейчас - разобраться в структуре типичной программы на Java, а заодно научиться писать, компилировать и запускать программы. Это краткое введение также поможет вам понять сходства и различия между Java и другими языками программирования, которые вы, возможно, использовали раньше.

Первым нашим шагом будет установка на компьютер компилятора и оболочки времени выполнения языка Java. Установленное программное обеспечение мы протестируем самым простым и очевидным способом - написав и выполнив программу на Java. Затем мы изучим структуру Java-программ и то, как происходит их выполнение. В заключение мы рассмотрим основные понятия, относящиеся к апплетам, и научимся помещать готовые апплеты на World Wide Web, чтобы любой пользователь Интернет мог получить к ним доступ.

В целом эта глава содержит очень много материала по языку Java. Однако не беспокойтесь, если вы не найдете здесь ответы на какие-то из ваших вопросов. Разработке и освоению материала, представленного в этой главе, посвящены все последующие главы - вплоть до главы 5, "Апплет в работе". В наших ближайших планах - написать несколько простых программ на Java и начать освоение этого замечательного языка.

Где найти файлы с примерами

Заголовок "Пример" над фрагментом текста программы в этой книге означает, что мы подготовили файл, который поможет вам быстрее прогнать пример на своем компьютере. Пользователи Windows 95 и Macintosh найдут файлы на прилагаемом к книге диске CD-ROM, а пользователи UNIX смогут получить файлы с примерами с помощью страницы в WWW, озаглавленной Online Companion и расположенной по адресу http://www.vmedia.com/java.html (на этой странице собраны ссылки на файлы с примерами).

Первые шаги

Решив начать программировать на языке Java, вы первым делом должны установить на свой компьютер компилятор и оболочку времени выполнения для этого языка. Эти компоненты входят в состав комплекта разработчика Java (Java Developers Kit, JDK) - пакета программ, который бесплатно распространяется фирмой Sun Microsystems. Версию 1.01 JDK вы найдете либо на прилагаемом к книге диске CD-ROM (для пользователей Macintosh, Windows 95 и Windows NT), либо на странице Online Companion (для пользователей UNIX). Новые версии JDK, по мере их появления, мы также будем делать доступными через Online Companion, поэтому вам имеет

смысл проверить, какую версию можно взять с Online Companion, прежде чем приступать к инсталляции с прилагаемого к книге диска.

Оболочки для программирования на Java третьих фирм

Многие фирмы - поставщики программного обеспечения (в частности, Borland и Symantec) в настоящее время заняты разработкой своих оболочек для программирования на Java. Все эти фирмы планируют включить в состав своих оболочек графический интерфейс пользователя, который сделает эти оболочки более удобными в использовании в сравнении с JDK. Разумеется, все эти продукты уже не будут бесплатными. Чтобы не сужать круг читателей, мы в этой книге решили ограничиться описанием только JDK фирмы Sun. Когда новые оболочки программирования для языка Java будут появляться на рынке, мы будем помещать краткие рецензии на них на странице Online Companion.

Выбрав источник инсталляции - прилагаемый к книге диск или Интернет, - вы должны перенести содержимое JDK в свою файловую систему. При установке с CD-ROM эта работа сводится к копированию файлов на жесткий диск, а выбрав в качестве источника установки Интернет, вы должны будете скачать файл с дистрибутивом JDK из сети и разархивировать его. Информацию о том, как это сделать на конкретной компьютерной платформе, вы найдете в одном из двух приложений - Приложении А, "О странице Online Companion", или Приложении Б, "Диск CD-ROM". Перенеся файлы JDK на свой компьютер, вы можете приступать к установке. В следующих разделах собраны подробные инструкции по установке JDK для каждой из платформ, для которых существует версия Набора разработчика. По мере появления версий для других платформ инструкции по установке для них мы также поместим на страницу Online Companion.

Инсталляция для Windows 95/Windows NT

Прежде чем следовать нижеприведенным инструкциям, вы должны поместить все файлы JDK на свой жесткий диск, скопировав их с диска CD-ROM. Поместить эти файлы можно в любой каталог. Находясь в этом же каталоге, вы должны установить значение переменной окружения CLASSPATH, которая позволит компилятору Java найти вспомогательные классы, нужные ему для компиляции Java-программ. Например, если вы поместили файлы дистрибутива в каталог C:\JAVA, вы должны установить переменную CLASSPATH, напечатав следующую команду в строке приглашения DOS:

```
C: SET CLASSPATH=.;C:\JAVA\LIB
```

Эту команду имеет смысл поместить в файл AUTOEXEC.BAT. Кроме того, вам, вероятно, покажется удобным добавить каталог с исполняемыми файлами JDK в путь поиска, задаваемый командой PATH. Если вы установили JDK в каталог C:\JAVA, то все исполняемые файлы будут помещены в каталог C:\JAVA\BIN, который и нужно будет добавить к списку каталогов команды PATH в файле AUTOEXEC.BAT.

Установив значения переменных окружения, вы можете приступать к программированию на языке Java. Первое, что вам понадобится для этого, - текстовый редактор. Практически единственное требование к текстовому редактору для написания программ - это возможность сохранять файлы в простом текстовом формате и с расширением .java. Желательно также, чтобы выбранный вами текстовый редактор был достаточно быстрым, в то время как, например, возможности оформления текста различными шрифтами совсем не обязательны. К примеру, Microsoft Word 7.0 лучшим выбором для программиста никак не назовешь. Идеальный текстовый редактор позволит вам также осуществлять простейшую проверку синтаксиса (например, парность скобок) одновременно с вводом текста программы. Если у вас установлена одна из оболочек для программирования на C/C++, вы можете попробовать использовать встроенный текстовый редактор этой системы.

Power PC Macintosh

Как только вы разархивируете файлы JDK на вашей файловой системе, Набор разработчика будет готов к работе. Обращайтесь к Приложениям А и Б за инструкциями по копированию файлов с диска CD-ROM или скачиванию файлов со страницы Online Companion и их разархивированию. Версия JDK для Macintosh, которую вы найдете на прилагаемом к книге диске, может использоваться только для создания апплетов. Если вы помните, в первой главе мы говорили о том, что Java-программы выполняются в рамках так называемой виртуальной машины, а виртуальная машина для запуска апплетов отличается тем, что она изолирует их и не позволяет им причинить какой-либо вред компьютеру. К моменту сдачи этой книги в типографию для компьютеров Macintosh существовала версия только такой виртуальной машины, которая поддерживает апплеты. Это означает, что вам понадобится программа просмотра апплетов

(appletviewer) для запуска многих простых программ, которые иллюстрируют в этой главе принципы языка Java. Для запуска этих программ вы должны сделать следующее:

1. Щелкните по значку программы просмотра апплетов на верхнем уровне разархивированного JDK.
2. Щелкните мышью по файлу index.html. Это приведет к запуску апплета под названием ProgramRunnerApplet.java, который, в свою очередь, позволит вам запускать простые программы, как если бы вы управляли ими из командной строки на компьютере с Windows или UNIX.

Конечно, рано или поздно у вас появится возможность запускать самостоятельные Java-программы на компьютере Macintosh. Но пока, чтобы познакомиться с приводимыми здесь примерами, вам придется пользоваться программой просмотра апплетов.

Однако прежде чем запускать Java-программы, вы должны иметь возможность вводить их текст. Для этого вам понадобится текстовый редактор, который обязательно должен уметь сохранять файлы в простом текстовом формате. Очень полезной будет также возможность производить несложную проверку синтаксиса одновременно с вводом программного кода. Если у вас на компьютере уже установлена одна из оболочек программирования для C++, вы можете использовать встроенный в нее текстовый редактор для ввода Java-программ.

UNIX

Прежде чем приступать к выполнению нижеприведенных инструкций, поместите все файлы, входящие в JDK, на свой жесткий диск. Для этого вам нужно будет скачать дистрибутив JDK со страницы Online Companion (о том, как это сделать, вы прочтете в Приложении A).

На жестком диске вы можете поместить файлы JDK в любой каталог. Находясь в этом же каталоге, вы должны установить переменную окружения CLASSPATH, которая позволит компилятору Java находить вспомогательные классы, которые нужны ему для компиляции Java-программ. Предположим, что вы поместили файлы дистрибутива в каталог /usr/local/java. Затем, находясь в строке приглашения оболочки, вы должны установить значение переменной CLASSPATH. Команды для различных оболочек UNIX, позволяющие сделать это, перечислены в табл. 2-1 (вы можете добавить эту команду в свой файл .login).

Таблица 2-1. Установка значения переменной окружения CLASSPATH

Оболочка	Команда
C shell	promptsetenv CLASSPATH /usr/local/java:.
Bourne shell	promptCLASSPATH=/usr/local/java:. promptexport CLASSPATH
Korn shell	promptexport CLASSPATH=/usr/local/java:.
bash	promptexport CLASSPATH=/usr/local/java:.

Вам, вероятно, покажется удобным добавить каталог с исполняемыми файлами Набора разработчика в путь поиска. Если вы установили JDK в каталог /usr/local/java, то все исполняемые файлы будут помещены в каталог /usr/local/java/bin, который и нужно будет добавить к списку каталогов пути поиска в файле .login.

Следующий шаг - выбор текстового редактора. На эту роль подойдут такие распространенные редакторы, как Emacs, vi или pico.

Первая программа на Java

Теперь нам предстоит проверить работоспособность установленного программного обеспечения. Текст примера 2-1, как и всех остальных примеров в этой книге, вы найдете на диске CD-ROM и на странице Online Companion (см. врезку "Где найти файлы с примерами"). Пользователи Macintosh должны загрузить в программу просмотра апплетов файл /Chapter2/Example1/appleProgram.html с диска CD-ROM. Если ваш компьютер - не Macintosh, вам лучше ввести этот первый пример самостоятельно, чтобы вы могли заодно опробовать в работе свой текстовый редактор.

СОВЕТ Пользователи Macintosh должны будут прибегнуть к помощи текстового редактора позднее, в следующих разделах этой главы.

Пример 2-1. Простейшая программа "Hello, Java!" (OurPrimaryClass.java).

```
import java.util.*;
public class OurPrimaryClass {
public final static void main(String S[]) {
        System.out.println("Hello, Java!");
        Date d=new Date();
        System.out.println("Date: "+d.toString());
                        }
                }
```

Введя текст этого маленького примера, сохраните его в файле OurPrimary- Class.java. Каждый раз, когда вы определяете общедоступный класс, вы должны сохранять его в файле, имя которого совпадает с именем класса (подробнее о том, что такое общедоступный класс, мы поговорим ниже). Прежде всего мы должны скомпилировать нашу программу с помощью компилятора Java, который называется javac. Пользователи UNIX и Windows для этого должны ввести в командной строке следующую команду:

javac OurPrimaryClass.java

Пользователям Macintosh достаточно щелкнуть по значку компилятора Java, а затем открыть файл OurPrimaryClass.java для компиляции.

Если компилятор обнаружит ошибки, проверьте правильность ввода текста программы (или просто возьмите готовый файл с прилагаемого диска или из Интернет со страницы Online Companion). Если вы по-прежнему не можете довести компиляцию до успешного конца, проверьте, выполняются ли следующие условия:

- Ваша система знает, где находится javac. На компьютерах с UNIX или Windows компилятор javac должен для этого находиться в каталоге, включенном в путь поиска (пользователи Macintosh могут об этом не беспокоиться).
- Компилятор javac должен быть в состоянии найти остальные файлы, входящие в JDK. На компьютерах с UNIX и Windows должно быть правильно установлено значение переменной окружения CLASSPATH (см. выше; пользователи Macintosh могут об этом не беспокоиться).

Если же компиляция прошла успешно, вы можете переходить к запуску программы. Пользователи UNIX и Windows должны для этого ввести следующую команду:

java OurPrimaryClass

Эта команда запускает оболочку времени выполнения, которая так и называется - java. Эта оболочка загружает класс OurPrimaryClass и выполняет входящий в него метод main. Вывод программы выглядит следующим образом:

Hello, Java!

после чего следует текущая дата. Если вместо описанного вывода вы получаете сообщение об ошибке, то, вероятнее всего, оболочка времени выполнения не может найти файл OurPrimaryClass.class, который был сгенерирован компилятором javac. В этом случае пользователи UNIX и Windows должны убедиться, что в значение переменной CLASSPATH входит текущий каталог. Пользователи Macintosh должны запускать этот пример в качестве апплета, как описано выше.

Разбор параметров в командной строке

Если только вы не пользуетесь компьютером Macintosh для писания программ на Java, вы можете также передать нашей простейшей программе какие-нибудь параметры в командной строке. Для этого нужно воспользоваться переменной - массивом строк, которую мы объявим как String S[] и в элементах которой будут содержаться отдельные параметры. Вот как выглядит вариант метода main, который печатает на выходе все, что передается ему в командной строке.

Пример 2-2. Метод main, осуществляющий разбор командной строки.

```
public class PrintCommandLineParameters {
public final static void main(String S[] ) {
        System.out.println("Hello, Java!");
        System.out.println("Here is what was passed to me:");
        for(int i=0;i.length;i++)
        System.out.println(S[i]);
        }
```

```
}
```

Наша программа теперь будет печатать на выходе все переданные ей параметры командной строки. Например, если вы запустите эту программу такой командой:
java PrintCommandLineParameters parameter1 parameter2 parameter3 parameter4
то на выходе вы получите следующее:

```
Hello, Java!
Here is what was passed to me:
parameter1
parameter2
parameter3
parameter4
```

СОВЕТ Метод main аналогичен функции main, которая должна присутствовать в любой программе на C или C++.

Простой текстовый вывод

Как вы уже, вероятно, догадались, метод System.out.println позволяет выводить текст на экран. Этот метод очень удобен для текстового вывода в несложных программах вроде тех, с которыми мы сейчас работаем. Когда мы с вами перейдем к созданию апплетов, мы должны будем научиться выводить графическую, а не только текстовую информацию. А сейчас давайте познакомимся поближе со свойствами метода System.out.println.

Как вы уже видели, если передать методу System.out.println строку символов, заключенную в пару двойных кавычек, этот метод выведет данную строку на экран, завершив ее переводом строки. Кроме того, этот метод можно использовать для печати значений переменных - как по отдельности, так и совместно со строками символов в кавычках.

Пример 2-3. Использование метода System.out.println.

```
public class PrintlnExample {
public static void main(String ARGV[]) {
        System.out.println("This example demonstrates the use");
        System.out.println("of System.out.println");
        System.out.println("\nYou can output variables values");
        System.out.println("like the value of an integer:\n");
        int i=4;
        System.out.println("variable i="+i);
        System.out.println(i);
        }
}
```

Как устроена Java-программа

Мы с вами только что написали простую программу на языке Java. Давайте воспользуемся этой программой, чтобы уяснить, из каких основных строительных блоков состоят Java-программы. Наша программа содержит не все из этих блоков, поэтому мы сейчас приступим к ее расширению и усложнению, чтобы задействовать в нашей программе большинство элементов, которые применяются во всех Java-программах.

Впрочем, не ждите, что, прочитав следующие несколько страниц, вы уже будете понимать основы этого языка. Объектно-ориентированные свойства языка Java рассматриваются в следующей главе, а формальный синтаксис языка обсуждается в главе 4. Цель же этой главы - помочь вам понять, что представляет собой Java-программа в целом. При этом важно помнить, что все программы, которые мы будем рассматривать в этом разделе, не относятся к апплетам. (Простейший апплет, выводящий надпись "Hello, Applet!", мы с вами напишем ближе к концу этой главы.) Тем не менее почти все, о чем мы здесь будем говорить, в равной мере относится и к апплетам, которые представляют собой полноправные Java-программы - пусть и запускаемые не из командной строки, а на Web-странице.

Обзор структуры Java-программы

Все Java-программы содержат в себе четыре основные разновидности строительных блоков: классы (classes), методы (methods), переменные (variables) и пакеты (packages). На каком бы языке вы ни программировали до сих пор, вы скорее всего уже хорошо знакомы с методами, которые есть не что иное, как функции или подпрограммы, и с переменными, в которых хранятся данные. С другой стороны, классы представляют собой фундамент объектно-ориентированных свойств языка. Пока что для простоты можно сказать, что класс - это некое целое, содержащее в себе переменные и методы. Наконец, пакеты содержат в себе классы и помогают компилятору найти те классы, которые нужны ему для компиляции пользовательской программы. Как мы увидим в главе 3, "Объектная ориентация в Java", классы, входящие в один пакет, особым образом зависят друг от друга. Однако пока, опять-таки для простоты, мы можем рассматривать пакеты просто как наборы классов. Даже простейшая программа, которую мы написали, чтобы протестировать установленный JDK, содержит в себе все эти составные части.

Все, о чем мы сейчас говорили, присутствует в любой Java-программе. Однако Java-программы могут включать в себя составные части и других видов, о которых мы сейчас подробно говорить не будем. Некоторые сведения об этих разновидностях составных частей приведены в табл. 2-2. Все, что описано в этой таблице, не обязательно требуется для каждой Java-программы, однако во многих программах из тех, которые мы будем писать, без этих составных частей будет не обойтись.

Таблица 2-2. Базовые понятия языка Java, не обсуждаемые в этой главе

Понятие	Для чего используется	Где в книге описывается
Интерфейсы	Позволяют реализовать полиморфизм	Полиморфизм и интерфейсы обсуждаются в главе 3
Исключения	Позволяют организовать эффективную обработку ошибок	Глава 4
Потоки	Позволяют одновременно выполнять больше одного фрагмента кода	Вводятся в главе 5, а подробно обсуждаются в главе 10

Java-программа может содержать в себе любое количество классов, но один из этих классов всегда имеет особый статус и непосредственно взаимодействует с оболочкой времени выполнения. В качестве такого особого класса оболочка времени выполнения всегда воспринимает первый из классов, определенных в тексте программы. Мы будем называть этот класс первичным классом (primary class). В примере 2-1 первичным классом был OurPrimaryClass. В первичном классе обязательно должны быть определены один или несколько специальных методов.

Когда программа запускается из командной строки, как мы делали это с примером 2-1, системе требуется только один специальный метод, который должен присутствовать в первичном классе, - метод main. Ниже, когда мы приступим к программированию апплетов, мы увидим, что первичный класс в апплете должен содержать уже несколько таких специальных методов.

Теперь мы перейдем к подробному рассмотрению каждого из четырех основных блоков Java-программы - переменных, методов, классов и пакетов.

Переменные

Видимо, понятие переменной не требует слишком подробных объяснений; удобнее всего представлять себе переменную как хранилище для единицы данных, имеющее собственное имя. Любая переменная в языке Java, как и в большинстве других языков программирования, принадлежит к определенному типу. Тип переменной определяет, какого рода информацию можно в ней хранить. Например, переменные типа int используются для хранения целочисленных значений. Приведем пример использования переменной этого типа.

Пример 2-4. Использование переменной.

```
public class UsesInt {
public static void main(String S[]) {
        int i=4;
        System.out.println("Value of i="+i);
        }
}
```

В этом примере мы использовали знак операции присваивания = для того, чтобы присвоить переменной i значение 4, а затем вывели значение этой переменной с помощью метода System.out.println. Тип переменной, который мы использовали в данном примере, относится к одной из двух больших групп типов, используемых в Java, - к примитивным типам (primitive

types). Другая большая группа типов объединяет в себе ссылочные типы (reference types), которые включают в себя типы, определенные пользователем, и типы массивов. К примитивным типам относятся стандартные, встроенные в язык типы для представления численных значений, одиночных символов и булевских (логических) значений. Напротив, все ссылочные типы являются динамическими типами. Главные различия между двумя упомянутыми группами типов перечислены в табл. 2-3.

Таблица 2-3. Сравнение примитивных и ссылочных типов

Характеристика	Примитивные типы	Ссылочные типы
Определены ли в самом языке Java?	Да	Нет
Имеют ли предопределенный размер?	Да	Нет
Должна ли для переменных этих типов выделяться память во время работы программы?	Нет	Да

СОВЕТ Примитивные и ссылочные типы также различаются по тому, как переменные этих типов передаются в качестве параметров методам (то есть функциям). Переменные примитивных типов передаются по значению, тогда как ссылочные переменные всегда передаются по ссылке. Если вы еще не знакомы с этой терминологией, не беспокойтесь - мы будем подробно говорить о передаче параметров в разделе "Методы" этой главы.

На практике самым важным различием между примитивными и ссылочными типами является то, о чем свидетельствует последняя строка табл. 2-3, а именно - что память для переменных ссылочного типа должна выделяться во время выполнения программы. Используя переменные ссылочных типов, мы должны явным образом запрашивать требуемое количество памяти для каждой переменной прежде, чем мы сможем сохранить в этой переменной какое-либо значение. Причина этого проста: оболочка времени выполнения сама по себе не знает, какое количество памяти требуется для того или иного ссылочного типа. Рассмотрим пример, иллюстрирующий это различие. При чтении примера имейте в виду, что все типы массива относятся к ссылочным типам, а также обратите внимание на строки комментариев, отбитые //.

Пример 2-5. Примитивные и ссылочные переменные.

```
public class Variables {
public static void main(String ARGV[]) {
int myPrimitive;
// переменная примитивного типа
int myReference[];
// переменная ссылочного типа
myPrimitive=1;
// сразу после объявления мы можем записывать данные в переменную
// примитивного типа
myReference=new int[3];
// однако, прежде чем сохранять данные в переменной ссылочного типа, мы
должны
// выделить память под эту переменную...
myReference[0]=0;
myReference[1]=1;
myReference[2]=2;
// ...и только после этого мы можем записывать в нее данные
        }
}
```

Поскольку тип int относится к примитивным типам, оболочка времени выполнения с самого начала знает, сколько места нужно выделить для каждой такой переменной (а именно, четыре байта). Однако когда мы объявляем массив переменных типа int, оболочка времени выполнения не может знать, сколько места потребуется для хранения этого массива. Поэтому прежде, чем мы сможем поместить что-либо в переменную myReference, мы должны запросить у системы определенное количество памяти под эту переменную. Этот запрос осуществляется с помощью оператора new, который заставляет оболочку времени выполнения выделить для переменной соответствующее количество памяти.

Заметим, что переменные-массивы и переменные определенных пользователем типов лишь указывают на то место в памяти, где содержатся собственно данные, тогда как переменные

примитивных типов ни на что не указывают, а просто содержат в себе соответствующие данные, имеющие определенный фиксированный размер.

СОВЕТ Как вы можете видеть, ссылочные типы очень похожи на указатели, применяющиеся в C/C++. Однако есть и серьезные отличия. Во-первых, используя ссылочные типы, вы не можете получить доступ к фактическим адресам данных в памяти. А во-вторых, невозможность получить доступ к адресу в памяти в языке Java означает, что в этом языке полностью отсутствует арифметика указателей.

Примитивные типы

Сначала рассмотрим примитивные типы языка Java. С одним из этих типов - типом int - мы уже познакомились выше на конкретном примере. Всего в языке Java определено восемь примитивных типов, которые перечислены в табл. 2-4.

Таблица 2-4. Примитивные типы языка Java

Тип	Размер в байтах	Диапазон значений	Примеры значений
int	4	от -2147483648 до 2147483647	200000, -200000
short	2	от -32768 до 32767	30000, -30000
byte	1	от -128 до 127	100, -100
long	8	от -922372036854775808 до 922372036854775807	1000, -1000
float	4	зависит от разрядности	40.327
double	8	зависит от разрядности	4000000.327
Boolean	1 бит	true, false	true, false
char	4	все символы стандарта Unicode	

Первые шесть типов из перечисленных в таблице предназначены для хранения численных значений. С переменными этих типов вы можете использовать знаки операций +, -, *и /, предназначенные соответственно для сложения, вычитания, умножения и деления. Полностью синтаксис записи выражений, содержащих числовые значения, приведен в главе 4. По большей части правила этого синтаксиса аналогичны правилам языка C. Давайте рассмотрим подробнее булевский тип, который в явном виде отсутствует во многих других языках программирования. Вот как осуществляется присвоение значения булевской переменной.

Пример 2-6. Присвоение значения булевской переменной.

```
boolean truth=true;
System.out.println(truth);
boolean fallicy=false;
System.out.println(fallicy);
truth=(1==1);
fallicy=(1==0);
System.out.println(truth);
System.out.println(fallicy);
```

Если мы поместим этот фрагмент кода в метод main из примера 2-1, то вывод программы будет иметь следующий вид:

```
true
false
true
false
```

Как видите, булевским переменным можно присваивать результат операции сравнения. В языке Java знаки операций !, != и == работают с булевскими значениями так же, как одноименные операторы работают с целочисленными значениями в языке C. Полное описание синтаксиса и семантики для булевского типа, как и для остальных примитивных типов, вы найдете в главе 4.

СОВЕТ В этой книге вы не раз столкнетесь с приводимыми в качестве примеров фрагментами кода, такими как пример 2-6. Эти фрагменты не могут компилироваться сами по себе, так как они не представляют собой законченных программ. Как вы понимаете, если бы мы приводили в книге только программы целиком, во многих случаях иллюстративная ценность примеров была бы снижена, и примеры эти занимали бы в книге слишком много места. В то же время в файлах на диске CD-ROM, прилагаемом к книге, все такие фрагменты кода включены в состав самостоятельных программ, каждую из которых можно скомпилировать и запустить отдельно.

Ссылочные типы

Как вы уже знаете, ссылочные типы отличаются от примитивных тем, что они не определены в самом языке Java, и поэтому количество памяти, которое требуется для переменных этих типов, заранее знать невозможно. В примере мы уже встречались с одним из ссылочных типов - типом массива. Массивы в языке Java могут состоять из переменных любого другого типа этого языка, включая типы, определенные пользователем (которые составляют большинство типов, используемых на практике).

Прежде чем мы перейдем к подробному рассмотрению ссылочных типов, вы должны освоиться с некоторыми терминами, относящимися к этой области. Когда мы выделяем память для переменной ссылочного типа с помощью оператора new, то мы тем самым реализуем этот ссылочный тип. Таким образом, каждая переменная ссылочного типа является реализаций или экземпляром соответствующего типа.

Эта терминология может показаться вам новой и непривычной, поэтому стоит рассмотреть процесс реализации подробнее. Проблема заключается в том, что язык Java не позволяет нам просто объявить переменную ссылочного типа и сразу же начать записывать в нее значение. Мы должны сначала запросить у оболочки времени выполнения некоторый объем памяти, а оболочка, в свою очередь, должна сделать запись в своих внутренних таблицах, что мы активизировали переменную данного ссылочного типа. Весь этот процесс в целом и называется реализацией переменной. После реализации, когда мы имеем в своем распоряжении экземпляр переменной данного типа, мы уже можем использовать этот экземпляр для хранения данных. Важно понимать, что экземпляр переменной и сам ссылочный тип, к которому эта переменная относится, являются качественно различными понятиями - для хранения переменной можно использовать только реализованный экземпляр переменной ссылочного типа.

Теперь мы переходим к рассмотрению типов, определенных пользователем, после чего мы познакомимся со свойствами массивов в Java.

Типы, определенные пользователем

Большинство языков позволяют программисту определять новые типы. Например, в языке C новые типы можно создавать с помощью оператора struct, а в Паскале - с помощью записей (records). Язык Java позволяет определять новые типы с помощью классов, о которых мы будем говорить в этом разделе, а также с помощью интерфейсов (interfaces), речь о которых пойдет в главе 3, "Объектная ориентация в Java".

На простейшем уровне рассмотрения классы похожи на структуры или записи - они тоже позволяют хранить наборы переменных разных типов. Однако есть и важное отличие: классы помимо переменных могут включать в себя также и методы. Ниже приведен пример объявления нового типа, названного "MyType". Ключевое слово public, которое стоит перед определением типа, является так называемым модификатором доступа (access modifier) и указывает на то, что доступ к данным членам класса могут получить методы, не входящие в данный класс. Подробнее о модификаторах доступа мы будем говорить ниже в этой главе.

Пример 2-7a. Объявление нового типа.

```
class MyType {
        public int myDataMember=4;
        public void myMethodMember() {
                System.out.println("I'm a member!");
                System.out.println("myData="+myDataMember);}
        }
```

Вы, вероятно, обратили внимание на то, что этот пример напоминает по структуре собственно программы на языке Java, которые мы писали ранее. Это сходство отражает ту двойную роль, которую классы играют в языке Java.

В программах, которые приводились выше в качестве примеров, классы использовались как

средство организации содержимого - данных и алгоритмов каждой программы. Но классы могут также использоваться и для определения новых типов. Переменные типов, определенных через классы, называются объектами, реализациями или экземплярами соответствующих классов. Создание, или реализация, объекта осуществляется с помощью того же оператора new, а доступ к членам (составным частям) класса - с помощью оператора "точка" (.).

Пример 2-7b. Реализация объекта.

```
public class RunMe {
public static void main(String ARGV[]) {
        MyType Mine=new MyType();
        int i=Mine.myDataMember;
        Mine.myMethodMember();
        }
}
```

Пример 2-7 иллюстрирует три основных вида действий, которые можно производить с объектом: создание объекта, доступ к членам-переменным объекта и доступ к членам-методам этого объекта. Последняя строчка кода в этом примере вызывает метод myMethodMember, который выводит на экран следующее:

I'm a member!
myData=4

Поскольку тип myDataType является ссылочным типом, мы должны использовать оператор new. Этот оператор запрашивает у системы определенное количество памяти для хранения нашего объекта. Кроме того, мы можем определить, какие еще действия должны выполняться в момент реализации класса, определив так называемый конструктор (constructor). Вот как выглядит конструктор для типа myDataType, единственная функция которого - сообщить о том, что происходит реализация класса.

Пример 2-8a. Конструктор, сообщающий о реализации класса.

```
public class MyType {
int myDataMember=0;
public MyType() {
        System.out.println("Instantiation in process!");}
}
```

Конструкторы можно использовать также для инициализации (присвоения начальных значений) членов-переменных данного класса. Вот пример конструктора, который присваивает переменной myDataMember целочисленное значение, переданное этому конструктору через аргумент.

Пример 2-8b. Конструктор, который инициализирует значение переменной, входящей в класс.

```
public MyType(int val) {
        System.out.println("setting myDataMember="+val);
        myDataMember=val;}
```

Теперь представим, что оба приведенных выше конструктора определены в нашем классе myDataType. Вот еще один фрагмент программы, в котором используются оба эти конструктора.

Пример 2-8c. Программа, использующая оба этих конструктора.

```
public class RunMe {
public static void main(String ARGV[]) {
        MyType instance1=new MyType();
        MyType instance2=new MyType(100);
        }
}
```

Вывод этой программы будет иметь следующий вид:

```
Instantiation in progress!
I'm a member!
myDataType=4
setting myDataType=100
I'm a member!
```

myDataType=100

Стандартные типы, определенные пользователем

Работая с определенными пользователем типами, вы должны помнить одну важную вещь: название этих типов совсем не подразумевает, что каждый пользователь должен сам определять для себя все типы, которые ему понадобятся. В состав JDK входят десятки готовых классов, которые вы можете использовать в своих программах. По сути дела изучение Java по большей части сводится к знакомству с этими предопределенными классами и изучению их свойств и применимости. Эти стандартные классы входят в интерфейс прикладного программирования (Application Programming Interface, API). Подробнее мы будем говорить об API в главе 6.

Тип String

До сих пор мы говорили о примитивных типах и о типах, определенных пользователем. Теперь рассмотрим один особый тип, который представляет собой гибрид этих двух типов, - тип String (тип строковых переменных). В основе своей тип String является типом, определенным пользователем, так как он определяется как одноименный класс String, содержащий в себе методы и переменные. Но в то же время этот тип проявляет некоторые свойства примитивного типа, что выражается, в частности, в том, как осуществляется присвоение значений переменным этого типа.

String myString="Hello!";

Несмотря на то, что такой способ объявления и инициализации переменных типа String является не совсем законным с точки зрения синтаксиса типов, определенных пользователем, нельзя не признать, что для такого часто встречающегося в программировании объекта, как строки символов, этот способ является самым очевидным и удобным. Кроме того, для конкатенации (сложения) двух строк можно использовать знак операции +.

int muInt=4;

String anotherString=myString+"myInt is "+myInt;

После выполнения указанных действий значение переменной anotherString будет "Hello! myInt is 4". Однако поскольку anotherString является в то же самое время и объектом, мы можем вызывать методы - члены класса String. Так, чтобы вырезать первые пять символов строки anotherString, нужно написать следующее выражение:

String helloString=anotherString.substring(5);

Как видите, реализация переменных типа String не требует применения оператора new. С точки зрения практики программирования это очень удобно, поскольку строковые переменные используются очень часто. Однако, программируя на языке Java, вы всегда должны помнить о том, что тип String является особым - это единственный определенный пользователем тип, переменные которого могут объявляться и использоваться без применения оператора new.

Типы массива

Типы массива используются для определения массивов - упорядоченных наборов однотипных переменных. Вы можете определить массив над любым существующим в языке типом, включая типы, определенные пользователем. Кроме того, можно пользоваться массивами массивов или многомерными массивами (об этом см. в главе 4, "Синтаксис и семантика"). Коротко говоря, если мы можем создать переменную некоторого типа, значит, мы можем создать и массив переменных этого типа. Вместе с тем создание массивов в языке Java может показаться вам непривычным, так как оно требует применения оператора new.

Пример 2-9a. Выделение памяти для массива целых чисел myIntArray[].

```
int myIntArray[];
myIntArray=new int[3];
MyType myObjectArray[];
myObjectArray=new MyType[3];
```

Оператор new дает команду оболочке времени выполнения выделить необходимое количество памяти под массив. Как видно из этого примера, необязательно объявлять размер массива тогда же, когда вы создаете переменную-массив. После того как вы создали массив оператором new, доступ к этому массиву осуществляется точно так же, как в языках C или Паскаль.

Пример 2-9b. Присвоение значений элементам массива myIntArray[].

```
myIntArray[0]=0;
myIntArray[1]=1;
myIntArray[2]=2;
```

```
myObjectArray[0]=new MyType();
myObjectArray[1]=new MyType();
myObjectArray[2]=new MyType();
myObjectArray[0].myDataMember=0;
myObjectArray[1].myDataMember=1;
myObjectArray[2].myDataMember=2;
```

Массивы в языке Java имеют три важных преимущества перед массивами в других языках. Во-первых, как вы только что видели, программисту необязательно указывать размер массива при его объявлении. Во-вторых, любой массив в языке Java является переменной - а это значит, что его можно передать как параметр методу и использовать в качестве значения, возвращаемого методом (подробнее об этом преимуществе мы будем говорить в следующем разделе, посвященном методам). И в-третьих, не составляет никакого труда узнать, каков размер данного массива в любой момент времени. Например, вот как определяется размер массива, который мы объявили выше.

Пример 2-9c. Получение длины массива.

```
int len=myIntArray.length;
System.out.println("Length of myIntArray="+len);
```

Методы

Метод в языке Java представляет собой подпрограмму, аналогичную функциям языков C и Паскаль. Каждый метод имеет тип возвращаемого значения и может вызываться с передачей некоторых параметров.

Для простоты все методы, которые мы будем использовать в наших примерах, будут объявлены статическими (static). Модификатор static, как и другие модификаторы методов, влияет на то, как будет вести себя данный метод в объектно-ориентированной программе (подробнее об этом ниже).

Для начала давайте разберемся с синтаксисом объявления метода. Модификаторы, если они есть, предшествуют указанию типа возвращаемого значения, за которым следует имя метода и список параметров в круглых скобках. Следующее затем тело метода заключено в пару фигурных скобок:

```
<модификаторы_метода> тип_возвращаемого_значения имя_метода (<параметры)> {
        тело_метода
}
```

Тело метода может содержать объявления переменных и операторы. В отличие от языка C объявления переменных могут располагаться в любом месте тела метода, в том числе и после каких-то операторов.

Ниже мы рассмотрим вопросы, связанные со значениями, возвращаемыми методами, и с передачей параметров методам. В конце раздела мы познакомимся с особым свойством языка Java, которое называется совмещением методов (method overloading), благодаря которому можно давать одно и то же имя нескольким методам, различающимся между собой списком принимаемых параметров.

Возвращаемые значения

С каждым методом должен быть соотнесен тип возвращаемого им значения. Тип void, который был приписан нашему методу main в примерах этой главы, является специальным способом указать системе, что данный метод не возвращает никакого значения. Методы, возвращаемый тип которых объявлен с помощью ключевого слова void, аналогичны процедурам языка Паскаль. Методы, у которых возвращаемое значение принадлежит к любому другому типу, кроме void, должны содержать в своем теле оператор return. Возвращаемое значение может принадлежать к любому из типов, о которых мы говорили в разделе "Переменные", - включая как примитивные типы, так и типы, определенные через класс. Ниже приведены примеры методов, не возвращающих никакого значения, и методов, возвращающих значение определенного типа.

Пример 2-10. Вызов методов.

```
public class MethodExamples{
static void voidMethod() {
        System.out.println("I am a void method");
```

```
        }
static int returnInt() {
        int i=4;
        System.out.println("returning 4");
        return i;}
static public final void main(String S[]) {
        System.out.println("Hello, methods!");
        System.out.println("Calling a void method");
        voidMethod();
        int ans=returnInt();
        System.out.print("method says -.-");
        System.out.println(ans);
        }
}
```

Как вы, вероятно, заметили, вызов методов в этом примере осуществлялся точно так же, как мы вызывали бы функции или процедуры в другом, не объектно-ориентированном языке. Это связано с тем, что в нашем примере статические методы вызывали другие статические методы, принадлежащие к тому же классу. То же самое верно и для тех случаев, когда нестатические (динамические) методы вызывают другие динамические методы. Однако когда динамические методы вызывают статические методы и, наоборот, когда возникает необходимость вызвать метод из другого класса, синтаксис вызова меняется. Об этих изменениях мы поговорим в следующем разделе.

Передача параметров

В качестве параметров в языке Java можно передавать переменные любого типа, включая типы, определенные через классы, и массивы переменных любого типа и размера. Однако переменные примитивных типов, передаваемые в качестве параметров, ведут себя иначе, чем переменные ссылочных типов в том же контексте. Сначала рассмотрим передачу переменных примитивных типов.

Все переменные примитивных типов передаются методам по значению (by value). Это означает, что в момент вызова метода делается копия переменной, передаваемой методу. Если метод будет изменять в своем теле значение переданной ему в качестве параметра переменной, то содержимое исходной переменной изменяться не будет, так как все действия будут производиться с ее копией. Проиллюстрируем это примером.

Пример 2-11. Передача в качестве параметров переменных примитивных типов.

```
class ParameterExample {
static int addFour(int i) {
        i=i+4;
        System.out.println("local copy of i="+i);
        return i;}
public final static void main(String S[]) {
        System.out.println("Hello, parameter passing!");
        int i=10;
        System.out.print("Original value of i="+i);
        int j=addFour(i);
        System.out.println("value of j="+j);

        System.out.println("Current value of i="+i);
        }
}
```

Вывод этой программы имеет следующий вид:

```
Hello, parameter passing!
Original value of i=10
value of j=14
Current value of i=10
```

Как видите, значение переменной i не изменилось, хотя метод addFour прибавил к значению своего параметра 4. Напротив, значения переменных ссылочного типа, переданных в качестве параметров, можно изменить в теле метода. Рассмотрим пример с массивом целых чисел.

Пример 2-12. Передача в качестве параметра переменной ссылочного типа.

```
public class ReferenceParameterExample {
static void changeArray(int referenceVariable[]) {
        referenceVariable[2]=100;}
public static void main(String ARGV[]) {
        int anArray[]=new int[3];
        anArray[2]=10;
        System.out.println("anArray[2]=");
        System.out.println(anArray[2]);
        changeArray(anArray);
        System.out.println(anArray[2]);}
}
```

Вывод программы выглядит так:

```
anArray[2]=
10
100
```

Когда мы передаем методу в качестве параметра переменную ссылочного типа, мы явным образом меняем то, на что указывает эта переменная, - в нашем случае массив целых чисел.

Строковые переменные и передача параметров

Несмотря на то, что тип String является определенным пользователем типом, он не ведет себя как ссылочный тип при передаче параметров. Переменные типа String в качестве параметров метода всегда передаются по значению, - то есть передав методу строковую переменную, вы в теле метода будете фактически работать с копией этой строковой переменной. Другими словами, изменение значения строковой переменной в теле метода не влияет на значение этой же переменной снаружи метода.

Совмещение методов

Вы наверняка сталкивались с необходимостью создавать две или несколько функций, выполняющих, по сути, одни и те же действия, но имеющих различные списки параметров. Язык Java дает в таких ситуациях более изящный выход из положения. В этом языке вы можете присвоить одно и то же имя нескольким методам, которые различаются списками своих параметров. Например, пусть у нас есть метод, предназначенный для сравнения двух целых чисел.

Пример 2-13a. Сравнение двух целых чисел.

```
public static String compareNums(int i, int j) {
if (i==j) {
        return "Numbers "+i+" and "+j+" are equal";}
if (ij) {
        return "Number "+i+" greater than "+j;}
        return "Number "+j+" greater than "+i;
}
```

Теперь представьте, что нам в программе понадобилось сравнить не два, а три целых числа. Конечно, можно было бы определить для этого новый метод с именем типа compareThreeNums. Но, к счастью, язык Java позволяет обойтись без умножения количества имен в программе.

Пример 2-13b. Совмещение метода с дополнительными параметрами.

```
public static String compareNums(int i, int j, int k) {
String S=compareNums(i,j);
S=S+"\n";
S=S+compareNums(i,k);
return S;}
```

Составляя каждый раз иной список параметров, мы таким образом можем определить любое количество методов с одним и тем же именем compareNums. Это становится особенно удобным в тех случаях, когда требуется произвести одно и то же действие над переменными разных типов. Как вы узнаете из главы 4, Java не позволяет передавать, к примеру, переменные типа double методу, параметры которого имеют тип int. Однако ничто не мешает нам прибегнуть к совмещению, определив еще один метод с тем же именем и со списком параметров типа double (или любого другого типа).

Пример 2-13c. Совмещение метода с параметрами другого типа.

```
public static String compareNums(double i, double j, double k)
{
if (i==j) {
        return "Numbers "+i+" and "+j+" are equal";}
if (i>j) {
        return "Number "+i+" greater than "+j;}
        return "Number "+j+" greater than "+i;
}
```

Выгоды совмещения методов особенно очевидны для тех, кому приходится этими методами пользоваться: вместо того чтобы помнить несколько имен разных методов, можно ограничиться запоминанием только одного имени, общего для методов с разными параметрами. В обязанности компилятора входит выяснение того, какой именно метод требуется вызвать в каждом случае.

Пример 2-13d. Вызов совмещенных методов.

```
public static void main(String ARGV[]) {
        int a=3;
        int b=4;
        int c=5;
        double d=3.3;
        double e=4.4;
        double f=5.5;
String S=compareNums(a,b);
        System.out.println(S);
        S=compareNums(a,b,c);
        System.out.println(S);
        S=compareNums(d,e,f);
        System.out.println(S);}
}
```

Классы

Теперь настало время заполнить некоторые пробелы в том, что вы уже знаете о классах. Как вы помните, наше знакомство с классами началось с того, что классы могут содержать в себе переменные и методы. В примерах, с которыми мы до сих пор имели дело, этого простейшего объяснения было вполне достаточно. Однако, с другой стороны, классы лежат в фундаменте объектно-ориентированных свойств языка Java, и теперь мы рассмотрим их с этой стороны.

Статические и динамические члены

Когда выше шла речь о переменных, мы видели, что с помощью классов можно определять новые типы. Теперь пора выяснить, что же означает модификатор static, который использовался в объявления методов в наших примерах. До сих пор мы использовали этот модификатор только при определении методов, поэтому сначала мы познакомимся с той стороной значения ключевого слова static, которая имеет прямое отношение к методам. Как вы увидите ниже, модификатор static может также использоваться с переменными, но при этом он имеет иное значение, нежели с методами.

Если в определении метода не использовать ключевое слов static, то этот метод будет по умолчанию динамическим (dynamic). Динамические методы и переменные всегда являются членами объектов, и доступ к ним осуществляется через переменную-объект. Напротив, статические методы не могут быть членами объектов. В табл. 2-5 указан синтаксис вызова динамических и статических методов.

Таблица 2-5. Синтаксис вызова динамических и статических методов

Тип метода	Модификатор	Синтаксис
Динамический	никакого (по умолчанию)	объект.имя метода (список параметров)
Статический	static	имя класса.имя метода (список параметров)

Проиллюстрируем это примером.

Пример 2-14a. Определение статических и динамических методов.

```
public class StaticVsDynamic {
        int i=0;
        public static void staticMethod(int j) {
                System.out.println("A static method");
                System.out.println("j="+j);
                }
// динамические методы
        public void setInt(int k) {
                i=k;
                System.out.println("setting i to "+k);
                }
        public int returnInt() {
                return i;}
}
```

Класс, определенный в этом примере, включает в себя один статический и один динамический метод. При этом статический метод не знает о существовании динамических членов класса setInt, returnInt и i. Вот как будет выглядеть первичный класс, иллюстрирующий различный синтаксис вызова статических и динамических методов.

Пример 2-14b. Вызов статических и динамических методов.

```
public class RunMe {
        public static void main(String S[]) {
        int i=0;
        StaticVsDynamic.staticMethod(10);
         // чтобы вызвать статический метод, не обязательно создавать объект
        StaticVsDynamic A=new StaticVsDynamic();
        // прежде чем вызывать динамический метод, требуется реализовать
экземпляр
        // объекта
        A.setInt(20);
        System.out.println("A.i = "+A.returnInt());
        }
}
```

Модификатор static и метод main

Теперь вам должно быть понятно, почему модификатор static всегда присутствует в объявлении метода main. Дело в том, что, когда мы вводим команду "java primaryClass", оболочка времени выполнения языка Java загружает класс primaryClass в память в виде типа, а не в виде объекта. После этого оболочка просто вызывает метод main в виде "primaryClass.main (S)", где S - массив параметров командной строки.

Кроме того, модификатор static можно использовать при объявлении переменных. Синтаксис обращения к переменной похож на правила вызова функции:
<имя класса>.<имя переменной>

Поскольку все методы и переменные должны принадлежать к какому-то классу, модификатор static используется для указания на те методы и переменные, которые не играют роль части объекта. Это делает их в какой-то мере эквивалентными глобальным подпрограммам и переменным в каком-нибудь не объектно-ориентированном языке - за тем исключением, что мы все-таки должны знать имя класса, в котором они содержатся, чтобы получить к ним доступ.

Доступ к членам класса

Java позволяет контролировать доступ к методам и переменным, входящим в тот или иной класс. До сих пор все члены классов, которые мы объявляли, были общедоступными (public). Модификатор public указывает на то, что значение данной переменной можно изменять из любого места нашей программы. Однако существует возможность ограничить доступ к методам и переменным с помощью модификаторов, перечисленных в табл. 2-6.

Таблица 2-6. Модификаторы доступа

Модификатор	Описание
public	Член класса доступен из любого места программы
private	Член класса доступен только в пределах данного класса
protected	Член класса доступен из любого места своего пакета, но недоступен за пределами пакета

В дополнение к этим трем существует еще один модификатор доступа, о котором мы будем говорить в главе 3, а именно private protected. Цель всех модификаторов доступа - защитить объекты от взаимодействия с теми членами классов, с которыми они не должны взаимодействовать.

Возможно, кому-то покажется, что тем самым мы наделяем программы, которые пишем, человеческими чертами и приписываем объектам программы свои собственные желания и свое собственное поведение. В конце концов, разве не программист обладает высшей властью над тем, что делает его программа?

Конечно, это так. Но модификаторы доступа - это единственное, что может дать нам абсолютную гарантию того, что объект будет вести себя так, как нам нужно. Если фрагменты вашего кода будет использовать кто-то другой (или даже вы сами, но спустя некоторое время, когда вы уже не будете помнить всех тонкостей своей программы), такое ограничение доступа, вполне возможно, избавит вас от многих неприятностей. Без этого ограничения довольно сложно было бы гарантировать, что новые классы, добавляемые в программу, не взаимодействуют каким-нибудь нежелательным способом с членами других классов, совсем для этого не предназначенными.

Что означает модификатор public в объявлении класса?

Как вы, вероятно, заметили, в примерах этой главы классы также объявляются с ключевым словом public. Это означает, что к ним могут получить доступ классы, не входящие в тот же пакет, что и объявляемый класс. В отличие от методов и переменных, классы могут либо иметь в своем объявлении ключевое слово public, либо не иметь его. Остальные три модификатора доступа в объявлении классов использоваться не могут.

Эта практика носит название "затенение данных" (data hiding) и играет важную роль в объектно-ориентированных свойствах языка, о которых мы будем говорить в главе 3. Здесь для иллюстрации этого понятия мы приведем один простой пример. Предположим, что мы пишем класс, в задачи которого входит отслеживание количества денег, вырученных магазином, и количества обслуженных покупателей. (Разумеется, в реальной жизни вам наверняка понадобилось бы отслеживать и множество других вещей - например, что именно, когда и кому продано, - но для нашего примера можно ограничиться простейшим случаем.) Вот как выглядит объявление этого класса.

Пример 2-15. Общедоступные и затененные члены класса.

```
public class SaleProcessor {
        private int Revenue=0;
        private int numSales=0;
public void recordSale(int newRevenue) {
        Revenue=Revenue+newRevenue;
        numSales=numSales+1;}
public int getRevenue() {
        return Revenue;}
public int getNumSales() {
        return numSales;}
}
```

Каждый раз, когда производится покупка, программа должна вызывать метод recordSale, который увеличивает на нужную величину сумму выручки и инкрементирует счетчик покупок. Объявив переменные, в которых хранятся выручка и число покупок, с модификатором private, мы гарантируем, что их значение не будет меняться кем-либо, кроме как специально разработанными для этого методами, входящими в этот класс.

Наследование классов

Как видите, модификаторы доступа делают классы более устойчивыми и надежными в работе, так как гарантируют, что снаружи класса можно будет получить доступ только к некоторым из методов и переменных. Наследование (inheritance), в свою очередь, упрощает практическое использование классов, так как позволяет расширять уже написанные и отлаженные классы, добавляя к ним новые свойства и возможности. Мы можем с легкостью создавать новые классы, которые будут содержать все члены, входившие в исходный класс, плюс любое количество новых членов.

Рассмотрим класс saleProcessor, объявленный в примере 2-15. Представьте себе, что начальник, ознакомившись с вашей работой, выражает желание иметь в своем распоряжении другой класс, позволяющий отслеживать деньги в кассе магазина. Мы можем взять существующий класс saleProcessor и, пользуясь методикой объектно-ориентированного программирования, расширить его возможности. Для простоты мы не будем учитывать медные деньги (меньше доллара), банкноты стоимостью свыше десяти долларов и необходимость возвращать сдачу.

Пример 2-16. Наследование классов.

```
class CashRegister extends SaleProcessor{
private int Ones=0;
private int Fives=0;
private int Tens=0;
CashRegister(int startOnes, int startFives, int startTens){
        Ones=startOnes;
        Fives=startFives;
        Tens=startTens;}
public void sellToCustomer(int newOnes, int newFives, int newTens)
        {
        int thisSum=0;
        Ones=Ones+newOnes;
        thisSum=newOnes;
        Fives=Fives+newFives;
        thisSum=thisSum+(newFives*5);
        Tens=Tens+newTens;
        thisSum=thisSum+(newTens*10);
        recordSale(thisSum);
        }
public int numOnes() {return Ones;}
public int numFives() {return Fives;}
public int numTens() {return Tens;}
}
```

Конструкторы и наследование

В приведенном выше фрагменте кода мы определили конструктор, входящий в подкласс. Как уже упоминалось выше, все классы в языке Java имеют по умолчанию простейший конструктор, который вызывается без каких-либо параметров. Оказывается, этот факт имеет свое объяснение: дело в том, что все классы языка Java являются расширениями специального класса под названием Object. Именно в классе Object и определен этот конструктор по умолчанию.

В этом примере, расширяя класс saleProcessor, мы получаем возможность пользоваться всеми написанными и отлаженными функциями этого класса вместо того, чтобы писать их заново. Это свойство, называемое повторным использованием кода (code reuse), является одним из главных преимуществ объектно-ориентированного программирования.

Пакеты

Итак, мы с вами познакомились с ядром языка Java. Как видите, классы являются основным строительным блоком любой Java-программы. В сравнении с классами пакеты выполняют чисто утилитарную функцию. Они просто содержат в себе наборы классов, а также объектов двух других видов, о которых мы еще не говорили, - исключения и интерфейсы. Кроме того, пакеты позволяют определять защищенные (protected) члены классов, которые доступны всем классам, входящим в один и тот же пакет, но недоступны каким бы то ни было классам за пределами этого пакета.

Сначала рассмотрим функцию, которую пакеты выполняют, будучи контейнерами для какого-то содержимого. Здесь пакеты играют простую, но очень важную роль: они позволяют компилятору найти классы, необходимые для компиляции пользовательской программы. Вы,

конечно, помните метод System.out.println, с помощью которого мы осуществляли текстовый вывод в наших примерах программ. На самом деле System представляет собой класс, входящий в пакет java.lang наряду с еще одним знакомым нам классом, String. С помощью оператора import программа может получить доступ к этим классам. В самом первом из наших примеров с помощью такого оператора import мы осуществляли доступ к классу Date:
import java.util.*;

Символ * в этом операторе означает, что компилятор должен импортировать все классы, входящие в пакет java.util. Этот пакет является одним из нескольких пакетов, входящих в API (интерфейс прикладного программирования), о котором мы будем говорить в главе 6, "Интерфейс прикладного программирования".

Компилятор Java самостоятельно определяет пакет со всеми классами, расположенными в текущем каталоге, и импортирует этот пакет во все программы, которые вы пишете и храните в этом каталоге. Вот почему у нас не возникало необходимости объединять явным образом классы, которые мы до сих пор написали, в какой-то пакет. Если же вам потребуется явным образом включить некий класс в некий пакет, это делается так:

Пример 2-17. Включение класса в пакет.

```
package simplePackage;
class simpleClass1 {
        public pubMethod() {
                System.out.println("This is a public method");}
        protected protectedMethod() {
                System.out.println("This is a protected method");}
        }
```

В этом примере мы поместили класс simpleClass в пакет simplePackage. Чтобы добавить в этот пакет еще один класс, достаточно поместить строку "package simplePackage" в начало файла, содержащего этот класс. Все классы, входящие в данный пакет, будут иметь доступ к защищенному методу protectedMethod, а классы, не входящие в пакет, не будут иметь доступа к этому методу.

Оболочка времени выполнения Java

Вам предстоит прочесть еще несколько глав, прежде чем вы сможете сказать, что понимаете все тонкости языка Java. Однако понимание базовой структуры и основных элементов, составляющих Java-программу, у вас должно быть уже сейчас. Прежде чем мы перейдем к созданию нашего первого апплета, давайте обсудим оболочку времени выполнения языка Java. Как вы, вероятно, помните из главы 1, "World Wide Web и Java", программы на языке Java выполняются в рамках виртуальной машины. Все, что программа имеет возможность знать об окружающем мире, содержится в оболочке времени выполнения (runtime environment), которая создается для этой программы виртуальной машиной. Сам по себе язык Java имеет достаточно свойств, которые делают его мечтой всякого программиста, - к примеру, объектную ориентированность, встроенную обработку ошибок и возможность одновременно выполнять фрагменты одной и той же программы (многопотоковость). Однако главное преимущество этого языка, которое ставит его в совершенно особое положение, - это полнейшая независимость от компьютерных платформ, которая целиком обеспечивается оболочкой времени выполнения. Давайте поговорим о том, как оболочка времени выполнения способна изменить жизнь программистов, переходящих на язык Java.

Процессы компиляции и выполнения

Java относится к частично компилируемым (semi-compiled) языкам. В отличие от "просто компилируемых" языков, компилятор Java не создает окончательно скомпилированный файл, готовый к запуску на компьютере. Вместо того он создает файл, который может исполнять специальная система - оболочка времени выполнения. Это означает, что вы можете написать и скомпилировать Java-программу на одной платформе, затем перенести ее на другую платформу и сразу же запустить без повторной компиляции.

Файл с расширением .class, создаваемый компилятором Java, состоит из так называемых байтовых кодов (bytecodes). Байтовые коды представляют собой не что иное, как инструкции для оболочки времени выполнения, в чем-то подобные инструкциям на машинном языке, из которых состоит скомпилированная программа на C. Единственное отличие - то, что если машинные инструкции исполняются операционной системой и собственно процессором компьютера, байтовые коды целиком обрабатываются оболочкой времени выполнения. Например, когда

программа запрашивает какое-то количество памяти или хочет получить доступ к устройству ввода-вывода (скажем, клавиатуре или монитору), реагировать на эти запросы будет именно оболочка времени выполнения. Сама программа никогда не имеет прямого доступа к компонентам системы. Таким образом, оболочка времени выполнения Java надежно изолирует Java-программу от аппаратуры компьютера. Эта изоляция приобретает особое значение для апплетов - как вы понимаете, пользователям вряд ли понравится, если в то время, как они читают Web-страницу, встроенный в эту страницу апплет займется, к примеру, форматированием жесткого диска.

Определение байтовых кодов

В этой книге вы не найдете подробной спецификации байтовых кодов. Вам достаточно знать, что исходный текст на языке Java преобразуется компилятором в байтовые коды, которые, в свою очередь, исполняются оболочкой времени выполнения Java. В действительности байтовые коды сами по себе представляют особый язык программирования - правда, вряд ли вы когда-нибудь захотите писать программы прямо на этом языке.

Некоторым из читателей, возможно, покажется, что Java благодаря этому можно считать интерпретируемым языком, - таким же как, к примеру, Perl или Бейсик. И в самом деле, .class-файл подвергается интерпретации точно так же, как программа на языках Perl или Бейсик. Однако программы на Java в сравнении с этими языками выполняются гораздо быстрее, поскольку компьютеру намного удобнее интерпретировать байтовые коды, чем исходный код на языках Perl или Бейсик, приспособленный для чтения человеком.

В каком-то смысле .class-файл можно считать сжатой формой представления .java-файла с исходным кодом программы, - причем сжатой таким образом, что оболочке времени выполнения Java не составляет никакого труда выполнить содержащиеся в этом файле инструкции. В то же время это "сжатие" совсем не является оптимальным. Любой .class-файл, созданный javac или любым другим корректно работающим компилятором языка Java, всегда содержит больше информации, чем необходимо для выполнения запрограммированных в нем действий. Эта дополнительная информация вводится в байтовый код для того, чтобы защититься от программ, которые могли бы попытаться "обмануть" оболочку времени выполнения Java.

О каком обмане идет здесь речь? К примеру, можно представить себе программу, которая попытается получить доступ к компонентам системы в обход оболочки времени выполнения. Написать такую программу непосредственно на Java невозможно, но какой-нибудь компьютерный взломщик может попытаться написать что-либо подобное прямо в байтовых кодах. Здесь и вступает в игру упомянутая выше дополнительная информация, включенная в .class-файлы: написать вручную, без использования компилятора Java, такой байтовый код, дополнительная информация в котором прошла бы проверку на подлинность в оболочке времени выполнения, очень сложно. Как мы увидим далее, это один из трех основных методов, которые реализованы в оболочке времени выполнения, чтобы обеспечить безопасность запуска апплетов и надежную изоляцию от них компонентов системы.

Несмотря на присутствие этой дополнительной информации, программы, написанные на языке Java, выполняются все же значительно быстрее, чем программы на полностью интерпретируемых языках типа Perl или Бейсик. С другой стороны, по скорости выполнения Java-программы все же проигрывают в сравнении с программами на языках, которые компилируются прямо в машинные коды (например, C и C++). К счастью, из этой ситуации есть оригинальный выход - недавно появившаяся технология так называемого "компилирования вовремя" (Just In Time, JIT). "Компилирование вовремя" означает, что байтовые коды действительно компилируются, а не интерпретируются в машинные инструкции при выполнении Java-программы. При первом же запуске программы оболочка времени выполнения компилирует ее в машинные инструкции, а при последующих запусках управление сразу получает этот скомпилированный код. Таким образом, при первом исполнении каждой новой Java-программы на данном компьютере ее скорость работы будет понижена из-за затрат времени на компиляцию, но зато при последующих запусках эта программа будет выполняться так же быстро, как если бы она была написана на полностью компилируемом языке типа C и скомпилирована для данной компьютерной платформы.

Следите за новостями по JIT

Когда эта книга готовилась к печати, первые компиляторы Java, работающие по принципу JIT, уже появились на рынке. Однако они еще не получили сколько-нибудь широкого распространения среди программистов, пишущих на Java. Кроме того, эти компиляторы не перенесены на все те платформы, на которых в настоящее время поддерживается Java, и ни один из броузеров WWW не имеет встроенного компилятора JIT. Тем не менее фирма Sun Microsystems преследует цель сделать технологию "компилирования вовремя" неотъемлемой частью всех существующих и будущих реализаций языка Java. Последнюю информацию о компиляторах JIT

вы всегда найдете на нашей странице Online Companion.

Несомненно, JIT-компиляторы для языка Java вскоре получат широкое распространение. Однако для вас как программиста на языке Java это не создаст никаких проблем: поскольку "компилирование вовремя" реализовано как свойство оболочки времени выполнения, оно даст лишь ускорение работы программ и не потребует каких-либо изменений в исходных текстах.

Сборка мусора

Конечно, если вы всерьез собираетесь заняться программированием на Java, вы должны понимать некоторые аспекты архитектуры оболочки времени выполнения Java, о которой мы говорили в предыдущем разделе. Однако с чисто практической точки зрения структура и устройство этой оболочки не так уж важны для программиста, поскольку они не влияют непосредственно на написание программ. Достаточно знать, что .class-файл исполняется на виртуальной машине и содержит байтовые коды; если результат выполнения этого файла нас устраивает, то мы можем особо не интересоваться тем, что именно происходит с кодом в оболочке времени выполнения - интерпретация, компиляция, частичная компиляция или еще какая-нибудь волшебная трансформация.

Однако одна из функций, выполняемых оболочкой времени выполнения, все-таки влияет на то, как должны писаться программы на языке Java. Эта функция называется сборкой мусора (garbage collection), и она делает жизнь программиста намного более светлой и радостной. Как вы уже, наверное, догадались, эта Сборка мусора, осуществляемая оболочкой времени выполнения Java, не имеет никакого отношения к завалам мятых распечаток, банок из-под пепси-колы и упаковок растворимого кофе, которые окружают компьютер любого уважающего себя программиста. Мусор, который собирает оболочка Java, - это те переменные в вашей программе, которые выполнили свою функцию и больше не нужны.

Случалось ли вам когда-нибудь сталкиваться с "утечкой памяти"? Эта неприятность случается, когда программа запрашивает у операционной системы все новые и новые участки оперативной памяти, но никогда не освобождает и не возвращает их. Через какое-то время (иногда очень незначительное) такое поведение программы приводит к исчерпанию свободной памяти в системе и к печальному концу - зависанию программы, а нередко и всего компьютера.

Конечно, причиной этой утечки памяти всегда является ошибка программиста, причем этот тип ошибок бывает особенно трудно обнаружить и исправить. Достаточно просто запросить подо что-то память и забыть вернуть ее системе. Чтобы обнаружить, где же происходит утечка памяти, вы должны просмотреть буквально каждое место в программе, где происходит какое-то выделение памяти, и проверить, возвращается ли эта память системе после использования. В программах длиной во многие тысячи строк такое расследование может отнять огромное количество сил и времени.

Язык Java полностью избавит вас от таких забот. Оказывается, единственный способ, которым наша программа может запросить память у системы во время своего выполнения, - это присвоение значения переменной-объекту или создание массива. В обоих этих случаях память запрашивается неявным образом, то есть от вас не требуется вычислять нужное количество байтов и предусматривать вызов специальной функции выделения памяти. Разумеется, было бы очень странно, если бы при таком неявном выделении памяти освобождать ее приходилось бы явно. Кроме того, сборка мусора защищает программиста от еще одной распространенной ошибки - записи данных в тот участок памяти, который уже освобожден и возвращен системе.

Когда мы присваиваем значение какой-то переменной, оболочка времени выполнения помещает особый маркер на блок памяти, который выделен для присвоенного значения. Если объект или массив создается в программе для местного использования в пределах какого-то фрагмента кода (например, в теле метода), то память, занятая этим объектом или массивом, будет освобождена, когда этот фрагмент кода потеряет управление. Такой процесс и называется сборкой мусора. Однако если эта переменная передается для использования другим частям программы - например, если она возвращается из этого метода в операторе return или включается в состав объекта или массива, передаваемого в качестве параметра другому методу, - то система учтет это и не будет освобождать занятую под переменную память.

Хорошо, скажете вы, но как эта сборка мусора может повлиять на написание программ? Во-первых, вы можете больше не беспокоиться об утечках памяти, поскольку оболочка времени выполнения следит за тем, как память используется вашей программой, и освобождает всю память, которая программе больше никогда не понадобится. С другой стороны, система сборки мусора способна самостоятельно определить, что является и что не является "мусором". Рассмотрим пример типичной ошибки в языке C, которая может стоить начинающим программистам многих часов безуспешного поиска ошибки:

```
public char *thisWontWork () {
```

```
        char localArray [6];
        strcpy (localArray,"hello");
        return localArray;}
```

По замыслу программиста определенная таким образом функция должна создавать массив символов, заполнять его строчкой "hello" и возвращать это значение. К сожалению, на практике все происходит совсем не так. Поскольку localArray определен внутри функции, вся занятая им память автоматически освобождается по завершении работы этой функции, не обращая внимания на то, что содержащееся в localArray значение мы возвращаем во внешний мир.

Нередко дело принимает еще более скверный оборот. Если мы проверим значение, возвращенное функцией, сразу по завершении ее работы, то скорее всего обнаружим, что оно является именно тем, что нам нужно, - просто потому, что освобожденная память еще не занята никакими другими данными. Но рано или поздно на тот участок памяти будут записаны другие данные - и, вероятно, к тому моменту, как это произойдет, у нас уже будет очень мало шансов сообразить, какая именно часть программы ответственна за эту неприятность: ведь виновная функция отработала уже очень давно и возвратила, казалось бы, совершенно правильное значение!

Поскольку подсистема оболочки времени выполнения Java, ответственная за сборку мусора, отслеживает использование переменных динамически, то подобных неприятностей с вами больше никогда не случится. Рассмотрим фрагмент кода на Java, эквивалентный приведенному выше коду на C:

```
public char[] thisWillWork () {
        char localArray[6];
        localArray={'h','e','l','l','o'};
        return localArray; }
```

В этом случае сборщик мусора обязательно заметит, что переменная localArray возвращается в качестве значения и, таким образом, продолжает использоваться в программе. Поэтому освобождение памяти, занятой под localArray, произойдет только тогда, когда этой переменной не будет присвоено никакого значения либо по завершении программы.

Создание Java-апплетов

Возможно, основы языка Java, которые мы здесь изучаем в поте лица, даются вам не без труда. Что ж, теперь самое время немного развлечься. Итак - апплеты!

Поскольку апплеты встраиваются в Web-страницы, их разработка включает в себя несколько новых этапов, которых не было в привычном вам цикле "редактирование - компиляция - запуск", на котором строится разработка обычных программ. К концу этого раздела вы должны научиться составлять и запускать свои собственные апплеты. После этого вы сможете перейти к главе 5, "Апплет в работе", и пополнить ряды авторов апплетов для World Wide Web.

Ваш первый апплет

Итак, давайте напишем простейший апплет. Откройте свой любимый текстовый редактор и введите текст следующего примера.

Пример 2-18a. Ваш первый апплет.

```
import java.applet.*;
import java.awt.*;
public class FirstApplet extends Applet {
public void paint(Graphics g) {
        g.drawString("Hello, Applets!",50,50);}
}
```

Вы, конечно, заметили, что эта программа довольно сильно отличается от тех, что мы писали до сих пор. Подробнее об этих отличиях мы будем говорить ниже, а сейчас давайте проверим нашу программу в работе. Вот что вам нужно сделать:

1. Скомпилируйте класс FirstApplet.
2. Теперь вставьте готовый апплет в Web-страницу. Для этого еще раз откройте свой текстовый редактор и создайте файл со следующим содержимым.

Пример 2-18b. Web-страница со ссылкой на FirstApplet.

```
<APPLET CODE=FirstApplet.class WIDTH=200 HEIGHT=200>
Ваш броузер не поддерживает язык Java. Посетите
<a href="http://
www.netscape.com">Netscape</a> и скачайте Netscape 2.0
</APPLET>
```

Этот файл можно сохранить под любым именем, но обязательно с расширением .html. Текст между тегами <APPLET...> и </APPLET> предназначен для тех броузеров, которые не умеют запускать встроенные в страницу апплеты.

3. Найдите в дистрибутиве JDK программу под названием "appletviewer". На компьютерах с UNIX и Windows эта программа находится в подкаталоге bin. Если, как рекомендовалось выше, вы вставили этот каталог в путь поиска исполняемых файлов, то вам достаточно будет на следующем этапе напечатать "appletviewer" в командной строке. Пользователи Macintosh найдут программу appletviewer на верхнем уровне JDK.
4. С помощью программы appletviewer откройте .html-файл, который вы только что создали. На компьютерах с UNIX и Windows имя этого .html-файла нужно передать программе просмотра апплетов в командной строке, а пользователи компьютеров Macintosh должны запустить appletviewer и выбрать команду Open из меню File.

После этого вы должны увидеть на экране окно, похожее на то, что изображено на рис. 2-1.

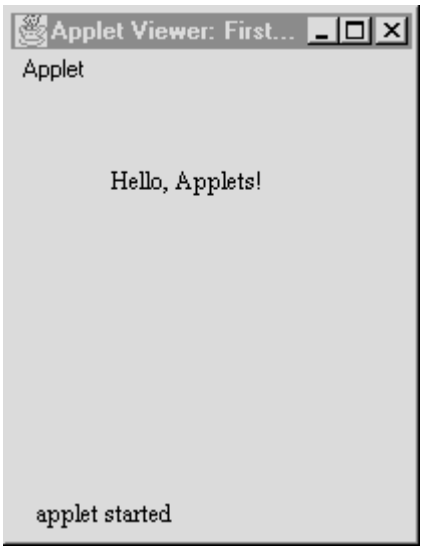

Рис. 2.1.

Вы можете поинтересоваться, нельзя ли использовать броузер Web, способный запускать Java-апплеты, для просмотра нашего примера. Это вполне возможно, но существует одно затруднение: ни программа просмотра апплетов, ни Netscape Navigator 2.0 не способны самостоятельно отследить перекомпиляцию кода апплета. Какой бы из этих двух программ вы ни пользовались, каждый раз, когда вы изменяете код апплета, вы должны будете выйти из программы и зайти в нее снова. А поскольку appletviewer - программа гораздо меньших размеров и быстрее запускающаяся, чем Netscape, выводы ее использования при разработке апплетов очевидны.

Как устроен апплет

Теперь, написав наш первый апплет, давайте разберемся, из каких частей он состоит. Класс, определенный в этом апплете, также является первичным классом, хотя он достаточно сильно отличается от первичных классов в примерах программ, которые мы писали ранее. В обычной Java-программе необходимо было определить только один обязательный метод в первичном классе - метод main. В классе апплета необходимо определить как минимум два метода. Как мы увидим в главе 5, "Апплет в работе", для создания некоторых специальных эффектов (например, мультипликации) может понадобиться определить и другие методы. Вы научитесь программировать апплеты не раньше, чем хорошо усвоите основы строения первичных классов в

апплетах. Вот основные различия между первичным классом апплета и первичным классом обычной Java-программы:

- Ни один из методов в первичном классе апплета не является статическим. Из этого можно сделать вывод, что этот класс должен быть в какой-то момент явным образом реализован. Однако в примере, который мы только что видели, оператора реализации класса нет. Отсюда следует, что оболочка времени выполнения, встроенная в Web-броузер, сама реализует первичный класс апплета.
- Первичный класс является расширением класса по имени Applet. Класс Applet, определенный в пакете java.applet, включает в себя те функции, которые должен иметь каждый апплет. Поэтому с формальной точки зрения апплеты представляют собой не что иное, как подклассы класса Applet.
- Результаты работы апплета показывают, что оба включенных в первичный класс метода отработали несмотря на то, что код самого апплета не содержал явных вызовов этих методов. Это объясняется тем, что точно так же, как оболочка времени выполнения Java сама ищет и вызывает метод main в первичном классе программы, оболочка времени выполнения апплета самостоятельно вызывает методы, входящие в подкласс класса Applet.

Чтобы хорошо понимать принцип функционирования апплетов, вы должны обратить особое внимание на последнее из этих трех замечаний. В программах, которые мы писали раньше, оболочка времени выполнения вызывала метод main, который вызывал остальные методы и реализовывал алгоритм программы. В отличие от этого, когда оболочка времени выполнения броузера запускает нашу программу-апплет, она прежде всего ищет и вызывает метод init. Однако в нашем примере метод init выполняет лишь служебные действия и совсем не отвечает за работу всей программы. Как же получает управление метод paint? Оказывается, система сама вызывает метод paint всегда, когда содержимое окна требуется обновить. Например, если вы закроете окно Web-броузера другим окном, а затем снова вытащите его на передний план, система сразу же вызовет метод paint, чтобы восстановить содержимое окна.

Класс Applet содержит большое количество методов, которые вызываются в ответ на действия пользователя (например, перемещения курсора мыши в пределах окна или нажатие определенных клавиш на клавиатуре). Все эти методы подробно описываются в главе 5. Здесь мы приведем в качестве примера использование метода mouseDown, который вызывается каждый раз, когда в пределах области, занятой апплетом, происходит нажатие левой кнопки мыши. Наша программа должна перерисовывать строчку"Hello, Applet!" в той точке, где пользователь щелкнул мышью.

Пример 2-19. Апплет, управляемый мышью.

```
import java.applet.*;
import java.awt.*;
public class SecondApplet extends Applet {
        int curX=50;
        int curY=50;
        public boolean mouseDown(Event e, int x, int y) {
                curX=x;
                curY=y;
                repaint();
                return true;}
public void paint(Graphics g) {
        g.drawString("Hello, Applets!",curX,curY);}
}
```

Обратите внимание, что в методе mouseDown вызывается метод repaint. Этот метод сообщает оболочке времени выполнения, что необходимо обновить картинку на экране. В ответ на это оболочка времени выполнения передает параметры экрана, содержащиеся в объекте типа Graphics, методу paint. Внешний вид этого апплета сразу после щелчка мышью показан на рис. 2-2.

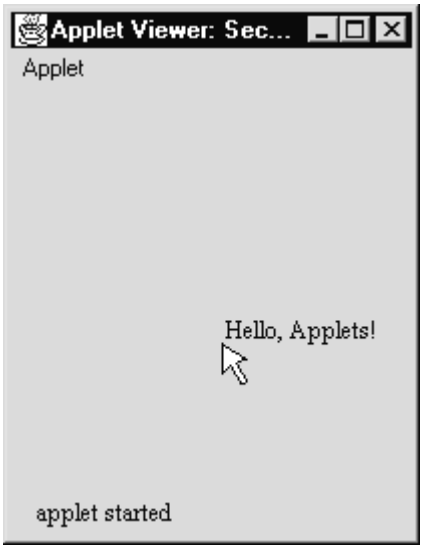

Рис. 2.2.

За исключением того факта, что оболочка времени выполнения сама вызывает требуемые методы во время работы программы, первичный класс апплета ведет себя так же, как первичные классы в тех программах, которые мы писали раньше. В этом первичном классе также можно определять новые методы (а не только переопределять методы, предопределенные в стандартном классе Applet) и реализовывать новые классы.

Интеграция апплетов в World Wide Web

До сих пор мы с вами использовали программу просмотра апплетов (appletviewer) для запуска апплетов, находящихся на нашем же компьютере. Теперь давайте обсудим, как можно сделать наши апплеты доступными всему миру через World Wide Web.

Вероятно, самый важный в этом отношении момент - доступ к Web-серверу. Вам понадобится перенести ваши .class-файлы и .html-файлы со ссылками на апплеты в то же место на сервере, где хранятся другие Web-страницы. Если вы работаете на той же машине, которая служит Web-сервером, вам, вероятно, нужно будет лишь скопировать файлы из одного каталога в другой. Если же сервер работает на другом компьютере, вам нужно будет перенести эти файлы на этот компьютер.

Вероятно, вам придется поговорить с администратором Web-сервера о том, как лучше всего установить файлы с апплетами и .html-файлы на сервере. Если ограничиться двумя примерами, которые мы только что рассмотрели (всего они занимают четыре файла), вам нужно будет лишь убедиться, что .class-файл каждого примера находится в том же каталоге, что и соответствующий .html-файл. Затем достаточно будет сообщить любому Web-броузеру адрес вашего .html-файла. Кстати, программа просмотра апплетов также может загружать апплеты из Интернет - вместо команды открытия файла можно приказать этой программе открыть URL-адрес.

Файл с расширением .class, относящийся к странице SecondApplet.html, должен находиться в том же каталоге, что и сам файл SecondApplet.html. Однако существует способ разместить .class-файл в любом другом месте дисковой системы, к которому имеет доступ Web-сервер. Для этого нужно использовать параметр CODEBASE тега <APPLET>. Ниже приведен пример .html-файла, содержащего атрибут CODEBASE.

Пример 2-20. Использование атрибута CODEBASE.

```
<APPLET CODE=ProgramPunnerApplet.class WIDTH=300 HEIGHT=150>
Ваш броузер не поддерживает язык Java. Посетите
<a href="http://www.netscape.com">Netscape</a> и скачайте Netscape 2.0
</APPLET>
```

В этом примере файл SecondApplet.class должен находиться в каталоге class_dir, входящем как подкаталог в каталог, содержащий сам .html-файл. Значением атрибута CODEBASE может быть также абсолютный путь к каталогу (то есть путь, начинающийся с корневого каталога), но при этом корневой каталог будет интерпретироваться как стартовый каталог Web-сервера, а не как корневой каталог диска. Помимо атрибута CODEBASE, существует еще несколько атрибутов, которые можно добавлять к тегу <APPLET>, изменяя внешний вид и поведение апплета. Эти атрибуты перечислены в табл. 2-7.

Таблица 2-7. Атрибуты тега <APPLET>

Атрибут	Значение	Является ли обязательным
CODE	Имя файла скомпилированного апплета (это должен быть файл с расширением .class).	Да
WIDTH	Ширина в пикселах того пространства, которое апплет будет занимать на Web-странице.	Да
HEIGHT	Высота в пикселах того пространства, которое апплет будет занимать на Web-странице.	Да
CODEBASE	Каталог на Web-сервере, где хранятся .class-файлы, на которые ссылается атрибут CODE.	Нет
ALT	Позволяет указывать альтернативный текст, который будет выведен на месте апплета в том случае, когда броузер распознает тег <APPLET>, но не поддерживает язык Java. В настоящее время не существует броузеров, которые обрабатывали бы атрибут ALT.	Нет
NAME	Позволяет задать имя для апплета. После этого другие апплеты на странице могут обращаться к этому апплету по имени и обмениваться с ним данными.	Нет
ALIGN	Позволяет выбрать режим выравнивания апплета на странице.	Нет
VSPACE	Позволяет задать величину в пикселах верхнего и нижнего полей вокруг апплета.	Нет
HSPACE	Позволяет задать величину в пикселах правого и левого полей вокруг апплета.	Нет

В дополнение к перечисленным в таблице атрибутам вы можете передавать апплету информацию с помощью специального тега <PARAM...>. Между тегами <APPLET...> и </APPLET> может находиться любое количество тегов <PARAM>. Внутри тега PARAM можно пользоваться атрибутами NAME и VALUE, создавая с их помощью пары "имя-значение". Во время своей работы апплет может получать информацию из этого источника. В главе 5 мы напишем несколько апплетов, в которых используется тег <PARAM>. Вот так выглядит .html-файл для одного из этих примеров.

Глава 3

Объектная ориентация в Java

В предыдущей главе мы касались объектно-ориентированных свойств языка Java при обсуждении переменных и классов. Объектно-ориентированное программирование (Object Oriented Programming, OOP) - настолько важная часть языка Java, что даже при написании очень простой программы на Java нам пришлось ввести некоторые понятия OOP. Прежде чем двигаться дальше, рассмотрим понятие объектной ориентации более подробно.

Начнем мы с общего объяснения того, что такое объектная ориентация и чем она улучшает Java. Мы введем также некоторые термины для описания классов, упоминавшиеся в главе 2, "Основы программирования на Java". После того как станет понятнее, что такое объектная ориентация и это понимание станет несколько более формализованным, мы сможем вернуться к вопросам, обсуждавшимся в главе 2, и рассмотреть их на новом уровне. После этого мы углубимся в специфические для Java объектно-ориентированные свойства этого языка.

Преимущества объектной ориентации

Объектная ориентация - возможно, самое популярное крылатое выражение в программировании. Как и у всех крылатых выражений, у него существует масса различных толкований. В главе 1 мы давали определение, которое принимается обычно: объектная ориентация - это подход, который упрощает решение задач. Но это определение описывает сам "продукт", который нам пытается продать тысяча напористых книг по менеджменту, видеозаписей и курсов. Давайте воспользуемся тем, что мы уже узнали из главы 2, и выработаем определение программистского уровня.

Классы - это альфа и омега объектной ориентации. Как вы помните, класс описывает тип, содержащий как подпрограммы (или методы), так и данные. Раз класс описывает тип, значит, мы можем создать переменные, содержащие и методы и переменные. Такие переменные являются объектами. Объекты отличаются от переменных в процедурных языках программирования тем, что могут определять, как можно менять данные. Подпрограммы в объектно-ориентированных языках отличаются от своих аналогов в процедурных языках тем, что они содержат наборы данных (элементы которых определены в том же классе), которые могут менять только эти подпрограммы, а другие методы не могут.

Это определение описывает реализацию подхода объектной ориентации с точки зрения программиста. Теперь мы можем приступить к обсуждению вопроса о том, что это дает программисту. Прежде чем двигаться дальше, посмотрим табл. 3-1, в которой приводится краткий перечень некоторых терминов Java, относящихся к объектной ориентации.

Таблица 3-1. Ключевые термины Java, относящиеся к объектной ориентации

Термин	Определение
Пакет	Набор структурных единиц языка Java, в том числе классов.
Класс	Тип данных, содержащий данные и подпрограммы.
Метод	Реализация подпрограммы на Java.
Конструкция	Создание класса в переменной; появляется в процессе выполнения программы.
Экземпляр, объект, реализация	Переменная типа класса, которая была создана.
Модификатор	Описывает, какие классы имеют доступ к элементу класса. Модификатор

доступа	доступа нужен также для указания того, что доступ к классу возможен извне пакета.

Модификатор доступа static

Вы, возможно, заметили, что табл. 3-1 не содержит ключевого слова static, описанного в главе 2, "Основы программирования на Java". Хотя объектная ориентация обладает множеством преимуществ, бывает, что простую подпрограмму не нужно жестко привязывать к конкретному набору данных. Это аналогично тому, что вы можете захотеть определить переменную, которая всегда будет иметь одно и то же значение, и нет никакого смысла возиться с реализацией объекта только для того, чтобы получить это значение. В таких случаях, когда объектная ориентация не дает преимуществ, используется модификатор static. Поскольку эти случаи не вписываются в объектно-ориентированный подход, мы в этой главе не рассматриваем статические методы и переменные.

Затенение данных

Помните модификатор доступа private, которым мы пользовались в главе 2? Переопределяя переменную с модификатором private, мы затеняем данные от всех подпрограмм в нашей программе, кроме подпрограмм, определенных в том же классе. Когда же имеет смысл затенять данные? Рассмотрим парадокс, часто возникающий при процедурном программировании.

Если вы когда-нибудь учились на курсах по основам программирования, в какой-то момент ваш учитель, возможно, советовал вам не делать переменную глобальной (то есть доступной для всех подпрограмм вашей программы). Если переменная является глобальной и из-за нее возникают ошибки, очень трудно проследить, в какой из подпрограмм эти ошибки возникли. Кто-нибудь, кто будет потом сопровождать вашу программу - будь она с ошибками или без, - будет долго мучиться, пытаясь понять, что происходит с этой переменной.

Это вполне справедливо. Но что если вы пишете программу, в которой, допустим, восемь подпрограмм, и в четырех из них используется одна и та же переменная? Если соблюдать запрет на глобальные переменные, вы должны пропустить эту переменную через четыре использующих ее метода. Но на самом деле это просто обходной путь вместо того, чтобы сделать переменную глобальной. В действительности все, что вам нужно, - это сделать переменную глобальной для тех четырех подпрограмм, в которых она используется.

В Java мы просто помещаем переменную в класс, переопределяем ее с модификатором private и добавляем к ней эти четыре метода. Если кто-то еще посмотрит текст нашей программы или если мы сами посмотрим его, когда уже давно забудем, как работает программа, мы поймем, что эти четыре метода - единственные, которым разрешено работать с этой переменной.

Инкапсуляция

В процессе затенения данных мы косвенно описали взаимоотношения между переменной и методами, содержащимися в том же классе. Если методу, не включенному в класс, требуется изменить эту переменную, он должен вызвать один из методов, определенных в классе. Такие взаимоотношения между элементами класса входят в понятие, называемое инкапсуляцией (encapsulation). Это понятие очень близко к затенению данных.

Рассмотрим значение инкапсуляции, когда мы имеем дело с одной переменной. Допустим, что это переменная целого типа и одна из наших подпрограмм печатает пустые строки в количестве, равном значению этой переменной. Если бы мы писали программу на процедурном языке, нам пришлось бы делать проверку, не является ли эта переменная отрицательной. Если мы инкапсулируем подпрограмму и переменную в класс, нам не нужно проверять, отрицательная ли переменная. Поскольку изменить значение переменной могут только методы, входящие в класс, мы просто напишем все наши четыре метода так, чтобы ни один из них не присвоил нашей целой переменной отрицательного значения:

```
public class printLines {
private int linesToPrint=0;
public void printSomeLines() {
        for (int i=0;i<<=linesToPrint;i++) {
        System.out.println("");}
        }
public void setLinesToPrint(int j) {
        if (j>>0) {
        linesToPrint=j;}
```

```
}
```

Поскольку переменная может измениться только в методе setLineToPrint, нам нужно проверить ее на отрицательность только в этом методе. Таким образом, нам не придется писать несколько лишних строк в программе. Если в нашем классе содержится несколько переменных, преимущества такого подхода станут еще очевиднее.

Чтобы понять причину этого, обратимся снова к режиму работы процедурного языка. Мы уже описали пример с одной переменной, которая используется в нескольких подпрограммах. Расширим эту ситуацию и предположим, что переменной является массив и вам нужно проследить за какой-то позицией массива. В этом случае при каждом обращении к этому массиву необходимо также обращаться к индексной переменной (то есть к совсем другому массиву и его индексу), что усложняет программу.

В результате мы получаем запутанный набор данных - каждый элемент этого набора связан со всеми остальными элементами. Это значит, что в процедурном языке каждая подпрограмма, использующая какой-то элемент этого набора данных, отвечает за сохранение связи этого элемента с другими. Поскольку каждая подпрограмма просто оперирует с данными, понимать, как выдерживаются эти связи, очень непросто. Если в каком-то месте связь порвалась, будет очень трудно определить, при каком именно действии это случилось, - совсем как с отношениями между людьми!

Поскольку наша личная жизнь выходит далеко за границы данного текста, сконцентрируемся на вопросе о том, каким образом объектная ориентация помогает нам сохранять связи между элементами программы. Рассмотрим простой пример. Допустим, у нас есть массив символов. Наша задача - начать с какой-то позиции в массиве и поменять символ, который стоит на этом месте, с каким-то другим символом. В следующий раз, когда нужно сделать перестановку, мы начнем со следующей позиции массива. Это выполняется таким классом:

```
public class replaceChars {
        private char myArray[];
        private int curPos=0;
        public reolaceChars(char someArray[]) {
                myArray=someArray;}
public boolean replaceNetChar(char c, char d) {
        if (newPositionSet(c)) {
        myArray[curPos]=d;
                return true;}
        else {return false;}
 }
private boolean newPositionSet(chae c) {
        int i=curPos;
        while (I<<myArray.length) {
                if (c==myArray[i]) {
                        curPos=i;
                        return true;}
                else {i++;}
                }
        return false;
        }
public boolean atEnd() {
        return(curPos==myArray.length-1);}
        // вычитаем 1, потому что позиции в массиве начинаются с нуля
        }
```

Написав несколько строк программы, мы сможем полностью решить нашу задачу. Заметим также, что новый метод PositionSet задан с модификатором private. Это снова возвращает нас к понятию затенения данных. Вместо того чтобы затенять данные, мы затеняем метод, который меняет данные. Возможно, мы не хотим менять позицию элемента во всех случаях, кроме тех, когда производится замена символа.

Теперь рассмотрим трудности, возникающие при решении нашей задачи в процедурном языке. Во-первых, мы не можем затенить наши данные или любую из участвующих подпрограмм. Это означает, что мы должны постоянно делать проверку того, что наша переменная curPos не вышла за границы значений. Во-вторых, у нас нет возможности следить непосредственно за текущей позицией в массиве.

Приведенная выше простая задача разрешима и средствами процедурных языков. Однако наш класс имеет то преимущество, что он является на самом деле определением типа. Когда мы

инкапсулируем наши методы и переменные в класс, мы фактически инкапсулируем наше решение. Раз решение - это тип, его легко снова использовать, просто реализовав другую переменную

```
replaceChars solution1=new replaceChars("Java - это здорово!")
replaceChars solution2=new replaceChars("Я хочу больше узнать о Java!");
while (!solution1.atEnd())
        {solution1.replaceNextChar('a','x');}
while (!solution2.atEnd())
        {solution2.replaceNextChar('o','y');}
```

Поскольку мы определяем методы, которые правильно взаимодействуют с набором данных, нам не нужно беспокоиться о деталях нашей программы. Если бы мы пытались решить нашу простую задачу средствами процедурного языка, нам все время приходилось бы сохранять связь между номером позиции и массивом. Если бы мы захотели произвести такую операцию над несколькими массивами, все усложнилось бы в несколько раз. Объектно-ориентированный подход позволяет использовать написанную программу на более высоком уровне абстракции.

Разумеется, абстракция - это не новшество при работе с ООР. Процедурные языки определяют последовательность действий в подпрограммах, и, таким образом, эти подпрограммы можно снова использовать. Кроме того, простые типы данных, например целочисленные переменные, суть абстрактное выражение того, как биты хранятся в памяти компьютера. Объектная ориентация просто поднимает эту абстракцию на новый уровень, связывая воедино абстракции типов данных и подпрограмм, в результате чего отношения между данными и действиями над ними могут быть повторно использованы.

Повторное использование через наследование

Когда мы решаем задачу инкапсуляцией методов и переменных, мы легко можем снова и снова применять это решение в своих программах. Но что если мы столкнулись с новой задачей, очень похожей на ту, которую мы уже решили? Объектная ориентация содержит особое средство - наследование (inheritance), - позволяющее использовать уже написанные программы для решения новых задач, сходных со старыми.

Рассмотрим процедуру наследования в действии. Во фрагменте кода, приведенном в предыдущем разделе, мы снова и снова вызывали replaceNextChar для двух одинаковых символов. Разве не удобнее было бы сделать это, используя какой-нибудь метод в классе replaceChar? Мы бы просто добавили этот метод в класс и снова откомпилировали программу. Но предположим, что кто-то еще использует начальный класс replaceChar. Тогда нам нужно поддерживать два класса с тем же именем, что может привести к путанице. Вместо этого мы можем создать новый класс, который унаследует характеристики нашего класса replaceNextChar:

```
class betterReplaceNextChar extends ReplaceNetChar {
        public int replaceAllChar(char c, char d) {
        int i=0;
        while(!atEnd()) {
                replaceNetChar(c,d);
                i++;}
        return i;}
}
```

Теперь мы получили новый класс, содержащий все методы класса ReplaceNext-Char плюс один дополнительный метод, который мы только что определили. Мы сумели инкапсулировать решение в новую задачу, расширив класс. Как мы видели при написании нашего первого апплета в главе 2, наследование - очень важное понятие в программировании на Java. Немножко дальше в этой главе мы рассмотрим его подробнее.

Возможности обслуживания и сопровождения

Мы уже неоднократно говорили о том, что объектно-орентированную программу легче сопровождать. Но что конкретно имеется в виду под сопровождением программы? В конце концов, после того как программа откомпилирована, она, по-видимому, должна работать вечно - а не как некий механизм, в котором рано или поздно начнет ощущаться усталость металла. Однако программное обеспечение тоже нуждается в подгонке под свою среду, хоть и иначе, чем физические конструкции.

Например, программа, изначально предназначавшаяся для того, чтобы следить за потребностями служащих предприятия, должна быть модернизирована с учетом заботы о здоровье людей. Или сетевая система, изначально созданная для того, чтобы просто передавать сообщения на соседние машины, теперь нуждается в том, чтобы ею можно было управлять с сервера, находящегося на Уолл-стрит. Можно было бы привести очень длинный список примеров, но наш главный тезис заключается в том, что программное обеспечение живет в сложном и бесконечно меняющемся мире. Возникают проблемы, которые программист, решавший начальную задачу, не предвидел, или на систему накладываются новые требования. Редко бывает, чтобы производственная компьютерная программа не изменялась в течение нескольких лет. Когда программа меняется, бывает, что изменения вносит новый человек или сам программист уже давно забыл хитросплетения собственной программы. В любом случае, кто бы ни вносил изменения, он предпочтет не начинать с разбора черновиков. Современные языки программирования должны давать возможность людям, занимающимся сопровождением программы, легко модифицировать программу для удовлетворения возникших новых потребностей.

Это основная цель объектно-ориентированных языков, и все вышеперечисленные свойства так или иначе преследуют ее. Например, возможность повторного использования явно означает удобство сопровождения программы. Если можно повторно использовать уже написанную программу для решения новых задач, значит, легче будет нарастить программу для того, чтобы она работала в изменившихся обстоятельствах. Кроме того, саму программу в этом случае легче понимать. Когда мы пользуемся затенением данных, определяя переменную внутри какого-то класса с модификатором private, любой свободно владеющий языком Java программист поймет, что только методы из этого класса могут влиять на данную переменную. Это подобно тому, как инкапсуляция методов и переменных облегчает изучение отношений между данными и действиями над ними.

Инкапсуляция, кроме того, упрощает добавление к программе новых свойств. Если класс работает так, как должен, то программисту, пытающемуся добавить новые свойства, не придется разбираться в основных деталях программы. Все, что ему нужно будет знать, - это как использовать общие методы и конструкторы.

Инкапсуляция имеет еще одно очевидное преимущество. Поскольку другие объекты в программе могут взаимодействовать с данным объектом только через общие методы и конструкторы, можно менять частные части системы и текст, создающий общие методы и конструкторы, не нарушая систему в целом.

Почему это является преимуществом? Рассмотрим задачу о 2000 годе. При наступлении нового тысячелетия многие хорошие программисты рассчитывают заработать по 500<|>$ в час, проверяя ошибки, допущенные компьютерами в разных организациях при распознавании смены тысячелетия. Почему? Существует огромное множество очень важных программ, которые не смогут правильно интерпретировать наступление нового тысячелетия, потому что они используют только две цифры для задания года. Это означает, что ваш банк может начать считать, что вам -73 года, или телефонный разговор между восточным и западным побережьем, начавшийся 31 декабря в 23:59, будет считаться продолжавшимся в течение 99 лет!

Это трудно исправить, потому что все эти программы были созданы до изобретения объектной ориентации. Они написаны на процедурных языках, и каждая из них использует собственный способ сравнения двух дат. Наши программисты высокого полета собираются корпеть над тоннами индивидуальных подпрограмм, выискивая места, в которых даты сравниваются неправильно. Давайте рассмотрим, как объектно-ориентированный подход устранил бы эту проблему. Ниже приводится класс Year (год), в котором мы специально сделали неправильное сравнение. (Чтобы наш пример не выставил нас полными идиотами, давайте считать, что нашим намерением было использовать как можно меньше места для хранения года.)

```
public class Year {
        private byte decadeDigit;
        private byte yearDigit;
        public Year(int thisYear) {
                byte yearsSince1900=(byte)thisYear-1900;
                decadeDigit=yearsSince1900/10;
                yearDigit=yearsSince1900-(decadeDigit*10);}
        public int getYear() {
                return decadeDigit*yearDigit;}
// другие методы
}
```

Теперь мы создаем десятки систем, которые доверяют этому классу хранить номер года, и, кроме того, этот класс используют другие программисты. Затем в один прекрасный день в декабре 1999 года мы понимаем, какую глупость мы совершили. Что делать - вызывать

консультанта за 500 $ в час? Конечно, нет! Все, что нам нужно, - это переписать заново реализацию класса. Если мы не будем менять описания общих методов, все в этих системах будет работать правильно:

```
public class Year {
        private byte centuryDigit;
        private byte decadeDigit;
        private byte yearDigit;
        public Year(int thisYear) {
                centuryDigit=(byte)thisYear/100;
                int lastTwo=thisYear-(centuryDigit*100);
                decadeDigit=(byte)lastTwo/10;
                yearDigit=(byte)(lastTwo-(decadeDigit*10)); }
        public int getYear() {
                return decadeDigit*yeaaDigit*centuryDigit;}
// другие методы
}
```

Теперь мы можем жить спокойно до 12799 года, и никому не придется нанимать человека для выполнения нудной работы по исправлению нашей программы!

СОВЕТ Java API, обсуждавшееся в главе 6, содержит класс Date, который не пострадает при смене тысячелетия.

Особенности объектов Java

Мы рассмотрели понятия, лежащие в основе некоторых частей программы, которые мы писали в главе 2, и надеемся, что убедили вас в том, что эти понятия отражают преимущества языка Java.

Иерархия классов Java

Используя термин "иерархия классов", мы описываем то, что происходит при наследовании. Допустим, у нас есть три класса: Mom, Son и Daughter (мама, сын и дочь). Классы Son и Daughter наследуют от Mom. Наша программа будет иметь следующий вид:

```
class Mom {
// описания, определения
}
class Son extends Mom {
// описания, определения
}
class Daughter extends Mom {
// описания, определения
}
```

Итак, мы создали иерархию классов. Точно так же, как и организационную иерархию, ее легко представить визуально.

В табл. 3-2 приведены некоторые термины, необходимые для описания нашей иерархии. Mom - это базовый класс, то есть класс, на котором базируются другие классы. Son и Daughter - это подклассы класса Mom, а Mom является суперклассом для Son и Daughter.

Таблица 3-2. Термины, связанные с иерархией классов

Термин	Определение
Иерархия классов	Группа классов, связанных наследованием.
Суперкласс	Класс, расширяемый неким другим классом.
Подкласс	Класс, расширяющий некий другой класс.

Базовый класс Класс в иерархии, являющийся суперклассом для всех остальных классов в этой иерархии.

Теперь, когда у нас уже есть некоторый словарь необходимых для работы понятий, мы можем поговорить конкретно об иерархии классов в Java. Во-первых, все классы в Java имеют ровно один непосредственный суперкласс. Как обсуждалось в главе 1, эта характеристика Java на языке объектной ориентации известна как единичное наследование. Разумеется, у класса может быть и больше одного суперкласса. Например, и Mom и Daughter являются суперклассами другого класса Granddaughter (внучка).

"Минуточку, - возможно, скажете вы, - но если все классы имеют в точности один непосредственный суперкласс, то каков же суперкласс класса Mom?" Дело в том, что иерархия классов, которую мы здесь описали, на самом деле является подмножеством другой огромной иерархии классов, которая содержит каждый единичный класс, когда-либо написанный на Java. На рис. 3-1 показано, как созданная нами маленькая иерархия классов встраивается в гораздо большую иерархию. На вершине этого класса находится специальный класс, называемый классом Object.

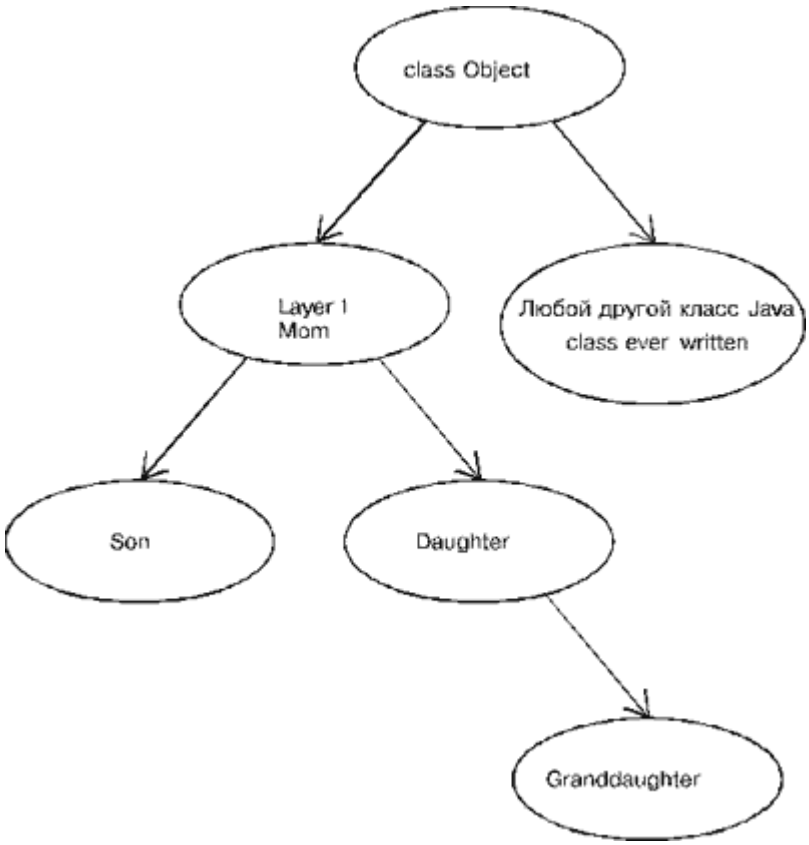

Рис. 3.1.

В том случае, когда мы объявляем класс, не указывая явно, расширением какого класса он является, компилятор Java подразумевает, то наш класс является расширением класса Object. Поэтому следующее объявление нашего класса Mom эквивалентно объявлению, данному выше:

```
class Mom extends Object {
// определения и объявления
```

Так чем же хороша эта всеобъемлющая иерархия классов? Поскольку все классы наследуют из класса Object, мы знаем, что всегда можно воспользоваться его методами. В состав методов класса Object входят, например, методы для установления равенства и методы, предназначенные для поддержки многопотоковости. Кроме того, нам не нужно заботиться об объединении нескольких различных иерархий объектов между собой, поскольку мы знаем, что все они - подмножества одной и той же глобальной иерархии. Наконец, иерархия классов гарантирует, что у каждого класса есть суперкласс, а это очень важно, как мы увидим в главе 6, когда будем рассматривать специальные классы-контейнеры.

Что входит в глобальную иерархию классов

Как мы уже говорили, любой класс в языке Java принадлежит одной и той же глобальной иерархии. Более того, чрезвычайно важная иерархия классов Java входит в состав JDK. Она называется "интерфейс прикладного программирования", или Java API. В главе 6 мы познакомимся с API подробнее.

Специальные переменные

Каждый класс Java содержит три заранее определенные переменные, которыми можно пользоваться: null, this и super. Первые две относятся к типу Object. Коротко говоря, null представляет собой несуществующий объект, а this указывает на тот же самый экземпляр. Переменная super разрешает доступ к методам, определенным в суперклассе. Ниже мы рассмотрим каждую из этих переменных.

Переменная null

Как вы помните из главы 2, "Основы программирования на Java", прежде чем использовать какой-то класс, его нужно реализовать. До этого класс имеет значение переменной null, и мы говорим, что объект равен нулю. Если объект равен нулю, доступ к его элементам не разрешен, потому что не был создан объект, с которым могли бы ассоциироваться эти элементы. Если мы попытаемся обратиться к элементам до того, как они были созданы, мы рискуем вызвать исключение NullPointerException, что остановит выполнение программы. Приведенный ниже метод действует довольно рискованно, потому что он принимает ReplaceNextChar в качестве параметра и использует его, не проверив на равенство нулю:

```
public void someMethod(ReplaceChars A) {
A.replaceNextChar('a','b');}
```

Следующая программа, вызывающая someMethod, приведет к NullPointerException, потому что не был создан ReplaceNextChar:

```
ReplaceChars B;
someMethod(B);
```

Чтобы уберечь программу от сбоев, необходимо перед использованием объектов проверять их на равенство нулю. Переписанный заново someMethod выполняет такую проверку, чтобы убедиться, что A не равно нулю, прежде чем обращаться к его элементам:

```
public void someMethod(replaceChars A) {
        if (A==null) {
                System.out.println("A пусто !!!");}
        else {
                A.replaceNextChar('a','b');
}
```

Переменная this

Иногда бывает необходимо передать другой подпрограмме ссылку на текущий объект. Это можно сделать, просто передав переменную this. Скажем, наши классы Son и Daughter определяют конструктор, который заключает переменную Mom в свой конструктор. Переменная this позволяет классам Son и Daughter следить за классом Mom, сохраняя ссылку на него в переменной, определенной с модификатором private:

```
public class Son {
        Mom myMommy;
public Son(Mom mommy) {
        myMommy=mommy;}
// методы
}
public class Daughter {
        myMommy=mommy;}
```

```
public Daughter(Mom Mommy) {
        myMommy=mommy;}
```

Когда класс Mom создает свои подклассы Son и Daughter, ему нужно передать своим конструкторам ссылку на себя. Mom делает это, используя переменную this:

```
public class Mom {
        Son firstSon;
        Son secondSon;
        Daughter fistDaughter;
        Daughter secondDaughter;
public Mom() {
        firstSon=newSon(this);
        secondSon=newSon(this);
        fistDaughter=newDaudther(this);
        secondDaughter=newDaudther(this);}
// другие методы
}
```

Для Mom, сконструированного таким образом:
Mom BigMama=new Mom();
рис. 3-2 представляет все взаимоотношения нашей семьи:

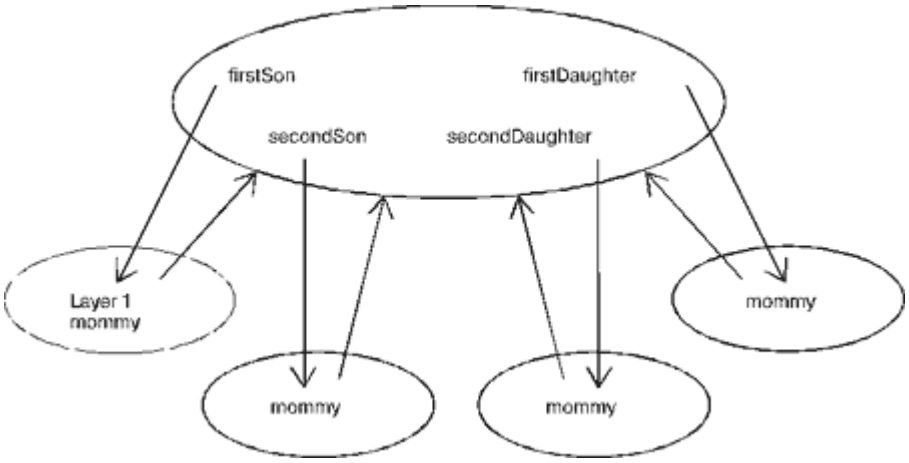

Рис. 3.2.

Переменная super

Вам часто придется обращаться к родительскому экземпляру метода. Допустим, вы реализовали конструктор, определенный в вашем классе-родителе. Возможно, вы решили присвоить начальные значения нескольким переменным, определенным в новом классе с модификатором private, а теперь хотите вызвать родительский конструктор. Именно здесь вам будет полезно воспользоваться переменной super. В следующем примере мы определим класс, который замещает свой родительский конструктор, а затем вызывает его, используя переменную super.

Обратимся снова к нашей иерархии Mom, Son и Daughter. Пусть Mom определяет метод мытья комнаты, называемый cleanUpRoom. Предполагается, что Son моет комнату в точности так, как определила Mom, после чего он должен сказать: "Моя комната вымыта!" Поскольку Mom определила метод мытья комнаты, Son может вызвать этот метод, используя переменную super, после чего выполнить дополнительное действие по выводу сообщения:

```
public class Mom {
// переменные, конструкторы
public void cleanUpRoom() {
// код для мытья комнаты
}
// другие методы
}
public class Son {
```

```
// переменные, конструкторы
public void cleanUpRoom() {
        super.cleanUpRoom();
        System.out.println("Моя комната вымыта!");}
// другие методы
}
```

СОВЕТ Внимание! Не следует считать, что переменная super указывает на совершенно отдельный объект. Чтобы ее использовать, не нужно реализовывать суперкласс. На самом деле это просто способ выполнения методов и конструкторов, определенных в суперклассе.

Конструкторы, так же как и методы, тоже могут использовать переменную super, как видно из следующего примера:

```
public class SuperClass {
        private int onlyInt;
        public SuperClass(int i) {
        onlyInt=i;}
public int getOnlyInt() {
        return onlyInt;}
}
```

Воспользовавшись переменной super, наш подкласс может повторно использовать программу, написанную для конструктора:

```
public class SubClass extends SuperClass {
private int anotherInt;
public SubClass(int i, int j) {
        super(i);
        anotherInt=j;}
public int getAnotherInt() {
        return anotherInt;}
}
```

На использование переменной super при обращении к конструкторам суперкласса накладываются два важных ограничения. Во-первых, переменную super можно использовать таким образом только внутри конструктора. И во-вторых, это выражение должно быть первым в конструкторе.

Реализация классов

Начиная с главы 2 мы занимались реализацией классов. Используя оператор new, мы оживляем наш класс как объект и присваиваем его переменной. Теперь рассмотрим некоторые спорные вопросы, касающиеся реализации, которые мы еще не обсуждали.

При первом описании реализации классов мы использовали задаваемый по умолчанию конструктор:

someClass A=new someClass();

Затем мы показали, что можно передать переменные конструктору. При этом мы воспользовались преимуществами совмещения конструкторов (constructor overloading), при котором класс определяет множество конструкторов с различными списками параметров. Поскольку конструкторы на самом деле являются просто методами специального типа, совмещение конструкторов работает так же, как совмещение методов, описанное в главе 2. Определенный ниже класс использует совмещение конструкторов:

```
public class Box {
        int boxWidth;
        int boxLength;
        int boxHeight;
public Box(int i) {
        boxWidth=i;
        boxLength=i;
```

```
        boxHeight=i;}
public Box(int i, int j) {
        boxWidth=i;
        boxLength=i;
        boxHeight=j;}
public Box(int i, int j, int k) {
        boxWidth=i;
        boxLength=j;
        boxHeight=k;}
// другие методы
}
```

В приведенном выше фрагменте кода определен класс, описывающий некий параллелепипед. Если передан только один параметр для конструктора, мы считаем, что параллелепипед является кубом. Если передано два параметра, считаем, что в основании параллелепипеда лежит квадрат, а второе целое число определяет его высоту. Если передаются три параметра, все они используются для описания параллелепипеда. Совмещение конструкторов позволяет нам предоставлять тем, кто будет использовать наш класс, различные способы создания класса.

Когда мы создаем переменную и не присваиваем ей определенного начального значения, Java присваивает значение за нас. Приведенным выше переменным были присвоены нулевые начальные значения. Вообще говоря, желательно убеждаться в том, что каждый конструктор присваивает значение каждой переменной в классе. Инкапсуляция данных зависит от того, являются ли эти данные допустимыми, и плохо работает, если переменным не присвоены начальные значения.

Но здесь возникает интересная ситуация - мы не можем ожидать, чтобы конструкторы, определенные в суперклассе, правильно присваивали начальные значения переменным, определенным в подклассе. Для разрешения этой проблемы набор правил Java для наследования конструктора отличается от правил наследования метода и наследования переменной. Если вы определяете какой бы то ни было конструктор, Java игнорирует все конструкторы в суперклассе.

Что происходит, когда объект больше не нужен

Как уже говорилось в главе 2, Java - язык, ориентированный на сборку мусора. Поскольку Java следит за тем, когда нужно освобождать память, необходимость в деконструкторе не очень велика. Тем не менее в Java есть метод, называемый finalize, который применяется, когда сборщик мусора перераспределяет память. Определяя этот метод, вы можете описать последовательность действий, которые должны быть выполнены, когда сборщик мусора выясняет, что данная переменная больше не будет использоваться.

Будьте осторожны при использовании метода finalize - он может ни разу не вызываться до окончания работы программы, и вы не сможете предсказать, в каком порядке будут восстановлены объекты, которые больше не используются.

Правила доступа

Когда мы обсуждали преимущества затенения данных, мы определили модификатор доступа private. Модификатор private разрешает доступ к переменной или методу из класса, в то время как модификатор public делает элемент доступным отовсюду. Существуют еще три других модификатора доступа, воздействующих на объектно-ориентированную природу элементов класса: protected, private protected и final. Мы перечислили их в том порядке, в котором они понятнее читателю - модификатор protected по своему действию ближе всего к модификаторам private и public, которые мы уже использовали, а модификатор final наиболее далек от всего, к чему мы привыкли. Итак, начнем с рассмотрения модификатора protected.

Модификатор доступа protected

Модификатор protected позволяет сделать элементы класса общими только для определенного набора классов - тех, что содержатся в том же пакете. Мы помещаем класс в пакет со следующим оператором в верхней строке файла:

```
package somePackage;
```

Если мы явно не поместили класс в специальный пакет, он помещается в пакет, заданный по умолчанию, а все классы определяются в текущем каталоге.

СОВЕТ Если вы явно не определили метод или переменную, компилятор считает, что вы хотите, чтобы они были определены с модификатором protected. Тем не менее, если потом вы решите поместить их в собственный пакет, элементы, которые до тех пор были доступны из классов того же рабочего каталога, больше не будут доступны. В таком случае всегда лучше явно определить элементы класса.

Модификатор доступа private protected

Модификатор private protected предоставляет меньше доступа, чем модификатор protected, но больше, чем модификатор private. Элемент, определенный с модификатором private protected, доступен только из подклассов некоего класса. Если другие модификаторы, которые мы использовали, соответствуют концепции затенения данных, то модификатор private protected наиболее важен при рассмотрении наследования классов.

Допустим, в некоем классе мы определяем переменную или метод с модификатором private. Если мы создаем подкласс в этом классе, этот подкласс не может обращаться к элементам, определенным с модификатором private, если суперкласс не входит в подкласс. Как будет объяснено в следующем разделе, часто бывает удобно разработать базовый класс, задача которого - просто существовать и содержать в себе несколько подклассов. В этом случае гораздо удобнее использовать не модификатор private, а модификатор private protected, чтобы подклассам не приходилось выполнять всю реальную работу через методы суперкласса, определенные с модификатором public.

Как работает наследование

Мы бегло ознакомились с тем, что скрывается под словами "объектная ориентация Java". Надеемся, что теперь вы уверенно владеете двумя ключевыми понятиями объектной ориентации - затенением данных и инкапсуляцией, и знаете, как использовать их в Java. Мы познакомили вас также с понятием наследования. Давайте теперь более глубоко рассмотрим механизм наследования. В данном разделе будет показано, как можно улучшить качество программирования за счет использования наследования при формировании иерархии классов. В Java существуют абстрактные классы и методы, которые помогут нам в структурировании иерархий классов.

Структурирование иерархий классов

В начале этой главы, при обсуждении возможности повторного использования, было показано, что наследование позволяет надстраивать уже написанные классы. Но наш пример продемонстрировал только одну часть повторного использования программы. Повторное использование - лишь одно из преимуществ наследования. С помощью наследования мы можем гораздо разумнее расположить ключевые модули нашей программы.

Обратимся к примеру, использованному нами в главе 1, "World Wide Web и Java", при объяснении того, что такое объектная ориентация. Как вы, возможно, помните, мы привели простую задачу "пойди в магазин и купи молока" и показали, как сформулировать ее в терминах объектной ориентации. Рассмотрим один из компонентов этой задачи, а именно пакет молока.

Допустим, что мы программируем целую систему. Мы могли бы написать класс, описывающий молоко. Но существует несколько различных видов молока, например обезжиренное и молоко с добавлением шоколада. И даже если бы все виды молока объединились в одну единицу, эта единица была бы только одной в множестве молочных продуктов. Таким образом, мы могли бы создать иерархию классов, подобную той, что приведена на рис. 3-3.

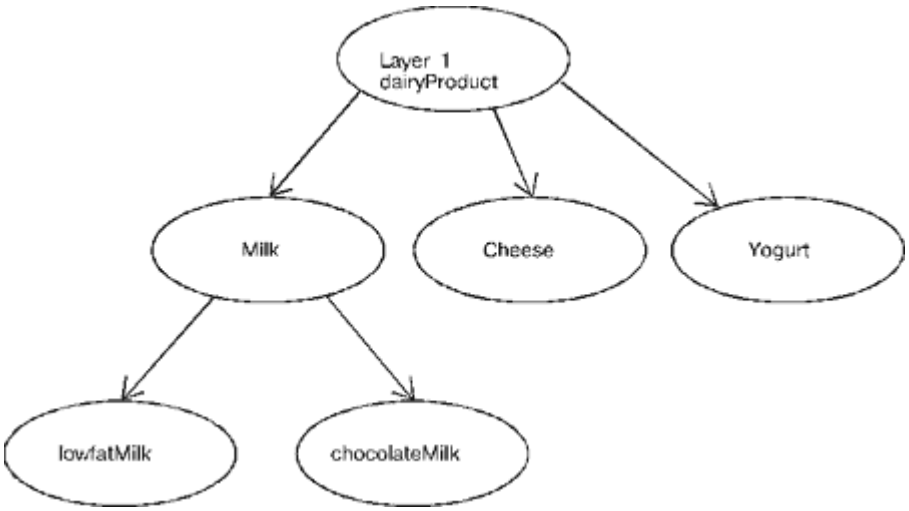

Рис. 3.3.

К счастью, это не просто упражнение в логическом мышлении. Java позволяет реализовать подкласс, а затем привести его к такому типу, чтобы он действовал как переменная суперкласса. Это очень ценно, если нас волнует только один общий аспект, определенный в верхней части иерархии классов, - например, скиснет ли наш молочный продукт на этой неделе. Допустим, наш класс dairyProduct (молочный продукт) содержит следующий метод:

```
public class dairyProduct {
// переменные, конструкторы
        public boolean sourThisWeek() {
                // соответствующий текст программы
        }
// другие методы
public void putOn Sale() {
        // программа покупки молочного продукта в магазине
}
```

Вот тут-то и возникает необходимость приведения (casting). Если у нас уже есть переменная - например, lowfatMilk (обезжиренное молоко), - мы можем привести ее к переменной типа dairyProduct:

```
lowfatMilk M=new lowfatMilk();
dairyProduct D=M;
if (D.sourThisWeek()) {
        System.out.println("Не покупайте");}
```

В чем преимущество такого подхода? Скажем, директор магазина хочет выяснить, какие пакеты с молоком скиснут на этой неделе. Те, что могут скиснуть, будут пущены в продажу. Директор должен просто перенести все объекты lowfatMilk, Milk, Cheese и Yogurt в следующий метод:

```
public void dumpSourGoods(dairyGood d) {
        if (d.sourThisWeek()) {
                d.putOnSale();}
}
```

Если бы для начала мы не построили иерархию классов, нам пришлось бы писать свой метод для каждого вида молочного продукта.

Абстрактные классы и методы

В предыдущем примере мы создали иерархию классов. Но наш класс dairyProduct содержит методы, у которых нет тела. Когда мы его написали, мы просто упомянули, что эти методы будут переопределены в подклассе. Однако, просмотрев нашу программу, трудно понять, что мы намереваемся это сделать. Для оказания помощи в этой ситуации в Java существует модификатор

abstract.

При использовании модификатора abstract с методами все подклассы должны переопределить абстрактный метод. Вот как можно сделать абстрактные методы для нашего класса dairyProduct

```
public class dairyProduct {
// переменные, конструкторы
public abstract boolean sourThisWeek();
// другие методы
public abstract void putOnSale();
}
```

Класс dairyProduct по-прежнему может быть реализован - просто теперь нельзя обратиться к абстрактным методам через объект класса dairyProduct. Однако мы можем также воспользоваться модификатором abstract для того, чтобы показать, что мы не хотим, чтобы объект был создан непосредственно:

```
public abstract myAbstractClass {
// программа
}
```

Когда мы определяем класс как абстрактный, мы можем объединить в нем методы и переменные. Когда мы создаем в классе подкласс, он наследует все элементы абстрактного класса по тем же правилам наследования, что мы уже описали.

Полиморфизм и интерфейсы Java

Объясняя преимущества объектной ориентации Java, мы столкнулись с понятиями, которые легко можно объяснить на основе того, что мы уже знаем из главы 2, "Основы программирования на Java". Теперь введем понятие полиморфизма и остановимся на том, как структурные механизмы Java - интерфейсы - позволяют включать полиморфизм в программу.

Полиморфизм - это процесс, с помощью которого мы можем вызвать один и тот же метод на группе объектов, причем каждый объект будет реагировать на вызов метода по-своему. В нашем примере с dairyGoods мы уже имели дело с полиморфизмом. Например, методы putOnSale и sourThisWeek определены во всех классах иерархии. Мы можем вызывать эти методы на всех объектах - что мы и делали, когда выставили на продажу все продукты, которые скоро могут скиснуть, - и каждый класс определяет, как будут в действительности реагировать его реализации.

Однако полиморфизм - понятие в какой-то степени ограниченное. Мы можем быть уверены только в том, что классы одного экземпляра будут содержать все методы, определенные в суперклассе. Но часто бывает, что некоторые подклассы должны содержать методы, не содержащиеся во всей иерархии. Например, поскольку молоко и йогурт являются жидкими продуктами, нам могут потребоваться методы cleanUpSpill (мытье пролитого) на случай, если пакет упадет. Но глупо было бы определять для класса Cheese метод мытья пролитого сыра. Кроме того, в магазине могут пролиться и различные немолочные продукты.

Хорошо структурированная иерархия классов не решает эту задачу. Даже если у нас есть класс storeGood (хранение продукта), расположенный над всеми классами, определяющими продукты в нашем магазине, не имеет смысла определять метод cleanUpSpill в верхней части, потому что многие продукты в магазине не могут пролиться. Что нам нужно (и это есть в Java), так это способ определения набора методов, реализуемых некоторыми, но не всеми классами иерархии. Такая структура называется интерфейсом.

Начнем мы исследование интерфейсов с определения интерфейса для наших жидких продуктов, которые могут пролиться:

```
interface spillable {
        public void cleanUpSpill();
        public boolean hasBeenSpilled();
}
```

Как вы можете видеть, данные методы определяются аналогично тому, как мы определяли абстрактные методы. Разумеется, эти методы абстрактные - они должны быть определены внутри класса, реализующего интерфейс. Заметим, кроме того, что у нас нет ни переменных, ни конструкторов. Те и другие не разрешается использовать в интерфейсе, потому что интерфейс -

это всего лишь набор абстрактных методов. Вот пример использования интерфейса для нашего класса Milk:

```
public class Milk extends dairyProduct
        implements Spillable {
        // переменные, конструкторы
public boolean hasBeenSpilled {
        // соответствующая программа
        }
public void cleanUpSpill {
        // соответствующая программа
        }
// другие методы
}
```

Ключевое слово implements показывает, что класс Milk определяет методы в интерфейсе Spillable (проливаемые продукты). Разумеется, если у нас есть экземпляр класса Milk, мы можем вызвать методы hasbeenSpilled (пролитый продукт) и cleanUpSpill. Преимущество подобных интерфейсов в том, что они, так же как и классы, относятся к типу данных. Хотя мы и не можем непосредственно их реализовать, мы можем представить их в виде переменных:

```
class Milk m+new Milk();
Spillable S=(Spillable)M;
if (S.hasBeenSpilled())
        {s.cleanUpSpill();}
```

Теперь мы можем через тип данных Spillable обратиться ко всем методам, имеющим дело с проливанием, и при этом нам не нужно будет определять все методы в базовом классе для всех продуктов в магазине - как жидких, так и твердых.

Кроме того, мы можем реализовать больше одного интерфейса в классе. Например, мы можем написать интерфейс Perishable (скоропортящиеся продукты), описывающий все продукты, которые могут испортиться. Наш класс Milk реализует оба интерфейса со следующим описанием класса:

```
public class Milk omplements Spillable, Perishable {
// определение класса
}
```

На самом деле было бы лучше реализовать интерфейс Perishable на уровне класса dairyGoods, потому что все молочные продукты являются скоропортящимися. Но не стоит беспокоиться - подклассы наследуют интерфейсы экземпляров их суперклассов.

Обзор понятий и пример

В этой главе мы рассмотрели множество понятий. Вы узнали, почему техника объектного ориентирования полезна вообще, как определять и использовать объекты и применять в Java такие фундаментальные методы ООР, как наследование и совмещение. Для того чтобы показать, что могут делать объекты для самого языка, были введены массивы. Ниже приведены табл. 3-3, суммирующая все понятия объектного ориентирования, и пример, который все эти понятия использует.

Таблица 3-3. Понятия и терминология объектного ориентирования

Понятие	Описание
Класс	Тип, определяющий некие данные и группу функций, действующую на этих данных.
Объект, экземпляр, реализация	Переменная типа class, которая появляется после реализации класса.
Затенение данных	Метод, позволяющий затенить переменную от других объектов. Затенение данных обычно облегчает процесс изменения внутренних структур данных.
Инкапсуляция	Заключение функций и данных в одни пакет.

Модификаторы доступа	Операторы, описывающие, какие классы могут обращаться к переменным или методам, определенным в классе.
Реализация	Создание объекта из класса. Реализация создает объект класса.
Конструктор	Раздел программы инициализации, вызываемый при реализации класса.
Иерархия классов	Многоуровневая диаграмма, показывающая взаимоотношения между классами.
Наследование	Создание нового класса расширением функций другого класса.
Суперкласс	Класс, унаследованный от другого класса.
Подкласс	Класс, наследующий от другого класса.
Переопределение метода	Переопределение методов подкласса, определенных в суперклассе.

Чтобы объединить все эти понятия в программе, мы создадим небольшую иерархию классов. Следующие группы объектов реализуют низкоуровневую графическую систему. Допустим, наша клиентка попросила нас написать программу рисования. Она хотела бы иметь возможность перемещать элементы рисунка как самостоятельные объекты. Первая демонстрационная версия будет включать примитивные формы, но окончательный проект может содержать множество сложных форм и растровых изображений. Если мы сможем сделать демо-версию к началу следующей недели, мы заключим контракт; в противном случае нам придется еще шесть месяцев корпеть над строками технического сопровождения. Ужасная перспектива, так что давайте поскорее сделаем работающую демо-версию.

Тот факт, что мы не знаем все формы, которые мы должны реализовать, усложняет нашу задачу. Нам придется применить наши знания методов OOP, чтобы сделать программу возможно более открытой. Одно из самых мощных наших орудий - наследование. Если мы правильно спроектируем иерархию объектов, у нас будет основа, в которую можно будет добавлять любое количество новых форм.

Помните про интерфейс? Он используется для того, чтобы группа объектов могла подчиняться стандартному набору правил. Эта возможность понадобится нам для создания программы рисования. Каждая форма должна будет иметь дело с несколькими важными подпрограммами. Нам нужно, чтобы каждая форма могла быть изображена на экране, чтобы это изображение можно было спрятать и переместить в другое место. Для такого основного набора операций мы можем написать простую программу рисования. Назовем наш интерфейс Shape (форма). Вот определение Shape:

```
interface Shape {
        public void show();
        public void hide();
}
```

Чтобы добавить в код новую форму, программа рисования должна будет выполнить только эти подпрограммы. С остальными подпрограммами будут иметь дело другие объекты иерархии. Следующий объект будет следить за местоположением формы. Любой текст программы, которым мы захотим описать формы, будет храниться в этом классе. Назовем этот класс BaseShape (базовая форма); он определен ниже. Заметим, что этот класс абстрактный и содержит абстрактные методы:

```
abstract class BaseShape {
        protected int x,y;
        public void setPos(int newX, int newY) {
                x = newX;
                y = newY;
        }
}
```

Теперь у нас есть общий интерфейс для каждой формы и базовый класс, откуда можно наследовать. Любой метод, который понадобится реализовать во всех формах, будет помещен в интерфейс. Общая часть программы для форм помещается в класс BaseShape. Последняя часть программы предназначена для реализации индивидуальных форм и небольшой демо-версии.

Следующий текст показывает, как реализуются некоторые формы, а именно прямоугольник и круг. Для каждой формы могут понадобиться дополнительные элементы данных и методы для

реализации данного конкретного рисунка. Чтобы воспользоваться удобным методом затенения данных, определим эти переменные и методы с модификатором private:

```
class Rectangle extends BaseShape implements Shape {
        private int len, width;
        Rectangle(int x, int y, int Len, int Width) {
                setPos(x,y);
                len = Len;
                width = Width;
        }
        public void show() {
                System.out.println("Прямоугольник(" + x + "," + y + ")");
                System.out.println("Длина=" + len + ", Ширина=" + width);
        }
        public void hide() {}
}
class Circle extends BaseShape implements Shape {
        private int radius;
        Circle(int x1, int y1, int Radius) {
                setPos(x1,y1);
                radius = Radius;
        }
        public void show() {
                System.out.println("Круг(" + x + "," + y + ")");
                System.out.println("Радиус=" + radius);
        }
        public void hide() {}
}
```

Последнее, что нам осталось, - сама программа рисования. Представьте себе, как долго вы могли бы мучиться с такой программой. Поскольку мы хотим хранить каждую форму отдельно, нам нужен способ хранения индивидуальных компонентов. Комбинация этих форм создает некий рисунок. Преимущество нашего подхода состоит в том, что мы легко можем перемещать или копировать элементы рисунка в другие места. Для этого нам нужен способ хранения элементов рисунка.

Здесь возникает следующая проблема. Какой тип структуры данных можно использовать, чтобы хранить множество объектов различных типов? Самым простым типом является массив. Массивы в Java позволяют хранить любой тип данных. Данные могут быть простого типа, например целые, более сложного типа, например объекты, или, как в нашем случае, интерфейсы, то есть определенного программистом типом. Мы определяем массив, который будет хранить объекты, реализующие интерфейс shape. Это позволит нам вызывать любой из определенных методов форм, не зная точно, какого типа этот объект. Мы можем продолжать создавать новые формы, не меняя нашей программы рисования. Это огромное достижение объектно-ориентированных языков!

```
class testShapes {
public static void main(String ARGV[]) {
        Shape shapeList[] = new Shape[2];
        int i;
        shapeList[0] = new Rectangle(0,0,5,5);
        shapeList[1] = new Circle(7,7,4);
        for(i=0, i<<2, i++) {
                shapeList[i].show();
                }
        }
}
```

Итак, вот она - простая программа, выполняющая основную работу нашей программы рисования. Добавим к ней графический раздел программы и получим удобную в употреблении и открытую для добавлений программу рисования. Когда наша клиентка попросит внести изменения в начальную программу, мы будем к этому готовы. Созданный каркас станет основой для реализации постоянно улучшающейся программы рисования. Хватит заниматься техническим сопровождением - у нас контракт!

Что дальше?

Мы надеемся, что теперь у вас сложилось четкое понимание ключевых понятий объектной ориентации и того, как они используются в Java. В следующей главе мы потратим некоторое время на описание синтаксиса языка. Хотя отчасти это описание будет носить обзорный характер, некоторые разделы будут совершенно новыми для вас. Надеемся, что к тому моменту, когда мы перейдем к самой сути - написанию применений и апплетов Java, - вы станете хорошо разбираться в основных понятиях языка Java и будете готовы приступить к настоящей работе.

Глава 4

Синтаксис и семантика

Язык Java в значительной своей части основан на языках C и C++, поэтому тот, кто хорошо знает эти языки, может считать эту главу почти что повторением пройденного. Разработчики Java поставили перед собой очевидную цель: создать язык, максимально похожий на C/C++, но который можно было бы эффективно использовать в программировании приложений для сети Интернет. Однако, чтобы достичь этой цели, им пришлось восполнить серьезнейшие пробелы C/C++ в таких областях, как безопасность, переносимость и удобство обслуживания программ. Кроме того, для создания Интернет-приложений потребовалось ввести в язык Java многопотоковость и обработку исключительных ситуаций. Таким образом, большинство отличий языка Java от языков C и C++ подпадают под одну из вышеперечисленных категорий.

Большинство информации, приведенной в этой главе, извлечено из официального документа под названием Java Language Specification (Спецификация языка Java) версии 1.0. Развитие языка продолжается, и поэтому вы наверняка будете сталкиваться с изменениями в будущих версиях Java. К примеру, в языке в его теперешнем состоянии присутствует несколько ключевых слов, значение которых не определено. Фирма Sun уже упоминала некоторые из возможных

изменений и дополнений в будущих версиях языка. Мы будем выносить информацию об изменениях языка по мере их появления на страницу Online Companion в World Wide Web.

Основное назначение этой главы - служить справочником, к которому вы сможете постоянно обращаться в своей практической работе. Однако здесь вы найдете и кое-какую общую информацию о языке. Нам бы хотелось, чтобы вы по крайней мере просмотрели эту главу, прежде чем переходить к изучению остальных глав книги. Особое внимание следует уделить разделам, посвященным массивам и исключительным ситуациям. Дело в том, что работа с массивами и обработка исключительных ситуаций в языке Java организованы не так, как в других языках программирования. В частности, понятие исключительных ситуаций является одним из ключевых понятий этого языка, и значительная часть материала всей книги имеет отношение к этому понятию.

В этой главе мы изучим синтаксис языка Java для следующих элементов языка:

- идентификаторы и использование стандарта Unicode,
- ключевые слова,
- типы данных,
- примитивные типы данных,
- преобразование примитивных типов данных,
- объявление переменных,
- знаки операций,
- пакеты,
- классы,
- преобразование ссылочных типов данных,
- интерфейсы,
- массивы,
- передача управления,
- исключения.

СОВЕТ Информацию о последних изменениях в стандарте языка Java вы всегда сможете найти на странице Online Companion по адресу http://www.vmedia.com/java.html.

Идентификаторы и использование стандарта Unicode

Идентификаторами называют имена, присваиваемые различным элементам программы. Любой объект, создаваемый в Java-программе, - переменная, метод или класс - имеет свое имя, которое представляет собой не что иное, как идентификатор. Идентификаторы в языке Java строятся из символов стандарта Unicode. "Что же такое Unicode? - спросите вы. - Еще один стандарт, который мне придется учить?" Вовсе нет! Весьма вероятно, что после того, как вы освоите материал этой главы, вам больше никогда в жизни не придется встречаться со стандартом Unicode.

Разработчики Java поставили перед собой цель сделать язык максимально переносимым, то есть таким, чтобы его могли с равным успехом использовать программисты, работающие не только на разных компьютерных платформах, но и живущие в разных странах и говорящие на разных языках. Поддержка иностранных языков стала актуальной в последние годы, когда многие компьютерные компании двинулись на завоевание рынков сбыта других стран и континентов. В настоящее время большинство компьютерных программ пишется с использованием английского языка, поэтому программисты в странах, где этот язык является иностранным, вынуждены фактически работать на чужом языке. Изучение компьютеров и без того дается многим людям с большим трудом. Зачем же еще больше усложнять жизнь? Стандарт Unicode и был разработан именно для того, чтобы помочь людям в других странах работать с компьютерами.

Стандарт Unicode был разработан организацией под названием Консорциум Unicode (Unicode Consortium), и его первая версия была опубликована в 1990 г. Этот стандарт унифицирует кодировку символов алфавитов большинства современных и древних языков. Каждый символ по стандарту Unicode кодируется 16 битами. Большинство пользователей работают с символами, закодированными по стандарту ASCII, в соответствии с которым каждый символ кодируется 7 битами. Увеличение количества битов на символ в стандарте Unicode позволяет расширить набор кодируемых символов, добавив к нему буквы других алфавитов и буквы с диакритическими значками, которые используются во многих языках. Все Java-программы кодируются с использованием Unicode, и все строки и одиночные символы, используемые в программах, хранятся в памяти в виде 16-битовых кодов.

Значит ли это, что вам придется учить еще одну кодировку символов? Конечно, нет. Более

того, использование Unicode скорее всего вообще никак не повлияет на вашу практическую работу как программиста. Если вы не используете в своих программах никаких символов, выходящих за пределы латинского алфавита, то вы можете вообще не задумываться об этом и продолжать писать программы так же, как делали это всю жизнь. Преобразованием вашего ASCII-файла в файл, закодированный по стандарту Unicode, займется компилятор Java, так что исходные тексты программ вам не придется хранить в каком-то специальном формате.

Таким образом, решение фирмы Sun использовать стандарт Unicode не окажет большого влияния на жизнь программистов-практиков. Нужно лишь помнить, что определенные типы данных занимают теперь больше места в памяти - а именно, все строки и одиночные символы увеличиваются в размерах в два раза. Конечно, на первый взгляд это может показаться недостатком языка. Однако вспомните, что при разработке Java главной целью было вовсе не экономное расходование памяти, а эффективное программирование для Интернет и возможность создания переносимых программ. То, что эти главные задачи успешно решены, позволяет мириться с несколько неэффективным расходом памяти для символьных значений. Вполне возможно, что, когда вам понадобится локализовать свою программу для использования в других странах, вы возблагодарите судьбу за то, что программисты фирмы Sun приняли в свое время столь дальновидное решение.

СОВЕТ Если вам понадобится узнать подробнее о стандарте Unicode, загляните на страницу по адресу http://unicode.org. Там вы найдете информацию о стандарте, сведения о том, как заказать бумажную копию стандарта Unicode, и узнаете, что нужно для того, чтобы стать членом Консорциума Unicode.

Комментарии

Java поддерживает все способы оформления комментариев, принятые в C/C++, и добавляет к ним еще один новый способ, ориентированный на автоматизированное документирование программного кода. Разумеется, какой бы стиль оформления комментариев вы ни использовали, на содержательную сторону программ это никак не влияет: компилятор игнорирует все возможные виды комментариев.

Комментарии в стиле C/C++ можно оформлять одним из следующих способов:

// текст

Весь текст, заключенный между этими сочетаниями символов, будет проигнорирован. Такой комментарий может распространяться на несколько строк.

// текст

Весь текст, следующий после // до конца строки, игнорируется.

В языке Java добавлен третий вариант оформления комментариев, используемый для автоматического документирования программ с помощью утилиты javadoc. Эта утилита, входящая в JDK, создает Web-страницу с описанием вашего кода; основой текста на этой странице как раз и будут комментарии в тексте программы, оформленные таким образом. Эти комментарии имеют следующий вид:

/** текст */

Текст в этом комментарии относится к переменной или методу, расположенному сразу после комментария.

Однако это еще не все. Авторы любого языка, а особенно такого, где допустимо несколько способов оформления комментариев, должны явным образом задать правила интерпретации для некоторых особых ситуаций, как, например, вложенных комментариев. В этом отношении между разными языками, включая Java, наблюдаются серьезные разногласия. В Java эти правила формулируются так:

- Комментарии не могут вкладываться друг в друга.
- Комментарии не могут быть частью строк или символьных констант.
- Сочетания символов /* и */ не имеют никакого специального значения в комментариях, отбитых символами //.
- Сочетание символов // не имеет никакого специального значения в комментариях, заключенных между /* и */.

Чтобы пояснить действие этих правил, достаточно одного примера. Следующая строка будет интерпретироваться как один вполне законный комментарий:

/* Это обычный комментарий, содержащий сколько угодно //, /*, /**. Чтобы закончить его, нужно написать */

Ключевые слова

В любом языке есть группа особых идентификаторов, зарезервированных для использования самим компилятором. Эти идентификаторы, обычно называемые ключевыми словами (keywords), не могут поэтому служить именами для каких-либо объектов вашей программы. Ключевые слова, зарезервированные компилятором Java, перечислены в табл. 4-1.

Таблица 4-1. На момент выхода этой книги ключевые слова, помеченные звездочкой, были зарезервированы для использования в будущем

Ключевые слова Java	**+**	**+**	**+**	**+**
abstract	do	implements	package	throw
boolean	double	import	private	throws
break	else	*inner	protected	transient
byte	extends	instanceof	public	try
case	final	int	*rest	*var
*cast	finally	interface	return	void
catch	float	long	short	volatile
char	for	native	static	while
class	*future	new	super	
*const	*generic	null	switch	
continue	*goto	operator	synchronized	
default	if	*outer	this	

Звездочками в этой таблице отмечены те ключевые слова, которые в текущей реализации языка, хоть и являются зарезервированными, не имеют никакого значения. Некоторые из них, например const и goto, зарезервированы с единственной целью сделать сообщения об ошибках, связанные с использованием этих конструкций, более осмысленными; другие относятся к тем механизмам, которые фирма Sun, вероятно, реализует в будущих версиях языка.

Типы данных

Тип данных, к которому принадлежит какая-либо переменная, может относиться к одному из четырех видов: классы, интерфейсы, массивы либо примитивные типы. На этапе компиляции каждая переменная воспринимается компилятором как принадлежащая либо к примитивному, либо к ссылочному типу данных. Слово "ссылочный" в словосочетании "ссылочный тип" говорит о том, что элементы таких типов (к которым относятся классы, интерфейсы и массивы) являются лишь указателями на некий объект. К примитивным типам относятся целые числа, числа с плавающей точкой, символы и булевские значения. Примитивные типы данных не являются ссылками на что-либо; их размер всегда известен заранее и никогда не меняется.

СОВЕТ Примитивные типы рассматриваются в следующем разделе "Примитивные типы данных", а информацию о ссылочных типах вы найдете в разделах "Классы", "Интерфейсы" и "Массивы".

Примитивные типы данных

Примитивные типы лежат в фундаменте любого языка программирования. Это те типы, о которых компилятор знает все, что ему нужно знать без каких-либо предварительных объявлений или спецификаций. Любой элемент, принадлежащий к типу, определенному пользователем, может быть разложен на составляющие примитивных типов. Примитивные типы - это те строительные блоки, без которых не обходится даже самая простая программа. Сейчас мы познакомимся по очереди с каждым из примитивных типов языка Java.

СОВЕТ В языке Java любой примитивный тип имеет заранее известный и никогда не меняющийся размер. Ничего похожего на машинозависимые размеры типов, от которых страдают

программисты на C/C++, в Java нет, как нет и никакой нужды использовать функцию sizeof. Наконец-то сделан решительный шаг в сторону истинной переносимости!

Целые числа

Целыми называют числа, не имеющие дробной части, поэтому изменять значение целочисленной переменной можно только на целое же число единиц. Компьютер использует целые числа в подавляющем большинстве своих операций. Целочисленные значения различаются по размеру отведенной для них памяти. В Java целое число может быть представлено последовательностью битов длиной от 8 до 64 битов. То, какой именно разновидностью целого типа вы будете пользоваться, определяет максимальное и минимальное значение, которые вы сможете хранить в переменной этого типа. В Java не поддерживаются беззнаковые целые, поэтому, вероятно, в некоторых случаях вам понадобятся более длинные целые, чем если бы вы работали с другим языком программирования. В табл. 4-2 приведены характеристики каждого из целых типов.

Таблица 4-2. Целые типы

Тип	Размер в битах	Минимальное значение	Максимальное значение
byte	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-922372036854775808	922372036854775807

Целочисленные константы могут задаваться в программе одним из трех способов: в виде десятичных, шестнадцатеричных и восьмеричных значений. По умолчанию все числа интерпретируются как десятичные и относятся к типу int, если только вы не припишете в конце числа букву "l", что означает "long".

Шестандатеричная цифра может иметь значение от 0 до 15, причем для значений от 0 до 9 используются обычные десятичные цифры, а для значений от 10 до 15 - первые буквы латинского алфавита с A до F. Числа в шестнадцатеричной записи часто используются для записи больших чисел или для ввода значений, для которых более естественно двоичное представление. Поскольку каждая цифра представляет не 10, а 16 возможных значений, большое число в шестнадцатеричной записи занимает меньше места, чем в десятичной.

Возьмем для примера число 32767, записанное десятичными цифрами. Это число - наибольшее значение, которое может принимать тип short. В шестнадцатеричной записи оно имеет вид 0x7FFF. Всегда, когда вам нужно указать на то, что число записано в шестнадцатеричной системе, вы должны приписать к нему спереди пару символов "0x". Регистр букв в шестнадцатеричных числах значения не имеет.

Восьмеричная цифра принимает значения от 0 до 7. Число в восьмеричной записи должно начинаться с нуля, за которым следует одна или несколько восьмеричных цифр. Например, десятичное число 32767 в восьмеричной записи выглядит как 077777. Иначе говоря, если число начинается с нуля, для компилятора это служит сигналом к тому, что цифры этого числа должны интерпретироваться как восьмеричные.

Все целочисленные значения обладают свойством возврата к началу диапазона (wrapping). Это значит, что если вы попытаетесь увеличить или уменьшить целое число, уже находящееся на самой границе диапазона возможных значений, это число перескочит в противоположный конец своего диапазона. К примеру, возьмем переменную типа byte, имеющую значение 127. Если прибавить к этой переменной единицу, то ее значение станет -128. Никакой ошибки здесь нет - целые переменные в таких ситуациях всегда меняют свое значение с наибольшего положительного на наименьшее отрицательное значение. И наоборот: если от -128 отнять 1, мы получим 127. Мораль проста: чтобы не столкнуться с подобной неприятностью, вы должны заранее оценивать, какой диапазон целых значений вам понадобится, и выбирать для своих переменных соответствующий тип.

Числа с плавающей точкой

Язык Java поддерживает числа с плавающей точкой обычной и двойной разрядности в соответствии со стандартом IEEE на двоичную арифметику с плавающей точкой (IEEE Standard for Binary Floating-Point Arithmetic). Соответствующие типы называются float и double. Тип float представляет собой 32-битное число с плавающей точкой обычной разрядности, а тип double - 64-битное число с плавающей точкой двойной разрядности.

Переменные с плавающей точкой могут хранить не только численные значения, но и любой из особо определенных флагов (состояний): отрицательная бесконечность, отрицательный нуль, положительная бесконечность, положительный нуль и "отсутствие числа" (not-a-number, NaN). Поскольку все эти флаги определены в языке Java, вы можете предусматривать в своем коде соответствующие проверки. Как правило, эти особые состояния являются результатом ошибочных действий; например, если 0 поделить на 0, результатом будет NaN, и вы сможете в программе явным образом выяснить это. Таким образом, поддержка языком этих особых состояний существенно облегчает поиск ошибок.

Все символьные константы с плавающей точкой подразумеваются принадлежащими к типу double, если не указано обратное. Чтобы задать 16-битное число с плавающей точкой типа float, вы должны приписать в конец его цифровой записи букву "f". После этого компилятор будет считать эту константу принадлежащей к типу float. Поскольку Java требует точного согласования типов, вы обязательно должны будете прибегнуть к этому приему, чтобы инициализировать переменную типа float.

Например, следующая строка кода приведет к ошибке при компиляции из-за несоответствия типов:

```
float num = 1.0;
```

Все константы с плавающей точкой по умолчанию относятся компилятором к типу double, поэтому, чтобы явным образом указать, что данная константа имеет тип float, припишите к цифровой записи этого числа букву "f":

```
float num = 1.0f;
```

СОВЕТ Необходимо быть крайне осторожным при использовании чисел с плавающей точкой в операторах сравнения. Помните, что два числа с плавающей точкой могут совпадать во многих своих десятичных знакоместах, однако стоит им разойтись на единичку в самой последней цифре, как с точки зрения оператора сравнения они перестанут быть равными. Сравнение двух значений с плавающей точкой в операторе if или использование переменной с плавающей точкой в качестве счетчика цикла может из-за этого привести (помимо снижения скорости работы) к появлению ошибок, которые очень трудно обнаружить.

Символы

Символы в Java реализованы с использованием стандарта Unicode (см. раздел "Идентификаторы и использование стандарта Unicode" выше). Это означает, что для хранения каждого символа отводится по 16 бит. Кодировка Unicode позволяет специфицировать множество самых экзотических непечатаемых символов и букв иностранных алфавитов. Чтобы задать константу-символ в программе, вы можете использовать как обычный символ, так и escape-последовательность для прямого указания кода в Unicode. В любом из этих случаев вы должны заключить символьное значение в пару апострофов.

Escape-последовательности в Unicode могут задаваться одним из двух способов. Первый из них должен показаться знакомым программистам на C/C++. Этот способ заключается в указании после обратной косой черты (\) некоторой буквы или символа, как показано в первом столбце табл. 4-3.

Таблица 4-3. Escape-последовательности в Unicode

Escape-последовательность	Функция	Значение в Unicode
\b	Забой (backspace)	\u0008
\t	Горизонтальная табуляция (horizontal tab)	\u0009
\n	Перевод строки (linefeed)	\u000a
\f	Перевод страницы (form feed)	\u000c
\r	Возврат каретки (carriage return)	\u000d
\"	Двойная кавычка (double quote)	\u0022
\'	Апостроф (single quote)	\u0027
****	Обратная косая черта (backslash)	\u005c

Вы можете также пользоваться другим способом записи escape-последовательностей: парой символов "\u", за которой следует четырехзначное шестнадцатеричное число, представляющее собой код нужного вам символа в Unicode. Число это может принимать значения от 0000 до 00FF. Вот несколько примеров симольных констант:

- 'a' - символ "a".
- '\n' - escape-последовательность для символа новой строки.
- '\\' - escape-последовательность для символа обратной косой черты.
- '\u0042' - escape-последовательность для символа с Unicode-кодом 0042.

Тип boolean

Переменные булевского типа могут иметь лишь одно из двух значений - true или false. Единственный способ присвоить значение переменной булевского типа - использование констант true и false. В отличие от C вы не можете присваивать булевским переменным целочисленные значения.

Однако чтобы как-то приблизиться к автоматическому преобразованию типов языка C, вы можете прибегнуть к сравнению целочисленного значения с нулем. Как известно, в языке C целочисленное значение 0 соответствует булевскому значению false, а все другие целочисленные значения - булевскому true. Чтобы преобразовать в соответствии с этими правилами целую переменную i к булевскому значению, вы можете использовать такую запись:

```
int i;
boolean b;
b = (i != 0);
```

Здесь используется оператор проверки неравенства, посредством которого i сравнивается с нулем. Круглые скобки в этом выражении необходимы для того, чтобы составные части выражения вычислялись в нужном порядке.

СОВЕТ Тип boolean играет в языке Java важную роль. Многие конструкции этого языка, такие как операторы цикла и условные операторы, могут пользоваться только выражениями, имеющими тип boolean. Ничего сложного в правилах работы с этим типом нет, однако если вы привыкли к тому, как в таких случаях вынуждает вас поступать C/C++, концепция булевских типов языка Java может потребовать некоторых усилий для овладения.

Преобразование примитивных типов данных

Преобразование между двумя примитивными типами данных встречается на практике очень часто. Главное при этом, как и везде, правильно представлять себе, что в действительности происходит. Невнимательность может привести к потере информации или к получению неверных результатов.

Язык Java требует обязательного соответствия типов. Это значит, что компилятор не будет автоматически преобразовывать вам данные из одного типа в другой. Вы должны преобразовывать типы явным образом с помощью механизма приведения типа (type cast), который позволяет указать компилятору, к какому типу следует преобразовать те или иные данные.

В Java приведение типа осуществляется так же, как и в C/C++. Для этого достаточно указать идентификатор требуемого типа в круглых скобках перед приводимым выражением. Если затребованное преобразование типа возможно, оно будет осуществлено, и вы получите значение нужного типа. Допустим, у нас есть две переменные shortVar и intVar, первая из которых принадлежит к типу short, а вторая - к типу int. Между этими типами существуют два возможных преобразования:

```
short shortVar = 0;
int intVar = 0;
intVar = shortVar;
shortVar = intVar; // несовместимые типы в операторе присваивания
```

При компиляции этого кода присваивание значения intVar переменной shortVar приведет к выдаче сообщения об ошибке. Дело в том, что вы пытаетесь тем самым присвоить значение с большим диапазоном переменной с меньшим диапазоном. Такой тип преобразования называется сужающим (narrowing conversion), так как при этом уменьшается количество бит, отведенных для хранения данных. Понятно, что при сужающем преобразовании вы можете потерять часть информации, содержащейся в числе, - либо изменив его значение, либо (в случае числа с

плавающей точкой) уменьшив разрядность и тем самым точность представления.

Язык Java заставляет вас расписаться в том, что вы отдаете себе отчет, что происходит при таком преобразовании: при любых сужающих преобразованиях вы должны прибегать к явному приведению типа. Это еще один пример того, как Java пытается почти насильственными методами внедрить хороший стиль программирования. Каждый раз, когда вы преобразуете, к примеру, значение типа long (64 бита) в значение типа int (32 бита), вы должны предусмотреть, что именно должна будет делать программа при возможности потери информации. Если вы абсолютно уверены, что значение приводимого целого типа long будет всегда попадать в диапазон типа int, то смело прибегайте к приведению типа; в противном случае вам нужно будет предусмотреть какие-то дополнительные действия (например, выдачу предупреждения пользователю о возможной потере данных).

Так, чтобы заставить компилироваться приведенный выше фрагмент, нужно добавить в него явное приведение типа. Вот как это будет выглядеть:

```
short shortVar = 0;
int intVar = 0;
intVar = shortVar;
shortVar = (short) intVar;
```

Теперь ничто не мешает компьютеру произвести требуемое присваивание. Значение intVar будет при этом преобразовано к типу short, в результате чего старшие биты числа будут отброшены, но знак числа при этом сохранится. В нашем случае это приведет к тому, что 32-битное целое число превратится в 16-битное.

В табл. 4-4 перечислены все преобразования примитивных типов, допустимые в языке Java. Буква "C" в клетке таблицы означает, что для данного преобразования требуется явное приведение к типу, иначе компилятор выдаст сообщение об ошибке. Буква "L" означает, что при данном преобразовании может измениться величина или уменьшиться разрядность числа из-за частичной потери информации. Буква "X" обозначает, что данное преобразование типов в Java запрещено.

Таблица 4-4. Преобразование примитивных типов

Исходный тип	**Тип, к которому происходит преобразование**	**+**	**+**	**+**	**+**	**+**	**+**	**+**
^	byte	short	int	long	float	double	char	Boolean
byte							C	X
short	C, L						C	X
int	C, L	C, L					C, L	X
long	C, L	C, L	C, L		C, L	C	C, L	X
float	C, L	C, L	C, L	C, L			C, L	X
double	C, L	C, L	C, L	C, L	C, L		C, L	X
char	C, L	C	C	C	C	C		X
boolean	X	X	X	X	X	X	X	

Буква "C" означает, что для данного преобразования требуется явное приведение типа; буква "L" означает, что при данном преобразовании может измениться величина или уменьшиться разрядность числа из-за частичной потери информации; буква "X" обозначает, что данное преобразование типов в Java запрещено.

Преобразование значений с плавающей точкой в целочисленные значения

При преобразовании значения с плавающей точкой в любой из целых типов вы теряете информацию о дробной части числа. Java отсекает дробную часть, округляя таким образом число в направлении к нулю. После этого получающееся в результате целое число будет преобразовано к требуемому целому типу путем увеличения или уменьшения количества бит.

Преобразование числа с плавающей точкой двойной разрядности к обычной разрядности

Преобразование числа типа double к числу типа float происходит в соответствии с требованиями режима "округления к ближайшему" (round-to-nearest) стандарта IEEE 754. Если преобразуемое значение слишком велико и выходит за пределы диапазона типа float, результатом преобразования будет положительная или отрицательная бесконечность. Преобразование значения NaN дает в результате также NaN.

Преобразования типа boolean

Тип boolean не допускает преобразования в какой-либо другой тип, так же как и никакой из типов Java не может быть преобразован к булевскому типу. Если вам требуется преобразовать целое значение в булевское или булевское в строковое, вы можете присвоить нужное значение вручную с помощью, например, такого фрагмента кода:

```
boolean bool;
int i = 15;
String st = null;
if (i == 0) bool = false; else bool = true;
if (bool) st = "true"; else st = "false";
```

СОВЕТ Информацию о преобразованиях ссылочных типов данных (классов, интерфейсов и массивов) вы найдете в соответствующем разделе этой главы.

Объявление переменных

Переменная в Java может принадлежать к примитивному типу либо быть объектом или интерфейсом. Создавать новые переменные можно в любом месте программы. За оператором объявления новой переменной может следовать оператор инициализации, с помощью которого созданной переменной присваивается начальное значение.

Мы уже не раз создавали новые переменные в рассматриваемых примерах. Вот еще несколько примеров объявления и инициализации переменных в Java:

```
int i = 42;
String st = "Hello World";
float pi = 3.14f;
boolean cont;
```

СОВЕТ В отличие от языков C и Паскаль, переменные в Java могут объявляться действительно в любом месте программы. Совсем не обязательно собирать все объявления переменных в начале программы, функции или процедуры.

Область действия

Любое объявление переменной имеет свою область действия, границы которой зависят от того, где именно расположено это объявление. Всякий раз, когда вы помещаете фрагмент кода в пару фигурных скобок { }, вы тем самым вводите новый уровень группирования. Границы этого нового уровня определяют, где созданные в нем переменные будут доступны, а где станет возможным их уничтожение. Переменная доступна только в том случае, если она определена на текущем уровне группирования или на одном из вышестоящих уровней.

Когда текущий уровень завершается, все объявленные в нем переменные становятся недоступными. Однако это не значит, что они обязательно уничтожаются, так как правила уничтожения переменных более сложны и учитывают дополнительные обстоятельства. При объявлении переменной, как вы уже знаете, для нее выделяется участок памяти. Когда текущий блок, в котором эта переменная была объявлена, заканчивается, она становится доступной для уничтожения, и теперь решение о ее уничтожении будет принимать сборщик мусора. Уничтожение произойдет в тот момент, когда на эту переменную больше никто не будет ссылаться. Это означает, что примитивные типы данных всегда уничтожаются сразу же, как только кончается соответствующий блок. Напротив, уничтожение переменных ссылочных типов

может быть отложено.

Уровни группирования и области действия объявлений легче всего представить себе в виде дерева, где каждый блок соответствует одной из ветвей. Чем больше вложенных друг в друга уровней группирования, тем выше будет это воображаемое дерево. Давайте возьмем для примера фрагмент программы и нарисуем соответствующее ему дерево уровней группирования:

```
class foo {
        int cnt;
        public void test1 {
                int num;
        }
        public void test2;
                for(int cnt=0; cnt < 5; cnt++) {
                        System.out.println(cnt);
                }
        }
}
```

На рис. 4-1 показано дерево уровней группирования для приведенного выше фрагмента программы.

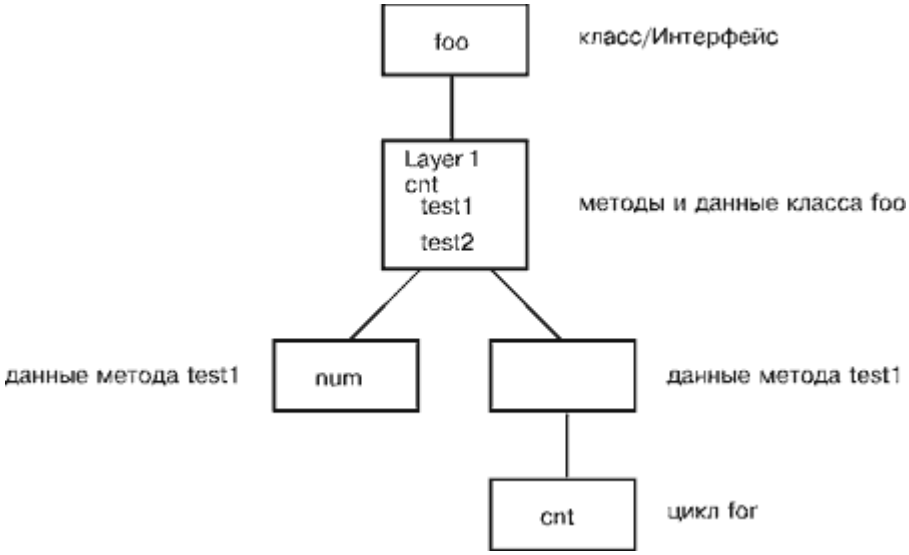

Рис. 4.1.

Обратите внимание на то, что каждый новый блок создает новую область действия переменных. На самом верхнем уровне такой областью является пакет, в котором объявляются классы и интерфейсы. На следующем уровне объявляются входящие в класс переменные и методы. Наконец, каждый метод класса образует собой еще один уровень группирования, в котором доступны локальные переменные, объявленные в этом методе. Обратите особое внимание на метод test2: на уровне группирования, соответствующем этому методу, нет объявления новых переменных, однако есть цикл for, который создает еще один вложенный уровень. Внутри этого подчиненного уровня определена переменная cnt, так что внутри тела цикла идентификатор cnt относится к этой локальной переменной, а не к одноименной переменной, объявленной на уровне класса (в таких случаях говорят, что одна переменная затеняет другую).

Для тех, кто незнаком с языком C, синтаксис цикла for может показаться странным. Мы будем подробно говорить о циклах с for и о других типах циклов в разделе "Передача управления" данной главы.

Итак, возможность создания нескольких уровней группирования позволяет производить затенение переменных. Например, если на уровне класса объявлена переменная cnt, ничто не мешает вам объявить переменную с тем же именем внутри какого-нибудь из методов этого класса, которая затенит вышестоящую переменную и сделает ее недоступной. Внутри метода идентификатор cnt будет относиться именно к локальной переменной cnt, объявленной в этом методе. Однако существует и возможность получить доступ к переменной класса, даже если она затенена. Для этого нужно воспользоваться особой ссылочной переменной this. Переменная this всегда указывает на текущий класс, поэтому, чтобы получить доступ к затененной переменной, объявленной в текущем классе, нужно явным образом указать принадлежность переменной к

классу, а не к методу. Вот как это делается:
this.cnt = 4;

СОВЕТ В большинстве случаев лучше избегать затенения переменных. Если же без этого никак не обойтись, помните, что доступ к затененной переменной, объявленной на уровне класса, возможен с помощью ссылочной переменной this.

Правила именования переменных

Имя переменной должно начинаться с буквы. Оно может быть любой длины и содержать в себе буквы, цифры и любые символы пунктуации за исключением точки. Имя переменной не может совпадать с каким бы то ни было идентификатором, уже существующем на данном уровне группирования. Это значит, в частности, что имена переменных не могут совпадать с именами:

- меток;
- других переменных на данном уровне группирования;
- параметров текущего метода.

Если вы в своей программе пользуетесь расширенной кодировкой Unicode, чтобы работать с символами иностранных алфавитов, то вы должны знать, как Java осуществляет сравнение символов. Один идентификатор признается равным другому только в том случае, если Unicode-коды всех символов в этих двух идентификаторов совпадают. Это означает, что латинская заглавная "A" (код \u0041) отличается от греческой заглавной "A" (код \u0391) и от всех других букв "A", имеющихся в алфавитах мира. Отсюда также следует, что язык Java чувствителен к регистру, то есть что заглавное "A" и строчное "a" с точки зрения Java являются разными символами.

Знаки операций

Язык Java поддерживает большое количество знаков операций (operators). Знак операции представляет собой специальный символ и предназначен для выполнения какого-то действия над одной, двумя или более переменными. К самым распространенным знакам операций относятся плюс (+), минус (-) и знак равенства (=).

Знаки операций подразделяются по количеству своих аргументов (операндов). Некоторые знаки операций, например минус, имеют разный смысл в зависимости от того, относятся они к одному или двум операндам.

Таблица 4-5. Приоритет знаков операций

.	[ ]	( )		
++	-	!	~	instanceof
*	/	%		
++	-			
<<<	>>>	>>>>		
<	>	<=	>=	
==	!=			
&				
^^				
&&				
||				
?:				
=	op=			

Порядок приоритета в таблице - сверху вниз; знаки операций, расположенные в одной строке, имеют равный приоритет.

СОВЕТ Знак операции op= является сокращенной записью целого класса комбинированных знаков операций, примером которых может служить +=.

Выражением (expression) называется конструкция, построенная из знаков операций и их операндов. Вычисление выражений определяется набором правил приоритета. Когда говорят, что одна операция имеет приоритет над другой, это означает, что в выражении она будет выполнена раньше. Пример, который должен быть известен вам еще с начальной школы, - разница между сложением и умножением. Поскольку в выражении x=2+4*3 умножение выполняется раньше сложения, значением x будет 14, а не 18.

В сложных выражениях для обеспечения правильного порядка вычисления операций лучше использовать скобки. Это не только гарантирует вам нужный порядок вычислений, но и сделает выражения более удобочитаемыми. Полная сводка правил приоритета приведена в табл. 4-5. Знаки операций, расположенные в этой таблице в одной строке, имеют равный приоритет, и выражения выполняются слева направо.

Знаки операций с числовыми аргументами

Знаки операций, аргументами которых являются числа, разделяются на две категории: унарные (unary) знаки операций с одним аргументом и бинарные (binary) - с двумя аргументами. Бинарные знаки операций подразделяются далее на операции с числовым результатом и операции сравнения, результатом которых является булевское значение.

Результат операции будет всегда принадлежать в тому же типу, что и больший из операндов. Например, если мы складываем два целых числа, одно из которых типа short, а другое типа long, результат будет иметь тип long. Правила выбора типа результата обобщены в табл. 4-6, из которой также следует, что наименьшим результатом, возвращаемым при операциях с целыми числами, является тип int и что при сложении любого числа с числом с плавающей точкой результат будет иметь тип float или double. Теперь перейдем к рассмотрению унарных операций.

Таблица 4-6. Выбор типа результата операции

Тип 1	Тип 2	Тип результата
byte	byte	int
byte	short	int
byte	int	int
byte	long	long
short	short	int
short	int	int
short	long	long
int	int	int
int	long	long
int	float	float
int	double	double
float	float	float
float	double	double

Унарные знаки операций

Унарные знаки операций имеют один аргумент. После выполнения операции результат подставляется в выражение на место операнда. Тип результата всегда совпадает с типом операнда, а потеря значимых цифр или изменение битовой длины числа при этом невозможны. Унарные операции Java перечислены в табл. 4-7.

Таблица 4-7. Унарные знаки операций

Знак операции	Описание
-	унарный минус
+	унарный плюс
~	побитовое дополнение
++	инкремент

-- декремент

Вот примеры выражений с унарными знаками операций:

- i++
- -i
- ~i

Унарные минус и плюс

Знак операции унарный минус (-) используется для изменения знака числа (отрицательное число становится положительным, а положительное - отрицательным). В противоположность этому, знак операции унарный плюс (+) фактически не выполняет никаких действий и предусмотрен лишь для сохранения симметрии.

Изменение знака целого числа можно представить как вычитание этого числа из нуля. Это справедливо для всех чисел, кроме максимальных по модулю отрицательных чисел. Дело в том, что в любом из целых типов отрицательных чисел на одно больше, чем положительных, поэтому унарный минус не может превратить наибольшее по модулю отрицательное число в положительное, сохраняя его тип. Например, приведенный ниже фрагмент кода даст совершенно неожиданный результат - на печать будет выведено значение -128, а не 128:

```
byte i=-128
System.out.println(-i);
```

Применяя унарный минус к числам с плавающей точкой, следует помнить о некоторых особых случаях. Если значением числа с плавающей точкой является NaN, то это значение не изменится от применения унарного минуса. Однако как положительный и отрицательный нуль, так и положительная и отрицательная бесконечности меняют при этом свой знак на противоположный.

Побитовое дополнение

Знак операции побитового дополнения применяется только к целым типам. Эта операция интерпретирует целое число как набор битов и меняет в этом наборе все нули на единицы, а все единицы на нули. Применив этот знак операции к числу x, вы получите в результате число (-x)-1. Те, кому не доводилось работать с данными на уровне битов, возможно, найдут эту операцию непривычной для себя. Область ее применения - те задачи, в которых нас не интересует числовое значение целой переменной: мы используем эту переменную просто как набор отдельных битов. Например, число типа short, равное нулю (0x0000), после операции побитового дополнения превращается в -1 (0xFFFF).

Знаки операций инкремента и декремента

Термины "инкремент" и "декремент" означают прибавление и вычитание единицы. Знаки операций инкремента и декремента могут размещаться как до, так и после переменной. Эти варианты называются соответственно префиксной и постфиксной записью этих операций.

Знак операции в префиксной записи возвращает значение своего операнда после вычисления выражения. При постфиксной записи знак операции сначала возвращает значение своего операнда и только после этого вычисляет инкремент или декремент. Рассмотрим фрагмент кода, иллюстрирующий эту разницу:

```
int i=0;
int j=0;
System.out.println(++i);
System.out.println(j++);
```

На выходе этой программы вы получите сначала 1, а затем 0. В первом операторе печати мы использовали префиксную запись операции, при которой переменная i сначала инкрементируется, а затем ее значение выводится на печать. Во втором случае переменная j сначала выводится на печать, а затем инкрементируется. В обоих случаях после обоих операторов печати значением как i, так и j будет единица.

Бинарные знаки операций

Бинарные знаки операции имеют два операнда и возвращают некий результат. Как уже упоминалось выше, результат будет иметь тот же тип, что и больший из операндов - например, сложение целых чисел типа byte и типа int даст в результате тип int. Сами операнды при этой операции не изменяются. Все бинарные знаки операций можно разделить на те, что вычисляют некий числовой результат, и те, что предназначены для сравнения операндов. Знаки операций, возвращающие числовой результат, приведены в табл. 4-8.

Таблица 4-8. Бинарные знаки операций, возвращающие числовые значения

Знак операции	Описание
++	сложение
-	вычитание
*	умножение
/	деление
%	остаток от деления
&	побитовое И
|	побитовое ИЛИ
^	побитовое исключающее ИЛИ
<<	побитовый сдвиг влево с учетом знака
>>	побитовый сдвиг вправо с учетом знака
>>>	побитовый сдвиг вправо без учета знака
op=	комбинация присваивания и одного из знаков операций

Сложение и вычитание

Если один из операндов операций + или - является числом с плавающей точкой, то перед выполнением операции оба операнда преобразуются в числа с плавающей точкой. Все целые типы, кроме типа long, при сложении и вычитании приводятся к типу int - иными словами, операции + и - не могут возвращать значения типа byte или short. Чтобы присвоить результат операции переменной одного из этих типов, вы должны будете прибегнуть к явному приведению типа.

Операции сложения и вычитания для чисел с плавающей точкой реализованы в языке Java в соответствии со стандартом IEEE. Это значит, что в большинстве случаев результаты будут совпадать с теми, которые интуитивно ожидаются. Однако если вы захотите выяснить, как Java обрабатывает особые случаи (например, сложение двух бесконечностей), обращайтесь к официальной спецификации языка Java.

Умножение, деление и нахождение остатка

Эти знаки операций изменяют типы операндов так же, как и знаки операций сложения и вычитания. Операции с целыми числами всегда ассоциативны, но в случае чисел с плавающей точкой из этого правила могут быть исключения. Допустим, мы объявили два числа с плавающей точкой, одно из которых равно единице, а второе - максимально представимому положительному числу типа float:

```
float one = lf;
float max = 2^24e104;
```

После этого, если мы вычислим значение выражения

```
one + max - one == max - one + one
```

то результатом его будет false. Дело здесь в том, что в левой части операции сравнения мы сначала прибавляем единицу к максимально возможному числу с плавающей точкой. Возникает переполнение, и результатом первой операции сложения будет поэтому положительная бесконечность. Вычитание затем единицы из бесконечности даст в результате ту же бесконечность. Поэтому в левой части сравнения мы получаем положительную бесконечность. В правой же части все идет так, как задумано: вычтя и прибавив единицу к значению max, мы получим в результате то же значение max.

Операция нахождения остатка от деления (%), или деление по модулю, определена следующим образом: (a/b)*b + (a%b) = a. Эта операция возвращает положительный результат, если делимое положительно, и отрицательный результат в случае отрицательного операнда. Результат этой операции по абсолютному значению всегда меньше делителя.

Вот пример использования операции нахождения остатка:

```
int i=10;
int j=4;
int k=-4;
System.out.println(i % j); // =2
System.out.println(i % k); // = -2
```

СОВЕТ Поведение операции нахождения остатка в Java отличается от требований стандарта IEEE 754. Однако вы можете пользоваться другой операцией, определенной в полном соответствии с этим стандартом. Эта операция определена в библиотеке math под названием Math.IEEEremainder.

Побитовые знаки операций

К побитовым знакам операций относятся те, что оперируют с числами в их битовом представлении. Эти операции применяются только к целочисленным значениям. Поскольку в числах с плавающей точкой битовые цепочки кодируют не само число, а его особое математическое представление, применение побитовых операций к таких числам не имеет смысла. Как правило, побитовые операции логического И (&), логического ИЛИ (|) и исключающего ИЛИ (^) используются для изменения и получения значения отдельных битов числа.

Представьте, что каждый бит целого числа хранит в себе какой-то флаг. Работая с целыми числами типа byte, мы можем хранить в них восемь таких флагов, по одному на каждый бит. Ниже показаны некоторые распространенные манипуляции с флагами, упакованными в целом числе:

```
byte flags=0xff; // исходное значение флагов 11111111
byte mask=0xfe; // битовая маска 11111110
flads = flags & mask; // установить флаг номер 8
flads = flags | mask; // сбросить флаг номер 8
flads = flags ^ mask; // = 00000001
```

Вниманию пользователей C/C++

Язык Java не поддерживает битовые поля. Чтобы получить доступ к отдельным битам,, вы должны пользоваться либо битовыми масками,, либо средствами класса java.util.BitSet.

Знак операции сдвига

Java поддерживает три операции сдвига - сдвиг влево (<<), сдвиг вправо (>>) и сдвиг вправо без учета знака (>>>>). Запись этих операций имеет следующий вид: сначала идет выражение, над которым производится сдвиг, затем знак операции сдвига и наконец количество битов, на которое производится сдвиг. Оба операнда должны быть целыми числами. Если вы хотите произвести сдвиг с числом с плавающей точкой, вы должны прибегнуть к явному преобразованию типа.

Операции сдвига с учетом знака сохраняют знак операнда. Сдвиг без учета знака заменяет каждую освобождающуюся при сдвиге позицию нулевым битом, игнорируя бит знака. Сдвиг на ноль битов в любую сторону допустим, хотя и не изменяет значения операнда.

С помощью операции сдвига можно эффективно производить умножение и деление на степени двойки. Например, сдвиг вправо на один бит равнозначен делению целого числа нацело на 2. При этом теряется крайний правый бит числа, поэтому нечетное число при этой операции даст тот же результат, что и меньшее на единицу четное число:

```
int i = 129; // в двоичном представлении это число равно 10000001
i = i >> 1; // теперь мы имеем 1000000, или 64
```

Комбинированные знаки операций

Некоторые знаки операций можно комбинировать со знаком операции присваивания, так что получающийся в результате комбинированный знак операции сначала вычисляет результат операции, а затем присваивает его переменной в левой части оператора присваивания. Такое комбинирование допустимо для всех знаков операций, возвращающих численный результат, - иными словами, для всех операций Java за исключением операций сравнения (см. раздел "Знаки операций сравнения" ниже).

Программист должен отдавать себе отчет в том, как происходит загрузка операндов в

регистровую память процессора при выполнении таких комбинированных операций. Если в выражении, стоящем в правой части, вы измените значение переменной, стоящей в левой части оператора присваивания, то это изменение будет проигнорировано. Лучше всего проиллюстрировать это примером:

```
int i = 0;
i += ++i;
```

На первый взгляд этот фрагмент кода должен присваивать переменной i значение 2. Однако это не так. Вычисление производится следующим образом. Прежде всего, текущее значение i загружается в регистр. Затем вычисляется выражение ++i, его результат присваивается переменной i и используется как второй операнд операции сложения. Тонкость этого момента в том, что первым операндом в сложении служит исходное значение i, то есть 0. В результате мы получим значение 1, которое и будет окончательно занесено в переменную i. Таким образом, приведенный выше фрагмент программы присваивает i значение 1.

Знаки операций сравнения

Помимо знаков операций, производящих вычисления с числами, в Java есть группа знаков операций, предназначенных для сравнения двух значений. Эти операции так и называются - операции сравнения. Они имеют по два параметра и возвращают булевское значение, соответствующее результату сравнения. Знаки операций сравнения языка Java перечислены в табл. 4-9.

Таблица 4-9. Знаки операций сравнения

Знак операции	Описание
<	меньше чем
>	больше чем
<=	меньше или равно
>=	больше или равно
==	равно
!=	не равно

СОВЕТ Существует одна очень распространенная ошибка, связанная с использованием операций сравнения. Многие программисты, особенно начинающие, пытаются использовать в качестве знака операции проверки равенства одиночный, а не двойной знак равенства. Помните, что одиночный знак равенства используется только в операции присваивания, а в операции сравнения нужно использовать двойной символ знака равенства. Это соглашение заимствовано из C/C++, и хотя компилятор Java, в отличие от этих языков, способен сам обнаружить такую ошибку, разумнее иметь в виду этот момент заранее.

Булевские знаки операций

Булевские знаки операций аналогичны соответствующим знакам операций для чисел. Все булевские операции возвращают значения типа boolean. Список булевских знаков операций языка Java приведен в табл. 4-10.

Таблица 4-10. Булевские знаки операций

Знак операции	Описание
!	отрицание
&	логическое И
|	логическое ИЛИ
^	исключающее ИЛИ
&&	условное И
||	условное ИЛИ
==	равно
!=	не равно
op=	комбинация оператора присваивания и одной из операций

?: условный оператор

Условный оператор - это единственная операция языка Java, имеющая три операнда. Этот оператор записывается в форме a?b:c. При этом сначала вычисляется выражение a, которое должно дать булевское значение, а затем, в соответствии с полученным результатом, возвращается либо b, если a имеет значение true, либо c, если a имеет значение false. По своей функции этот оператор аналогичен оператору if:

```
int i;
boolean cont=false;
// обычный оператор if
if (cont) i=5; *
else i=6;
// сокращенная запись с помощью условного оператора
i = (cont?5:6);
```

В этом фрагменте кода переменной i присваивается значение либо 5, либо 6 в зависимости от того, чему равна булевская переменная cont. Если cont имеет значение true, i получает значение 5; в обратном случае i получает значение 6. Условный оператор позволяет достичь этого результата быстрее и удобнее.

Знаки операций над символьными значениями

Особых знаков операций, возвращающих символьные значения, в языке Java не существует. Большинство знаков операций, которые мы обсуждали выше, возвращают целочисленные значения. Если вам потребуется применить эти знаки операций к символьным значениям, вы должны будете явным образом привести результат этих операций обратно к типу char. Если одним из операндов знака операции является символ, он перед выполнением операции приводится к типу int. Очевидно, что это преобразование никогда не приведет к потере информации.

Предположим, что нам требуется написать код для перевода символа из верхнего в нижний регистр. Воспользуемся тем, что код заглавной буквы A (как в ASCII, так и в Unicode) равен 98, а код строчной буквы a равен 65. Если мы возьмем любую заглавную букву и вычтем из ее кода разницу между кодами заглавной и строчной букв A, то в результате мы получим код, соответствующий строчной букве. Проиллюстрируем этот принцип примером:

```
char c='B';
c = (char) c - ('A' - 'a');
```

Знаки операций над объектами

Объекты в языке Java могут объединяться с помощью следующих знаков операций: =, ==, !=, instanceof. Как правило, все остальные операции, такие как сложение, не имеют никакого смысла с объектами. Исключением служит особый случай сложения двух строк.

Знак операции присваивания используется для присвоения указателя на объект ссылочной переменной. При этом новой копии объекта не создается - просто ссылочная переменная начинает указывать на другой объект. Когда все созданные таким образом ссылки на объект потеряют силу (например, когда закончатся области действия всех соответствующих переменных), данный объект будет уничтожен сборщиком мусора. Предположим, что у нас есть класс foo и программа содержит следующую цепочку операторов присваивания:

```
foo test = new foo();
foo test2 = null;
test2 = test;
```

В этом примере показаны допустимые варианты потребления операторов присваивания с объектами. В первой строке оператор присваивания объединен с созданием нового экземпляра объекта. Во второй строке создаваемой ссылочной переменной test2 присваивается null (нулевой указатель). Это означает, что теперь test2 не указывает ни на какой объект и любая попытка использовать эту переменную приведет к возникновению исключительной ситуации NullPointerException. В последней строке значение переменной test присваивается переменной test2. Теперь эти две переменные указывают на один и тот же экземпляр объекта.

К объектам применимы две операции сравнения: проверка на равенство (==) и на

неравенство (!=). Фактически эти операции проверяют равенство не объектов, а указателей на них, то есть возвращают свое значение в зависимости от того, указывают ли сравниваемые переменные на один и тот же объект в памяти. Никакого сравнения отдельных компонентов объекта при этом не производится. Это значит, что два объекта с одним и тем же содержимым, но являющиеся разными экземплярами класса, не будут равны друг другу с точки зрения операции сравнения. Пусть, например, мы имеем два экземпляра класса foo, определенные следующим образом:

```
foo test = new foo();
foo test2 = new foo();
foo test3 = test;
```

Теперь выпишем в виде таблицы результаты проверки на равенство этих трех ссылочных переменных. В табл. 4-11 на пересечении строки и столбца, соответствующих сравниваемых переменным, стоит тот из знаков операций, который возвращает для этих переменных значение true.

Таблица 4-11. Равенство объектов

	test	**test2**	**test3**
test	==	!=	==
test2	!=	==	!=
test3	==	!=	==

Операция instanceof используется для определения типа объекта во время выполнения программы. К ней приходится прибегать, поскольку другим способом определить тип ссылочной переменной при выполнении программы иногда бывает невозможно. К примеру, представьте, что у вас есть класс под названием shape, реализациями которого являются различные геометрические формы. Подкласс этого класса, предназначенный для хранения многоугольников, называется polygonShape. Если у вас есть переменная, являющаяся экземпляром класса shape, определить с гарантией, является ли этот объект многоугольником, можно с помощью операции instanceof:

```
shape shapeHolder;
if (shapeHolder instanceof polygonShape) {
        polygonShape polygon = (polygonShape) shapeHolder;
        // объект является многоугольником, и с ним можно производить
соответствующие
        // действия
        ...
}
```

В этом примере мы имеем экземпляр некоего обобщенного класса shape. Если этот объект принадлежит в то же время к конкретному классу polygonShape, то, согласно алгоритму, с ним требуется произвести некие действия. Чтобы получить доступ, например, к функциям - членам класса polygonShape, мы должны не только определить, принадлежит ли наш объект к этому типу, но и явным образом преобразовать ссылочную переменную к типу ссылки на polygonShape.

Такого рода ситуации встречаются довольно часто, особенно в тех случаях, когда в программе объявлены структуры данных, содержащие объекты, которые являются подклассами некоего общего класса-родителя. Представьте, что вы пишете векторный графический редактор, используя объектно-ориентированный подход, и геометрические фигуры, из которых слагается создаваемый пользователем рисунок, хранятся в некоей структуре данных. Чтобы напечатать рисунок, вам нужно будет использовать цикл, рассматривающий все компоненты этой структуры данных и печатающий каждый из них по очереди. Если каждая из геометрических форм требует применения особых инструкций для вывода на печать, то для сортировки форм по типам нам придется прибегнуть к операции instanceof.

Операции над строками

В разделе "Строки" ниже в этой главе мы будем подробно рассматривать этот тип данных и увидим, что строки в Java представляют собой некий гибрид примитивных типов и объектов. С точки зрения пользователя строка выглядит чаще всего как объект, однако компилятором предусматривается несколько особых ситуаций, в которых строки ведут себя как переменные

примитивных типов. Эта двойственность, хотя и была введена с целью облегчить операции со строками, часто сбивает с толку неопытных пользователей и является причиной ошибок.

Как мы говорили выше, объекты в Java не могут быть операндами для встроенных знаков операций, таких как + или -. Как правило, применение этих операций к объектам, даже если бы и было возможно, не имело бы никакого смысла. Однако существуют некоторые объекты (например, сложные конструкции, состоящие из чисел, такие как матрицы или комплексные числа), для которых удобно было бы иметь свои арифметические операции, желательно определенные с помощью тех же знаков операций. В других языках для этой цели применяется совмещение знаков операций, которое позволяет менять алгоритм действия операции в зависимости от типов операндов. Это позволяет использовать стандартные знаки операций, такие как + или -, для действий над составными объектами. Однако разработчики языка Java пришли к выводу, что такая возможность сделала бы Java-программы более трудными для написания, отладки и сопровождения. Поэтому в Java совмещение знаков операций отсутствует. Конечно, согласиться с правильностью этого решения трудно.

Один из самых частых случаев совмещения знаков операций - это определение операций для работы со строками. Поэтому в качестве компромисса авторы Java предусмотрели совмещение знака операции сложения, который благодаря этому может использоваться для конкатенации (сложения) строк. Если хотя бы один из двух операндов операции сложения является строкой, результат так же будет строкой. Это значит, что второй операнд, если он не принадлежит к типу String, будет искусственно приведен к этому типу. Правила преобразования операндов различных типов в строки суммированы в табл. 4-12.

Таблица 4-12. Правила преобразования нестроковых значений в строки для конкатенации

Операнд	Правило
Пустая переменная	Любая ссылочная переменная, не указывающая ни на какой объект, преобразуется в строку "null".
Целочисленное значение	Преобразуется в строку, содержащую десятичное представление данного целого числа, возможно, со знаком минус впереди. Возвращаемое значение никогда не начинается с символа "0" за единственным исключением: если преобразуемое значение равно 0, возвращается строка из одного символа "0".
Значение с плавающей точкой	Преобразуется в строку, представляющую данное число в компактной записи. Это значит, что если представление числа занимает более десяти символов, число будет преобразовано в экспоненциальную форму. Для отрицательных чисел возвращаемая строка начинается со знака минус.
Одиночный символ	Преобразуется в эквивалентную строку длиной в один символ.
Булевское значение	Преобразуется в одну из двух строк символов - "true" или "false", в зависимости от преобразуемого значения.
Объект	Для преобразования в строку вызывается метод toString(), определенный в данном объекте.

После того как оба операнда преобразованы к строковому типу, они конкатенируются. Вот несколько примеров того, как это происходит:

```
String foo = "Hello ";
String bar = "World";
int i = 42;
boolean cont = false;
String result = null;
result = foo + bar; // = "Hello World"
result = foo + i; // = "Hello 42"
result = foo + cont; // = "Hello false"
```

Как видите, использование знака операции плюс со строками весьма удобно. Однако зададимся вопросом: если знак операции плюс имеет со строками такое значение, то что должен в аналогичной ситуации делать знак операции минус? Ответ прост - он не делает ничего. А как насчет операций сравнения == и !=? Давайте проведем такой эксперимент:

```
String foo = "Hello";
String bar = "Hello";
if (foo == bar) System.out.println ("Равно");
else System.out.println ("Не равно");
```

В результате этой последовательности операторов на выходе вы получите строку "Равно". На первый взгляд все совершенно правильно - ведь наши строки в действительности равны друг другу. Однако давайте вспомним, как работает операция равенства для ссылочных переменных, указывающих на объекты. Как вы уже знаете, эта операция проверяет лишь, действительно ли сравниваемые объекты находятся в одном и том же месте памяти, а не то, равны ли они друг другу в каком-то ином смысле. Вот еще один пример применения знака операции равенства к строковым значениям.

```
class testString {
        String st = "Hello";
}
class testString2 {
        String st = "Hello";
        String st2 = "Hello";
        public static void main(String args[]) {
                testString test = new testString();
                testString2 test2 = new testString2();
                if (test.st == test.st2) System.out.println ("Равно");
                else System.out.println ("Не равно");
                if (test.st == test2.st) System.out.println ("Равно");
                else System.out.println ("Не равно");
        }
}
```

На сей раз результат может показаться неожиданным. Первое сравнение дает результат "Равно", а второе - "Не равно". Дело здесь в том, что компилятор в подобных случаях производит оптимизацию кода с целью уменьшения занимаемой программой памяти. В частности, переменные st и st2, объявленные в пределах одного класса, на самом деле указывают на один и тот же экземпляр объекта в памяти - увидев, что вы заводите две одинаковые строки, компилятор решает сэкономить и поместить в код только один экземпляр этой строки, на который будут ссылаться две строковые переменные. Вот почему, оказывается, операция сравнения со строками иногда работает правильно, а иногда - нет.

Мораль проста: операцию сравнения == нельзя использовать для сравнения двух строк. Вместо этого вы должны пользоваться методом equals, определенном в классе String. Используя приведенное выше объявление переменных, мы могли бы переписать метод main, чтобы получить верный результат сравнения, следующим образом:

```
public static void main(String args[]) {
                testString test = new testString();
                testString2 test2 = new testString2();
                if (test.st.equals(test.st2))
                        System.out.println ("Равно");
                else
                        System.out.println ("Не равно");
                if (test.st.equals(test2.st))
                        System.out.println ("Равно");
                else
                        System.out.println ("Не равно");
        }
}
```

Подробнее о классе String мы будем говорить в разделе "Строки" в этой главе, а также в главе 6.

Пакеты

Пакеты - это инструмент Java, предназначенный для организации содержимого программ. Пакет, как правило, представляет собой группу связанных по смыслу классов и интерфейсов. С одним из стандартных пакетов вы уже знакомы - это пакет java.lang. В этом пакете определено большинство стандартных функций языка. Классы Интерфейса прикладного программирования (API) также сгруппированы в пакеты. Таким образом, пакеты - гибкий и удобный инструмент, позволяющий создавать библиотеки кода для повторного использования в будущем.

Содержимое пакета может храниться в одном или в нескольких файлах. Каждый такой файл

должен начинаться с декларации пакета, к которому он принадлежит. В каждом файле может содержаться только один общедоступный класс. При компиляции этих файлов получающиеся в результате файлы с расширением .class будут помещены в каталоге, соответствующем имени пакета, все точки в котором заменены на символы /. Например, если нам нужно создать пакет, скомпилированные файлы которого будут размещаться в каталоге ventana/awt/shapes, то каждый из исходных файлов, входящих в этот пакет, должен начинаться со следующего объявления:
package ventana.awt.shapes;

Основное назначение пакетов - создание библиотек кода. Об этом мы будем подробно говорить в главе 10, "Структура программы".

Импорт

Допустим, у нас есть пакет. Как получить доступ к входящим в него классам и интерфейсам? Один из способов - использование полного имени нужного класса. Предположим, что мы реализовали упоминавшийся выше пакет ventana.awt.shapes и что этот пакет содержит два класса - circle и rectangle. Если нам потребуется создать новый экземпляр класса circle, то это можно сделать с помощью следующего выражения:
ventana.awt.shapes.circle circ = new ventana.awt.shapes();

Однако доступ к классам через их полные имена не особенно удобен. Существует более экономный способ - использование оператора import для импортирования содержимого пакетов. Импортировав таким способом те или иные классы или интерфейсы пакета, вы получаете возможность обращаться к ним по их кратким именам, без приписывания имени пакета. Например, вот как осуществляется импорт класса circle из пакета shapes:

```
import ventana.awt.shapes.circle;
class tryShapes {
        public static void main(String args[]) {
        circle circ = new circle();
        }
}
```

Как видите, эта возможность позволяет уменьшить объем текста программ и сделать его более удобочитаемым. С помощью этого механизма обычно осуществляется доступ к стандартным средствам языка Java - сначала вы импортируете нужный пакет, а затем пользуетесь его классами и интерфейсами.

Существует способ еще более сокращенной записи оператора import. Если вы собираетесь пользоваться большим количеством классов из какого-либо пакета, запись каждого из них в операторе import потребовала бы много места и сил. Вместо этого можно пользоваться символом *, который, будучи поставлен вместо имени класса или интерфейса в операторе import, заставляет Java импортировать все классы и интерфейсы из данного пакета. Так, чтобы получить доступ ко всему содержимому пакета shapes, можно написать следующий оператор:
import ventana.awt.shapes.*;

При этом можно не беспокоиться о бесполезном увеличении размера скомпилированной программы. Такой оператор импортирования загружает содержимое пакета только в символьную таблицу компилятора - нечто вроде большого словаря, с которым компилятор сверяется каждый раз, когда встречает в программе какой-либо идентификатор. Если на какой-то из импортированных классов ссылок не было, то он не включается в скомпилированный код. Таким образом, выбор между полной и сокращенной формой оператора import определяется исключительно соображениями удобства и экономии времени.

Вниманию пользователей C/C++

Оператор import напоминает директиву компилятора #include, применяющуюся в C/C++. Однако важной отличительной чертой оператора import является то, что сам по себе он не генерирует кода - вы никогда не сможете увеличить размер скомпилированной программы, просто добавляя в нее операторы import. В противоположность этому директива #include эквивалентна вставке в текущий файл содержимого другого файла, который вполне может генерировать при компиляции некий код. В языке Java подобная операция невозможна в принципе - вы не можете просто копировать функции из одного файла в другой, а можете пользоваться только механизмом наследования.

Классы

В приводимых выше примерах мы с вами уже не раз создавали новые классы. Теперь настало время познакомиться с формальным определением синтаксиса классов. Этот раздел не только познакомит вас с некоторыми новыми свойствами классов, но и будет служить справочником, к которому вы сможете обращаться по мере дальнейшего изучения языка. Если у вас вызывает затруднение понятие объекта, перечитайте главу 3, "Объектная ориентация в Java". Собственно говоря, без хорошего понимания, что такое объекты и зачем они нужны, вряд ли стоит двигаться дальше.

Класс - это основной строительный блок Java-программ. Любой класс состоит из данных и методов. Методы, входящие в каждый конкретный класс, как правило, определяют способы изменения и доступа к данным класса. Объединение в одном контейнере как самих данных, так и алгоритмов работы с ними - одно из ключевых свойств объектно-ориентированного программирования, значительно облегчающее повторное использование и обслуживание программного кода.

Конструкторы

Конструктор в языке Java - это особый метод, который вызывается с целью создания нового экземпляра объекта. Конструктор некоего класса должен иметь то же имя, что и сам класс, и не должен возвращать никакого значения. Класс может иметь несколько конструкторов, но все они должны различаться между собой по количеству и типам параметров. Имена параметров при этом не учитываются - иными словами, вы не можете объявить два конструктора к одному и тому же классу, имеющие одинаковое количество параметров одинаковых типов, но по-разному именованных. Вот несколько примеров объявлений конструкторов:

```
class foo {
        foo() {...}                 // конструктор без параметров
        foo(int n) {...}          // конструктор с одним параметром типа int
        foo(String s) {...}    // конструктор с одним параметром типа String
}
```

В этом примере объявление двух конструкторов с одним параметром допустимо, поскольку у каждого из конструкторов этот параметр имеет свой тип. В то же время вы не сможете объявить еще один конструктор как foo(int i), так как, хотя имя параметра и отличается, тип и количество параметров совпадают с одним из уже объявленных конструкторов.

Деструкторы

Любой класс может иметь один деструктор - метод, который вызывается в тот момент, когда объект становится доступным для сборщика мусора. Строго говоря, вы не можете точно предсказать момент, в который будет вызван деструктор. В деструктор удобно поместить такие действия, как закрытие файлов, отключение от сети и т. п. В то же время деструктор не должен производить никакого взаимодействия с пользователем или с другими объектами программы, поскольку вы не можете знать, какие из этих объектов будут доступны в момент вызова деструктора.

Деструктор в языке Java должен иметь имя finalize. Он не имеет параметров и не возвращает никакого значения. Так, к приводившемуся выше в качестве примера классу foo можно добавить деструктор следующим образом:

```
class foo {
        finalize() {...}         // действия, которые нужно выполнить перед
уничтожением объекта
 }
```

Модификаторы классов

В объявлениях класса можно использовать три модификатора - abstract, final или public. Они должны располагаться перед ключевым словом class. Вот, например, как объявляется класс foo с использованием двух из этих модификаторов:
public final class foo {...}

К общедоступному (public) классу можно получить доступ из других пакетов. Если же класс не объявлен как общедоступный, к нему могут обращаться только классы, входящие с ним в один пакет. В каждом пакете можно объявить только один общедоступный класс - вот почему файл с

исходным текстом может содержать только один общедоступный класс или интерфейс.

Модификатор final означает, что данный класс нельзя расширять, то есть нельзя строить на его основе новые классы. Некоторые из классов и интерфейсов прикладного программирования Java определены с этим модификатором. Например, классы Array и String определены с модификатором final, поскольку они являются гибридными классами - то есть экземпляры этих классов не являются в полном смысле слова объектами, а часть кода этих классов реализована непосредственно в компиляторе. Обычно объявление класса с модификатором final не имеет большого смысла, так как при этом вы теряете возможность определять подклассы данного класса. Тем самым вы лишаетесь преимуществ объектно-ориентированного подхода, и ни вы сами, ни другие люди не смогут пользоваться написанным вами кодом. Однако модификатор final может быть полезным в некоторых случаях, в которых переносимость и возможность наследования являются нежелательными.

Определяя класс с модификатором abstract, вы тем самым сообщаете компилятору, что один или несколько методов этого класса является абстрактными. Абстрактным методом называется такой, который в момент своего объявления не содержит никакого кода; код может добавляться в этот метод позднее в подклассах данного класса, которые унаследуют этот абстрактный метод. Абстрактный класс не может быть реализован (то есть нельзя создать экземпляр данного класса), но его можно расширять, создавая подклассы и заполняя в них абстрактные методы нужными алгоритмами. Подкласс абстрактного класса обязательно должен либо сам быть объявлен абстрактным, либо реализовать все абстрактные методы. Такой подход удобен в тех случаях, когда на ранних этапах работы ясна структура программы, но еще не выработаны конкретные алгоритмы. Если попытаться объявить целый класс (а не метод) абстрактным, такой класс будет называться интерфейсом (interface). Подробнее об интерфейсах говорится в разделе "Интерфейсы" данной главы, а также в главе 3, "Объектная ориентация в Java".

Ключевое слово extends

Отношение наследования реализуется с помощью ключевого слова extends. Любой класс может расширять (или, иными словами, быть наследником) не более одного другого класса. Таким образом, множественное наследование явным образом в языке Java не поддерживается. Тем не менее использование интерфейсов позволяет реализовать некоторые свойства множественного наследования.

У всех объектов в Java есть один общий класс-родитель, который называется Object. Если в спецификации класса не указан класс-родитель, то по умолчанию вновь создаваемый класс становится подклассом класса Object. Для явного указания класса-родителя применяется ключевое слово extends. Так, если мы определили выше класс foo, мы можем создать его подкласс bar следующим образом:

```
class bar extends foo {...}
```

Подкласс наследует все методы и переменные своего класса-родителя. Вы можете переопределить или затенить какие-то из этих переменных и методов, использовав в подклассе соответствующий идентификатор с другим значением. Чтобы при этом получить доступ к затененному идентификатору, можно воспользоваться особой переменной super, которая указывает на класс-родитель, ближайший к данному в иерархии классов. Допустим, что в классе foo есть метод под названием test, а в подклассе bar этот метод затенен созданием другого метода с тем же именем. Чтобы получить доступ к исходному методу test, определенному в foo, нужно прибегнуть к следующей записи:

```
class bar extends foo {
        void test() {
                super.test(); // вызов метода test, определенного в классе-
родителе (foo.test)
                ...
        }
}
```

СОВЕТ Попытка определить "порочный круг" зависящих друг от друга классов приведет к сообщению об ошибке при компиляции. Иными словами, класс Б не может быть подклассом А, если класс А уже определен как подкласс класса Б.

Ключевое слово implements

Класс может являться реализацией одного или нескольких интерфейсов. Интерфейсом называют класс, все методы которого абстрактны. Ключевое слово implements, за которым

следует имя интерфейса, должно стоять последним в объявлении класса. Таким образом, полный синтаксис объявления класса таков:

```
<модификаторы класса> class <имя класса> extends
<имя класса-родителя> implements <имя интерфейса> {...}
```

В этом объявлении все, кроме ключевого слова class и имени самого определяемого класса, является факультативным. Если класс является реализацией интерфейса, он должен заполнить каким-то кодом методы, определенные в данном интерфейсе. Единственным исключением из этого правила является случай, когда сам определяемый класс является абстрактным; при этом конкретная реализация методов интерфейса может быть переложена на подклассы данного класса.

Допустим, у нас есть интерфейс shapeInterface, который содержит два метода - draw и erase. Тогда мы можем определить класс с именем shape, реализующий этот интерфейс:

```
class shape implements shapeInterface {
        void draw() {...}
        void erase() {...}
}
```

Если вы хотите создать класс, реализующий сразу несколько интерфейсов, то имена этих интерфейсов нужно перечислить после ключевого слова implements через запятую. В таком случае создаваемый класс должен реализовать все методы каждого интерфейса. Допустим, мы имеем два интерфейса, называемые shapeInterface и moveableInterface. В этом случае мы можем определить класс dragDrop, реализующий оба этих интерфейса:

```
class dragDrop implements shapeInterface, moveableInterface
{...}
```

Более содержательное обсуждение интерфейсов вы найдете в главе 3, "Объектная ориентация в Java". Синтаксис объявления интерфейсов приведен ниже в этой главе в разделе "Интерфейсы".

Модификаторы объявления переменных

Определяя внутри класса переменные, вы можете воспользоваться некоторыми из модификаторов. Присутствие этих модификаторов изменяет такие свойства переменных, как доступность их из других классов, поведение переменных в условиях многопотоковости, а также то, является ли переменная статической или конечной (final). В объявлениях переменных можно указывать следующие модификаторы: public, private, protected, static, final, transient и volatile.

На доступность переменной из других частей программы влияют модификаторы public, protected, private protected и private. Переменная, объявленная с ключевым словом public, доступна как в том пакете, в котором она объявлена, так и в любом другом пакете. Из всех модификаторов данный накладывает наименьшие ограничения на доступность переменной.

Переменная, объявленная с модификатором protected в некоем классе C, доступна всем классам в данном пакете, а также во всех классах, являющихся подклассом класса C. Иными словами, доступа к этой переменной не имеют те классы, которые не входят в данный пакет и не являются подклассами того класса, в котором эта переменная определена.

Если же переменная в классе C объявлена как private protected, то это означает, что к ней могут получить доступ только подклассы класса C. Другим классам, входящим в тот же пакет, эта переменная недоступна. Таким образом, если вам нужно ограничить сферу действия переменной только самим классом и его подклассами, используйте сочетание ключевых слов private protected.

Наконец, модификатор, сильнее всего ограничивающий доступность переменной, - модификатор private - делает переменную невидимой нигде за пределами данного класса. Даже подклассы данного класса не смогут обращаться к переменной, объявленной как private.

Вот пример, в котором используются все четыре модификатора доступа:

```
class circle {
        public String className;
        protected int x,y;
        private protected float radius;
        private int graphicsID;
}
```

Если переменная объявлена с ключевым словом static, это означает, что данная переменная будет общей для всех реализаций этого класса. Место для такой переменной выделяется во время компиляции, поэтому вам не нужно будет создавать экземпляр класса, чтобы получить доступ к этой переменной. Например, таким образом в классе Math пакета java.lang определена переменная-константа PI. Без какой-либо реализации данного объекта мы можем сразу получить доступ к этой переменной:
System.out.println("PI = " + Math.PI);

Модификатор final говорит о том, что значение данной переменной не может быть изменено. Объявление этой переменной обязательно должно содержать инициализацию - присвоение начального значения, а любая попытка изменить значение переменной в других местах программы приведет к сообщению об ошибке при компиляции. Модификатор final обычно используется в определениях констант. Кроме того, неизменяемые константы обычно имеют модификаторы public и static. Так, в некоем классе foo можно определить константу Answer следующим образом:

```
class foo {
        public static final int Answer = 42;
}
```

Наконец, модификаторы transient и volatile относятся к той части языка, которая отвечает за многопотоковое исполнение программ. Основная цель этих модификаторов - облегчить компилятору оптимизацию многопотокового кода. Переменная, объявленная с ключевым словом transient, не может принадлежать объекту в резидентном состоянии (persistent state). Ключевое слово transient будет использовано для реализации некоторых функций в будущих версиях языка.

Переменная, объявленная как volatile, - это такая переменная, о которой известно, что она может изменяться асинхронно. Переменные, объявленные с этим ключевым словом, будут записываться на свое место в памяти после каждого использования и вновь загружаться по мере необходимости. Ключевые слова transient и volatile зарезервированы для использования в будущем, хотя в программах их можно употреблять уже сейчас. (Подробнее о переменных, объявленных с ключевым словом volatile, вы узнаете в главе 11, "Многопотоковость".)

Модификаторы методов

При объявлении метода могут использоваться модификаторы, перечисленные в табл. 4-13. Из них модификаторы public protected и private действуют точно так же, как и при объявлении переменных, и употребляются для ограничения доступа к методам.

Таблица 4-13. Модификаторы методов

public	protected	private	static
abstract	final	native	synchronized

Модификатор static позволяет сделать метод доступным даже в том случае, когда класс, к которому он принадлежит, не реализован. Методы, объявленные статическими, неявным образом используют также модификатор final - иными словами, переопределить статический метод невозможно. В пределах статического метода вы можете обращаться только к тем членам данного класса, которые также являются статическими.

Абстрактный метод, определенный с модификатором abstract, - это такой метод, который будет реализован не в экземпляре данного класса, а лишь в каком-то из его подклассов. Если хотя бы один из методов класса является абстрактным, этот класс также становится абстрактным и уже не может быть реализован. Если все методы класса являются абстрактными, то такой класс, вероятно, имеет смысл объявить как интерфейс.

Метод, определенный с модификатором final, не может быть переопределен в подклассе данного класса. По сути, тем же свойством обладает и метод, объявленный с модификатором private, - он также не может быть переопределен. Оптимизирующий компилятор, возможно, будет производить встраивание такого метода для повышения скорости работы программы - это значит, что во все места, где данный метод вызывается, компилятор вместо вызова будет копировать сам код метода. При этом за счет увеличения объема программы иногда удается получить заметный выигрыш в скорости. Многие компиляторы C/C++ также пользуются таким методом оптимизации.

Глава 5

Апплет в работе

К настоящему времени вы должны хорошо понимать различие между программированием сверху-вниз и объектно-ориентированным программированием и иметь представление о синтаксисе и семантике языка Java. Ну что ж, пора начинать программировать.

В ближайших главах будут объяснены основы написания апплетов. Начнем мы с того, как расширить класс Applet, и опишем важные обходные методы для получения нужного вам поведения апплета. Мы покажем, как использовать методы класса Applet, чтобы получить изображение и звук из сети. Вы узнаете, как получить параметры из HTML-кода, так что ваши апплеты смогут проявлять различные варианты поведения без необходимости перекомпиляции. Мы объясним, как заставить апплет ответить на действия мыши и ввод с клавиатуры. В заключение мы покажем, как оживить апплеты с помощью потоков и как избавить апплеты от раздражающего мерцания.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Что такое апплет?

Апплет объединяет в себе элементы сложного графического окна с легкостью использования и возможностями работы с сетями. В сущности, он является миниатюрным графическим интерфейсом пользователя, подобно Microsoft Windows или X11, который, как гарантируют разработчики, будет иметь в основном одни и те же функциональные возможности независимо от типа компьютера, им управляющего.

Апплеты очень полезны для написания прикладных программ для Интернет, потому что они могут быть вложены в HTML-документы и выполняться в броузерах Web, допускающих использование языка Java, - например, Netscape Navigator 2.0. Чтобы создать свои собственные апплеты, нужно расширить класс Applet и сослаться на новый класс на Web-странице. Давайте рассмотрим апплет "Hello World", подобный апплету, которым мы занимались в главе 2, "Основы программирования на Java".

Пример 5-1a. Апплет "Hello World".

```
import java.applet.*;
import java.awt.*;
public class HelloWorldApplet extends Applet {
        public void init() {
                resize(250,250);
        }
        public void paint(Graphics g) {
                g.drawString("Hello world!",25,25);
        }
}
```

Апплет "Hello World" расширяет класс Applet, а это означает, что все методы и переменные, доступные классу Applet, доступны и нашему расширению этого класса. К примеру, взяв два из этих методов - init и paint, - мы можем изменить их заданное по умолчанию поведение так, чтобы они делали то, что нам нужно. Рассмотрим HTML-код для Web-страницы, которая содержит апплет "Hello World".

Пример 5-1b. Web-страница "Hello World".

```
<HTML>
<HEAD>
<TITLE>Hello World Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="HelloWorldApplet.class"
WIDTH=250 HEIGHT=250>
</APPLET>
</BODY>
</HTML>
```

СОВЕТ Тег <APPLET> не был включен в существующие на момент написания книги стандарты HTML Консорциума W3 (W3C) - авторитетной группы, разрабатывающей стандарты для WWW. Этот синтаксис в настоящее время используется только броузером Netscape Navigator 2.0 и программой просмотра апплетов фирмы Sun, способными интерпретировать Java. Так что это будет, вероятно, нестандартный тег, существующий де-факто подобно другим, не соответствующим стандарту HTML маркировочным тегам, используемым Netscape, - тегам <CENTER> и <BLINK>. W3C в настоящее время предлагает тег <INSERT> для приложений, вставляемых в Web-страницы; последние новости по этой теме вы можете найти по адресу http://flwww.w3.orglpub/WWW/TR/WD-insert.html.

Параметр CODE внутри тега <APPLET> определяет полный URL-адрес к классу компилируемого апплета - здесь мы допускаем, что эта HTML-страница находится в том же самом каталоге, что и класс Hello World. Обратите внимание, что с помощью атрибутов WIDTH и HEIGHT нужно сообщить броузеру, насколько велик этот апплет, чтобы броузер мог правильно сформировать страницу. Под Netscape Navigator 2.0 вы можете задать для этих атрибутов значение 100%, что заставит броузер давать апплету столько места, сколько ему требуется.

Апплеты в Java стилистически отличаются от прикладных программ на других языках программирования. Java-код в значительной степени является событийно управляемым, подобно гипертексту в Web, вместо обычного линейного потока, как в традиционном программировании. Оболочка времени выполнения - Web-броузер - действует как интерфейс между кодом и компьютером, на котором он выполняется. Эта связь представлена на рис. 5-2.

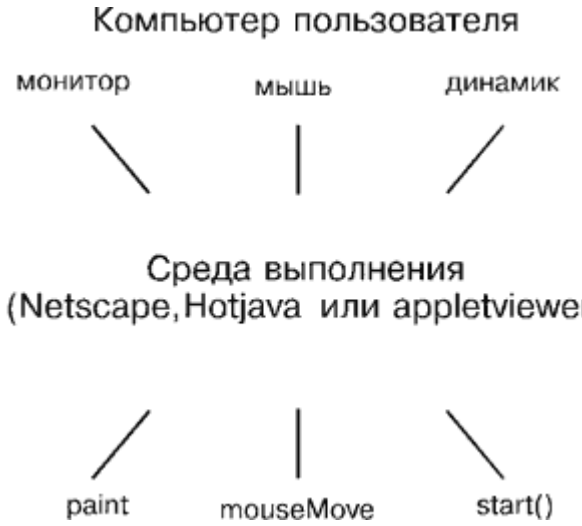

Рис. 5.2.

Оболочка времени выполнения понимает такие методы апплета, как paint и mouseMove, и вызывает их, когда требуется, - например, когда необходимо перерисовать экран или отразить перемещения мыши. По умолчанию при вызове этих методов апплет ничего не делает - программист должен сам переопределять эти методы, если хочет, чтобы апплет ответил на соответствующие события.

Интерфейс AppletContext

Оболочка времени выполнения создает класс Java, который осуществляет интерфейс AppletContext. Интерфейс AppletContext описывает несколько методов, с помощью которых апплеты могут запрашивать ресурсы из оболочки времени выполнения, например методы getImage и getAudioCIip (описанные позже в этой главе). Класс AppletContext наблюдает за событиями и вызывает соответствующие методы для их обработки. Его можно представить как трап между кодом вашего апплета и оболочкой времени выполнения. Апплеты могут запрашивать ссылку на их AppletContext, используя метод getAppletContext. Вам не нужно взаимодействовать с AppletContext непосредственно, если только вы не хотите заставить несколько апплетов общаться друг с другом. Мы опишем, как это делается, в главе 13, "Работа с сетью на уровне сокетов и потоков".

Стадии выполнения апплета

Когда Java-совместимый броузер Web загружает класс Applet, сначала он распределяет память для апплета и глобальных переменных. Затем выполняется метод init. (Вообще, программисты используют метод init, чтобы инициализировать глобальные переменные, получить ресурсы из сети и установить интерфейс пользователя.) После этого броузер вызывает метод start. Если часть броузера, содержащего апплет, видима (что обычно и случается, когда апплет только начинает свою работу), вызывается метод paint. Если пользователь уходит со страницы, содержащей апплет, броузер вызывает метод stop. Когда пользователь возвращается на страницу с апплетом, метод start, так же как и метод paint, вызывается снова. Следующий фрагмент кода иллюстрирует работу апплета в случае, если пользователь покидает страницу и затем возвращается на нее.

Пример 5-2. Апплет, считающий обращения к странице.

```
import java.applet.*;
import java.awt.*;
public class Count extends Applet {
        int InitCount=0;
        int StartCount=0;
        int StopCount=0;
        int PaintCount=0;
        public void init() {
                resize(250,75);
                InitCount = InitCount + 1;
        }
        public void start() {
                StartCount = StartCount + 1;
        }
        public void stop() {
                StopCount = StopCount + 1;
        }
        public void paint(Graphics g) {
                PaintCount++;
                String Output = new String(
                        "Inits: "+InitCount+
                        " Starts: "+StartCount+
                        " Stops: "+StopCount+
                        " Paints: "+PaintCount);
                g.drawString(Output,25,25);
        }
}
```

Вывод апплета после первого запуска показан на рис. 5-3. Апплет один раз был инициализирован, один раз запускался, ни разу не останавливался и, по крайней мере, один раз был перерисован.

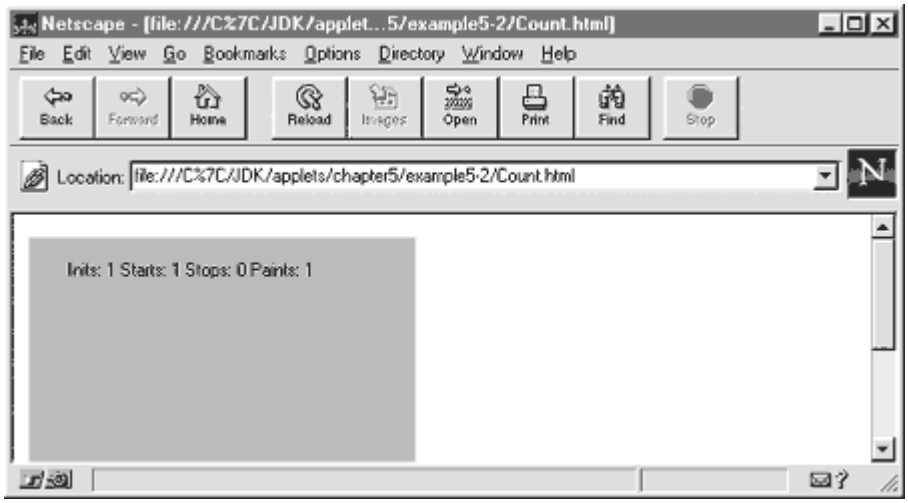

Рис. 5.3.

Если вы перейдете к другой Web-странице и затем возвратитесь обратно (без выхода из броузера), вы увидите, что апплет все еще был инициализирован только один раз, но запускался два раза, один раз останавливался и, по крайней мере, дважды перерисовывался. Это отражено на рис. 5-4.

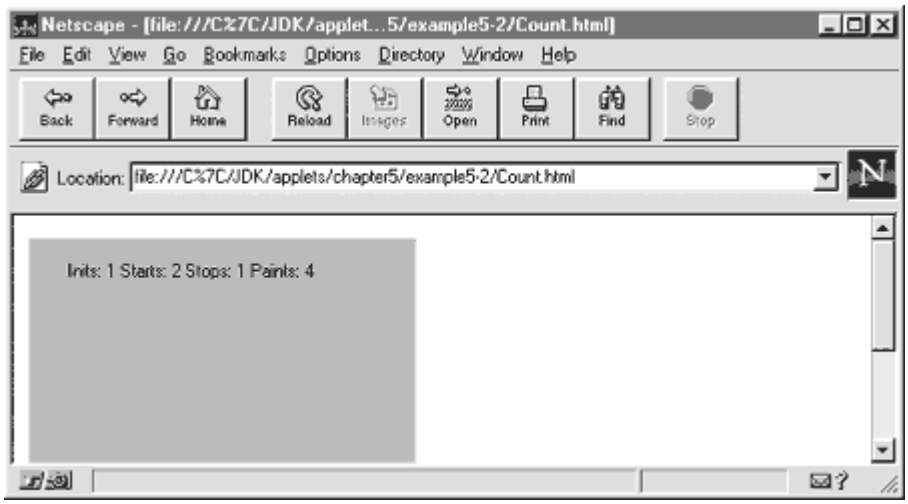

Рис. 5.4.

СОВЕТ Нажатие кнопки Reload в Netscape Navigator 2.0 заставит апплеты на текущей странице остановиться и запуститься снова.

Если вы заслоняете апплет, перемещая другое окно поверх окна с апплетом, а затем снова переводите его наверх, делая активным, вы обнаружите, что апплет не был запущен и не останавливался, но окно с ним было перерисовано. В табл. 5-1 приведены некоторые важные методы класса Applet.

Таблица 5-1. Основные методы класса Applet

Метод	Описание
init ()	Вызывается один раз при первой загрузке кода апплета.
start ()	Вызывается всякий раз, когда Web-страница, содержащая апплет, становится в броузере активной.
stop ()	Вызывается, когда Web-страница, содержащая апплет, больше не активна в броузере.
destroy ()	Вызывается при явном уничтожении апплета.
paint (Graphics g)	Вызывается, когда апплет должен повторно вывести графическое окно.

Замещение метода destroy

Апплеты могут замещать метод destroy, который вызывается после метода stop при явном уничтожении апплета. Метод destroy предназначен для освобождения ресурсов, которые использовал апплет. На практике этот метод замещается редко, потому что Java самостоятельно освобождает используемые ресурсы. Как только апплет уничтожается, все переменные в памяти теряют ссылки времени выполнения и становятся мусором, подчиняющимся встроенному сборщику мусора. Подпрограммы сборки мусора выполняются в фоновом режиме, когда система считает это необходимым, так что нет никакой реальной потребности в том, чтобы самостоятельно освобождать ресурсы. Замещение метода destroy может быть полезным в том случае, если, например, вам нужно удостовериться в том, что пользователь действительно хочет уничтожить апплет.

Интересы защиты информации сдерживают разработку апплетов в некоторых отношениях. Известно, что апплеты неспособны как-либо обращаться к локальному жесткому диску пользователя. Таким образом, любые большие объемы данных, необходимые вашим апплетам, должны быть получены из файлов компьютера-сервера через сеть, о чем мы будем говорить в следующем разделе.

Доступ к ресурсам

Одна из причин популярности World Wide Web - легкость, с которой авторы могут добавлять к своим Web-страницам изображения и звук, просто включая в код страницы указатели на местоположение графических и звуковых файлов, которые они хотят использовать. Использование языка Java дает еще более простой и намного более мощный способ.

HTML - язык описания документа; Java - добротный язык программирования. Ваши Java-апплеты могли бы использовать изображения как графические пиктограммы или спрайты в игре аркадного стиля. Следующий Java-апплет принимает из сети файл с изображением и звуком и отображает их.

Пример 5-3. Апплет для Web.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class WebApplet extends Applet {
        private Image myImage;
        private AudioClip mySound;
        private URL ImageURL;
        private URL SoundURL;
        public void init() {
                resize(250,250);
                try {
                        // привязываем URL к ресурсам
                        ImageURL = new
URL("http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/sample.gif");
                        SoundURL = new URL("http://www.vmedia.com/vvc/onlcomp/
java/chapter5/sounds/sample.au");
                }
                // следим за правильностью URL
                catch (MalformedURLException e) {}
                // загружаем изображение
                myImage = getImage(ImageURL);
                // загружаем звук
                mySound = getAudioClip(SoundURL);
        }
        public void start() {
                // запускаем проигрывание звука
                mySound.loop();
        }
        public void stop() {
                // останавливаем проигрывание звука
                mySound.stop();
        }
        public void paint(Graphics g) {
                // выводим изображение
                g.drawImage(myImage,0,0,this);
        }
}
```

В этом фрагменте кода присутствуют ссылки на три класса, которые могут быть вам незнакомы: java.awt.Image, java.applet.AudioClip и java.net.URL. Эти классы подобно большинству классов, определенных в Java API, делают более или менее то, что подразумевают их имена.

Класс Image

Класс Image определяет простое двумерное графическое изображение. Класс Graphics (используемый методом paint) может выводить изображения с помощью метода drawImage, как показано в следующем примере:

```
Image myImage;
myImage = createImage(50, 50);
```

```
g.drawImage(myImage, 0, 0, this);
```

Метод createImage определен для класса jawa.awt.Component, родителя класса Applet. Этот метод воспринимает два аргумента типа int и создает новое место для изображения с заданными размерами.

Метод drawImage принимает четыре параметра: непосредственно изображение, координаты расположения изображения в окне и класс ImageObserver. Мы подробно опишем этот класс в главе 9, "Графика и изображения", а сейчас просто подставьте ваш апплет непосредственно в качестве параметра ImageObserver при использовании drawImage.

СОВЕТ Если вам надо передать апплет как параметр, вы можете использовать ключевое слово this, о котором говорилось в главе 4, "Синтаксис и семантика".

Класс Applet использует метод getImage(URL) для получения изображения из сети. Этот метод выполнен с помощью интерфейса AppletContext (см. врезку "Интерфейс AppletContext" выше). Следовательно, Java-апплеты могут импортировать изображения любого графического формата, поддерживаемого Web-броузером. Наиболее часто используемые форматы - GIF и JPEG. Для загрузки изображениям обычно нужно некоторое время, но мы можем идти вперед и уже выводить изображения в нашем апплете; но фактически изображения появятся не сразу, а через некоторое время. Если вам нужен более полный контроль над изображениями, вы можете использовать класс MediaTracker, о котором мы будем говорить в главе 9.

Класс AudioClip

Класс java.applet.AudioClip - это представление высокого уровня для звуковых данных. В этом классе определены три метода: play, loop и stop. Класс апплета определяет метод getAudioClip, который по заданному URL возвращает звуковой файл. Подобно методу getImage, метод getAudioClip фактически выполняется контекстом апплета; так, апплет способен использовать любой звуковой формат, поддерживаемый броузером. К примеру, следующий фрагмент кода:

```
AudioClip mySound;
mySound.play();
```

проигрывает звук, представляемый данным AudioClip. Класс Applet определяет метод play(URL), который, когда ему передан URL звукового файла, проигрывает этот файл. Если звуковой файл отсутствует или его формат не поддерживается, ничего не произойдет. Класс AudioClip, подобно Image, может использоваться, как только звук затребован, но ему может понадобиться некоторое время, чтобы фактически загрузить и проиграть звук. К сожалению, класс MediaTracker, с помощью которого можно загрузить изображения до того, как они потребуются, еще не поддерживает AudioClip.

Класс URL

URL, или Uniform Resource Locators ("унифицированная ссылка на ресурс"), - полный адрес объекта в World Wide Web (например, http://www.vmedia.com/index.htall - адрес Web-страницы узла Ventana Online). В языке программирования Java есть отдельный класс для обработки URL. Экземпляр класса представляет собой объект в Web. Класс URL будет описан полностью в главе 14, "Работа с сетью на уровне URL", но вы можете начинать использовать его уже теперь. Самый простой способ создавать объект URL состоит в использовании конструктора URL (String):

```
URLmyObject;
myQbject = new URL ("http://www.vmledia.com/index.html");
```

К сожалению, этот фрагмент кода не завершен. Если бы мы пробовали его откомпилировать, Java-компилятор пожаловался бы на то, что мы не сумели обработать исключительную ситуацию MalformedURLException. URL может быть очень сложным, однако легко создать объект URL со строкой, которая напоминает URL, но им не является. Если это случится, конструктор URL потерпит неудачу и сообщит апплету, что строка URL, которую он проанализировал, неправильна. Мы должны быть подготовлены к этому обстоятельству и перехватывать ошибку, что и делается в следующем фрагменте кода:

```
URL myObject;
try {
 myObject = new URL ( "http: //www.vmedia.com/index.html");
 }
 catch (MalformedURLException e) {
 // код здесь выполняется, если строка URL неправильна
}
```

Мы будем подробно описывать исключения в главе 10, "Структура программы", так что не волнуйтесь, если пока это вам не очень непонятно. Только не забудьте, что, создавая новый URL, вы должны стараться перехватывать исключения. Если вы абсолютно уверены, что у вашего URL синтаксис правильный, вы можете не помещать никакого кода между вторыми фигурными скобками.

Есть еще другой важный конструктор URL, который получает URL и строку как параметры. URL указывает абсолютный базовый URL, а строка содержит путь к объекту относительно этого базового указателя. Например, если вы назначили http://www.vmedia.com/ourbook/ как URL и строку "images/pictures.gif", новый URL укажет на http://www.vmedia.com/ourbook/images/picture.gif. Если же вы укажете строку "/otherbook/index.html", новый URL укажет на http://www.vmedia.com/otherbook/index.html.

Этот конструктор полезен вместе с методом getCodeBase класса Applet, который возвращает URL файла класса Applet. Вы можете использовать getCodeBase и относительный конструктор URL, чтобы создавать URL на объекты без определения базового имени. Это особенно полезно, потому что апплетам в целях безопасности не позволено открывать сетевые соединения с удаленными главными компьютерами, за исключением того Web-сервера, с которого файл этого класса был загружен (мы обсудим эти ограничения более полно в главе 13, "Работа с сетью на уровне сокетов и потоков"). Итак, вы должны установить ресурсы, которые будет использовать ваш апплет, на том же Web-сервере, что и сам апплет. При построении URL к этим ресурсам в апплете просмотра самое простое и самое безопасное - использовать метод getCodeBase с относительным конструктором URL вместо того, чтобы использовать абсолютный URL. Это упростит и установку вашего апплета на новом Web-сервере.

Как вы можете заметить, мы вставили в наш пример апплета некоторые данные, а именно URL изображения и звукового файла. Если вы профессиональный программист или профессор информатики, вы, вероятно, отметили это и поморщились - для устоявшихся стандартов программирования это неприемлемо. Всякий раз, когда вы захотите изменить изображение или звуковой файл, используемые этим апплетом, вы должны будете изменить код и перекомпилировать этот класс. К счастью, в Java есть способ передавать параметры апплетам во время выполнения, что можно использовать для определения различных графических или звуковых файлов.

Доступ к параметрам

Особенность хороших языков высокого уровня - способность получать параметры из командной строки. Программисты используют это, чтобы изменить поведение программ, основываясь на вводимых пользователем данных, что уменьшает потребность в сложном пользовательском интерфейсе. Но Java-апплеты запускаются не из командной строки, поскольку они, как и их параметры "командной строки", являются вложенным внутренним HTML-кодом. Рассмотрим следующую Web-страницу.

Пример 5-4а. Web-страница с параметрами.

```
<HTML>
<HEAD>
<TITLE>Good Web Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="GoodWebApplet.class" WIDTH=250 HEIGHT=250>
<PARAM NAME="IMAGE"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/sample.gif">
<PARAM NAME="SOUND"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/sounds/sample.au">
</APPLET>
</BODY>
</HTML>
```

В этом фрагменте кода мы имеем вложенную переменную с именем IMAGE и значением ".. /images/sample.gif". Апплет может обращаться к переменной IMAGE, используя метод getParameter, который получает имя переменной в качестве параметра и возвращает строку, содержащую значение переменной. То же верно и для параметра SOUND. Все параметры представляются как строки.

СОВЕТ Из главы 6, "Интерфейс прикладного программирования", вы узнаете, как преобразовать переменные строкового типа к другим типам. Это вам пригодится, если вы захотите получить в качестве параметра число.

Используя эти новые функциональные возможности, мы можем переписать метод init нашего апплета для Web так, чтобы сделать его независимым от данных.

Пример 5-4b. Усовершенствованный апплет для Web.

```
public void init() {
                String ImageParam;
                String SoundParam;
                resize(250,250);
                // значения берутся из HTML-файла
                ImageParam = getParameter("IMAGE");
                SoundParam = getParameter("SOUND");
                try {
                        // используем параметры, чтобы взять URL
                        ImageURL = new URL(ImageParam);
                        SoundURL = new URL(SoundParam);
                }
                catch (MalformedURLException e) {}
                myImage = getImage(ImageURL);
                mySound = getAudioClip(SoundURL);
        }
```

Позволив определять расположение изображения и звукового файла автору Web-страницы, а не программисту, мы сделали апплет намного более полезным. Все, что автор должен сделать, чтобы поменять изображение или звук, используемые апплетом, - это изменить метку VALUE любого параметра. Когда у вас есть апплет, показывающий часто меняющиеся данные, или вы хотите, чтобы авторы Web-страниц могли легко изменять заданное по умолчанию поведение апплета, вы должны использовать соответствующие параметры.

Какими бы полезными ни были эти параметры, они позволяют апплетам взаимодействовать с пользователем только одним довольно узким способом - через выбор строк, записанных до начала выполнения программы. Но как же изменить поведение апплета во время его выполнения - когда пользователь перемещает мышь или нажимает какую-то клавишу? Такая реакция необходима для сложных прикладных программ, и обработка событий в языке Java облегчает выполнение этой задачи.

Взаимодействие с пользователем

Чтобы взаимодействовать с пользователем в реальном масштабе времени и отслеживать изменения в оболочке времени выполнения Java, апплеты используют понятие события (event). Событие - это информация, сгенерированная в ответ на некоторые действия пользователя (перемещение мыши или нажатие клавиши на клавиатуре). События также могут быть сгенерированы в ответ на изменение среды - к примеру, когда окно апплета заслоняется другим окном. Оболочка времени выполнения следит за происходящими событиями и передает информацию о событии особому методу, называемому обработчиком события (event handler). Многие обычно используемые обработчики событий предопределены для класса Applet. По умолчанию эти обработчики не делают ничего - чтобы использовать их, надо заместить соответствующий метод своим собственным кодом, например:

```
public boolean mouseMove(Event evt, int x, int y) {
 // код, расположенный здесь, выполняется, если переместилась мышь
 return true; // событие обработано
}
```

Метод mouseMove вызывается всякий раз, когда перемещается мышь. Обработав событие, он возвращает true.

События, генерируемые мышью

Обработчику событий mouseMove передаются три параметра: непосредственно событие, которое является классом, содержащим всю информацию, нужную для уникальной идентификации события, и две координаты - в данном случае новое расположение мыши внутри апплета. В табл. 5-2 приведены предопределенные обработчики событий.

Таблица 5-2. Часто используемые предопределенные обработчики событий

Обработчик событий	Описание
mouseDown (Event, int, int)	Нажата кнопка мыши. Целочисленные параметры указывают расположение мыши.
mouseUp (Event, int, int)	Кнопка мыши отпущена.
mouseMove (Event, int, int)	Перемещение мыши.
mouseDrag (Event, int, int)	Перемещение мыши с нажатой кнопкой.
mouseEnter (Event, int, int)	Перемещение мыши на окно апплета.
mouseExit (Event, int, int)	Мышь покинула окно апплета.
keyDown (Event, int)	Нажата клавиша перемещения курсора или функциональная клавиша. Целочисленный параметр указывает эту клавишу (см. табл. 5-3).
keyUp (Event, int)	Клавиша перемещения курсора или функциональная клавиша отпущена.

Замещая некоторые из этих предопределенных обработчиков событий, мы можем написать апплет отображения положения курсора показанный ниже. Этот апплет рисует изображение курсора, следуя за мышью в окне апплета. Такое поведение может быть выключено и снова включено нажатием кнопки мыши.

Пример 5-5. Апплет отображения курсора.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class CursorApplet extends Applet {
        // расположение мыши
        private int mouse_x, mouse_y;
        // хотите следовать за мышью?
        private boolean Follow = true;
        private Image CursorImage;
        public void init() {
                mouse_x = 125;
                mouse_y = 125;
                resize(250,250);
                String CursorFile = getParameter("CURSORFILE");
                try {
                        URL CursorURL = new URL(CursorFile);
                        CursorImage = getImage(CursorURL);
                }
                catch (MalformedURLException e) {
                        CursorImage = createImage(0,0);
                }
        }
        public void paint(Graphics g) {
                // простая рамка
                g.drawRect(0,0,249,249);
                // прорисовка курсора в месте расположения мыши
```

```
g.drawImage(CursorImage,mouse_x,mouse_y,this);
        }
        public boolean mouseMove(Event evt, int x, int y) {
                if (Follow) {
                        // обновление информации о местоположении
                        mouse_x = x;
                        mouse_y = y;
                        // перерисовка окна
                        repaint();
                }
                return true;
        }
        public boolean mouseDown(Event evt, int x, int y) {
                // здесь слежение отключается
                if (Follow) {Follow = false;}
                // а здесь - наоборот
                else {Follow = true;}
                return true;
        }
}
```

Проверяя событие нажатия кнопки мыши, вы можете определять различную обработку в зависимости от того, делает ли пользователь одиночный щелчок или двойное нажатие. Класс Event определяет переменную clickCount, которая установлена в 1 для одиночных нажатий и 2 для двойных. Фактически ей будет присвоено число щелчков, которое пользователь сумеет сделать прежде, чем событие будет сгенерировано. На практике полезны только одиночные и двойные щелчки. Следующий фрагмент кода иллюстрирует использование этой переменной:

```
public boolean mouseDown ( Event evt, int x, int y){ if (evt.clickCount==1) (
// одиночное нажатие
 } elseif (evt.clickCount==2) {
 // двойной щелчок
 } else {
 // слишком быстрые пальчики
 }
return true;
}
```

Этот метод вызывается при нажатии кнопки мыши. Если кнопка нажата один раз, апплет обрабатывает случай с одиночным щелчком; если кнопка нажата дважды, апплет обрабатывает двойное нажатие.

События, генерируемые клавиатурой

Методы keyUp и keyDown работают таким же образом, как и обработчики событий мыши, за исключением того, что им передается идентификатор клавиши, а не координаты события. Идентификатор клавиши - это целое число, которое соответствует нажатой клавише. Обычные клавиши пишущей машинки имеют значения, соответствующие их ASCII-кодам. Кроме того, Java поддерживает множество специальных клавиш, приведенных в табл. 5-3. Эти специальные клавиши фактически определены как статические целые переменные (константы) для класса Event. В главе 6, "Интерфейс прикладного программирования", будет показано, как преобразовать целые числа в строки, если вы захотите отображать ввод с клавиатуры. Следующий фрагмент кода проверяет нажатие клавиш управления курсором:

```
Public boolean keyDown(Event evt, lnt key) {
                switch(key) {
                case Event.UP:
                case Event.DOWN:
                case Event.LEFT:
                case Event.RIGHT:
                default:
        }
        return true;
```

```
}
```

Метод keyDown вызывается при нажатии клавиши. Оператор switch переключается между четырьмя стрелками и всеми другими клавишами и вызывает соответствующий участок кода для рассматриваемой клавиши.

Shift, Ctrl & Meta

События могут быть обработаны по-разному в зависимости от того, нажата или нет какая-то из специальных клавиш наложения маски на происходящее событие. Shift, Ctrl и Meta - клавиши наложения маски в языке Java. Клавиша Meta эквивалентна клавише Alt в Microsoft Windows; под оконной системой X11 она может быть соотнесена с различными клавишами. Класс Event обеспечивает методы shiftDown, controlDown и metaDown, которые возвращают булевскую переменную, указывающую состояние каждой из клавиш наложения маски.

Обработчики событий: что же происходит на самом деле?

Что же действительно происходит при возникновении события? AppletContext обращает внимание на то, что событие произошло внутри апплета (см. врезку "Интерфейс AppletContext" выше в этой главе). AppletContext создает новый экземпляр класса Event, присваивает значения соответствующим параметрам, чтобы апплет знал, какое событие случилось, и передает новое событие методу handleEvent. Класс Event содержит всю информацию, нужную для идентификации события; переменные, описывающие события, перечислены в табл. 5-4.

Таблица 5-4. Переменные события

Переменная	Описание
public Object target	Компонент, в котором произошло событие для апплета; обычно непосредственно сам апплет.
public long when	Время, в которое произошло событие, - целое число типа Long, содержащее число миллисекунд, прошедшее с полуночи 1 января 1970 года по Гринвичу. В Java есть метод java.lang.System.currentTimeMillis, который возвращает это значение.
public int id	Тип события (см. табл. 5-6).
public int x	Координата X события.
public int y	Координата Y события.
public int key	Идентификатор клавиши (для специальных клавиш, см. табл. 5-3).
public int modifiers	Состояние клавиш наложения маски.
public Object arg	Необязательный параметр. Он не используется для событий мыши и клавиатуры, но часто применяется при использовании графики, обеспечиваемой Java API (см. главу 9, "Графика и изображения").

Метод handleEvent проверяет тип события и вызывает соответствующий обработчик, передавая ему относящиеся к делу параметры (диаграмма этого процесса показана на рис. 5-5).

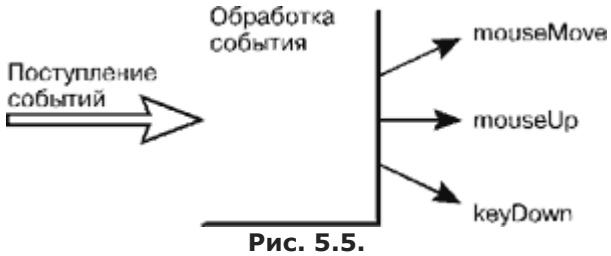

Рис. 5.5.

Например, при перемещении мыши возникает событие MOUSE_MOVE, которое и передается методу handleEvent. Последний вызывает соответствующий этому событию метод mouseMove, передавая ему положение мыши. Фактически передавать положение события в виде отдельных параметров не нужно, потому что эта информация для удобства программирования закодирована непосредственно в событии.

Вы можете заменить метод handleEvent, если хотите обработать большое количество различных событий без замены каждого конкретного метода. Это может быть также полезно, если вы хотите обрабатывать последовательности событий по-другому. Сделав это, не забудьте,

что предопределенные обработчики событий не будут вызываться, если вы не вызовете их явно в вашем новом методе handleEvent. В следующем примере вызывается родовой обработчик для событий мыши, а все другие события передаются оболочке времени выполнения:

```
public boolean handleEvent(Event evt) {
        switch (evt.id) {
                case Event.MOUSE MOUISE_DOWN:
                case Event.MOUSE MOUSE_UP:
                case Event.MOUSE MOUSE_MOVE:
                case Event.MOUSE MOUSE_DRAG:
                case Event.MOUSE MOUSE_ENTER:
        case Event.MOUSE MOUSE_EXIT:
        }
}
```

Вы можете генерировать события в своей программе. Для этого нужно просто создать новый экземпляр класса Event и объявить его в своем апплете (конструкторы событий перечислены в табл. 5-5). Генерация событий может быть полезной для написания системы макросов, которая, например, регистрирует движение мыши. Вы могли бы оформить рисунок, записав действия рисующего пользователя, могли бы скопировать этот рисунок в другое место. Следующий код заставит ваш апплет думать, что мышь передвинулась в новое положение:

```
Event FakeEvt;
long time = System.currentTimeMillis();
FakeEvt = new
Event(this, time, MOUSE_MOVE, new_x, new_y, 0, 0, null); postEvent(FakeEvt);
```

Таблица 5-5. Конструкторы событий

Конструктор события	Описание
Event(Object,long,int, int,int,int,int,Object)	Пункт назначения, время, тип события, x-координата, y-координата, идентификатор клавиши, модификаторы и параметр.
Event(Object, long, int, int, int, int, int)	Пункт назначения, время, тип события, x-координата, y-координата, идентификатор клавиши и модификаторы.
Event(Object, int, Object)	Пункт назначения, тип события и параметр.

Полный список типов событий приведен в табл. 5-6. Типы, отмеченные звездочкой, не относятся к апплетам. Типы SCROLL_ и LIST_ используются вместе с компонентами ввода пользователя, о которых мы будем говорить в главе 7, "Пользовательский интерфейс".

Таблица 5-6. События, отмеченные звездочкой, не используются в апплетах

Типы событий	+
WINDOW_DESTROY	MOUSE_EXIT
WINDOW_EXPOSE	SCROLL_LINE_UP
WINDOW_ICONIFY	SCROLL_LINE_DOWN
WINDOW_DEICONIFY	SCROLL_PAGE_UP
W lNDOW_MOVED	SCROLL_PAGE_DOWN
KEY_PRESS	SCROLL_PAGE_ABSOLUTE
KEY_RELEASE	LIST_SELEGT
KEY_ACTION	LIST_DESELECT
KEY_ACTION_RELEASE	ACTION_EVENT
MOUSE_DOWN	LOAD_FI LE *
MOUSE_UP	SAVE_FILE*
MOUSE_MOVE	GOT_FOCUS
MOUSE_DRAG	LOST_FOCUS
MOUSE_ENTER	

В хорошем Java-апплете большая часть кода пишется для методов, которые выполняют некоторые действия в ответ на события. Но предположим, что мы хотим обрабатывать некоторую последовательность событий независимо от других событий. К примеру, мы хотим, чтобы апплет

в одно и то же время отображал движущийся рисунок и отвечал на события. Если мы начнем цикл, чтобы отобразить мультипликацию в каком-то из наших методов, метод никогда не будет завершен, и апплет зациклится. К счастью, в Java есть очень хороший выход из этой ловушки - потоки.

Анимация при помощи потоков

Представьте себе оболочку времени выполнения как контору с одним диспетчером, который управляет методами апплета в ответ на различные события. Наш диспетчер очень методичен; он должен ждать завершения каждой задачи перед тем, как приступить к следующей. Предположим, что мы хотим дать ему метод, который является настолько сложным или повторяющимся, что это препятствовало бы диспетчеру делать что-нибудь еще. Решение в этом случае просто - нужно нанять второго диспетчера, чтобы он работал над другой задачей. Новый диспетчер - это новый поток.

Потоки отличаются от других классов, так как они, получив управление и начав действовать, выполняются независимо в среде метода, который их запустил. В этом отношении они походят на процессы в многопроцессорных операционных системах, подобных UNIX. Когда вызывается обычный метод, программа ждет, пока он не будет полностью выполнен. Но когда вызывается метод потока (обычно когда поток запущен), метод, вызвавший поток, продолжает выполняться во время выполнения метода потока. На рис. 5-6 показана схема выполнения одиночного потока и двух потоков.

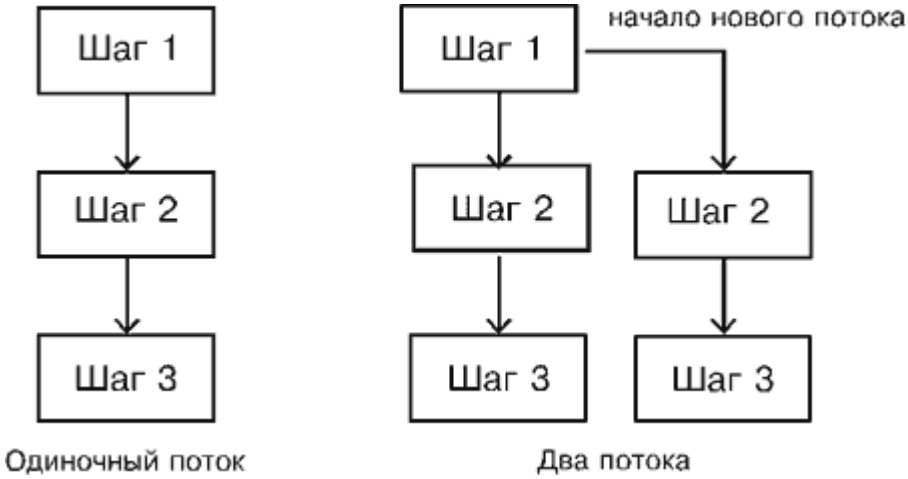

Рис. 5.6.

Вы можете использовать потоки в своих программах на Java, определяя расширения класса Thread. Подробно этот процесс описан в главе 11, "Многопотоковость". Но уже теперь мы объясним, как позволить апплету использовать потоки. Эта методика будет полезна, если, например, апплет отображает движущееся изображение, но вы хотите, чтобы апплет делал и другие полезные вещи во время проигрывания мультипликации.

Интерфейс Runnable

Как обсуждалось в главе 3, "Объектная ориентация в Java", язык Java не учитывает многократное наследование. Апплет не может непосредственно расширять ни класс Applet, ни класс Thread. Вы можете задать метод, который выполняется внутри потока, с помощью интерфейса Runnable. Этот интерфейс, подобно классу Thread, является частью пакета java.lang. Можно выбрать такой путь:

```
class ThreadedApplet extends Applet implements Runnable {
}
```

Интерфейс Runnable сообщает компилятору, что этот класс определит метод, который должен быть выполнен внутри потока. Выполняемый метод управляется экземпляром класса Thread, созданного непосредственно в апплете. Посмотрим, как можно использовать это на практике:

```
class ThreadedApplet extends Applet Implements Runnable {
        private Thread engine = null;
```

```
        // наш поток
        public void init() {
                engine = new Thread(this);
                // этот поток теперь управляет нашим методом
        }
        public void start() {
                englne.start();
                // метод начинает выполняться
        }
        public void stop() {
                if (engine!=null && engine.isAlive()) {
                // если нужно,
                        engine.stop();
                        // останавливаем выполнение метода
                }
        }
        public void run() {
                while (engine.isAlive()) {
                // код, расположенный здесь, выполняется, пока апплет не
остановится
                }
        }
}
```

Когда апплет инициализирован, он создает новый поток, привязанный к апплету. При запуске апплета начинается работа потока, который циклически выполняет свой код в методе run, пока апплет не остановлен.

Простые методы для работы с потоками

Класс Thread определяет множество методов для управления потоками. Мы подробно обсудим их в главе 11, "Многопотоковость". В табл. 5-7 кратко описаны наиболее важные методы, которые надо запомнить.

Таблица 5-7. Общие методы класса Thread

Метод	Описание
isAlive()	Возвращает булевскую переменную, указывающую, является ли поток действующим или нет.
sleep(long)	Просит, чтобы поток бездействовал определенное число миллисекунд. Этот метод выполнит InterruptedException, если получит сигнал прерывания от другого потока. (О прерываниях см. главу 11, "Многопотоковость".)
start()	Начинает выполнение потока.
stop()	Останавливает выполнение потока.
suspend()	Временно приостанавливает поток.
resume()	Продолжает выполнение потока после приостановки.

Используя потоки, мы можем расширить апплет отображения курсора (пример 5-5), показанный на рис. 5-7, чтобы отобразить движущееся изображение, которое следует за курсором. Мы изменим апплет так, чтобы дать пользователю некоторый контроль над ним: нажатием кнопки мыши останавливать и снова запускать мультипликацию.

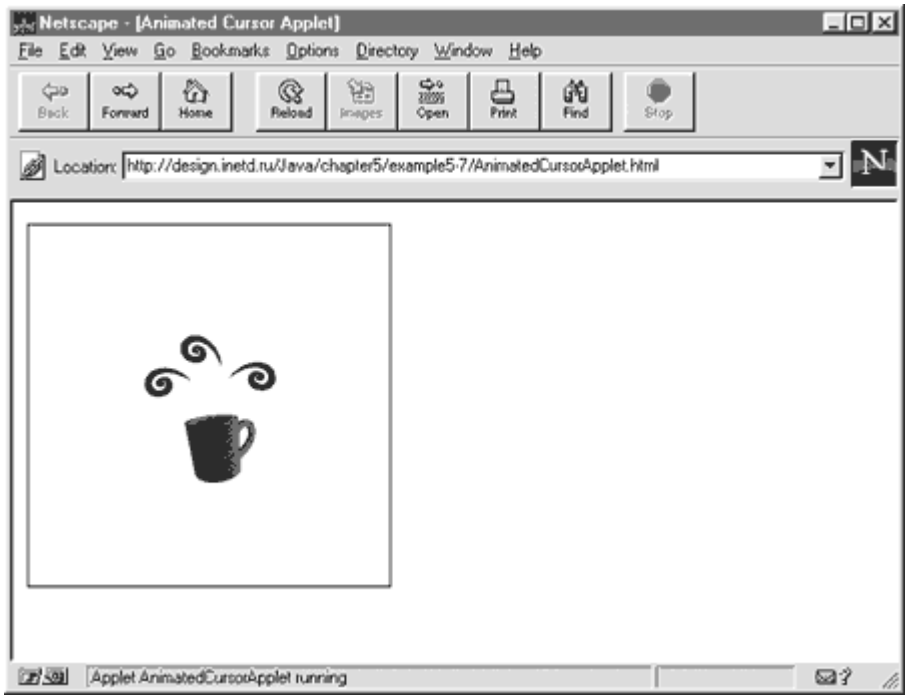

Рис. 5.7.

Пример 5-6а. Анимированный апплет отображения курсора.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
// Этот класс имеет метод, выполняемый внутри потока,
// и будет управляться через
// экземпляр потока, созданный в этом апплете.
public class AnimatedCursorApplet extends
Applet implements Runnable {
        private int mouse_x, mouse_y;
        // массив кадров мультипликации
        private Image CursorImages[];
        // индекс текущего изображения
        private int CursorIndex = 0;
        // поток управления выполнением метода
        private Thread anim = null;
        // мультипликация не приостанавливается
        private boolean paused = false;
        public void init() {
                resize(250,250);
                // привяжем экземпляр потока к апплету
                anim = new Thread(this);
                mouse_x = 125;
                mouse_y = 125;
                // для начала принимаем 5 изображений
                CursorImages = new Image[5];
                int i;
                String CursorParam;
                URL CursorURL;
                // заполним массив изображений
                for (i=0; i<5; i++) {
                        CursorParam = getParameter("CURSORFILE"+i);
                        try {
                         CursorURL = new URL(CursorParam);
                         CursorImages[i] = getImage(CursorURL);
                        }
                        catch (MalformedURLException e) {
                        // создаем пустой кадр, если URL неверен
                                CursorImages[i] = createImage(0,0);
                        }
```

```
                }
        }
        public void start() {
                // начало выполнения метода
                anim.start();
        }
        public void stop() {
                if (anim!=null && anim.isAlive()) {
                // остановка выполнения метода при необходимости
                        anim.stop();
                }
        }
        public void paint(Graphics g) {
                int px, py;
                // курсор указывает на текущее изображение
                Image Cursor = CursorImages[CursorIndex];
                g.drawRect(0,0,249,249);
                // центрируем изображение
                px = mouse_x - Cursor.getWidth(this)/2;
                py = mouse_y - Cursor.getHeight(this)/2;
                g.drawImage(Cursor,px,py,this);
        }
        public boolean mouseMove(Event evt, int x, int y) {
                mouse_x = x;
                mouse_y = y;
                return true;
        }
        public boolean mouseDown(Event evt, int x, int y) {
                // если метод был приостановлен, перезапустим его
                if (paused) {
                        anim.resume();
                        paused = false;
                }
                // иначе приостановим выполняемый метод
                else {
                        anim.suspend();
                        paused = true;
                }
                return true;
        }
        public void run() {
                while (anim!=null) {
                        try {
                                // приостановка на 50 миллисекунд
                                anim.sleep(50);
                        }
                        // если что-нибудь произошло
                        catch (InterruptedException e) {}
                        // двигаемся к следующему изображению
                        CursorIndex = CursorIndex + 1;
                        if (CursorIndex==5) {
                                // повторим снова с самого начала
                                CursorIndex = 0;
                        }
                        repaint();
                }
        }
}
```

Пример 5-6а. Web-страница для анимированного апплета отображения курсора.

```
<HTML><HEAD>
<TITLE>Animated Cursor Applet</TITLE>
</HEAD>
<BODY>
```

```
<APPLET CODE="AnimatedCursorApplet.class"
HEIGHT=250 WIDTH=250>
<PARAM NAME="CURSORFILE0"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/anim0.gif">
<PARAM NAME="CURSORFILE1"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/anim1.gif">
<PARAM NAME="CURSORFILE2"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/anim2.gif">
<PARAM NAME="CURSORFILE3"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/anim3.gif">
<PARAM NAME="CURSORFILE4"
VALUE="http://www.vmedia.com/vvc/onlcomp/java/chapter5/images/anim4.gif">
</APPLET></BODY></HTML>
```

Устранение мерцания

Если вы не работаете на удивительно суперскоростной графической рабочей станции, то при запуске этого апплета вы наверняка обратили внимание на несколько раздражающее мерцание, что является результатом усилий апплета перерисовать экран быстрее, чем это технически возможно. Классическое решение этой проблемы - двойная буферизация. Диаграмма этой процедуры приведена на рис. 5-8.

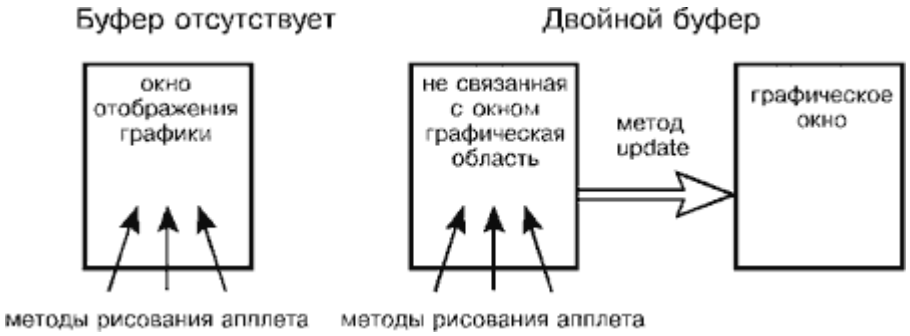

Рис. 5.8.

Когда оболочка времени выполнения должна повторно перерисовывать графические окна - например, когда окно было затенено или когда апплет запросил, чтобы окно было повторно перерисовано, - вызывается метод модификации апплета и ему передается объект Graphics графического окна. Апплет может запрашивать перерисовку графики методом paint. Метод paint вызывает метод модификации как можно скорее. Если апплет запрашивает другую перерисовку прежде, чем метод модификации вызвался из первой, апплет модифицируется только в первый раз. По умолчанию методы модификации просто передают объект Graphics методу paint.

В апплете можно определить виртуальный экран, чтобы рисовать на нем, вместо того чтобы обращаться непосредственно к графическому окну оболочки времени выполнения. Нарисованное изображение апплет может перенести на реальное окно с помощью метода getCraphics. Этот метод возвращает объект Graphics, связанный с изображением. Апплет будет рисовать на нем с той скоростью, с какой сможет. Метод модификации апплета выведет изображение на объект Graphics, вместо того чтобы разрешить апплету вывести его прямо на экран. В результате мерцание должно исчезнуть.

Есть одно значительное различие между закрашиванием объекта Graphics оболочки времени выполнения и закрашиванием объекта Graphics виртуального изображения. Окно оболочки времени выполнения очищено и должно быть повторно окрашено полностью при каждой модификации. Виртуальное изображение сохранит то, что на нем уже было нарисовано, если оно не окрашено явно. Если вы хотите использовать такое изображение для вывода движущегося рисунка, вы должны повторно перерисовывать фон вокруг рисунка каждый раз, когда вы модифицируете его, иначе ваш рисунок, перемещаясь, оставит за собой следы.

Теперь мы можем добавить двойную буферизацию к нашему апплету отображения курсора, используя виртуальное изображение экрана.

Пример 5-6а. Анимированный апплет отображения курсора без мерцания.

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class AnimatedCursorApplet extends Applet implements Runnable {
```

```
private int mouse_x, mouse_y;
private Image CursorImages[];
private int CursorIndex = 0;
private Thread anim = null;
private boolean paused = false;
private Image OffscreenImage;
// наш виртуальный экран
private Graphics OffscreenGraphics;
// наш интерфейс к нему
public void init() {
        resize(250,250);
        OffscreenImage = createImage(250,250);
        // новое изображение
        OffscreenGraphics = OffscreenImage.getGraphics();
        // привяжем наш интерфейс к изображению
        anim = new Thread(this);
        mouse_x = 125;
        mouse_y = 125;
        CursorImages = new Image[10];
        int i;
        String CursorParam;
        URL CursorURL;
        for (i=0; i<5; i++) {
                CursorParam = getParameter("CURSORFILE"+i);
                try {
                        CursorURL = new URL(CursorParam);
                        CursorImages[i] = getImage(CursorURL);
                }
                catch (MalformedURLException e) {
                        CursorImages[i] = createImage(0,0);
                }
        }
}
public void start() {
        anim.start();
}
public void stop() {
        if (anim!=null && anim.isAlive()) {
                anim.stop();
        }
}
public synchronized void update(Graphics g) {
        paint(OffscreenGraphics);
        // скажем апплету закрашивать наше изображение
        g.drawImage(OffscreenImage,0,0,this);
        // закрасим реальное графическое окно
}
public void paint(Graphics g) {
        int px, py;
        Image Cursor = CursorImages[CursorIndex];
        g.setColor(Color.white);
        g.fillRect(0,0,249,249);
        // рисуем по старому изображению
        g.setColor(Color.black);
        g.drawRect(0,0,249,249);
        // выведем рамку
        px = mouse_x - Cursor.getWidth(this)/2;
        py = mouse_y - Cursor.getHeight(this)/2;
        g.drawImage(Cursor,px,py,this);
}
public boolean mouseMove(Event evt, int x, int y) {
        mouse_x = x;
        mouse_y = y;
        return true;
```

```
        }
        public boolean mouseDown(Event evt, int x, int y) {
                if (paused) {
                        anim.resume();
                        paused = false;
                }
                else {
                        anim.suspend();
                        paused = true;
                }
                return true;
        }
        public void run() {
                while (anim!=null) {
                        try {
                                anim.sleep(50);
                        }
                        catch (InterruptedException e) {}
                        CursorIndex = CursorIndex + 1;
                        if (CursorIndex==5) {
                                CursorIndex = 0;
                        }
                        repaint();
                }
        }
}
```

Что дальше?

Прочитав это главу, вы можете программировать мультимедиа в Java-апплетах. Теперь вы можете получать изображение и звуковые данные из сети, а также параметры во время выполнения из HTML-кода, можете обрабатывать события интерфейса пользователя и оживлять свои апплеты при помощи потоков. Все это необходимо для создания апплетов, которые включают мультимедиа, но этого недостаточно для написания апплетов, которые могут взаимодействовать с пользователем на высоком уровне, используя текстовые и другие устройства ввода данных.

К настоящему времени вы должны получать удовольствие при использовании языка Java. В следующих главах обсуждается богатая иерархия классов Java, обеспечиваемых API. Эти классы и интерфейсы добавляют к языку Java полезные функциональные возможности, обеспечивающие удобную реализацию многих сложных структур данных, используемых программистами, и полный графический комплект инструментальных средств работы с окнами. Они являются также хорошим примером мощности расширяемых объектов многократного использования.

Глава 6

Интерфейс прикладного программирования

Вероятно, вы уже начали создавать свои собственные апплеты. Но Java - намного больше, чем прибор для создания мультипликации на Web-страницах. Придет время, и вы, закончив изучение этой книги, будете создавать апплеты и приложения, которые используют среду работы с окнами и общаются с сетью. Ваш рост от кодировщика апплета до программиста Интернет обеспечит твердое понимание API.

API - это огромный набор инструментальных средств с широкими функциональными возможностями, которые приходят с Java Developers Kit. В нем вы найдете инструментальные средства манипулирования строками, работы с сетями, математические возможности, а также "рабочие лошадки" - программы работы с хеш-таблицами (не путайте их с "кэшем" - это совсем другое) и стеками. Все это богатство, обеспечиваемое API, - фундамент Java. Фактически, мы уже его использовали. Классы Applet, URL, Image, String и System - это все члены API. Помните оператор import, которым мы пользовались в прошлой главе? Каждый раз, когда мы ставим за ним нечто, начинающееся с "java", мы обращаемся к API.

СОВЕТ В большинстве объектно-ориентированных языков программирования есть некоторая форма библиотеки классов - совокупность классов, которые написаны на языке и готовы к применению. Если вы привыкли использовать библиотеки, вы можете думать об API как о библиотеке классов Java - это два идентичных понятия. Однако, как вы увидите, с технической точки зрения это не так, потому что в Java API включены еще и интерфейсы. Мы будем придерживаться термина API в оставшейся части книги.

API можно использовать почти для любой создаваемой программы. Фактически, класс Object, который находится наверху иерархии классов Java, - это самостоятельный член API. Большая часть вашей работы как программиста Java будет заключаться в изучении того, какие средства имеются в вашем распоряжении в наборе инструментальных средств API. В этой главе внимание сосредоточено на той части API, которая сохранит вас от огромного количества рутинной работы, когда вы начнете создавать все более сложные апплеты. Например, когда мы будем описывать интерфейс пользователя в главе 7, изображения в главе 9 и работу апплетов с сетями в части IV, в действительности мы будем пользоваться кодом, уже содержащимся в API.

Основы API

Большинство языков программирования включают в себя нечто подобное Java API. В то время как спецификация языка говорит, как операторы и ключевые слова работают вместе, базовые функциональные возможности типа ввода и вывода включены в язык и обеспечиваются компилятором. Java - не исключение. Те из вас, кто знаком с языком C, вероятно, увидят множество аналогий между стандартными библиотеками C, которые приходят с любым компилятором, и Java API. Без библиотек C вы можете только управлять базовыми типами и не имеете простого пути для ввода или вывода.

Однако у Java API есть несколько преимуществ над стандартными библиотеками, подключаемыми компиляторами в других языках. Во-первых, API полностью объектно ориентирован. Так что вы не должны смешивать и, соответственно, согласовывать библиотеки классов с библиотеками ANSI C, как это делается при работе в C/C++. Во-вторых, независимость языка от платформы означает, что API также независим от платформы. Вам никогда не придется волноваться относительно библиотек на различных платформах, являющихся несовместимыми.

API настолько легок в использовании, что у вас не должны возникнуть какие-либо серьезные проблемы при изучении тех частей, которые являются абсолютно необходимыми. Если вы действительно с головой погрузитесь в исследование API, вы сможете выполнять множество базовых действий без написания своего собственного кода и достигнете хорошего понимания того, как использовать объектно-ориентированные особенности языка Java.

Сначала нам нужно разобраться в структуре API. Затем мы рассмотрим синтаксис Java так, чтобы вы могли использовать конкретные классы и интерфейсы в их наиболее полных вариантах.

Структура API

API - это совокупность нескольких десятков готовых классов, интерфейсов и исключений. Внутри API эти классы, интерфейсы и исключения сгруппированы в восемь пакетов: java.applet, java.awt, java.awt.image, java.awt.peer, java.io, java.lang, java.net и java.util. В табл. 6-1 кратко описано назначение каждого пакета.

Таблица 6-1. Пакеты API

Пакет	Описание
java.applet	Содержит классы и интерфейсы, которые запускают апплет (см. главу 5, "Апплет в работе").
java.awt	Позволяет создавать графические интерфейсы пользователя.
Java.awt.peer	Составлен полностью из интерфейсов, которые позволяют системе работать с окнами Java. Легко переносимы на другие платформы (но не представляют никакого интереса для среднего программиста).
java.awt.image	Предназначен для создания и манипулирования изображениями (см. главу 9, "Графика и изображения").
java.io	Обрабатывает ввод и вывод программы (см. главу 11, "Многопотоковость").
java.lang	Содержит основные элементы java.language, такие как Object, String, Excerption и Thread.

Java.net	Обрабатывает взаимодействие с сетью (см. главы 11, "Многопотоковость", и 12, "Программирование за рамками модели апплета").
Java.util	Содержит несколько вспомогательных классов, обычно использующих структуры данных.

Не забывайте, что пакет - это связанный набор классов и интерфейсов, которые позволяют обращаться к методам и переменным, определенным с модификатором protected. Во многих случаях члены входят в какой-то из пакетов, что объясняется соображениями дизайна, связанными с использованием ключевого слова protected. В других случаях какой-то из членов может принадлежать к некоторому пакету из-за того, что по своей цели он ближе всего к другим членам именно этого пакета. Используя API, мы можем думать о различных пакетах как о семействах функциональных возможностей. Не обманитесь на счет java.awt.image и java.awt.peer - их члены не имеют никакой специальной синтаксической связи с членами пакета java.awt. Создатели Java выбрали для API такие имена, чтобы показать, что члены двух пакетов являются подобными по функциональным возможностям.

СОВЕТ Подклассы других классов API не обязательно находятся в том же самом пакете, что и их родитель. Например, класс Applet существует в java.applet, но его суперкласс находится в java.awt.

При ежедневном программировании структура пакета важна для вас в двух случаях. Во-первых, вам нужно использовать оператор import, чтобы обратиться к классу в API. А во-вторых, вы должны быть хотя бы в общем знакомы с содержанием пакетов, потому что интерактивная документация Java Developers Kit, как показано на рис. 6-1, также организована в виде пакетов. Каждый класс, интерфейс и исключение имеют собственную Web-страницу с подробным описанием всех доступных общих методов и переменных.

Рис. 6.1.

Закончив изучение этой главы, вплотную пообщайтесь с интерактивной документацией JDK. Поскольку вы наверняка сталкиваетесь с ошибками во время компиляции своих Java-программ, вы найдете интерактивную документацию неоценимой. Жизнь этой книги продолжается на нашей странице Online Companion, где вы можете найти самую последнюю интерактивную документацию, обеспечиваемую фирмой Sun Microsystems.

Использование API

Первый шаг в использовании API - обращение к пакетам с помощью оператора import. После этого процесс вашего обучения разделится на два направления. Сначала вы узнаете, как использовать сформированные блоки в программировании на Java. Кроме этого, как только вы станете опытнее в использовании API, вы будете самостоятельно писать блоки Java, разрабатывая свой собственный стиль работы с языком Java.

API - это прекрасный способ познакомиться со специфическими особенностями языка Java. Вы можете учиться, используя то, что уже существует. Вероятно, вы не всегда будете соглашаться с решениями создателей проекта API, но вы достигнете того уровня, когда будете испытывать удовольствие от программирования на Java. Начав разрабатывать свои собственные классы, интерфейсы и пакеты, вы сможете использовать API как ссылку для своего проекта.

Импорт пакетов

Уже в главе 2 мы начали импортировать пакеты и описали синтаксис полностью в главе 3. Как вы помните, для этого достаточно написать:
import <название_пакета>. *;

Это не является необходимым для импорта пакетов, но делает ваш код проще для чтения. Как вы помните из главы 5, когда мы не импортировали пакет java.applet, мы должны были объявлять наш класс следующим образом:
class MyApplet extends java.applet.Applet {

Но для классов, которые мы будем часто использовать, не очень-то удобно каждый раз подробно выписывать полное имя класса.

СОВЕТ Пакет java.lang настолько необходим, что компилятор всегда импортирует его.

Рассмотрим рецепт наилучшего использования оператора import на примере класса java.util.Date, позволяющего получать текущее время и дату.

1. Определите пакет, содержащий класс, который вы хотите использовать. Это довольно просто: так как полное имя нашего класса - java.util.Date, следовательно, он входит в пакет java.util. В противном случае мы проконсультировались бы с интерактивной документацией по JDK, чтобы найти соответствующий пакет.
2. Импортируйте класс оператором import. Если мы используем только класс Date, мы могли бы сделать так:
   import java.util.Date;
   Обычно используется несколько классов из одного пакета. В этом случае лучше использовать шаблон для импорта всех классов из этого пакета:
   import java.util.*;
3. Теперь мы готовы использовать наш класс, чтобы обратиться к текущей дате.
   Использование оператора import довольно просто. Ловушка, которой нужно избежать, - импорт только пакета, а не класса. Попасть в эту ловушку можно со следующим оператором import:
   import Java.util;
   Этот оператор фактически не импортирует никаких классов. Добавляя шаблон или имя класса, мы можем объявить переменные следующим образом:
   Date d;
   или иначе:
   java.util.Date d;
   С оператором import java.util.* или без него последний код правилен. Однако такая форма объявления неуклюжа, и в ней легко ошибиться.

Статические методы против динамических

Как вы помните из главы 2, методы Java могут быть или статическими, или динамическими. Динамические методы намного более общие - они связаны с конкретизацией понятий класса. С другой стороны, статические методы являются методами, которые связаны с классом непосредственно. Метод является статическим, если в его объявлении присутствует модификатор static. На рис. 6-2 показан статический метод с именем parse в классе java.util.Date. Обратите внимание на присутствие ключевого слова static.

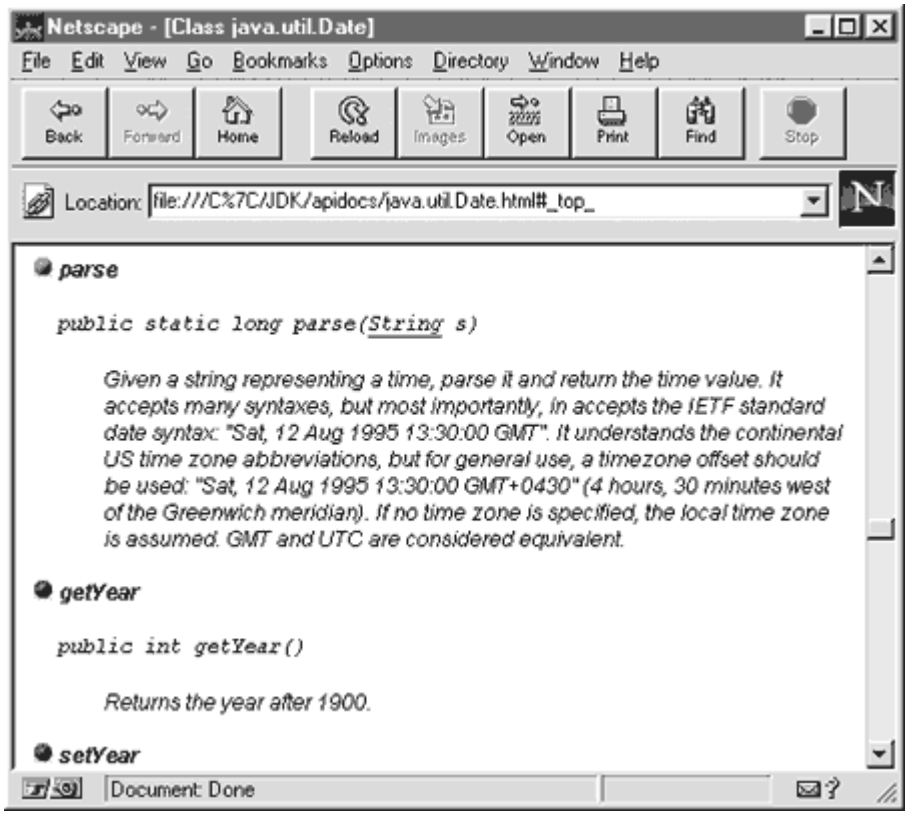

Рис. 6.2.

Используя статический метод типа parse, мы не должны создавать экземпляр класса. Взамен этого мы ссылаемся на класс следующим образом:

```
public long demoStatlc(String S) {
    return Date.parse(S);}
```

Многие из методов в API статические, потому что даже если бы они были динамическими, они привели бы к точно такому же результату. Следовательно, нам пришлось бы лишний раз создавать переменную, когда нам нужен был бы только метод, который получает что-то на входе и производит что-то на выходе, оставаясь безразличным к состоянию остальной части объекта.

Но это удобство часто вызывает ошибки у новичков в Java, потому что они пробуют обращаться к статическим методам динамически. Поэтому оказывается полезным думать о каждом классе как о состоящем из двух отдельных частей - той, что является динамической, и той, что является статической. Динамическая часть состоит из переменных и методов, которые являются частью экземпляра класса, - те, что не определены с модификатором static. В случае нашего класса Date это были бы методы, отличные от нашего метода parse.

С другой стороны, статические переменные и методы являются только утилитами, которые принадлежат некоему классу, потому что этот класс - то место, где вы вполне логично искали бы их. Давайте рассмотрим, к примеру, метод parse из класса Date. Этот метод берет строку, которая форматируется как дата, и возвращает вам число секунд, прошедших с полуночи 1 января 1970 года по Гринвичу.

Если вам нужны такие функциональные возможности и вы начали для их поиска просматривать интерактивную документацию API, вы будете, вероятно, заглядывать в класс Date. Просматривая описания методов класса Date, вы увидите, что такой метод уже существует и вам не нужно писать его самостоятельно.

Как только вы увидите, что метод существует, определите, является ли он статическим. Если это статический метод, вы будете знать, что:

- Вы не должны присваивать значение объекту, чтобы использовать этот метод.
- Вы не можете обращаться к статическому методу из объекта, подвергнутого обработке.
- Обращаться к методу нужно с синтаксисом <имя класса>.<имя метода>.

Обзор других модификаторов доступа

Обратившись к интерактивной документации, чтобы выяснить, является ли данный метод статическим, вы можете наткнуться на указатели для других модификаторов доступа. Все эти модификаторы определены в главе 3, "Объектная ориентация в Java". Здесь мы сделаем их обзор, учитывая их воздействие на нас как на пользователей API.

- Модификатор protected. Означает, что метод или переменная доступны только другим классам этого пакета. Так как мы создаем классы, которые не будут входить ни в какой из пакетов API, этот модификатор пока не накладывает на нас никаких ограничений. Мы доберемся до использования ключевого слова protected при проектировании и выполнении собственных пакетов в главе 11, "Многопотоковость".
- Модификатор final. Означает, что метод (или весь класс) не может иметь подклассы. Когда мы используем классы, которые являются частью API, это ключевое слово не имеет для нас никакого значения. Однако если мы хотим сделать свой собственный подкласс класса API, мы должны проверить, не определен ли тот с помощью модификатора final.
- Модификатор abstract. Если весь класс определен с модификатором abstract, мы можем реализовывать только его подклассы. Если только один из членов класса определен с модификатором abstract, мы можем обращаться только к тому члену подкласса, который заменяет этот специфический метод (см. главу 3, "Объектная ориентация в Java"). Когда вы видите модификатор abstract, не забудьте, что дело обстоит не так, как кажется на первый взгляд. Вы должны, вероятно, рассмотреть дерево API, которое включено в документацию, и исследовать подклассы, обеспечиваемые API. На рис. 6-3 показано дерево абстрактного класса java.awt.Component. Понимание подклассов этого класса - большая часть изучения Java AWT, и мы исследуем их полностью в следующих двух главах.

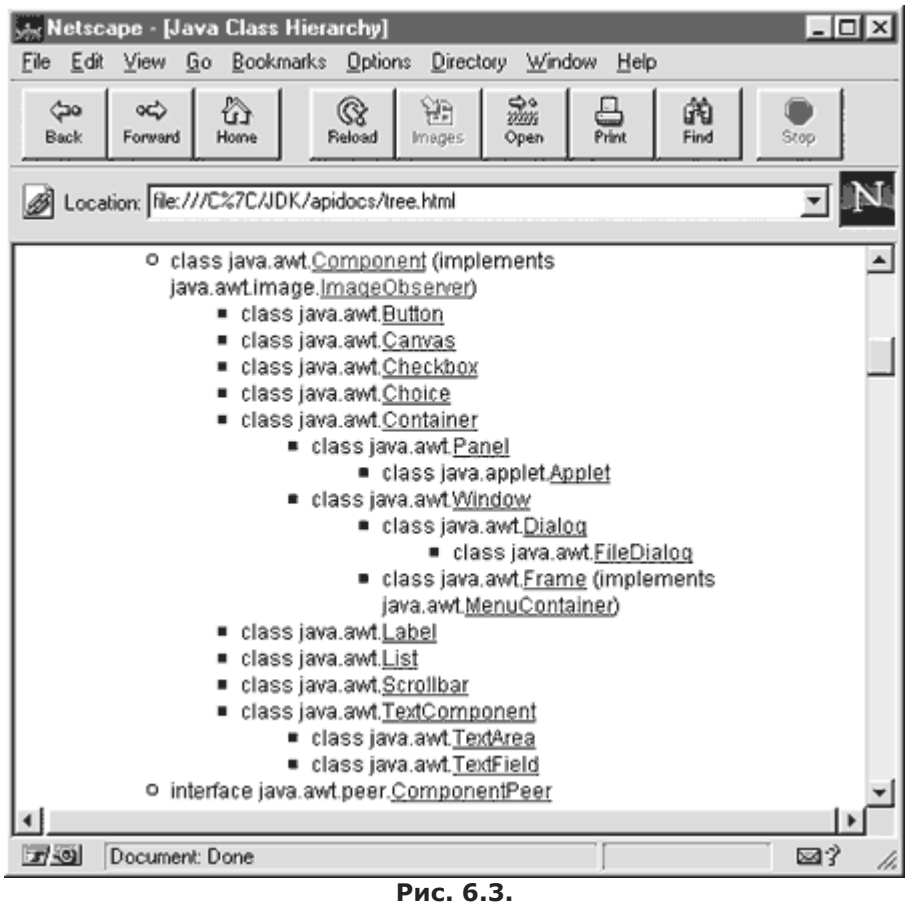

Рис. 6.3.

- Модификаторы synchronized и native. Не имеют никакого реального значения для пользователей API. Чтобы дать о них общее представление, скажем, что модификатор sinchronized означает, что к специфическому методу или переменной можно обратиться только в определенное время, что важно для многопотоковых программ (см. главу 11). Но теперь нам не нужно волноваться относительно этого модификатора.
- Модификатор native определяет, что метод фактически выполнен в динамической связанной библиотеке (DLL), которая не была написана на Java. Предполагается, что все методы, определенные с модификатором native, вставляются в код, написанный на Java.

Но пока нам не нужно обращать особое внимание в том случае, если метод, который мы хотели бы использовать, определен с модификатором native. (Более подробно мы рассмотрим такие методы в главе 12, "Программирование за рамками модели апплета".)

Исключения и API

Изучая интерактивную документацию по Java, вы можете обратить внимание на то, что некоторые методы и конструкторы вызывают исключения. Мы столкнулись с конструктором, вызывающим исключение, в главе 5, "Апплет в работе", - конструктор для класса URL, который содержится в пакете java.net. На рис. 6-4 приведена Web-страница класса URL, показывающая, что конструктор, который получает строку, вызывает исключение MalformedURLException.

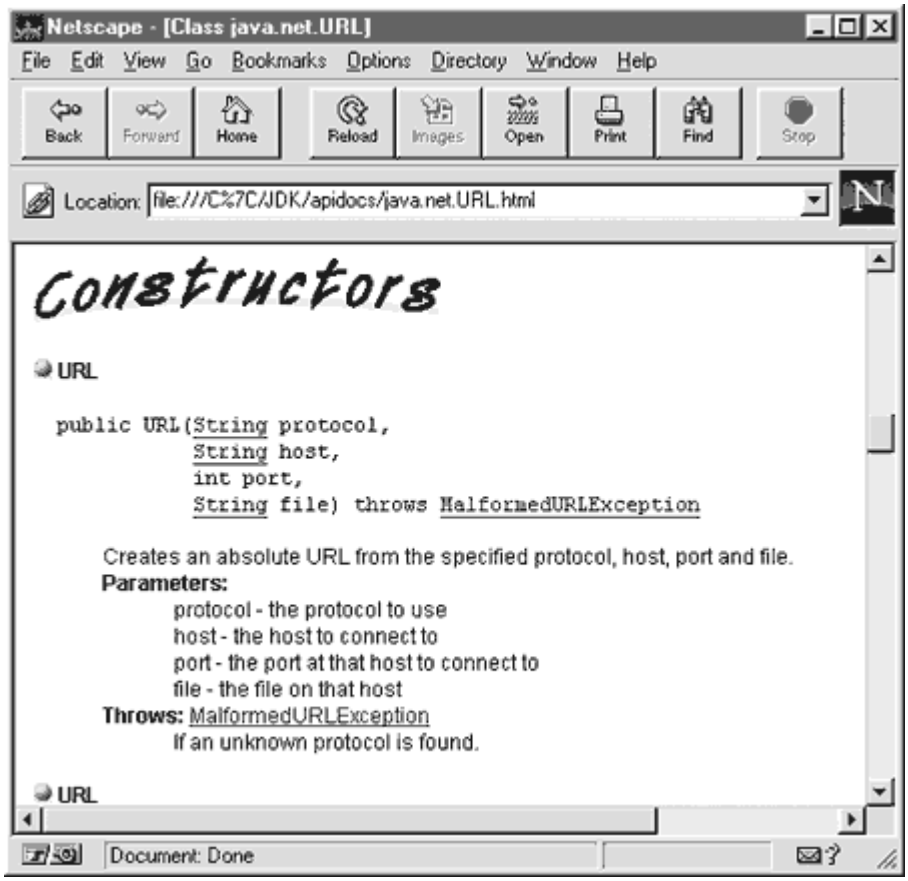

Рис. 6.4.

Чтобы эффективно использовать API, вы должны знать исключения, которые может вызывать метод или конструктор. Написание собственных подпрограмм, обрабатывающих исключения, будет рассмотрено в главе 10, "Структура программы". Сейчас же мы только представим вам рецепт для имеющихся методов API и конструкторов, вызывающих исключения. Основная стратегия показана на диаграмме потока на рис. 6-5.

Рис. 6.5.

Первый шаг довольно очевиден - мы сначала выясняем в интерактивной документации, вызывает ли наш метод исключение. Если ключевое слово throw присутствует в методе или объявлении конструктора, мы выполняем второй шаг, чтобы понять, какое исключение вызывается. Самый простой способ сделать это состоит в том, чтобы щелкнуть по имени исключения (в нашем случае для конструктора URL - MalformedURLException). Так как исключения являются классами, они имеют собственные Web-страницы.

Добравшись до Web-страницы исключения, мы определяем, что это за исключение, то есть проверяем, подклассом какого класса оно является. Если оно является потомком RuntimeException или Error, нам не нужно использовать блок обработки. Затем мы должны решить, нужно ли перехватить это исключение. Обычно нет никаких причин перехватывать исключение - оно вызывается, когда что-то идет действительно неправильно, например, управление передается за пределы памяти. Но бывают случаи, когда возникает необходимость перехватывать исключения во время выполнения апплета.

Зачем перехватывать исключения?

В API есть много методов, которые вызывают исключения во время выполнения программы, которые имеет смысл перехватывать. Ниже в этой главе мы рассмотрим методы, преобразующие строки в числовые значения типа float и int. Если строка задана некорректно, произойдет исключение NumberFormatException. Если мы попробуем преобразовать строку, не имеющую числового смысла, в число и не перехватим это, наша программа сработает неправильно.

Если нужно перехватить исключение во время выполнения программы, в текст метода, который вызывает исключение, вставляется блок обработки. В него помещается все, что зависит от нормального выполнения метода или конструктора. Кроме того, мы должны удостовериться, что мы действительно обрабатываем исключительную ситуацию в нашем блоке. Хотя следующий апплет откомпилируется и будет работать правильно, в нем дан пример плохого обращения с исключениями:

```
import java.applet.*;
class badApplet extends Applet {
Image img;
// описание переменных
public void init() { URL imgURL;
// Плохая идея! Должно быть в блоке обработки исключений.
try {
```

```
        imgURL=new URL(getParam)("image"));
        }
        catch (MalformedURLException e) {}
        // Плохая идея! Мы ничего не делаем в случае исключения.
img=getImage(imgURL);
}
// другие методы
}
```

Рассмотрим, что произойдет, если параметр "image" не будет верным URL-адресом. Наш блок проверки даже не сообщит пользователям, что что-то неправильно! Еще хуже то, что мы использовали imgURL вне блока обработки. Это означает, что наша программа даст сбой, когда мы попытаемся отыскать изображение.

Фрагмент кода, приведенный выше, - результат нежелания попробовать обрабатывать исключения в программе. Так как обработка исключений часто непонятна начинающему программисту Java, он может отнестись к ней, как к ненужным сложностям, и делать только то, что абсолютно необходимо. Однако, как мы увидим из главы 10, "Структура программы", обработка исключений чрезвычайно важна для написания надежных Java-программ. Мы не будем обсуждать это глубже, но, поскольку вы начинаете использовать API, попробуйте не приобретать привычки плохо обрабатывать исключения. Если вы пишете блоки обработки, берите за основу следующие советы:

- Если автор метода считает, что надо вызывать исключение, то это, вероятно, уже является хорошей причиной обработки исключения.
- Удостоверьтесь, что весь код, который зависит от метода или конструктора, вызывающего исключение, содержится в блоке обработки. Самый простой способ гарантировать это состоит в том, чтобы объявлять любые переменные, которые назначены специфическим методом или конструктором, внутри этого блока.
- Всегда используйте свои блоки обработки исключений, чтобы разумно отвечать на обстоятельства, вызывающие исключения. В нашем примере плохого апплета мы должны объяснить пользователю тот факт, что параметр "image" является недопустимым. В критических случаях вы можете завершить выполнение апплета в блоке обработки.

Класс java.lang.Object

Класс java.lang.Object - это фундаментальный класс Java. Помните ли вы из главы 3, "Объектная ориентация в Java", что все классы происходят из одного класса? И это - тот самый класс! Наиболее важное замечание относительно класса Object состоит в том, что он находится наверху иерархии. Однако он не является просто абстракцией - этот класс содержит несколько методов и может подвергаться обработке подобно другим классам. Так как все классы происходят из класса Object, его методы присутствуют в любом другом классе. Большинство (если не все) базовых методов, приведенных в табл. 6-2, обычно отменяются в подклассах.

Таблица 6-2. Базовые методы класса Object

Метод	Описание	Исключения
boolean equals (Object o)	Возвращает true, если эквивалентен заданному объекту.	Нет
String toString()	Возвращает строку с информацией о заданном объекте.	Нет
Object clone()	Клонирует заданный объект.	CloneNotSupportedException
void finalize()	Вызывается, когда заданный объект уже больше не нужен.	Throwable

Хотя в классе Object есть и другие методы, здесь приведены наиболее важные. Первые два отменяются любым классом в API. Метод clone - наш секрет для создания новых копий объектов. Этот метод возвращает нас к обсуждению оператора присваивания в главе 4. Давайте вспомним наш пример:

```
public void sampleMethod(someClass anObject) {
someClass anotherObject=anObject; }
```

Здесь anotherObject и anObject в действительности являются одним и тем же классом - присваивание не создавало новую копию. Если метод clone не изменялся, мы можем использовать его, чтобы сделать копию образца:

```
SomeClass aCopy=anObject.clone();
```

Здесь мы считаем, что автор someClass определил, как он хочет проклонировать экземпляр someClass. Однако определение способа клонировать себя для объектов - не всегда хорошая идея для класса (см. главу 10). Авторы специфических классов решают, могут ли клонироваться экземпляры этих классов, и многие из классов в API не клонируемы. Этот метод можно использовать, когда класс или один из его суперклассов осуществляет интерфейс клонирования и явно заменяет метод.

Но что можно сказать относительно других методов в классе Object? Их изучение выходит за рамки этой главы. В табл. 6-3 описано, что они делают и где они обсуждаются впоследствии.

Таблица 6-3. Другие методы класса Object

Метод	Описание	Где описан
wait, notify, notifyAll	Позволяет объекту общаться с потоками Java.	Глава 11, "Многопотоковость".
hashCode	Позволяет объекту использоваться совместно с java.util.Hashtabfe.	Глава 10, "Структура программы".
getClass	Используется для получения информации об этом объекте во время выполнения.	Глава 10, "Структура программы".

Работа со строками

Как вы знаете, строка - это просто последовательность символов. Фактически все языки программирования так или иначе поддерживают строки, и Java не является исключением. Класс String, чья Web-страница с документацией показана на рис. 6-6, находится в пакете java.lang и используется для представления строк. Этот класс обеспечивает такие функциональные возможности, как преобразование к числовым типам, поиск и замена подстроки, обращение к конкретным символам и незаполненному пространству. Вполне вероятно, что вы будете использовать класс String больше, чем любой другой класс в API, и вы должны хорошо знать входящие в него методы. В этом разделе мы исследуем класс String. Мы также рассмотрим некоторые другие классы в API, которые помогут вам манипулировать со строками.

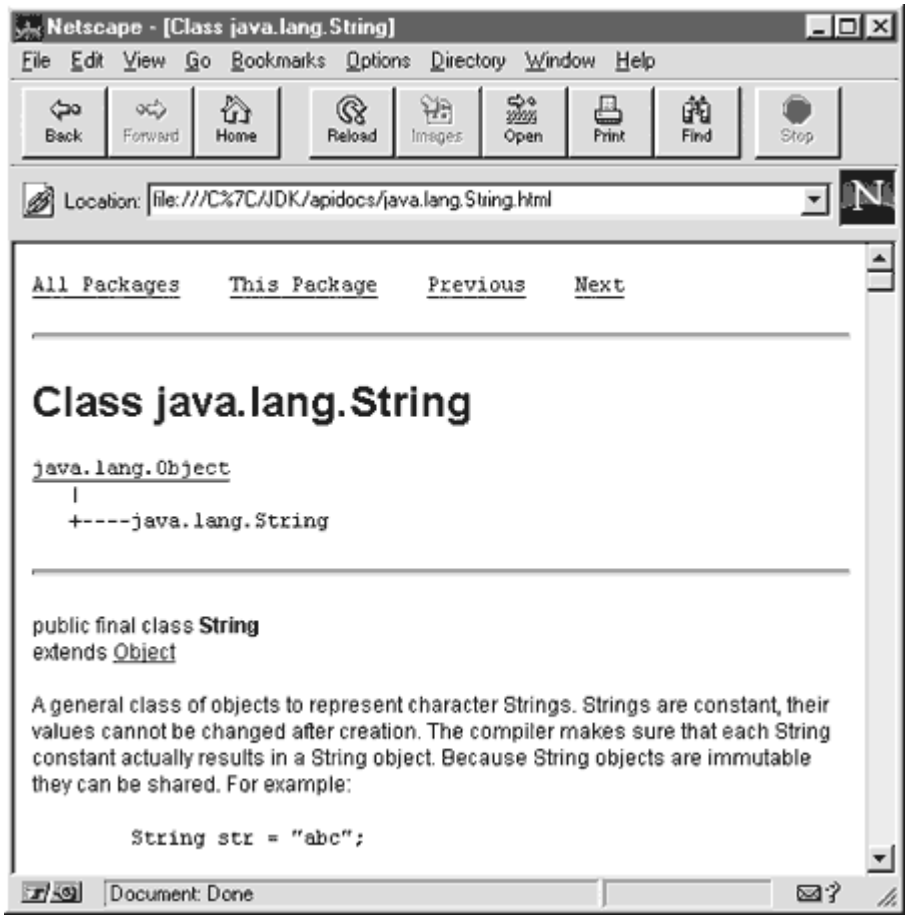

Рис. 6.6.

Создание строк

Строки можно создавать одним из двух способов:

- использованием одного из семи конструкторов класса String;
- использованием знака операции присваивания (=).

До сих пор для создания строк мы применяли знак операции присваивания. Оператор String S="My string,'';
является самым простым способом создать строку. Знак операции + также может использоваться со строками для простой конкатенации строк, как показано в примере 6-1.

Пример 6-1а. Формирование строк из простых типов.

```
String S2="another string";
String S3=S+S2+"and a primitive type:"
int i=4;
S3=S3+i
```

S3 будет теперь иметь значение "My string, another string and a primitive type: 4". Знак операции + работает со всеми примитивными типами. Однако знак операции = этого не делает. Следующий фрагмент кода не будет компилироваться из-за последних двух операторов:

```
int i=4;
String SI=i;
// Не будет компилироваться!
String S2=4;
// Не будет компилироваться!
```

Это ограничение неприятно, но его можно легко, хотя и не очень изящно, обойти. Надо создать пустую строку и добавить к ней нужное значение с помощью знака операции +.

Пример 6-1b. Присваивание значений примитивных типов непосредственно строкам.

```
int i=4;
String S1=""+i;
String S2=""+4;
```

Знак операции присваивания работает только со строковыми константами (то есть текстом, содержащимся внутри кавычек) и другими строками. Знак операции + будет работать со строковыми константами, другими строками и всеми примитивными типами. Кроме того, строки могут быть созданы с помощью конструкторов, описанных в табл. 6-4.

Таблица 6-4. Конструкторы строк

Конструктор	Объект, который он создает
String()	Пустая строка.
String (String S)	Новая строка, которая является копией S.
String(char charArray[])	Строка, основанная на массиве charArray.
String(char charArray[], int offset, int len)	Строка, основанная на подмассиве, начинающаяся со смещения ang и длиной len символов.
String(byte byteArray[], int hibyte)	Строка, основанная на массиве byteArray, где hibyte описывает старший байт Unicode каждого символа.
String(char charArray[], int hibyte, int offset, int len)	Строка, основанная на подмассиве, начинающаяся по смещению offset и длиной len байт.

Хотя знак операции присваивания не совместим с примитивными типами, это действительно единственное, что является интуитивно неясным при создании строк в Java. Как только мы проберемся через причуды сравнения строк, вы будете готовы спокойно использовать строки.

Сравнение строк

Теперь, умея создавать строки, мы должны научиться их сравнивать. В классе String есть два метода сравнения - equals и compareTo. Метод equals заменяет метод equals в классе Object и возвращает true, когда строки совпадают, в то время как метод compareTo при эквивалентности строк возвращает 0.

Пример 6-2a. Сравнение строк с помощью методов compareTo и equals.

```
if (S1.compareTo(S2)!=0)
        System.out.println("S1 is equivalent to the String S2");
if (S1.equals(S2))
        System.out.println("S1 is equivalent to the String S2");
```

Значение, возвращаемое методом compareTo, равно -1, если S1 - лексически меньше, чем S2, и 1, если S1 лексически больше, чем S2. Как можно видеть из нижеприведенного примера, "лексическое" сравнение по существу означает сравнение в алфавитном порядке. Различие состоит в том, что этот порядок простирается на цифры и другие небуквенные символы так, что "a113" оказывается лексически больше, чем "a112". Упорядочение определено значениями Unicode.

Пример 6-2b. Сравнение строк с помощью метода compareTo.

```
String S1="alpha";
String S2="alpha";
if (S1.compareTo(S2)==-1)
        System.out.println("this will always be printed");
if (S2.compareTo(S1)==1)
        System.out.println("this will always be printed");
```

Лексическое упорядочение получено из целочисленных значений символов Unicode. Метод compareTo последовательно сравнивает каждый символ строк, основываясь на его целочисленном значении. Когда он находит различие, то возвращает 1 или -1, как описано выше. Для цифр и букв упорядочение интуитивно понятно и просто для запоминания (рис. 6-7).

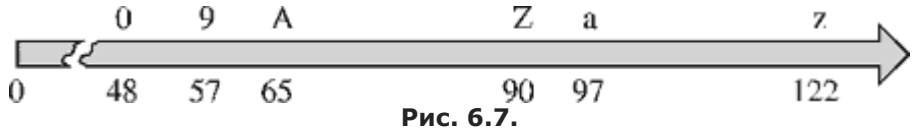

Рис. 6.7.

Более сложным является сравнение строк при использовании символов "неанглийского" алфавита и служебных символов типа %, ! или *. Самый простой способ узнать численные значения подобных символов, не показанных на рис. 6-7, - привести тип char к int и напечатать его значение:

```
public void strangeCharacter(char Strange) {
int val =( int) Strange;
System.out.println ("One strange character!'');
System.out.println ( "The char " +Strange+" = " +val); }
```

Знак операции == и сравнение строк

Вам может встретиться Java-код, который сравнивает строки с помощью знака операции ==. В простых случаях знак операции == работает для строк так же, как для примитивных типов. Но такое использование не является частью спецификации языка - оно зависит от компилятора и может не работать в некоторых случаях. Так что лучше им не пользоваться, чтобы не допустить неприятную и труднонаходимую ошибку, и всегда применять методы compareTo или equals.

В дополнение к двум методам сравнения, которые мы только что рассмотрели, есть еще шесть других, сравнивающих две строки различными способами. Все они помещены в табл. 6-5 в порядке увеличения сложности.

Таблица 6-5. Методы сравнения строк

Метод	Возвращает true, когда
equalsIgnoreCase(String S)	Строка равняется S, независимо от регистра символов.
beginsWith(String S)	Строка начинается с S.
beginsWith(String S,int len)	Строка начинается с первых len символов S.
endsWith(String S)	Строка заканчивается S.
regionMatches(int toffset, String S,int ooffset, int len)	Подстрока в строке, начиная с позиции со смещением toffset, соответствует подстроке, начинающейся по смещению ooffset, для len символов.
regionMatches(boolean ignoreCase, int toffset, String S, int ooffset, int len)	Тот же, что и предыдущий метод, но игнорирует регистр символов, когда ignoreCase == true.

Большинство методов сравнения строк интуитивно понятны, но методы regionMatches нуждаются в некотором объяснении. Они позволяют сравнивать любую из подстрок в S1 и S2. Параметр first определяет позицию в строке образца, с которой вы хотите начать сравнение. Второй параметр - некоторая другая строка с позицией, определенной в третьем параметре, с которой вы хотите начать сравнение в этой строке. Последний параметр - длина области сравнения. Использование методов regionMatches проиллюстрировано на рис. 6-8.

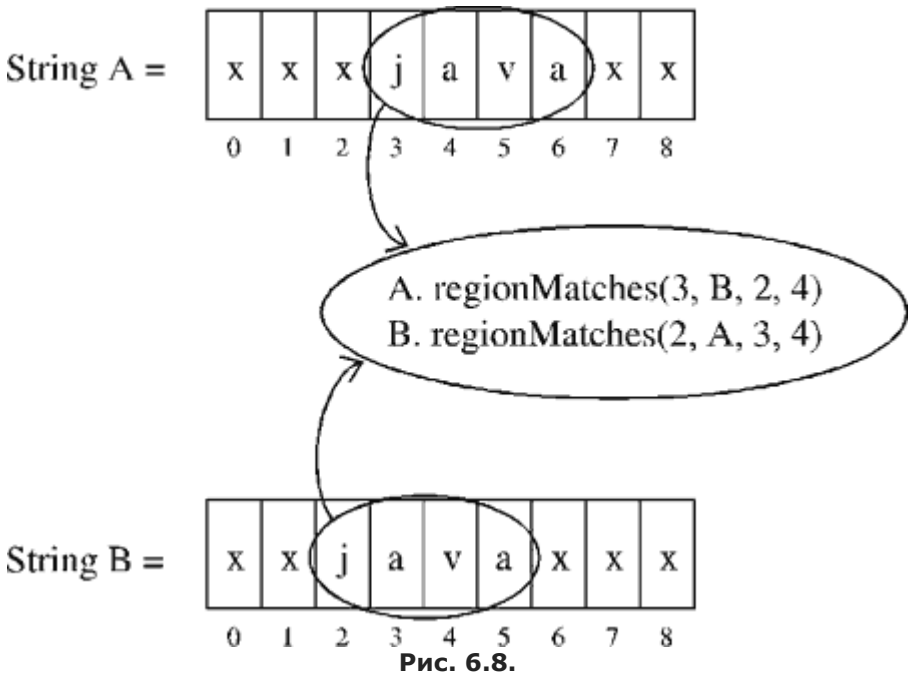

Рис. 6.8.

Когда мы начнем анализировать строки, вы увидите, что эти два метода сравнения чрезвычайно полезны. Их правые целые числа можно передавать как параметры. К примеру, их можно использовать для эмуляции методов endsWith и startsWith. Ниже приведен фрагмент кода, который всегда возвращает true.

Пример 6-3a. Подражание методу startsWith с помощью метода regionMatches.

```
int lengthB=B.length();
if (A.startsWith(B)==A.regionMatches(0,B,0,lengthB))
// всегда true
```

Еще лучше проиллюстрировать, как добиться правильных параметров, можно, сделав эквивалент метода endsWith при помощи метода regionMatches.

Пример 6-3b. Подражание методу endsWith с помощью метода regionMatches.

```
int lengthB=B.length();
int startA=A.length()-B.length();
if (A.endsWith(B)== A. regionMatches( startA, B, 0, IengthB))
DealingWith Substrings
```

Работа с подстроками

Мы только что видели, как класс String позволяет сравнивать подстроки. Существует также множество методов для извлечения подстрок. Давайте начнем с простых методов работы с подстроками:

public String substring(int start)
public String substring(int start, int len)

Первый метод из показанных выше возвращает новую строку, которая начинается с первой позиции в символьном массиве и содержит все символы после этой позиции. Второй метод возвращает только len символов после начала. Первый символ строки находится в нулевой позиции. Рассмотрим некоторые примеры:

```
String S="0123456789";
String S1=S.substring(0);
String S2=S;
// строки SI, S2 и S3 эквивалентны.
String S4=S.substrilng(4);
// S4 имеет значение "456789"
```

```
String S5=S.substring(4,3);
// S5 имеет значение "456"
```

Методы работы с подстроками неудобны единственно тем, что им всегда нужно передавать параметры. Это тот случай, когда можно применить методы indexOf. Методы indexOf позволяют вам исследовать класс String на индивидуальные символы или строки и возвращать индекс массива. Возвращенное целочисленное значение может затем быть передано методам работы с подстроками для извлечения или к методам regionMatches для сравнения. В табл. 6-6 описаны различные методы indexOf.

Таблица 6-6. Методы indexOf класса String

Объявление метода	Параметры	Возвращаемое значение
lilt indexOf(char a)	a - символ для поиска.	Индекс первого местонахождения.
IndexOf(char a, int start)	a - символ для поиска, start - начальная позиция для поиска.	Индекс первого местонахож-дения от начальной позиции.
IndexOf(String S)	S - строка для поиска.	Индекс первого местонахождения S.
indexOf(String S, int start)	S - строка для поиска, start - начальная позиция для поиска.	Индекс первого местонахождения от начальной позиции.

Метод lastIndexOf()

Для каждого метода indexOf имеется соответствующий метод lastIndexOf. Как и обязывает его имя,, метод lastIndexOf начинает поиск от конца строки.

Изменение строк

На протяжении этой главы мы изменяли строки следующими операторами:

```
String S="some string";
S=S+", more of the string",
S=S.substring(14,4);
// присвоение подстроки первоначальной строке
```

Обратите внимание, что, когда мы делаем эти перестановки, мы должны присвоить результат первоначальной строке. Этот шаг необходим, потому что, однажды созданный, экземпляр строки не может быть изменен. Так что мы создаем новую копию строки, вызывая один из методов класса String, и затем присваиваем новую копию обратно первоначальной переменной.

Кроме методов работы с подстроками, есть еще четыре других метода, которые мы можем использовать для перестановок. Эти методы приведены в табл. 6-7. Не забудьте, что в каждом случае создана копия строки, перестановки сделаны на этой копии и копия возвращена обратно. Ни один из методов в классе String фактически не изменяет строки непосредственно - взамен вы присваиваете новую копию или той же самой, или другой переменной.

Таблица 6-7. Методы изменения строк

Методы, используемые со строкой S1	Параметры	Возвращаемое значение
S1.concat(S2)	S2 - строка, которая будет конкатенирована к S1.	SI+S2
S1.replace(a,b)	Символ b заменяет символ a.	S1 с замененным символом b.
S1.toLowerCase()	нет	S1 без символов верхнего регистра.
S1.toUpperCase()	нет	S1 без символов нижнего регистра.
S1.trim()	нет	S1 без начальных или конечных пробелов.

Разбор строк

Теперь у нас есть все инструменты для анализа строк. Мы можем сравнивать и, используя методы indexOf, отыскивать подстроки. Но давайте допустим, что строка может иметь несколько

вхождений одной и той же подстроки. Например, математическое выражение может иметь несколько вхождений знака +. Компьютерные специалисты отнеслись бы к такой подстроке как к разделителю. Единицы между разделителями называются токенами.

Токены, разделители и компиляция

Слово "токен" может встретиться вам в литературе, связанной с компьютерными языками. Компиляторы должны разбить файл исходного текста, используя разделители, запятые и незаполненное пространство. Простой пример - процесс разбора списка параметров. Компилятор сначала изолирует символы внутри круглых скобок. Затем, используя запятую в качестве разделителя, он может идентифицировать токены - индивидуальные объявления переменных. После этого он может начинать компилировать остальную часть подпрограммы.

С помощью методов indexOf и методов работы с подстроками, о которых мы уже говорили, можно было бы написать некоторый код, который бы анализировал токены, основываясь на специфическом разделителе. Но это было бы нелегкой работой. К счастью, пакет java.util содержит класс StringTokenizer, который разобьет данную строку, основываясь на данном разделителе. На рис. 6-9 показана интерактивная документация для StringTokenizer.

Рис. 6.9.

Предположим, что у нас есть строка "3 + 4 + 2 + 7" и нам нужно вычислить сумму целых чисел. Класс StringTokenizer расчленит строку, основываясь на знаках +, начиная слева.

Пример 6-4. Разбор строки с использованием класса StringTokenizer.

```
public int getSum(String S) {
int runningSum=0;
StringTokenizer tok=new StringTokenizer(S,"+");
// S - строка для разбора
// "+" - разделитель
while (tok.hasMoreTokens()) {
String thisToken=tok.getNextToken();
runningSum=Add(thisToken);
}
return runningSum;}
```

```
private int Add(String S) {
// мы должны знать преобразование примитивного типа
// прежде, чем сможем выполнить его
}
```

Как вы можете видеть, с помощью методов indexOf и regionMatches сделать то же самое было бы намного труднее. Мы можем резервировать использование методов indexOf и regionMatches, если мы должны расчленить строку только один раз. В табл. 6-8 описаны конструкторы, а в табл. 6-9 показаны основные методы, которые вы должны знать, чтобы эффективно использовать этот класс.

Таблица 6-8. Конструкторы класса StringTokenizer

Конструктор	Описание
StringTokenizer(String S, String delim)	Разбивает S, основываясь на delim.
StringTokenizer(String S, String delim, boolean returnDetims)	Как и выше, но если delims == true, разделители возвращаются как токены.
StringTokenizer(String S)	Разбивает строку, основываясь на незаполненном пространстве ("\t\n\r").

Таблица 6-9. Методы класса StringTokenizer

Метод	Описание
String nextToken()	Возвращает следующий токен.
String nextToken(String newDelim)	Возвращает следующий токен после переключения разделителя на newDelim.
boolean hasMoreTokens()	Возвращает true, если имеются токены, которые не были возвращены.
int countTokens()	Возвращает число токенов в строке.

Преобразование строк в другие типы данных

Программистам часто приходится делать преобразования между строками и примитивными типами данных, например целыми числами. Чтобы преобразовывать примитивные типы данных в строки, используются методы valueOf. Конкретный метод valueOf имеется для каждого из примитивных типов данных, и все они работают сходным образом. Давайте покажем преобразование значений типов int, float, double, boolean и char.

Пример 6-5a. Преобразование примитивных типов данных в строки.

```
int someInt=1;
String StringInt=String.valueOf(someInt);
float someFloat=9.99f;
String StringAsFloat=String.valueOf(someFloat);
// StringAsFloat имеет значение 9.99
// Обратите внимание на конечный f,
// отличающий его от типа double.
double someDouble=99999999.99;
String StringAsDouble=String.valueOf(someDouble);
// StringAsDouble имеет значение 999999999.99
boolean someBoolean=true;
String StringAsBoolean=String.valueOf(someBoolean);
// StringAsBoolean имеет значение true
char someChar='a';
String StringAsChar=String.valueOf(someChar);
// StringAsChar имеет значение "a"
```

Эти методы не отличаются от того обходного пути, который мы использовали раньше, чтобы присвоить строке значение примитивного типа с помощью пустой строки и знака операции +. Возможно, вы найдете эти методы полезными, чтобы сделать свой код немного более понятным. Но как преобразовать строки в примитивные типы? Как и можно было ожидать, для этого в API есть специальные статические методы - но они не находятся в классе String. Они содержатся в классах-упаковщиках (wrapper) примитивных типов, которые мы обсудим после. К примеру, рассмотрим преобразования для числовых примитивных типов, исключая short и byte.

Пример 6-5b. Преобразование строк к примитивным типам.

```
String intString="10";
String floatString="10.1f";
String longString="999999999";
String doubleString="999999999.9";
try {
        int i=Integer.parseInt(intString);
        float f=Float.valueOf(floatString).floatValue();
        long l=Long.parseLong(longString);
        double d=Double.valueOf(doubleString).longValue();
        } catch (NumberFormatException e) {
                System.out.println("invalid conversion attempted");}
```

Здесь нам не нужно волноваться относительно того, что наши строки являются недопустимыми, потому что мы сами присваиваем им значение. Но во многих случаях (например, когда вы получаете данные от пользователя) нужно быть подготовленным к тому, что строки формируются неправильно. NumberFormatException является исключением времени выполнения, так что вы не должны были бы его перехватывать. Но если вы это не сделаете, в один прекрасный день ваша программа даст сбой.

Вышеприведенный пример не охватывает преобразования строки в значения типов short и byte. Запомните, что все числовые типы данных могут приводиться к типам short и byte, как показано в следующем фрагменте кода.

Пример 6-6a. Преобразование строки в значения типов byte и short.

```
String S="0";
try {
        byte b=(byte)Integer.parseInt(S);
        // Усечение значения int к байту;
        // информация может быть потеряна.
        short s=(short)Integer.parseInt(S);
        // Усечение значения int к short;
        // информация может быть потеряна.
        } catch (NumberFormatException e) {
                // Работая с недопустимым значением S,
                // вы могли бы снова вызвать исключение.
                }
        }
```

СОВЕТ В вышеприведенном фрагменте кода для преобразования строк к значениям типов byte и short мы усекаем значение от одного примитивного типа данных до другого. Если I.intValue() возвращает значение, большее 256 или меньшее -255, наше значение типа byte будет неправильным. Аналогично, значение, большее 32768 или меньшее -32767, даст неправильное значение для s. Самый лучший способ обрабатывать такие ситуации - создавать свои собственные обработчики исключений. Этот фрагмент кода будет расширен в главе 10, "Структура программы".

Единственные примитивные типы, которые мы еще не рассмотрели, - это boolean и char. Преобразование в char тривиально - только используйте метод charAt, чтобы возвратить первый символ строки:

```
public char stringToChar(String S) {
return S.charAt(0);}
```

Преобразование к типу boolean - скорее числовое преобразование. К сожалению, метод valueOf класса Boolean не вызывает исключения, когда строка не напоминает булевское значение. Он возвращает true, когда строка содержит "true", и false во всех других случаях:

```
String truth="true";
String fallicy="false";
String whiteLie="anything else";
String nothingness=null;
boolean b1=Boolean.valueOf(truth).booleanValue();
```

```
// b1==tru
boolean b2=Boolean.valueOf(fallicy).booleanValue();
// b2==false
boolean b3=Boolean.valueOf(whiteLie).booleanValue();
// b3==false
boolean b4=Boolean.valueOf(nothingness).booleanValue();
// b4==false
```

Упаковщики примитивных типов

Язык Java состоит из объектно-ориентированных переменных и примитивных типов данных boolean, char, double, float, int и long. Классы, которые мы будем рассматривать ниже, объединяют эти два мира. Это классы Boolean, Char, Double, Float, Integer и Long. Как было показано выше, они необходимы для преобразования строк в значения примитивных типов. Кроме того, они полезны при использовании векторов и стеков, которые будут описаны в следующем разделе, а также при математических вычислениях - они содержат константы, которые представляют максимум, минимум, бесконечное значение и не-число. Мы рассмотрим эти константы позже в разделе "Математика и API" этой главы.

Структуры всех этих "классов-оболочек" очень похожи. Каждый упаковщик объявлен с модификатором final, что означает, что вы не можете создать в нем подклассы. Каждый класс может быть создан или со строкой, или со значением соответствующего примитивного типа данных в качестве единственного параметра. И каждый имеет методы, которые связывают его с помощью интерфейса с соответствующим примитивным типом. В примере 6-7 показаны два способа создания для классов Integer и Float.

Пример 6-7. Конструкция упаковщика примитивного типа.

```
int i=2;
String intString="3";
Integer I1=new Integer(i);
Integer I2=new Integer(intString);
float f=2.1f;
String floatString="3.2";
Float F1=new Float(f);
Float F2=new Float(floatString);
```

Классы-контейнеры

Если вы когда-либо изучали информатику, вы, вероятно, знакомы с хеш-таблицами и стеками. Если нет, считайте, что вам повезло, - используя Java API, вы можете пожинать плоды применения этих мощных средств, не сдавая экзаменов профессору. Классы, которые здесь описаны, - реализации языка Java для некоторых из наиболее мощных структур данных. Эти классы, называемые классами-контейнерами (Object Container), используются для группирования различных типов объектов.

Контейнеры позволяют помещать в себя сколько угодно объектов и предоставляют очень полезные функциональные возможности для размещения и поиска объектов. Класс Hashtable позволяет делать пары ключ-значение и обращаться к объекту по ключу. Класс Vector полезен, когда нужно создать много копий одного класса, но заранее количество этих копий неизвестно. Класс Stack позволяет следить за набором объектов по принципу "первый вошел - первый вышел".

Примитивные типы и классы-контейнеры

Примитивные типы не могут использоваться с контейнерами, потому что классы структур данных имеют дело исключительно с объектами. Чтобы поместить значение типа char, boolean, float, int или double в одну из этих структур данных, сначала заключите его в упаковщик соответствующего примитивного типа.

СОВЕТ Прочитав о классах-контейнерах, вы, вероятно, захотите использовать их для управления вашими собственными объектами. Чтобы сделать это правильно, вам нужно заменить методы equals и hashCode в классе Object. Мы обсудим это в главе 10, "Структура программы".

Класс Vector

Давайте начнем с самого простого контейнера данных - класса Vector. Пусть его имя не вводит вас в заблуждение - этот класс не зарезервирован для сложных научных или графических программ. Класс Vector является, по существу, расширяемым массивом. Разобравшись в этом классе, вы найдете для него много применений. Вы когда-нибудь нуждались в массиве некоторого типа данных, но не знали его величины? Класс Vector заботится об этом. В отличие от массива вы можете добавлять к вектору столько элементов, сколько вам нужно. Закончив добавлять элементы в вектор, вы можете создать массив правильного размера и скопировать все элементы в него. В табл. 6-10 приведены основные методы, которые нужно знать для выполнения подобной работы.

Таблица 6-10. Базовые методы класса Vector

Метод	Описание
void addElement(Object o)	Добавляет объект.
void insertElementAt(Object o, int i)	Вставляет объект o в позицию i. Элементы не уничтожаются, размер увеличивается на единицу.
void copyInto(Object someArray[])	Копирует все элементы из вектора в некоторый массив.
int length()	Возвращает число элементов в векторе.

Давайте рассмотрим эту структуру данных в действии. Помните апплет мультипликации из главы 4, "Синтаксис и семантика"? Мы использовали метод getParam класса Applet, чтобы получить URL изображения из HTML-страницы. Мы произвольно устанавливали число изображений в пять. С другой стороны, мы могли бы подсчитать число изображений и затем создать массив этого размера. Создавая вектор, мы можем добавлять каждое изображение, с которым мы сталкиваемся, в соответствующем параметре на HTML-странице. Теперь мы можем написать апплет мультипликации, который загружает любое число изображений.

Пример 6-8. Апплет мультипликации с произвольным числом изображений.

```
public class AnimatedApplet extends Applet {
        private Image imgs[];
public void init() {
               if (getImages()0) {
                               }
               }
public int getImages() {
               int indx=0;
               Vector imageVec=new Vector();
               while (getParameter("image"+indx)!=null) {
                       String S=getParameter("image"+indx);
                       try {
                       URL U=new URL(getDocumentBase(),S);
                       Image im=getImage(U);
                       imageVec.addElement(im);
               catch { showStatus(S+"is not a valid URL -. skipping"); }
        indx=indx+1;
        }
imageVec.copyInto(imgs);
return indx;
}
```

Теперь у нас есть удобный способ сохранить столько изображений, сколько мы захотим, и, кроме того, мы можем поместить их все в массив. При желании можно было бы оставить их в векторе. Так как, используя метод insertElementAt, мы можем обращаться к любому из хранящихся в нем изображений, наш вектор также функционален как массив. Этот подход нужно использовать, если мы ожидаем поступления дополнительных изображений позже. Но если мы не ожидаем новых элементов, обычно лучше применить метод copyInto. Проще определить то, что программа делает с массивом, чем с вектором. Применение массива более эффективно и с точки зрения использования памяти, так как обычно приходится сильно преувеличивать размер массива при первом его вызове.

Метод copyInto и производительность программы

Применяя метод copyInto, описанный выше, вы можете использовать меньшее количество

памяти, чем вам могло бы потребоваться в других случаях, но при этом нужно рассматривать и вопрос эффективности. Метод copyInto создает массив и затем копирует в него ссылки к объектам, содержащимся в векторе. Если у вас есть вектор, содержащий N элементов, должны выполниться N операций копирования.

Операция копирования не занимает много ресурсов,, так как при этом делаются копии не объектов,, а только ссылок на них. Это означает,, что в общем случае копируется только N * 8 байтов,, в зависимости от конкретной реализации компьютера. Конечно,, для очень большого N это могло бы вызвать задержку времени выполнения.

Сохранение неопределенного числа объектов - наиболее мощное использование класса Vector, но его можно применять и для других целей. В табл. 6-11 описаны методы, позволяющие делать точную копию всех векторов, искать специфические элементы, обращаться к элементам по их индексам и удалять все элементы. Обратите внимание на невозможность удалять только часть элементов вектора - метод удаления работает по принципу "все или ничего".

Таблица 6-11. Дополнительные методы класса Vector

Метод	Описание
boolean contains(Object o)	Если объект находится в векторе, вызывает o.equals (eachlObject) для каждого элемента.
int indexOf(Object o)	Возвращает расположение объекта или -1, если его нет в векторе. Использует метод equals как вышеприведенный.
int indexOf(Object o, int i)	То же что и предыдущий метод indexOf, но начинает поиск с позиции i.
int lastIndexOf(Object o)	Действует подобно int indexOf (Object o), но поиск начинается с последнего элемента.
int removeAllElements ()	Удаляет все элементы в векторе.
boolean isEmpty()	Возвращает true, если вектор пуст.

Так как класс Vector не имеет предопределенного размера, вы можете задаться вопросом, насколько он эффективен. Так как он является динамической структурой, мы не можем ожидать, что он будет эффективен как массив, - так как он растет, он будет постоянно нуждаться во все большем количестве памяти. По умолчанию место резервируется для 10 объектов. Каждый раз, когда требуется больше места, внутренний размер вектора удваивается. Это важно помнить, так как вы можете управлять расширением векторов, используя конструкторы и методы, описанные в табл. 6-12. Конструктор Vector(size int) просто создает вектор определенного размера. Если вы знаете, что будете иметь по крайней мере 100 элементов, создавайте вектор этого размера. Вы можете регулировать увеличение вектора, передавая значение второму конструктору, чтобы использовать эту величину как приращение размера. Вектор затем предоставит место для этого числа элементов каждый раз, когда потребуется его увеличение, вместо обычного удвоения. Управлять размером вектора после того, как он создан, можно с помощью методов ensureCapacity или setSize. Метод setSize опасен, если ваш вектор больше, чем новый размер, который вы определяете, потому что все элементы после указанного размера будут потеряны. Так что лучше использовать метод ensureCapacity.

Таблица 6-12. Эффективность и класс Vector

Метод / Конструктор	Описание
Vector(int size)	Создает вектор с размером в size объектов.
Vector(int size, int inc)	Создает вектор с размером в size объектов и устанавливает приращение inc.
ensureCapacity(int size)	Если емкость меньше, чем size, обеспечивает достаточно места для size объектов.
setSize(int size)	Устанавливает размер size, возможно, удаляя конечные элементы.

Мы еще не обсуждали следующую особенность векторов: в одном векторе можно помещать различные типы объектов, так как методы класса Vector, позволяющие помещать элементы в векторы, - insertAt и addElement - берут тип объекта как параметр. Так как все классы происходят из класса Object, все наши объекты могут приводиться к Object. Все классы-контейнеры обеспечивают эту особенность (полноценный пример будет приведен ниже, при обсуждении хеш-таблиц).

Хеш-таблицы

Точно так же, как векторы больше всего подходят для хранения неопределенного числа объектов, хеш-таблицы созданы для хранения неопределенного числа пар объектов. Хеш-таблица, как показано на рис. 6-10, имеет два столбца. Один столбец содержит ключи, другой - элементы. Каждый ключ связан с одним и только одним элементом. Если вы попытаетесь добавить в таблицу пару ключ-элемент, когда такой ключ уже есть в хеш-таблице, элемент, соответствующий ему, будет заменен. Поместив пары в хеш-таблицу, вы можете обращаться к их элементам, используя связанные с ними ключи. Другими словами, вы можете передавать в хеш-таблицу некоторый объект, и если этот объект является одним из ключей, будет возвращен связанный с этим ключом элемент. Эти базовые действия могут быть выполнены методами, приведенными в табл. 6-13.

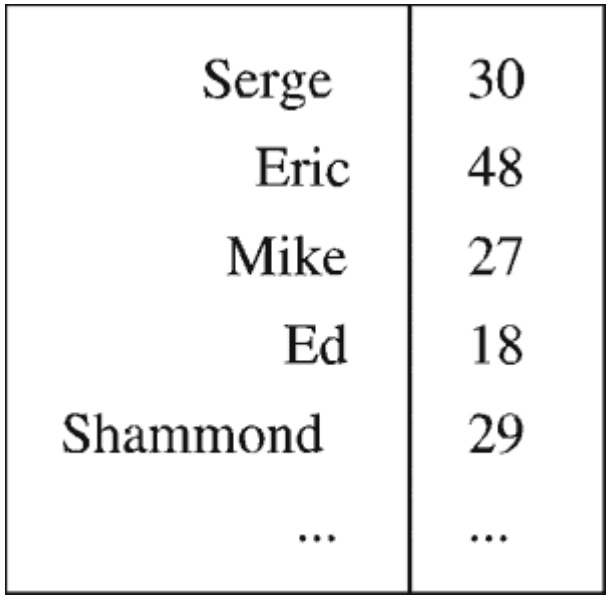

Рис. 6.10.

Таблица 6-13. Основные методы класса Hashtable

Метод	Описание
Object put(Object key, Object element)	Помещает пару ключ-элемент в хеш-таблицу. Если ключ уже был в таблице, возвращается связанный с ним элемент.
Object get(Object key)	Возвращает элемент, связанный с ключом.
boolean containsKey(Object key)	Возвраает true, если ключ находится в таблице.
Object remove(Object key)	Удаляет пару ключ-элемент из таблицы.

Давайте теперь сформируем хеш-таблицу, показанную на рис. 6-10. Сначала мы напишем простой код, который не делает никаких проверок, и затем сделаем его более устойчивым к ошибкам. Так как ключи представляют собой просто числа, мы должны использовать упаковщики примитивных типов. Обратите внимание на преобразование типов в этом примере. Завершив работу, мы сможем узнать возраст человека, если мы знаем его имя.

Пример 6-9а. Работа с хеш-таблицей.

```
class AgeManager {
private Hashtable H=new Hashtable();
// другие переменные
void updateAge(String person, int age) {
// имя не должно быть пустым!
Integer intWrap=new Integer(age);
        H.put((String)person,(Integer)intWrap);
return intWrap.intValue();
}
int getAge(String person) {
        // человек должен присутствовать в таблице!
        Integer intWrap=(Integer)H.get((String)person);
```

```
        return intWrap.intValue();
        }
```

Риск потерять смысл

Если вы явно не приводите переменные, когда помещаете их в хеш-таблицу, ваш код все равно будет откомпилирован. Хеш-таблица способна выбрать значения, основанные на ключах, - класс Object содержит необходимые методы, чтобы определить равенство. Однако ваш код будет проще, если вы будете следовать соглашению приведения переменных перед помещением их в хеш-таблицы.

Базовые методы класса Hashtable позволяют получать нужные числа из хеш-таблицы. Когда мы передадим имя объекту ageManager, он возвратит возраст. С другой стороны, мы должны создать объекты класса Integer, потому что хеш-таблицы не знают, как обращаться с примитивными типами данных. Теперь, когда мы знаем, как помещать и извлекать переменные из хеш-таблицы, можно сделать наши методы более устойчивыми.

Пример 6-9b. Более устойчивый getAge.

```
int getAge(String person) {
// Возвращает:
// возраст человека, если он имеется в таблице;
// -1, если человек не находится в таблице;
// -2, если person==null.
        if (person==null)
                return -2;
        if (!H.containsKey(person))
                return -1;
Integer intWrap=(Integer)H.get((String)person);
return intWrap.intValue();
}
```

Теперь мы должны защитить метод put от ошибок и дать вызывающей программе некоторую информацию о происходящем. Так как метод put возвращает объект, мы можем возвратить старое значение. Таким образом можно определить, добавляем ли мы новую пару возраст-имя или заменяем возраст кого-либо в нашей хеш-таблице.

Пример 6-9c. Измененный updateAge.

```
int updateAge(String person, int age) {
        // Возвращает:
        // возраст, если был изменен;
        // -1, если не изменен;
        // -2, если person==null.
if(person==null)
        return -2;
Integer intWrap=new Integer(age);
if (H.containsKey((String)person)) {
        intWrap=(Integer)H.put((String)person, (Integer)intWrap);
                return intWrap.intValue();}
        else {
        H.put((String)person,(Integer)intWrap);
return -1;
}
```

Как уже говорилось выше, приведение типов необходимо для соответствующей операции в хеш-таблице; теперь давайте посмотрим, почему оно выгодно. Соответствующее приведение типов позволяет нам помещать различные типы объектов в одну и ту же хеш-таблицу. Когда мы отыскиваем их, мы можем решить, объект какого типа мы хотим получить. Давайте покажем это в примере 6-10 с очень маленькой хеш-таблицей (здесь класс Vector - это суперкласс Stack).

Пример 6-10. Приведение типов и хеш-таблица.

```
Vector Vec=giveAsAVector();
Stack St=giveAsAStack();
// Stac  подкласс класса Vector
H.put("vec", (Vector)Vec);
H.put("stack",(Stack)St);
```

```
Stack st1=(Stack)H.get((String)"vec");
Stack st2=(Stack)H.get((String)"stack");
```

Мы получили два разных стека из нашей хеш-таблицы, хотя помещали только один:

```
Hashtable H=new Hashtable();
H.put((String)"stack", (Stack)st1);
H.put((String)"stack", (Stack)st2);
```

То же самое мы делаем со значением ключа, которое передаем методу get.

Итак, мы продемонстрировали основные функциональные возможности хеш-таблиц. Оставшиеся методы, определенные в классе Hashtable, приведены в табл. 6-14. С той информацией, что вы узнали из нашего обсуждения и из табл. 6-13 и 6-14, вы уже сможете широко использовать хеш-таблицы с классами API. Но для того, чтобы использовать их с классами, которые вы создаете сами, вам нужно разобраться в методе hashCode и в том, как он работает с хеш-таблицами (что будет описано в главе 10).

Таблица 6-14. Другие методы класса Hashtable

Метод	Описание
void clear()	Очищает хеш-таблицу.
Hashtable clone()	Делает копию хеш-таблицы.
boolean contains(Object o)	Возвращает true, если o - элемент хеш-таблицы.
boolean isEmpty()	Возвращает true, если хеш-таблица пуста.
int size()	Возвращает размер хеш-таблицы.

Стеки

Если вы заглянете внутрь любой сложной программы, вы где-нибудь да найдете стек. Стек в Java - просто стек объектов. Обращаться можно только к верхнему обьекту и добавлять объекты только к верхней части. Стеки наиболее полезны для слежения за прошлыми действиями программы. В табл. 6-15 приведены все методы этого класса.

Таблица 6-15. Методы класса Stack

Метод	Описание
Object push(Object o)	Помещает o в стек.
Object pop()	Выталкивает верхний элемент из стека.
Object peek()	Возвращает верхний элемент без его удаления.
boolean empty()	Возвращает true, если стек пуст.
int search(Object o)	Если o находится в стеке, возвращается расстояние до него от верха стека.

Давайте сделаем простой калькулятор с кнопкой Undo. В переменной sum будет храниться текущее значение, а класс Operator будет представлять собой суперкласс, из которого мы сможем получать допустимые операторы. У каждого оператора есть обратный метод. Допустим, что вся проверка ошибок выполнена до вызова вычисляющего метода:

```
class Calculator {
Double sum=null;
Stack prevActlvities;
// другие объявления
void compute(Operator op, Double nextValue)
        { sum=op.newValue(op, nextValue);
        prevActivities.push( (Operator)op);
        prevActivities.push( (Double)nextVal ue);
}
boolean undo() {
if (!prevActivities.empty()){
// Достаточно одной проверки, так как в стек
// два числа помещаются одновременно
        Double lastValue=(Double)prevActivities.pop();
```

```
        Operator lastOp=(Operator)prevActivities.pop();
        sum=op.newValue(lastOp.inverse(), lastValue);
        return true;
        }
else return false;}
}
```

Как вы можете заметить, стеки - это хороший способ следить за прошлыми действиями пользователя по принципу "первый вошел - первый вышел" (FIFO). Иногда возникает необходимость следить за действием пользователя на основе принципа "последний вошел - первый вышел" (LIFO). В этом случае нужно использовать класс Vector.

Интерфейсы API

Как уже говорилось в главе 3, "Объектная ориентация в Java", интерфейсы Java позволяют использовать набор методов класса без создания экземпляра класса. Интерфейс объявляет методы, а некоторый класс их реализует. Интерфейсы играют определяющую структурную роль в API, и есть два интерфейса в java.util - Enumeration и Observer, - которые являются особенно полезными. Изучив структурную роль интерфейсов и применение этих двух сервисных интерфейсов, вы сможете использовать интерфейсы в их наиболее полной форме, перемещаясь по другим Java-пакетам и разрабатывая свои собственные.

Особо важные интерфейсы

В исходных текстах для классов API можно часто встретить интерфейсы, используемые следующим способом:

```
public class Applet extends Panel {
// основной класс Applet
        private AppletStub stub;
{
public final void setStub(AppletStub stub) {
        this.sttub=(Appl etStub)stub;
{
// остальная часть класса Applet
}
```

Код берется непосредственно из основного класса Applet, который мы использовали для создания наших собственных апплетов в главе 5, "Апплет в работе". Что же делает этот код? По существу, он позволяет оболочке времени выполнения обрабатывать платформозависимые методы, определенные в интерфейсе AppletStub. Например, когда вы используете метод getParameter, фактически вызывается stub.getParameter:

```
public String getParametertString name) {
        return stub.getParameter(name);
}
```

При обращении к заглушке (stub) интерфейса оболочка времени выполнения должна иметь дело с проблемой восстановления параметра из броузера Web. Наши апплеты вообще не затрагиваются - оболочка времени выполнения передает им заглушку, и они используют ее.

Все интерфейсы в java.applet и java.awt.peer подчинены этой цели - независимости классов API от среды, в которой они выполняются. Хотя эти интерфейсы важны, они не имеют никакого реального применения, когда мы просто используем API. Например, если мы не будем программировать оболочку, в которой выполняются апплеты, мы никогда не столкнемся с AppletStub. Когда мы начнем разрабатывать наши собственные пакеты в главе 10, "Структура программы", мы, возможно, вспомним эту методику. Но пока постарайтесь не запутаться в интерфейсах из этих двух пакетов.

Интерфейс Enumeration

В отличие от интерфейса AppletStub интерфейс java.util.Enumeration полезен непосредственно в наших программах. Когда класс API, например Vector или Hashtable, возвращает Enumeration, обычно вы получаете список объектов. Вы можете затем работать с этим списком, выполняя нужную операцию на каждом из объектов. Красота этого решения состоит в том, что вы можете использовать методы интерфейса Enumeration со своим списком независимо от происхождения этих объектов.

Предположим, что нам нужно выполнять одну и ту же операцию с вектором, ключом или значением хеш-таблицы. Можно было бы помещать все ключи или все значения хеш-таблицы в вектор и затем проходить по всем его элементам, но во многих случаях удобнее использовать вместо этого интерфейс Enumeration.

Пример 6-11a. Интерфейс Enumeration.

```
public void printEnumeratlon(Enumeration e) {
        while (e.hasMoreElements())
                System.out.println(e.nextEIement.toString);
}
```

Чтобы использовать printEnumeration с хеш-таблицами или векторами, вы должны применить один из методов, приведенных в табл. 6-16.

Таблица 6-16. Возвращаемые элементы Enumeration

Класс	Метод	Возвращаемое значение
Vector	Enumeration elements()	Перечисление всех элементов вектора.
Hashtable	Enumeration keys()	Перечисление всех ключей хеш-таблицы.
Hashtable	Enumeration elements()	Перечисление всех элементов хеш-таблицы.

Чтобы использовать метод printEnumeration на практике, достаточно любого из следующих операторов (для хеш-таблицы H и вектора V):

Пример 6-11b. Использование интерфейса Enumeration.

```
Hashtable H=giveMeHashtable();
Vector V=giveMeVector();
printEnumeration( V.elements()); printEnumeration( H.keys() );
printEnumeration( H.elements());
```

В нашем простом примере printEnumeration мы использовали все методы интерфейса Enumeration. Так что мы не можем слишком полагаться на интерфейс Enumeration. Хотя он дает удобный способ работы с хеш-таблицами и векторами, мы можем пройтись по списку только один раз.

Интерфейсы java.lang.Clonable и java.lang.Runnable

Эти два интерфейса играют, прежде всего, структурную роль, но не такую примитивную, как в java.applet и java.awt.peer. Когда мы видим, что некий класс реализует интерфейс Cloneable, это значит, что мы можем использовать метод clone для создания копии этого класса. Давайте создадим копию экземпляра класса Vector, который позволяет производить клонирование (здесь V1 - экземпляр этого класса, существовавший до того, как мы попробовали сделать копию).

Пример 6-12. Использование метода clone.

```
Vector V1=new Vector();
String V1Element="First element in V1";
V1.addElement(V1);
Vector V2=(Vector)V1.clone();
String V2Element="Second element in V2";
System.out.println("Vector V1");
EnumerationPrinter.printEnumeration(V1.elements());
System.out.println("-----");
System.out.println("Vector V2");
EnumerationPrinter.printEnumeration(V2.elements());
System.out.println("-----");
```

Здесь мы используем метод printEnumeration, чтобы отобразить содержание вектора. Вывод этого фрагмента кода следующий:

```
Vector VI
First Element in VI
----------------------
Vector V2
First Element in VI
Second Element in V2
----------------------
```

Если бы мы заменили оператор clone простым назначением, V1 стал бы идентичен V2 и наш вывод был бы следующим:

```
Vector VI
First Element in VI
First Element in V2
----------------------
Vector V2
First Element in VI
First Element in V2
```

Хотя интерфейс Cloneable важен, он не так важен, как метод clone. Когда мы видим, что класс реализует интерфейс Cloneable, мы знаем, что автор этого класса обеспечивает клонирование. Это правильно и для интерфейса Runnable, но важно то, что класс Runnable поддерживает потоки. Мы уже видели пример этого в главе 5, "Апплет в работе", когда сделали наш апплет способным к управлению мультипликацией. Мы полностью исследуем интерфейс Runnable и проблемы, вызываемые наличием многопотоковых программ, в главе 10, "Структура программы".

Обработка событий при помощи java.util.Observer

Интерфейс java.util.Observer разработан для того, чтобы помочь нашей программе управлять объектами. Объединив его с классом java.util.Observable, мы сможем создавать семейство классов, которые следят друг за другом. Экземпляр класса или подкласса Observable следит за реализациями классов, которые выполняют интерфейс Observer, как показано на рис. 6-11.

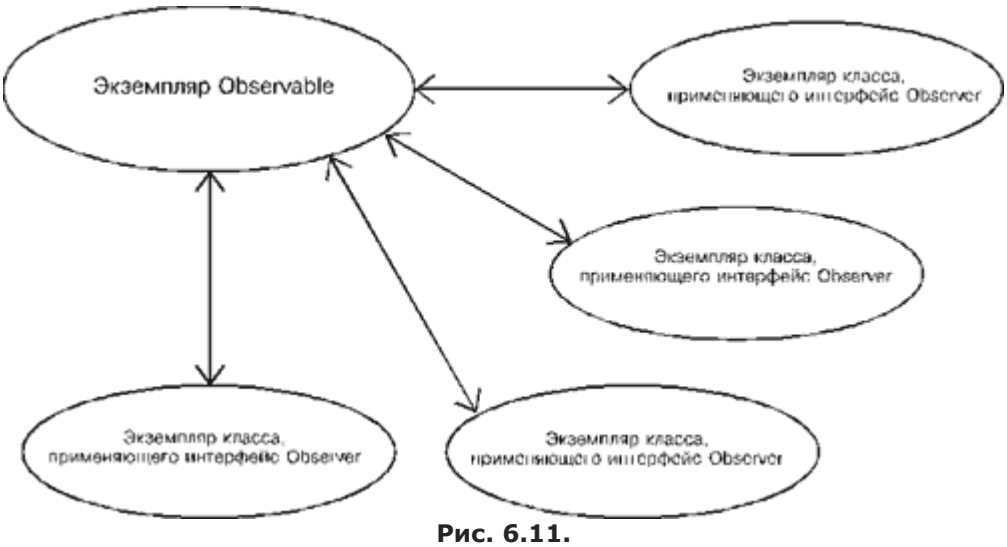

Рис. 6.11.

Используя этот интерфейс, мы можем создать нашу собственную систему обработки событий, подобную той, что мы использовали в классе Applet. Предположим, что нам нужна программа, в которой действия одного объекта имеют связь со многими другими объектами. Чтобы использовать подходящий реальный, всем понятный пример, скажем, что нам нужен объект, представляющий игру в баскетбол. В мире баскетбола игра имеет отношение к большому количеству объектов - болельщикам, прессе, корзине, мячу, ООН, орбитальным космическим кораблям. Если мы создаем классы, представляющие каждый из этих объектов, мы должны

выполнить интерфейс Observer так, чтобы каждый объект в нашей системе мог быть легко информирован о ходе игры. Затем мы напишем наш игровой класс, который, когда что-нибудь происходит, сообщает объекту Observable информацию для всех других объектов.

Пример 6-13a. Использование класса Observable.

```
public class BigGame {
        private Observable onlookers;
        private int teamAScore;
        private int teamBScore;
        private String teamWinning;
        // переменные, конструкторы
        public void setObservable(Observable o) {
                onlookers=o;}
        public void updateScore(int teamA, int teamB) {
                teamAScore=teamA;
                teamBScore=teamB;
                // должен быть выполнен, когда счет изменен
                onlookers.notifyObservers(this);
                // передаем игру всем зрителям
                }
public String getWinningTeam() {
        return new String("Carolina");}
}
```

Как следует из определения метода update, различные классы, которые выполняют Observer, определяют то, что они должны делать, когда им сообщают об изменениях в игре.

Пример 6-13b. Использование интерфейса Observer.

```
class UNCFan implements Observer{
        // переменные, конструкторы
        public void update(Observable o, Object Game) {
                BigGame BG=(BigGame)Game;
                String winningTeam=BG.getWinningTeam();
                if (winningTeam.equals("UNC"))
                        System.out.println("yeaaaa!!!");
                else
                        System.out.println("booo!!!!!");
                }
        // другие методы
        }
class DukeFan implements Observer {
        // переменные, конструкторы
        public  void update(Observable o, Object Game) {
        BigGame BG=(BigGame)Game;
        String winningTeam=BG.getWinningTeam();
                if (winningTeam.equals("Duke"))
                        System.out.println("yeaaa!!!!");
                else
                        {System.out.println("booo!!!");
                        }
}
```

В начале игры мы создали бы класс Observable и использовали метод addObserver, чтобы включить зрителей в игру.

Пример 6-13c. Добавление объектов выполнения Observer.

```
Observable onlookers=new Observable();
onlookers.addObserver(UNCFanInstance);
on lookers.addObserver(DukeFanlnstance);
on lookers.addObserver(Press);
on Iookers.addObserver(UnitedNations);
// и т. д, и т. п.
```

Кроме метода addObserver, есть еще несколько других методов, помогающих управлять объектами. Они приведены в табл. 6-17.

Таблица 6-17. Методы класса Observable

Метод	Описание
void addObserver(Observer o)	Добавляет наблюдателя, чтобы следить за экземпляром Observable.
void deleteObserver(Observer o)	Удаляет наблюдателя.
void notifyObservers(Object arg)	Сообщает наблюдателям, что что-то случилось; чтобы узнать, что именно, они могут исследовать arg.
int countObservers()	Возвращает число наблюдателей.
boolean hasChanged()	Возвращает true, если что-то изменилось; setChanged и clearChanged должны использоваться, чтобы устанавливать и сбрасывать флажок.
void setChanged()	Заставляет hasChanged возвращать true.
void clearChanged()	Заставляет hasChanged возвращать false.

Математика и API

Мы рассмотрели базовые операторы в главе 2, "Основы программирования на Java", но рано или поздно вам понадобится выполнять более сложные операции. Класс Math пакета java.lang содержит 30 арифметических методов, например, методы нахождения максимума или минимума пары чисел или получения натурального логарифма. Он также содержит значения для чисел e и пи в виде статических переменных.

На рис. 6-12 показана Web-страница класса Math в интерактивной документации по Java. Вместо того чтобы пытаться описать методы более подробно, мы сосредоточимся на том, как использовать их на практике.

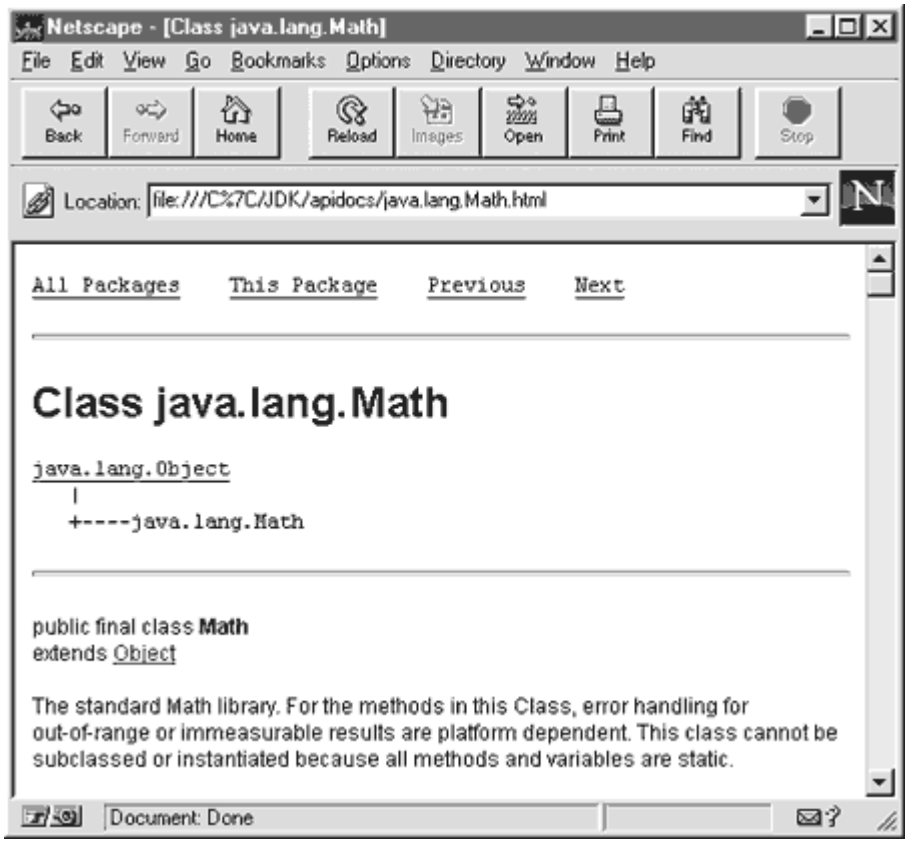

Рис. 6.12.

Пример 6-14. Использование класса Math.

```
public class DoSomeMath {
// другие объявления
public void useMathMethods(int i1,int i2, double d, float f,long l) {
```

```
int intAbsValue=Math.abs(i1);
double doubAbsValue=Math.abs(d);
float floatAbsValue=Math.abs(f);
long longAbsValue=Math.abs(l);
// Они возвратят абсолютное значение.
// Метод abs !!!перегружен.
int minInt=Math.min(i1,i2);
double maxDoub=Math.max(i1,d);
// Методы min и max перегружены для каждого числового типа.
// Во второй строке i1 приводится к типу double.
}
```

Как мы говорили выше, мы можем получать значения e и пи из класса Math. Классы-упаковщики также содержат некоторые значения, представляющие для нас интерес, например статические переменные для максимального значения каждого примитивного типа и другие полезные величины. Все они описаны в табл. 6-18.

Таблица 6-18. Полезные статические переменные

Переменная	Значение	Классы, содержащие данную переменную
E	Основание естественного логарифма.	Math
PI	Значение числа пи.	Math
MIN_VALUE	Минимальное значение для данного типа.	Integer, Float, Long, Double
MAX_VALUE	Максимальное значение для данного типа.	Integer, Float, Long, Double
NEGATIVE_INFINITY	Представляет отрицательную бесконечность.	Float, Double
POSITIVE_INFINITY	Представляет положительную бесконечность.	Float, Double
NaN	"Не число"; может использоваться, чтобы представить неназначенные значения.	Float, Double

Что дальше?

Мы только начали исследовать API. Теперь, когда вы узнали основы API, мы можем перейти к пакетам абстрактного набора инструментальных средств для работы с окнами. Мы опишем их в главе 7, "Пользовательский интерфейс", главе 8, "Еще об интерфейсе пользователя", и главе 9, "Графика и изображения". Вся часть книги о работе с сетями (часть IV, "Java и Сеть") посвящена пакетам java.io и java.net.

Глава 7

Пользовательский интерфейс

В этой главе мы приступаем к изучению особого компонента языка, называемого "Advanced Windowing Toolkit" ("Инструментарий для построения оконного интерфейса", AWT). Начнем мы с основных элементов экранного интерфейса пользователя (Graphical User Interface, GUI), определенных в пакете java.awt. Апплеты, которые мы напишем на протяжении этой главы, будут использовать следующие компоненты AWT:

- ввод с клавиатуры,
- поля для текстового ввода из одной или нескольких строк,
- кнопки,
- флажки,
- списки,
- всплывающие меню,
- полосы прокрутки,
- статичные текстовые надписи.

Java позволяет использовать эти базовые элементы интерфейса пользователя с гораздо большим удобством и эффективностью, чем HTML. Одним из очень важных инструментов, доступных Java-программисту, является эмуляция ввода пользователя. Все перечисленные выше элементы позволяют отслеживать перемещение курсора мыши, что делает их более интерактивными. Например, программа может вывести какой-нибудь поясняющий или образцовый текст в поле редактирования в тот момент, когда пользователь проносит над ним курсор мыши. Следующая глава, "Еще об интерфейсе пользователя", развивающая материал данной главы, посвящена тому, как реализовать более сложные элементы интерфейса. Затем, когда мы перейдем к изучению взаимодействия с сетью (часть 4, "Java и Сеть"), вы уже будете уметь создавать эффективный и профессионально выглядящий интерфейс для своих программ и сможете приступить к созданию апплетов, реализующих связь с удаленными сетевыми серверами (в том числе и серверами протокола HTTP).

На протяжении этой главы мы создадим несколько версий одного и того же апплета, демонстрируя применение различных элементов экранного интерфейса. Апплет наш предназначен для пересчета денежных сумм из одной валюты в другую; его основная функция - принять от пользователя число (денежную сумму в некоей валюте), умножить его на определенный курс (коэффициент пересчета) и возвратить полученный результат.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Сравнение интерфейса пользователя Java и бланков на языке HTML

Если вам приходилось работать с бланками на языке HTML и составлять программы для работы на сервере, удовлетворяющие стандарту Common Gateway Interface (CGI), учтите, что возможности Java в этом отношении не идут ни в какое сравнение с возможностями HTML. Java обеспечивает более широкий набор инструментов и функций, а также обладает всеми преимуществами объектно-ориентированного подхода к дизайну интерфейса. Кроме того, уже написанные приложения, реализующие интерфейс CGI, могут использоваться совместно с Java-апплетами, включенными в HTML-страницу. Таким образом, вы можете перейти к разработке Java-апплетов вместо существующих HTML-бланков без необходимости существенно менять CGI-программы, обрабатывающие пользовательский ввод. Так, практический пример апплета, описанного в главе 17, использует ссылку на URL-адрес и метод POST протокола HTTP для посылки данных, введенных пользователем, специализированной CGI-программе на сервере.

Более того, теперь при необходимости обрабатывать данные из заполненных бланков вам даже не обязательно прибегать к CGI-программированию. Вы можете обрабатывать эти данные прямо в Java-апплете, работающем на компьютере пользователя. Тем самым можно заметно снизить нагрузку на Web-сервер - теперь ему не придется принимать данные и запускать CGI-программы каждый раз, когда пользователю потребуется поработать с бланком. Дополнительным преимуществом этого подхода является то, что Java-апплет может напрямую обращаться к любому Web-серверу. Характерный пример, в котором эта возможность может оказаться полезной, - интерфейс к базе данных, обычно реализуемый с помощью CGI: после того как пользователь заполнил бланк и отослал его, на Web-сервере запускается специальная CGI-программа, принимающая эти данные и отсылающая их на другой сервер, на котором и расположена база данных. Теперь Java-апплет может сразу обращаться к серверу базы данных, не загружая без нужды Web-сервер. Java позволяет напрямую обращаться к серверам баз данных с запросами, и уже существуют пакеты, предназначенные специально для этого.

Апплет пересчета денежных сумм

Представьте, что мы выполняем заказ туристического агентства, которое хочет, чтобы его клиенты через World Wide Web могли знакомиться с предлагаемыми маршрутами путешествий и ценами на билеты. Чтобы привлечь больше посетителей на свою страницу, агентство хочет предусмотреть удобную возможность пересчитывать денежные суммы из одной валюты в другую (например, для клиентов из других стран, которые хотят знать цены на услуги в знакомых им денежных единицах). Вы как администратор Web-сервера получаете задание разработать соответствующий механизм. Обдумав возможные варианты, вы приходите к выводу, что Java-апплет позволит сделать это гораздо удобнее и эффективнее, чем HTML-бланк в сочетании с CGI-программой. Процесс разработки такого апплета мы и рассмотрим в этой главе. По ходу дела вы познакомитесь с содержимым и возможностями классов AWT.

Следующий раздел мы начнем с выяснения того, как реализовать в апплете перехват ввода пользователя с клавиатуры, а затем рассмотрим общую структуру апплета.

Ввод с клавиатуры

В главе 5, "Апплет в работе", мы рассматривали концепцию событий и их обработки Java-программой. Как вы помните, события одного типа, такие как проход курсора мыши над какой-то областью, могут относиться к некоторому объекту, в то время как другие осуществляют взаимодействие между объектами. Кроме того, события можно использовать для построения интерфейса пользователя, а именно - для перехвата воздействий пользователя на объекты интерфейса.

На рис. 7-1 показан простейший прототип апплета пересчета денежных сумм. В этой версии пользователь просто печатает число в текстовом поле ввода и нажимает Enter, чтобы увидеть результат пересчета. Для перехвата нажатия клавиши мы используем обработчик событий keyDown, в котором производится проверка на равенство нажатой клавиши значению "\n", что позволяет отфильтровать нажатие Enter и предусмотреть на эту клавишу особую реакцию. (Подробное описание классов TextField и TextArea вы найдете в следующем разделе.) Исходный текст этой версии нашего апплета приведен в примере 7-1.

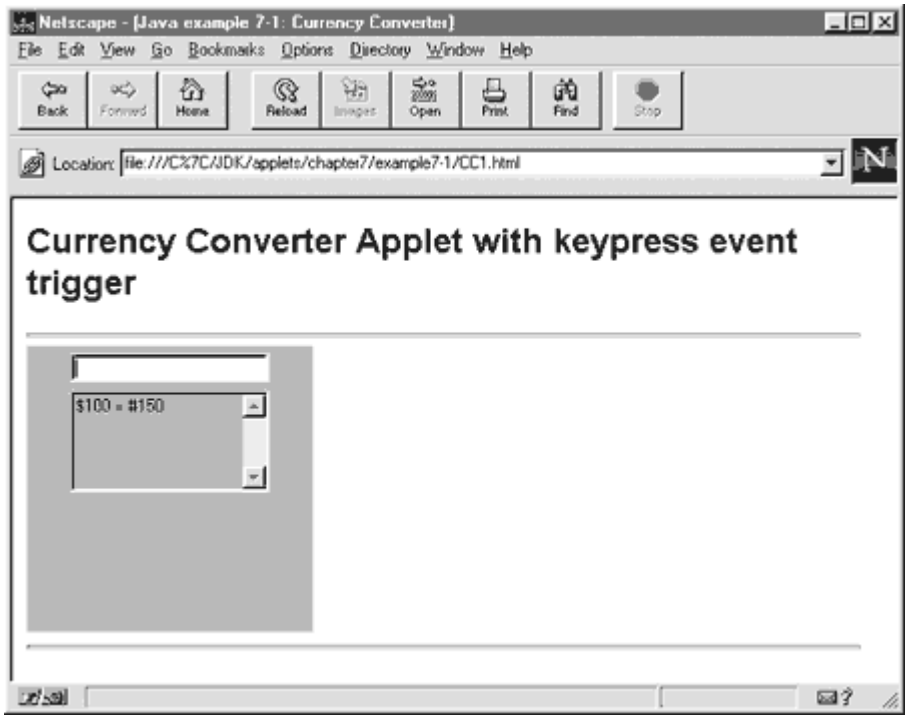

Рис. 7.1.

Пример 7-1. Апплет пересчета денежных сумм, перехватывающий событие нажатия клавиши.

```
import java.awt.*;
import java.applet.Applet;
public class currencyConverter1 extends java.applet.Applet {
        float conversion_ratio = 1.50f;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        // элементы экранного интерфейса для ввода/вывода
        // (см. следующий раздел)
public void init() {
        field1.setEditable(true);
        field2.setEditable(false);
        field1.resize(field1.preferredSize());
        field2.resize(field2.preferredSize());
        add(field1);
        add(field2);
show();
        } // конец init
public void convert() {
        float currency1, currency2;
```

```
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString =
        "$" + InputString + " = " + "#" + Float.toString(currency2) + "\n";
        field2.appendText(OutputString);
} // конец convert
public boolean keyDown(Event evt, int key)
        // обработчик события нажатия клавиши
{char c=(char)key;
        // преобразуем код клавиши к типу char для сравнения
        if (c == '\n')
        // если нажата клавиша Enter, запускаем convert()
        { convert();
        return true;
        }
        else { return false; }
        } // конец KeyDown()
} // конец апплета currencyConverter1
```

В этом примере мы используем обработчик событий keyDown, перехватывающий событие нажатия клавиши. Ниже, в разделе "Флажки", мы также будем использовать нажатие клавиши Enter для запуска процедуры пересчета, но тогда мы будем пользоваться методом handleEvent. Этот метод представляет собой универсальный обработчик событий, с помощью которого можно перехватывать даже такие события, для которых в программе уже есть другие обработчики, и, таким образом, отбирать события, реакцию на которые вы хотите предусмотреть. (В примере, использующем флажки, вы увидите, как можно пользоваться этим обработчиком для перехвата нажатия клавиши Enter и щелчка мыши по флажку.) Таким образом, для перехвата событий существует несколько способов, и во многих случаях они вполне взаимозаменяемы как по своим функциям, так и по скорости работы. В тех апплетах, где вам будет нужно перехватывать много разных событий, наилучшим выбором будут универсальные обработчики событий, такие как handleEvent или action. Такие обработчики позволяют собрать весь код обработки событий в одном месте, что обычно выгодно с точки зрения удобочитаемости текста апплета и скорости его работы. Передача управления фрагменту, отвечающему за обработку конкретного события, чаще всего осуществляется оператором switch.

Кроме того, с помощью метода handleEvent можно обрабатывать целые последовательности событий, а также учитывать пространственные координаты событий в рамках апплета. Мы воспользуемся этим в примере, использующем флажки, в котором обработчик, перехватив событие, проверяет координаты курсора мыши, чтобы понять, где произошел щелчок - над флажком или где-то в другом месте. Множество других примеров обработки событий вы найдете как в главе 8, "Еще об интерфейсе пользователя", так и в других главах книги, особенно тех, в которых разрабатываются учебные апплеты.

Поля редактирования текста

Поле редактирования типа TextArea может использоваться в апплете как для вывода, так и для ввода и редактирования текста. Поля редактирования этого типа состоят из нескольких строк текста и имеют полосы прокрутки. Напротив, поля редактирования типа TextField состоят из одной строки и не имеют полос прокрутки. Как TextField, так и TextArea являются наследниками класса TextComponent и, за исключением упомянутых различий, во всем аналогичны друг другу. Поле типа TextField использовалось в приведенном выше примере апплета:

```
TextField field1 = new TextField(20);
field1.setEditable(true);
```

В первой строке создается поле типа TextField шириной 20 символов. Во второй строке этому полю присваивается атрибут, разрешающий редактирование текста в нем. Если заменить во второй строке true на false, пользователь не сможет редактировать текст в поле.

Затем мы создаем поле типа TextArea шириной в 20 символов и высотой в 4 строки:

```
TextArea field2 = new TextArea(4,20);
```

Чтобы инициировать полосы прокрутки у этого поля редактирования, в программе ничего предусматривать не нужно, так как любой объект класса TextArea при создании автоматически получает этот элемент интерфейса.

Вывод текста в поле редактирования из программы осуществляется одним из двух методов:

```
field1.setText("");
```

```
field2.appendText(OutputString);
```

Метод setText позволяет установить содержимое поля редактирования (в нашем случае пустая пара двойных кавычек позволяет очистить поле редактирования). Метод appendText позволяет приписать текстовую строку в конец текущего содержимого поля редактирования. Однако функция appendText() не начинает новую строку, если поле редактирования имеет в высоту несколько строк; чтобы перейти на новую строку, вы сами должны вставить в редактируемое значение символ новой строки.

Теперь нам следует позаботиться о размерах созданных полей редактирования. Дело в том, что, хотя каждое поле имеет свой изначальный, "естественный" размер, условия работы апплета могут привести к тому, что фактический размер будет другим. Чтобы явным образом приказать полю редактирования принять его натуральный размер, мы воспользуемся методом resize, а естественные размеры поля редактирования выясним с помощью метода preferredSize:

```
field1.resize(field1.preferredSize());
field2.resize(field2.preferredSize());
```

Теперь, когда объекты TextField и TextArea созданы и инициализированы, мы должны добавить их к интерфейсу нашего апплета. Для этого используется метод add. И наконец, чтобы вывести на экран все добавленные таким образом компоненты интерфейса, используется метод show. Вот что мы должны написать, чтобы увидеть созданные объекты field1 и field2:

```
add(field1);
add(field2);
show();
```

Существует много других методов, которые можно использовать для работы с полями редактирования. В табл. 7-1 перечислены методы, имеющиеся в обоих классах - как в TextField, так и в TextArea (все эти методы унаследованы из класса TextComponent).

Таблица 7-1. Методы класса TextComponent

Метод	Описание
getSelectedText()	Возвращает фрагмент текста, выделенный в содержимом поля.
getSelectionEnd()	Возвращает номер последнего выделенного символа.
getSelectionStart()	Возвращает номер первого выделенного символа.
getText()	Возвращает все содержимое поля.
isEditable()	Возвращает булевское значение, показывающее, разрешено ли редактирование в поле.
paramString()	Возвращает значение типа String, содержащее параметры поля.
select(int, int)	Возвращает фрагмент текста поля, расположенный между указанными начальным и конечным символами.
selectAll()	Выделяет весь текст в поле.
setEditable(boolean)	Запрещает или разрешает редактирование в поле в зависимости от передаваемого булевского значения.
setText(String)	Выводит указанный текст в поле.

Класс TextField содержит меньше специфических методов, чем класс TextArea (так как некоторые из операций с многострочным полем редактирования не имеют смысла для поля из одной строки). Методы класса TextField перечислены в табл. 7-2.

Таблица 7-2. Методы класса TextField

Метод	Описание
echoCharIsSet()	Возвращает true, если данное поле имеет эхо-символ.
getColumns()	Возвращает ширину поля в символах.
getEchoChar()	Возвращает эхо-символ.
minimumSize(int)	Возвращает минимальный размер, который может иметь поле с указанной шириной.
minimumSize()	Возвращает минимальный размер данного поля.
paramString()	Возвращает значение типа String, содержащее параметры поля.
preferredSize(int)	Возвращает естественный размер поля с указанной длиной.

preferredSize()	Возвращает естественный размер данного поля.
setEchoCharacter(char)	Устанавливает эхо-символ поля.

Класс TextArea имеет несколько разновидностей конструкторов. Один из них стоит в приведенном выше примере программы. В табл. 7-3 перечислены конструкторы класса TextArea, а в табл. 7-4 - методы этого класса.

Таблица 7-3. Конструкторы класса TextArea

Конструктор	Описание
TextArea()	Создает новый объект типа TextArea.
TextArea(int, int)	Создает новый объект с указанным числом строк и символов в строке.
TextArea(String)	Создает новый объект, содержащий указанный текст.
TextArea(String, int, int)	Создает новый объект с указанным числом строк и символов в строке, содержащий указанный текст.

Таблица 7-4. Методы класса TextArea

Метод	Описание
appendText(String)	Добавляет указанный текст в конец текста, содержащегося в поле.
getColumns()	Возвращает число символов в строке поля.
getRows()	Возвращает число строк в поле.
insertText(String, int)	Вставляет указанный текст после символа с указанным номером.
minimumSize(int, int)	Возвращает минимальный размер, который может иметь поле с указанным количеством строк и символов в строке.
minimumSize()	Возвращает минимальный размер данного поля.
paramString()	Возвращает значение типа String, содержащее параметры поля.
preferredSize(int, int)	Возвращает естественный размер поля с указанным количеством строк и символов в строке.
preferredSize()	Возвращает естественный размер данного поля.
replaceText(String, int, int)	Заменяет текст, расположенный между символами с указанными номерами, на указанный текст.

Кнопки

Предыдущая версия апплета пересчета денежных сумм использовала для запуска пересчета нажатие клавиши Enter. Давайте теперь изменим апплет, добавив в него кнопку (button), с помощью которой пользователь сможет инициировать пересчет после ввода исходного значения. Вот как создается кнопка с надписью "Convert":

```
Button ConvertButton = new Button("Convert");
```

Именно так создается кнопка в примере 7-2 ниже. В этой версии мы не будем предусматривать перехват нажатия Enter, поскольку теперь вычисления запускаются только щелчком мыши по кнопке. Внешний вид этой версии апплета показан на рис. 7-2.

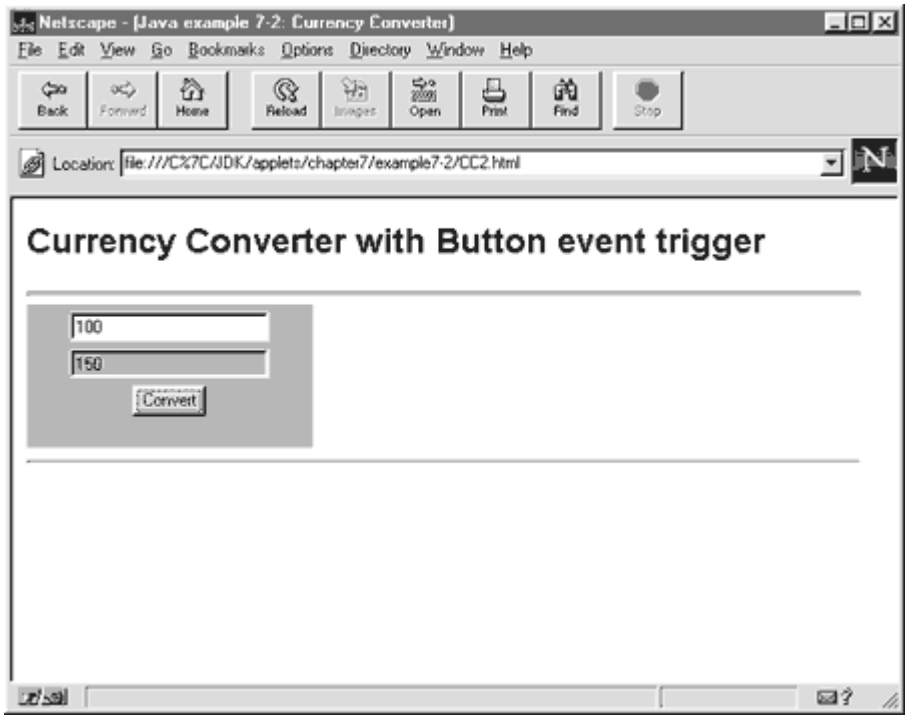

Рис. 7.2.

Пример 7-2a. Апплет пересчета денежных сумм, содержащий кнопку.

```
import java.awt.*;
import java.applet.Applet;
public class currencyConverter2 extends java.applet.Applet {
        float conversion_ratio = UKratio;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
Button ConvertButton = new Button("Convert");
public void init() {
        field1.setEditable(true);
        field2.setEditable(false);
        field1.resize(field1.preferredSize());
        field2.resize(field2.preferredSize());
        add(field1);
        add(field2);
        add(ConvertButton);
        show(); // вывод на экран созданных элементов
} // конец метода init
```

Этот фрагмент кода обеспечивает создание кнопки с надписью "Convert", при нажатии которой введенное значение пересчитывается и выводится в соответствующее поле. Метод init используется для создания апплета. В поле field1 пользователь может вводить свое значение, а поле field2 предназначено для вывода и защищено от редактирования. Элементы экранного интерфейса добавляются в апплет с помощью метода add. Любой элемент для вывода на экран обязательно должен быть добавлен к некоторому контейнеру (в нашем случае контейнером по умолчанию является окно апплета). Метод resize приводит текстовые поля редактирования к заказанному для них размеру (в нашем случае 20 символов) с учетом ширины окна броузера.

Собственно пересчет осуществляется следующей функцией.

Пример 7-2b. Вычисления.

```
public void convert() {
        float currency1, currency2;
        String InputString = field1.getText();
        currency1 =
        Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString =
        Float.toString(currency2);
        field2.setText(OutputString);
```

```
} // конец convert()
```

Здесь создается строковая переменная, в которую записывается значение из поля field1. Затем переменная inputString преобразуется в число с плавающей точкой, и результат этого преобразования присваивается переменной currency1. Умножив это значение на conversion_ratio, мы присваиваем результат переменной currency2, а затем, преобразовав его к типу String, записываем строку результата в переменной outputString. Наконец, полученное значение вводится в поле field2. Метод, в котором происходит этот пересчет, вызывается, когда обработчик событий, приведенный ниже, фиксирует нажатие кнопки Convert.

Пример 7-2c. Перехват нажатия кнопки.

```
public boolean action(Event evt, Object obj)
        {
        if ("Convert".equals(obj))
        {
                convert();
                return true;
        }
        else { return false; }
        } // конец action()
} // конец апплета currencyConverter
```

Этот метод - перехватчик события используется для задания реакции на нажатие кнопки Convert. Для этого мы должны переопределить обработчик событий с именем action. Внутри обработчика событий производится сравнение идентификатора объекта, с которым произошло событие, со строкой "Convert" - надписью на нашей кнопке. На самом деле здесь можно было бы обойтись без сравнения, поскольку в нашем примере происходит всего одно событие, на которое мы должны отреагировать. Однако впоследствии, когда нам понадобится добавить в апплет другие кнопки, а также обрабатывать события, связанные с мышью, мы будем пользоваться этим механизмом для распознавания происходящих событий. В табл. 7-5 перечислены методы класса Button.

Таблица 7-5. Методы класса Button

Метод	Описание
getLabel()	Возвращает надпись на кнопке.
paramString()	Возвращает строку параметров кнопки.
setLabel(String)	Устанавливает надпись на кнопке.

Теперь нужно завести HTML-страницу, в которую будет встроен наш апплет пересчета денежных сумм. Создайте пустую страницу и поместите в нее фрагмент кода, приведенный ниже. Затем сохраните HTML-файл в том же каталоге, что и файл со скомпилированным апплетом currencyConverter2.class. После этого HTML-страницу можно загрузить в Netscape Navigator с помощью команды Open File.

```
<applet code="currencyConverter2.class" height=200 width=200>
</applet>
```

Переключатели

Чтобы привлечь клиентов со всего мира, наше туристическое агентство должно обеспечить возможность пересчитывать денежные суммы по курсам как можно большого количества разных валют. Давайте создадим пару переключателей (checkboxes), чтобы пользователь мог выбирать, сумму в какой валюте пересчитывать в доллары США. Как и одноименные органы управления, создаваемые с помощью HTML, переключатели в Java позволяют пользователю выбирать одну из нескольких возможностей, щелкая по соответствующему кружку. Каждый переключатель должен принадлежать какой-то группе, причем в каждой из этих групп можно пометить только один из переключателей. Вот как создается новый переключатель:

```
Checkbox Check1 = new Checkbox("UK Pound");
```

В этой строчке создается новый переключатель под названием Check1 с надписью "UK Pound", и его значение по умолчанию устанавливается в false (переключатель не выбран). Чтобы создать переключатель с указанием группы, к которой он должен принадлежать, и его исходного состояния, используйте следующую форму записи конструктора:

```
Checkbox UKCheck = new Checkbox("UK Pound",
CurrencyCheckGroup, false);
```

Здесь создается переключатель с надписью "UK Pound", принадлежащий группе CurrencyCheckGroup и имеющий по умолчанию значение false. Группа, объединяющая переключатели, является экземпляром класса CheckboxGroup. Вот как создается использованная в предыдущем примере группа CurrencyCheckGroup:
CheckboxGroup CurrencyCheckGroup = new CheckboxGroup();

Помните, что помимо создания переключателя вы должны также явным образом добавить его в свой апплет. В приведенном ниже примере это делается с помощью метода add, который использовался и для добавления всех остальных компонентов. Сначала создадим еще один переключатель, принадлежащий к той же группе:

```
CheckboxGroup FrancCheck = new Checkbox("Franc",
CurrencyCheckGroup, true)
```

Теперь напишем проверку, позволяющую узнать, какой из двух переключателей - фунт стерлингов или франк - выбран:

```
if UKCheck == CurrencyCheckGroup.getCurrent()
        { System.out.println("UK Pound is checked"); }
        else
                System.out.println("Franc is checked")
```

Теперь нам остается лишь переделать апплет пересчета денежных сумм так, чтобы он использовал разные коэффициенты пересчета в зависимости от выбранного переключателя. Текст этой версии апплета приведен в примере 7-3. Здесь мы снова будем пользоваться перехватом нажатия клавиши Enter, так как кнопка Convert в этой версии отсутствует. Как и в первой версии, нажатие Enter запускает пересчет (подробнее см. раздел "Ввод с клавиатуры" выше). Внешний вид апплета, использующего переключатели, показан на рис. 7-3.

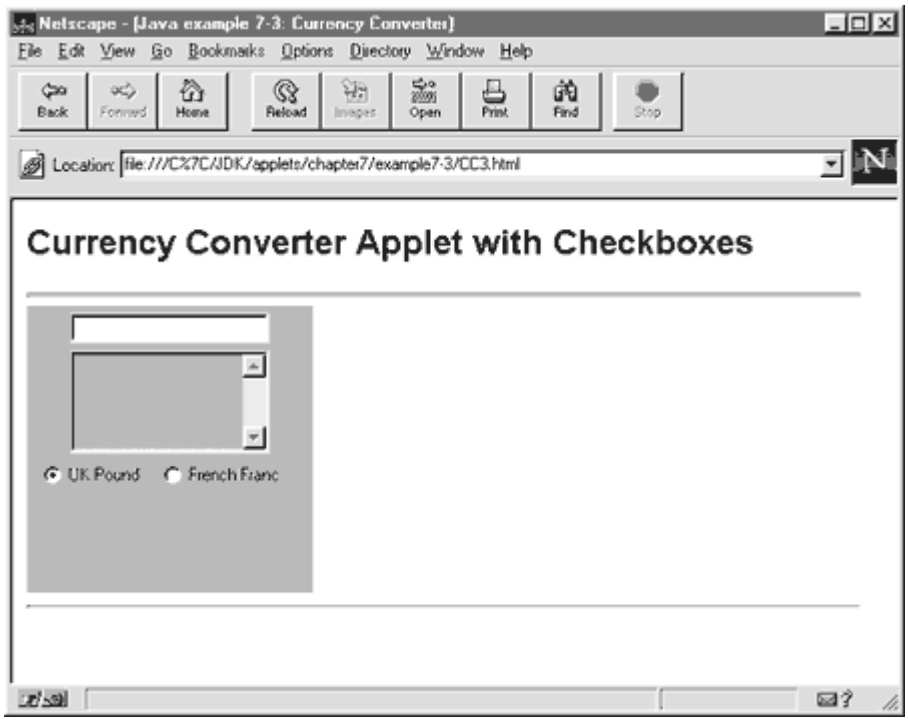

Рис. 7.3.

Пример 7-3. Апплет пересчета денежных сумм, использующий переключатели.

```
import java.awt.*;
import java.applet.Applet;
public class currencyConverter3 extends java.applet.Applet {
        final float UKratio = 1.50f;
        final float FRANCratio = 6.00f;
        float conversion_ratio = UKratio;
        TextField field1 = new TextField(20);
```

```
        TextArea field2 = new TextArea(4,20);
        CheckboxGroup CurrencyGroup = new CheckboxGroup();
        Checkbox UK_Pound = new Checkbox("UK Pound", CurrencyGroup, true);
        Checkbox French_Franc = new Checkbox("French Franc", CurrencyGroup,
false);
public void init() {
        field1.setEditable(true);
        field2.setEditable(false);
        add(field1);
        add(field2);
        add(UK_Pound);
        add(French_Franc);
resize(field1.preferredSize());
        resize(field2.preferredSize());
        show();
        }
public void convert() {
// функция convert() не изменилась
        float currency1, currency2;
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString =
        "$" + InputString + " = " + "#" + Float.toString(currency2) + "\n";
        field2.appendText(OutputString);
} // конец convert
public boolean handleEvent(Event evt) {
// перехват события щелчка мыши по одному из переключателей
if (evt.target == CurrencyGroupList)
        {if ( "UK Pound" == CurrencyGroupList.getSelectedItem() )
                { conversion_ratio=UKratio; }
        if ( "French Franc" == (CurrencyGroupList.getSelectedItem()) )
        { conversion_ratio=FRANCratio; }
// проверка, какой из переключателей нажат,
// и выбор соответствующего коэффициента пересчета
if (evt.target == field1)
        { char c=(char)evt.key;
                if (c == '\n')
                        {convert();
                        return true;}
                else { return false; }
        }
        return false;
        }
} // конец апплета
```

Что можно делать с группами переключателей?

При создании переключателя вы обязательно должны приписать его к одной из существующих групп переключателей. Однако вам не нужно пользоваться методом add, чтобы добавить саму эту группу в апплет. Иными словами, группа служит лишь неким общим идентификатором для нескольких переключателей. Ниже, когда мы будем изучать простые и выпадающие списки, мы увидим, что объекты, соответствующие этим спискам, необходимо добавлять в апплет с помощью метода add. Разница между списком и группой переключателей, таким образом, заключается в том, что элементы списка добавляются именно в список, а не непосредственно в апплет.

В табл. 7-6 перечислены методы класса CheckboxGroup, а в табл. 7-7 - методы класса Checkbox.

Таблица 7-6. Методы класса CheckboxGroup

Метод	Описание
getCurrent()	Возвращает выбранный на настоящий момент переключатель.
setCurrent	Переносит выбор на переключатель, на который ссылается

(CheckboxInstance)	CheckboxInstance.
toString()	Возвращает строковое представление значений данной группы переключателей.

Таблица 7-7. Методы класса Checkbox

Метод	Описание
getCheckboxGroup()	Возвращает группу, к которой принадлежит переключатель.
getLabel()	Возвращает текстовую надпись, идентифицирующую данный переключатель.
getState()	Возвращает булевское значение, соответствующее состоянию переключателя.
paramString()	Возвращает строку параметров переключателя.
setCheckboxGroup (CheckboxGroupInstance)	(Пере)приписывает переключатель к указанной группе.
setLabel(String)	Устанавливает текстовую надпись для данного переключателя.
setState(boolean)	Устанавливает состояние переключателя.

Списки

Списком (list box) называется набор элементов, выстроенных по вертикали, один или несколько элементов из которого могут быть выбраны (выделены). В предыдущем примере пользователь выбирал коэффициент пересчета для требуемой валюты с помощью переключателей. Однако если вы собираетесь обслуживать много разных валют, переключатели могут оказаться не самыми подходящими для этой цели. Намного удобнее будет собрать варианты выбора в список. По сравнению с группой переключателей список позволяет расположить варианты выбора более удобно и единообразно.

В примере апплета ниже список используется для выбора валюты, сумму в которой пользователь хочет пересчитать в доллары. Конструктор List, содержащийся в AWT, создает прокручиваемое окно списка, для которого можно задать количество видимых одновременно элементов (то есть его высоту). Для прокручиваемого списка можно также разрешать или запрещать выбор нескольких элементов одновременно. Вот как может выглядеть конструктор, создающий новый список:

List CurrencyList = new List(3, false);

Этим выражением создается новый список под названием CurrencyList, количество одновременно отображаемых элементов которого равно трем и в котором можно выбрать только один элемент за раз. Если бы вместо false стояло true, это означало бы, что в списке можно, напротив, выбирать несколько элементов одновременно. Добавление элемента к списку требует несколько большего количества подготовительных операций, чем добавление переключателя или кнопки. Текст новой версии нашего апплета, использующей для выбора одной из валют список, приведен в примере 7-4.

Пример 7-4a. Апплет пересчета денежных сумм, использующий список.

```
import java.awt.*;
import java.applet.Applet;
public class currencyConverter4 extends java.applet.Applet {
        final float UKratio = 1.50f;
        final float FRANCratio = 6.00f;
        float conversion_ratio = UKratio;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        List CurrencyGroupList = new List(2, false);
public void init() {
        field1.setEditable(true);
        field2.setEditable(false);
        CurrencyGroupList.addItem("UK Pound");
        CurrencyGroupList.addItem("French Franc");
        CurrencyGroupList.select(0);
        add(field1);
        add(field2);
```

```
        add(CurrencyGroupList);
        field1.resize(field1.preferredSize());
        field2.resize(field2.preferredSize());
        show();
} // конец init
```

В этом фрагменте кода создается новый список высотой в два элемента, позволяющий выбирать только один из элементов. Создав список, мы должны заполнить его элементами. Это делается иначе, чем в предыдущем примере, в котором мы добавляли переключатели в группу. Новый элемент добавляется непосредственно в экземпляр класса List - тогда как переключатели мы создавали по отдельности и только затем объединяли их в группы. Здесь же нам достаточно один раз создать экземпляр класса List, после чего можно добавлять в него любое количество элементов. Затем мы устанавливаем курсор в списке на первый элемент с помощью метода select(0). После этого весь список как одно целое добавляется в апплет методом add.

Пример 7-4b. Добавление списка в апплет.

```
public void convert() {
// функция convert() не изменилась
        float currency1, currency2;
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString =
        "$" + InputString + " = " + "#" + Float.toString(currency2) + "\n";
        field2.appendText(OutputString);
} // конец convert
public boolean handleEvent(Event evt) {
// перехват события выбора одного из элементов списка
        if (evt.target == CurrencyGroupList)
                {if ( "UK Pound" == CurrencyGroupList.getSelectedItem() )
                        { conversion_ratio=UKratio; }
        if ( "French Franc" == (CurrencyGroupList.getSelectedItem()) )
        { conversion_ratio=FRANCratio; }
}
if (evt.target == field1)
        { char c=(char)evt.key;
                if (c == '\n')
                        {convert();
                        return true;}
                else { return false; }
        }
        return false;
        }
} // конец апплета
```

В табл. 7-8 приведены описания некоторых методов из класса List.

Таблица 7-8. Полезные методы класса List

Метод	Описание
addItem(String)	Добавляет указанную строку как новый элемент в конец списка.
addItem(String, int)	Добавляет указанную строку как новый элемент после элемента с указанным номером.
allowsMultipleSelections()	Возвращает true, если для списка разрешен одновременный выбор нескольких элементов.
clear()	Очищает список, уничтожая все его элементы.
countItems()	Возвращает количество элементов в списке.
delItem(int)	Уничтожает указанный элемент списка.
delItems(int, int)	Уничтожает элементы, попадающие в диапазон с указанными границами.

deselect(int)	Снимает выделение с элемента с указанным номером.
getItem(int)	Возвращает текст элемента с указанным номером.
getRows()	Возвращает количество одновременно видимых элементов списка (его высоту).
getSelectedIndex()	Возвращает номер выбранного элемента или -1, если ни один из элементов не выбран.
getSelectedIndexes()	Возвращает номера выбранных элементов списка, если в списке разрешен выбор нескольких элементов
getSelectedItem()	Возвращает строку с текстом выделенного элемента или null, если ни один из элементов не выбран.
getSelectedItems()	Возвращает строки с текстом выбранных элементов списка, если в списке разрешен выбор нескольких элементов.
getVisibleIndex()	Возвращает номер элемента, который был сделан видимым при последнем вызове метода makeVisible.
isSelected(int)	Возвращает true, если элемент с указанным номером выбран, и false в обратном случае.
makeVisible(int)	Прокручивает список так, чтобы элемент с указанным номером был виден.
minimumSize(int)	Возвращает минимальный размер, который будет занимать список с указанным количеством одновременно видимых элементов.
minimumSize()	Возвращает минимальный размер данного списка.
paramString()	Возвращает строку параметров списка.
preferredSize(int)	Возвращает естественный размер для списка с указанным количеством одновременно видимых элементов.
preferredSize()	Возвращает естественный размер для данного списка.
replaceItem(String, int)	Заменяет текст элемента с указанным номером.
select(int)	Помечает как выбранный элемент с указанным номером.
setMultipleSelections(boolean)	Разрешает или запрещает выбор нескольких элементов одновременно.

Выпадающие списки

Мы уже умеем программировать выбор из нескольких возможностей с помощью переключателей и списков. Еще один метод, который позволяет уменьшить размер органов управления апплета, - это выпадающий список (choice). Этот компонент интерфейса, как и список, содержит набор элементов; кроме того, он может находиться в раскрытом либо закрытом состоянии, причем в последнем случае в списке виден только выбранный элемент (рис. 7-5). Как и в предыдущей версии нашего апплета, где мы работали со списками, нам нужно будет создать экземпляр класса Choice и заполнить его элементами.

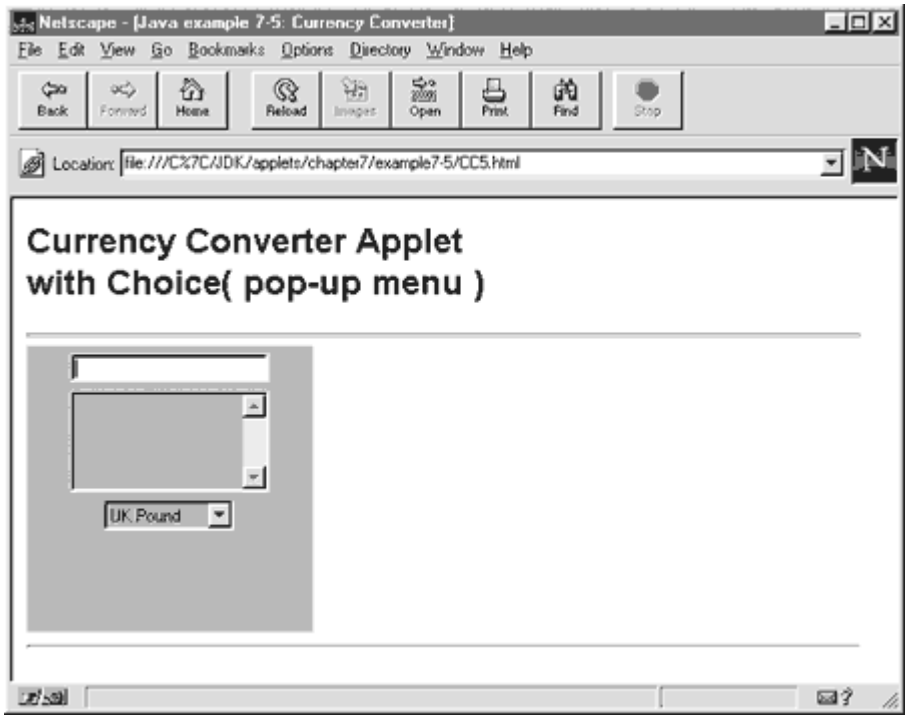

Рис. 7.5.

Выпадающий список создается точно так же, как и обычный список, с которым вы уже знакомы. Взяв за основу предыдущую версию апплета, нужно будет лишь заменить все случаи вхождения идентификатора CurrencyGroupList на идентификатор CurrencyGroupChoice. Фрагмент кода, создающий новый выпадающий список, приведен в примере 7-5.

Пример 7-5. Добавление в апплет выпадающего списка.

```
Choice CurrencyGroupChoice = new Choice();
CurrencyGroupList.addItem("UK Pound");
CurrencyGroupList.addItem("French Franc");
```

Текст из примера 7-5 нужно вставить в апплет вместо объявления CurrencyGroupList, расположенного в самом начале кода прошлой версии, в которой использовались обычные списки (см. пример 7-4). В табл. 7-9 перечислены некоторые из методов класса Choice.

Таблица 7-9. Полезные методы класса Choice

Метод	Описание
addItem(String)	Добавляет новый элемент к выпадающему списку.
countItems()	Возвращает количество элементов в выпадающем списке.
getItem(int)	Возвращает строковое представление элемента с указанным номером.
getSelectedIndex()	Возвращает номер выбранного элемента.
getSelectedItem()	Возвращает строковое представление выбранного элемента.
paramString()	Возвращает строку параметров выпадающего списка.
select(int)	Переносит выбор на элемент с указанным номером.
select(String)	Переносит выбор на элемент, которому соответствует указанная строка.

Полосы прокрутки

Полосы прокрутки (scrollbars) автоматически добавляются к полям редактирования, однако вы можете использовать их и отдельно. Например, к нашему апплету для пересчета денежных сумм можно добавить полосу прокрутки, с помощью которой пользователь сможет изменять коэффициент пересчета (рис. 7-6). Вот как выглядит конструктор, создающий полосу прокрутки:

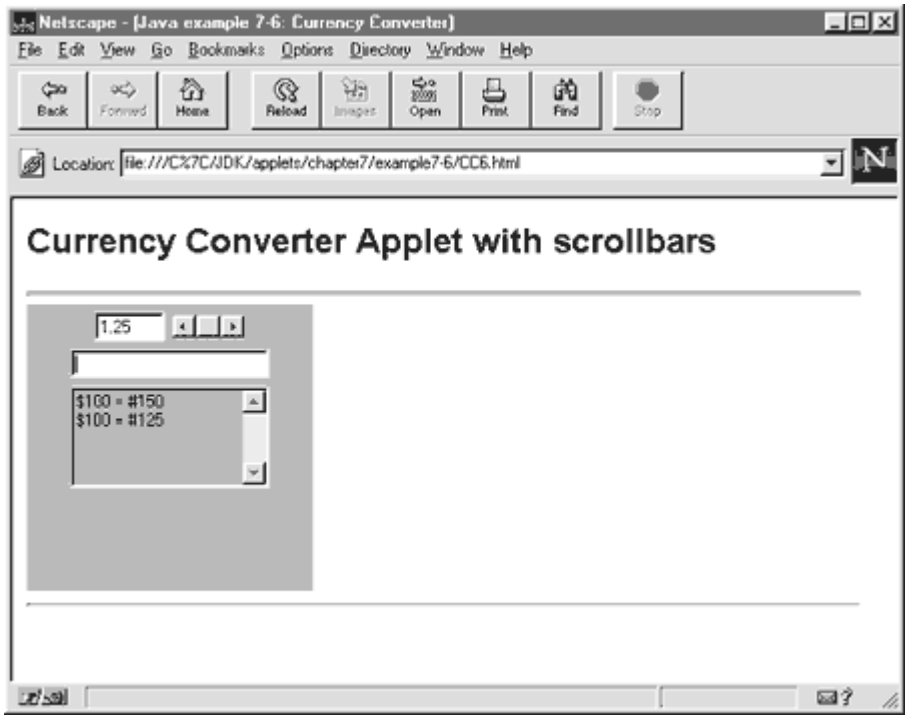

Рис. 7.6.

Scrollbar ratioScrollBar = new Scrollbar(
Scrollbar.HORIZONTAL, 150, 25, 50, 250);

Здесь создана горизонтальная полоса прокрутки, начальное положение движка которой установлено в 150. Длина полосы прокрутки на экране равна 25, минимальное значение, соответствующее крайнему левому положению движка, равно 50, а максимальное - 250 (эти величины, как и текущее положение движка, могут быть представлены только целыми числами).

Давайте добавим такую полосу прокрутки в наш апплет пересчета курса валют. Как вы помните, до сих пор мы использовали коэффициент пересчета, равный 1,5, но теперь мы позволим пользователю изменять эту величину, уменьшая ее на 0,1 при перемещении движка прокрутки влево и увеличивая на 0,1 при перемещении его вправо. В расположенном рядом поле редактирования мы будем отображать устанавливаемые значения, чтобы пользователь мог видеть, что он делает.

Пример 7-6а. Апплет пересчета денежных сумм с полосой прокрутки.

```
import java.awt.*;
import java.applet.Applet;
public class currencyConverter5 extends java.applet.Applet {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
Scrollbar ratioScrollBar = new Scrollbar(
Scrollbar.HORIZONTAL, 150, 25, 50, 250 );
public void init() {
        field1.setEditable(true);
        field2.setEditable(false);
        add(ratioField);
        add(ratioScrollBar);
        add(field1);
        add(field2);
        ratioField.setText("1.50");
        resize(field1.preferredSize());
        resize(field2.preferredSize());
        show();
} // конец init
```

Коэффициент пересчета, изменяемый с помощью полосы прокрутки, динамически отображается в созданном специально для этого поле редактирования. Сама горизонтальная

полоса прокрутки шириной в 25 единиц имеет диапазон изменения от 50 до 250 и начальное значение, равное 150.

Теперь нам нужно предусмотреть обработку событий, связанных с полосой прокрутки.

Пример 7-6b. Обработка событий, связанных с полосой прокрутки.

```
public void convert() {
        float currency1, currency2;
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString =
        "$" + InputString + " = " + "#" + Float.toString(currency2) + "\n";
        field2.appendText(OutputString);
} // конец convert
public boolean handleEvent(Event evt) {
        if (evt.target == ratioScrollBar)
                { int in;
                        in = ratioScrollBar.getValue();
                                conversion_ratio = in/100f;
ratioField.setText(Float.toString(conversion_ratio));
                        }
        if (evt.target == field1)
                { char c=(char)evt.key;
                        if (c == '\n')
                                {convert();
                                return true;}
                        else { return false; }
                }
                return false;
        } // конец handleEvent()
} // конец Applet()
```

Мы должны отслеживать все события, связанные с объектом ratioScrollBar, и постоянно запрашивать значение, выставленное в настоящий момент на полосе прокрутки, поскольку любое из событий может привести к изменению этой величины. Возвращаемое полосой прокрутки значение может быть только целым числом, так что нам придется разделить это значение на 100, чтобы привести его к нужному нам диапазону (от 0,5 до 2,5). Кроме того, при каждом изменении коэффициента мы должны обновлять содержимое поля редактирования. Некоторые из методов класса Scrollbar приведены в табл. 7-10.

Таблица 7-10. Полезные методы класса Scrollbar

Метод	Описание
getLineIncrement()	Возвращает величину инкремента, применяемую при нажатии кнопок по краям полосы прокрутки ("строчный" инкремент).
getMaximum()	Возвращает максимальное значение, возвращаемое полосой прокрутки.
getMinimum()	Возвращает минимальное значение, возвращаемое полосой прокрутки.
getOrientation()	Возвращает ориентацию полосы прокрутки.
getPageIncrement()	Возвращает величину инкремента, применяемого при щелчке по самой полосе прокрутки ("страничный" инкремент).
getValue()	Возвращает текущее значение на полосе прокрутки.
getVisible()	Возвращает ширину полосы прокрутки (то есть величину ее видимой части).
paramString()	Возвращает строку параметров полосы прокрутки.
setLineIncrement(int)	Устанавливает строчный инкремент полосы прокрутки.
setPageIncrement(int)	Устанавливает страничный инкремент полосы прокрутки.
setValue(int)	Устанавливает текущее значение полосы прокрутки.
setValues(int, int, int, int)	Изменяет параметры полосы прокрутки, задаваемые при ее создании.

Надписи

Этот последний из базовых элементов интерфейса позволяет вводить в апплет статические текстовые надписи (labels). Давайте воспользуемся надписями для добавления в наш апплет пояснений к органам управления. В табл. 7-11 перечислены некоторые методы класса Label. На рис. 7-7 показан внешний вид новой версии нашего апплета, использующей надписи. Приведенный ниже исходный текст представляет собой модификацию предыдущей версии апплета (см. пример 7-6), использующую текстовые надписи.

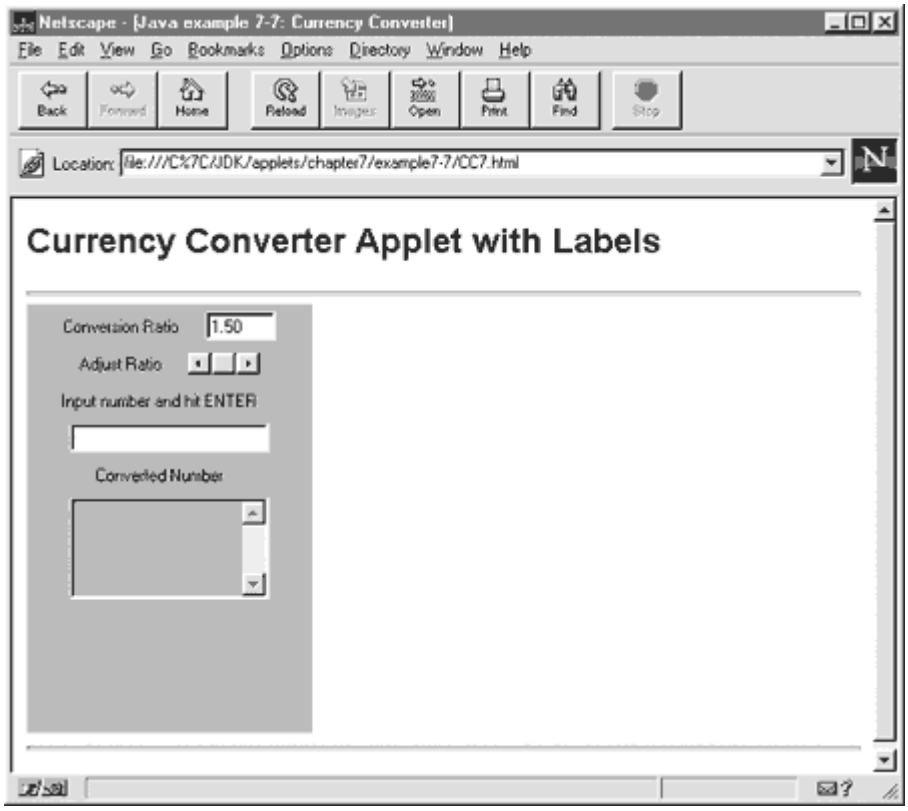

Рис. 7.7.

Пример 7-7. Апплет пересчета денежных сумм с текстовыми надписями.

```
public class currencyConverter76 extends java.applet.Applet {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
Scrollbar ratioScrollBar = new Scrollbar(
Scrollbar.HORIZONTAL, 150, 25, 50, 250 );
public void init() {
        field1.setEditable(true);
        field2.setEditable(false);
        add(new Label("Conversion Ratio"));
        add(ratioField);
        add(new Label("Adjust Ratio"));
        add(ratioScrollBar);
        add(new Label("Input number and hit ENTER"));
        add(field1);
        add(new Label("Converted Number"));
        add(field2);
        ratioField.setText("1.50");
        resize(field1.preferredSize());
        resize(field2.preferredSize());
        show();
                } // конец init
```

Таблица 7-11. Полезные методы класса Label

Метод	Описание

getAlignment()	Возвращает текущий режим выравнивания для данной надписи.
getText()	Возвращает текст надписи.
paramString()	Возвращает строку параметров надписи.
setAlignment(int)	Устанавливает режим выравнивания для надписи.
setText(String)	Устанавливает текст надписи.

Что дальше?

В этой главе мы познакомились с основными компонентами интерфейса пользователя, применяемыми в апплетах. В следующей главе, используя полученные знания, мы приступим к изучению более сложных компонентов, а также обсудим некоторые общие вопросы дизайна и компоновки пользовательского интерфейса.

Глава 8

Еще об интерфейсе пользователя

Разработка интерфейса программы - задача ответственная и весьма непростая. Инструменты, входящие в набор разработчика Java и предназначенные для организации внешнего вида программ, помогут вам успешно решить эту задачу. Эти инструменты позволяют свободно размещать элементы на экране без необходимости указывать их точные координаты и размеры. Выгода такого подхода не только в том, что разработчику не нужно тратить драгоценное время на подсчет количества пикселов и тому подобных параметров низкого уровня, но и в том, что запрограммированный таким образом интерфейс будет выглядеть более-менее одинаково на разных платформах. Конечно, в мелочах внешний вид программы на разных платформах будет варьироваться, так как эти мелочи сильно зависят от операционной системы, а еще больше - от ее графической оболочки (например, Openwin, Windows 95 или Macintosh).

Один из самых ценных инструментов JDK, предназначенных для создания пользовательского интерфейса, - это набор менеджеров размещения (layout managers), предназначенных для визуальной организации элементов интерфейса. Менеджеры размещения обязательно используются в любом апплете; даже в простейшем апплете, на котором мы с вами в главе 7 знакомились с основными элементами интерфейса, использовался менеджер размещения по умолчанию.

В этой главе мы обсудим все существующие менеджеры размещения, а также контейнеры для экранных элементов, к которым относятся панели (panels), диалоговые окна (dialogs) и фреймы (frames). Здесь вы также научитесь встраивать в свой апплет меню, получать список доступных в системе шрифтов и устанавливать шрифты для отдельных элементов интерфейса. Примеры в этой главе также используют апплет пересчета денежных сумм из одной валюты в другую, с которым мы работали в главе 7, однако здесь мы добавим к нему много новых функций и значительно усовершенствуем его внешний вид, пользуясь более сложными компонентами интерфейса и уделяя большее внимание их размещению и компоновке.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Программирование внешнего вида апплета

Прежде чем приступать к заданию внешнего вида апплета в программе, рекомендуется составить набросок его экранного интерфейса (или, по крайней мере, представить себе такой набросок мысленно). Это значительно облегчит написание кода, а также подскажет вам порядок, в котором удобнее всего добавлять компоненты в код. Кроме того, такой набросок позволит вам понять, какие группы компонентов удобнее всего объединить с помощью панелей - специальных контейнеров, заключающих в себе другие компоненты интерфейса.

Ваш апплет может использовать окна, которые никак не будут связаны с окном Web-броузера, - с этой целью можно использовать фреймы или диалоговые окна, содержащие те или иные элементы интерфейса вашего апплета. Создание нескольких окон позволит вам также лучше отобразить в интерфейсе логическую структуру апплета, разделив его интерфейс на блоки, каждый из которых пользователь сможет убрать или минимизировать, если он больше не нужен. Например, в учебной программе - клиенте шахматного сервера, которую мы будем разрабатывать в главе 18, список пользователей не должен постоянно присутствовать на экране, поэтому мы поместим его в отдельный фрейм, выводимый на экран по нажатию специальной кнопки. Ознакомившись со списком, пользователь сможет убрать его, просто закрыв окно.

Выбор размещения экранных элементов - задача очень ответственная, требующая взвешенного подхода, особенно для достаточно сложных по структуре апплетов. Если ваш апплет предоставляет пользователям множество опций и органов управления и выводит на экран много разнородной информации, следует сравнить по меньшей мере несколько вариантов размещения материала, пытаясь выбрать тот, который сделает работу с апплетом как можно более комфортной и интуитивно очевидной. Например, чтобы представить информацию о версии апплета, его назначении и авторе, лучше всего воспользоваться пунктом меню Help, так как любой хоть сколько-нибудь опытный пользователь будет искать эту информацию именно там.

При разработке интерфейса вы должны пытаться поставить себя на место типичного пользователя, имеющего самые минимальные знания о своем компьютере и о сети Интернет. Например, типичный пользователь, которому захочется заглянуть в систему помощи, будет искать соответствующее меню в правом верхнем углу окна, поэтому разумнее всего именно туда его и поместить. Подобные традиции построения интерфейса, которые обязательно нужно соблюдать, ни в коей мере, конечно, не ограничивают вашу творческую свободу как дизайнера. Хотя дружественность к пользователю и должна стоять на первом месте (если только вы

рассчитываете, что вашим апплетом будет пользоваться хоть кто-нибудь, кроме вас), она не мешает вложить в дизайн апплета известную долю оригинальности.

Контейнеры

Для объединения связанных друг с другом органов управления Java API предлагает три вида контейнеров: панели, диалоговые окна и фреймы. Все контейнеры являются наследниками класса Component и используются как составные объекты, в которые можно добавлять другие элементы экранного интерфейса. В случаях сложного интерфейса контейнеры позволяют также по-разному размещать группы элементов относительно друг друга. Например, группу флажков или переключателей с относящейся к ним кнопкой можно собрать вместе и разместить определенным образом относительно другого элемента интерфейса - например, поля редактирования.

Панели

Панели (panels) являются наиболее общим видом контейнеров в Java. Так, в приведенном выше примере с группой флажков и кнопкой удобнее всего было бы использовать именно панель. Панель является объектом, относящимся непосредственно к классу Container. Панель можно использовать как внутри другого контейнера (например, фрейма), так и непосредственно в окне Web-броузера. Когда ваш интерфейс состоит из большого количества элементов, почти всегда есть смысл объединить группы связанных по смыслу элементов с помощью панелей. Панель может также иметь свой собственный менеджер размещения, независимый от менеджера размещения контейнера, в который эта панель входит. Предположим, например, что мы пользуемся простейшим менеджером размещения FlowLayout для создания фрейма, содержащего компоненты различных размеров. Теперь вам потребовалось поместить в этот фрейм группу из пяти кнопок одинаковых размеров. Это можно сделать быстро и удобно, собрав эти кнопки в панель и применив в ней менеджер размещения GridLayout (который как раз и предназначен для компоновки элементов, имеющих одинаковый размер), а затем поместив готовую панель во фрейм.

Окна

Окна, как и панели, представляют собой общий класс контейнеров. Но в отличие от панелей окно Java действительно представляет собой окно - объект операционной системы, существующий отдельно от окна броузера Web или программы просмотра апплетов. Непосредственно класс Window никогда не используется, а используются только его три подкласса - Frame, Dialog и FileDialog. Каждый из этих подклассов содержит все функции исходного класса Window, добавляя к ним несколько специфических свойств. Например, класс Frame реализует контейнер с меню, позволяющий добавлять панель меню в создаваемые фреймы. Все окна, как и положено, имеют стандартные кнопки для свертки, максимизации и уничтожения, а также управляющее меню (control menu). Кроме того, у всех окон есть панель заголовка, которую вы должны заполнить при создании экземпляра окна.

Фреймы

Фрейм (frame) - это объект, который может существовать без всякой связи с окном броузера Web. С помощью класса Frame можно реализовать интерфейс независимого апплета. Сразу после создания фрейм представляет собой пустое окно, в которое вы должны самостоятельно добавлять компоненты интерфейса.

Во фреймах можно располагать любые элементы пользовательского интерфейса. Так, в примере апплета пересчета денежных сумм, с которым мы будем работать в этой главе, весь пользовательский интерфейс заключен во фрейм. Фреймы можно также создавать для отдельных компонентов интерфейса (например, список пользователей, открываемый нажатием кнопки в окне броузера Web; с таким применением фреймов мы познакомимся в главе 18). Создав фрейм, вы можете также предусмотреть обработчики для событий нажатия кнопок свертки, максимизации и уничтожения окна, автоматически добавляемых к панели заголовка созданного окна. Кроме того, фреймы являются реализацией класса MenuContainer, позволяющего добавлять в окно панель меню. Меню и класс MenuContainer будут обсуждаться ниже в этой главе.

Диалоговые окна

Диалоговые окна (dialogs) используются в основном для одноразового запроса информации у пользователя или для вывода небольших порций информации на экран. Диалоговые окна во всем подобны фреймам, но имеют два важных отличия: во-первых, они не являются реализацией класса MenuContainer, и, во-вторых, они могут иметь модальность - это значит, что можно сконструировать диалоговое окно, которое запретит пользователю обращаться к другим окнам (включая и окно броузера Web) до тех пор, пока пользователь не произведет требуемого действия в этом диалоговом окне (например, не нажмет кнопку).

При создании диалогового окна его модальность задается указанием особого параметра вызова конструктора. Таким образом, диалоговое окно дает вам выбор: позволить ли пользователю продолжить работу, не обращая внимания на диалоговое окно, или же потребовать от него особой реакции на сообщение в этом окне.

Диалоговые окна удобны для приглашения пользователя к выполнению какого-то действия и для подтверждения того, то пользователь ознакомился с сообщением программы. Диалоговые окна в отличие от фреймов не реализуют панель меню автоматически. Поскольку диалоговые окна обычно меньше по размерам, чем фреймы, они бывают удобны для быстрого диалога с пользователем.

При работе с диалоговыми окнами важно помнить одно правило: каждое диалоговое окно обязательно должно иметь фрейм в качестве родителя. Это значит, что диалоговое окно нельзя открыть непосредственно из главного окна апплета или из окна броузера Web. Чтобы создать диалоговое окно, вы должны сначала завести фрейм, даже если его единственным назначением будет служить родителем для диалогового окна. Только если ваш апплет уже использует фреймы, вы можете обойтись без этой подготовительной стадии. Пример создания пустого фрейма, играющего роль родителя диалогового окна, вы найдете в разделе "Диалоговый апплет: BorderLayout" в этой главе.

Файловые диалоги

Файловый диалог нельзя использовать в апплетах, поскольку он предназначен только для программ, работающих непосредственно в программе просмотра апплетов или в интерпретаторе командной строки "java". Этот объект используется для передачи дескрипторов файлов для загрузки или сохранения через потоковый класс. Подробнее о файловых диалогах вы узнаете в главе 12, "Программирование за рамками модели апплета".

Установка цветов

С помощью методов setForeground и setBackground можно установить цвет для любого компонента интерфейса. Оба эти метода имеют единственный параметр, который представляет собой объект Color, например:

```
setForeground(Color.gray);
```

Приведенный здесь оператор устанавливает серый цвет для текущего компонента. Возможные значения цветов следующие: черный (black), синий (blue), циан (cyan), темно-серый (darkGray), серый (gray), зеленый (green), светло-серый (lightGray), фиолетовый (magenta), оранжевый (orange), розовый (pink), красный (red), белый (white) и желтый (yellow).

Кроме того, вы можете создавать свои собственные цвета, конструируя новый объект класса Color и устанавливая значения для красной, зеленой и синей составляющих цвета:

```
Color MyColor = new Color(100, 100, 100);
```

Три числовых параметра в вызове конструктора Color представляют собой величины красной, зеленой и синей составляющих цвета. Поскольку цветовые значения в Java занимают 24 бита, каждая цветовая составляющая имеет размер в один байт, и поэтому допустимые значения для этих параметров лежат в диапазоне от 0 до 255.

Меню

Меню давно уже стали стандартным элементом интерфейса современных программ. В Java меню особенно удобно использовать в сочетании с одним из менеджеров размещения - CardLayout, который позволяет реализовывать динамичный функциональный интерфейс (см. раздел "Добавление меню: CardLayout" в этой главе). Меню в Java генерируют события, для которых программист создает обработчики, - так что работа с меню в апплете не должна представлять для вас ничего сложного. Классы MenuBar и Menu позволяют конструировать удобные меню и интегрировать их в структуру апплета.

Все создаваемые фреймы автоматически реализуют класс MenuContainer. Пожалуй, фреймы - это единственный класс объектов, в которых стоит использовать меню: как вы понимаете, добавление второй панели меню в окно броузера не имеет большого смысла, а диалоговое окно с меню уже, строго говоря, не является диалоговым окном. Именно поэтому Frame является

единственным контейнером, который автоматически реализует класс MenuContainer, - несмотря на то, что теоретически меню может быть добавлено в любой из контейнеров.

Добавление меню в апплет

Отдельные выпадающие меню добавляются в панель меню - объект класса MenuBar, после чего этот объект добавляется к соответствующему фрейму. Прежде всего нам нужно создать экземпляр MenuBar. После этого мы создаем экземпляры класса Menu и добавляем в них команды.

СОВЕТ Когда вы добавляете новые меню в объект MenuBar, они появляются на экране слева направо в том порядке, в каком вы их вставляли, поэтому порядок операторов добавления меню в программе следует планировать заранее.

Панель меню

Прежде всего создадим новый объект класса MenuBar под названием Bar:

```
MenuBar Bar = new MenuBar();
```

Для добавления меню к созданной панели меню используется такой метод:

```
Bar.add(m);
```

Параметр m представляет собой объект класса Menu. Меню добавляются в панель меню слева направо в том порядке, в каком в программе расположены вызовы указанного метода. Для удаления меню с панели меню служит метод remove. Еще одна интересная возможность - присвоение одному из меню статуса "меню помощи", которое осуществляется следующим образом:

```
setHelpMenu(m);
```

Метод setHelpMenu превращает меню m в меню помощи, перенося его на правый конец панели, отдельно от других меню. Обратите внимание, что, прежде чем вызывать setHelpMenu, нужно сначала добавить это меню на панель. Кроме того, не забудьте, что метод setHelpMenu входит в класс MenuContainer, реализацией которого является объект Frame. Если вы хотите вызвать этот метод за пределами кода, относящегося к конкретному фрейму, вы должны написать что-нибудь вроде следующего:

```
frameNameInstantiation.setHelpMenu(m)
```

СОВЕТ Если вы хотите использовать метод setHelpMenu, вы должны сначала добавить нужное меню к панели. Только после этого можно вызывать setHelpMenu.

Наконец, нам остается приказать объекту Frame использовать данный экземпляр MenuBar. Для этого применяется метод setMenuBar, который, как и setHelpMenu, относится к классу Frame:

```
setMenuBar(Bar);
```

Этот оператор позволяет добавить указанную панель меню к текущему фрейму. Как правило, вызов этого метода размещается в самом конце кода, отвечающего за создание меню: сначала создается экземпляр класса MenuBar, затем создаются контейнеры Menu, потом к созданным меню добавляются команды, и только после всего этого созданные меню добавляются к панели меню.

Создание команд меню

При создании меню необходимо реализовать класс Menu для каждого выпадающего меню и класс MenuItem для каждой команды в этом меню. Вот как выглядит объявление нового меню:

```
Menu m = new Menu("Operation");
```

Здесь мы создаем новое меню с заголовком "Operation" и сохраняем его в переменной m. Поскольку для доступа к объекту вполне достаточно его заголовка (например, заголовка меню или текста команды), мы можем объединить этапы создания новой команды и добавления ее в меню:

```
m.add(new MenuItem("Adjust Ratio"));
```

В этом операторе создается новая команда меню с текстом "Adjust Ratio" и сразу же добавляется в меню m. Еще одна интересная возможность - объявление "западающего" (tear-off) меню. Это значит, то такое меню останется открытым даже после того, как кнопка мыши будет отпущена. Западающее меню создается с помощью следующего конструктора:

new Menu("Operation", true);

Команды добавляются в меню в направлении сверху вниз. Вы можете также включать в меню горизонтальные разделительные линии с помощью метода addSeparator, который вставляет на текущую позицию в меню разделитель. Кроме того, вы можете создавать вложенные меню благодаря тому факту, что класс Menu является всего лишь расширением класса MenuItem. Достаточно создать новый экземпляр Menu и добавить его в другое меню точно так же, как вы бы сделали это с командой. Мы будем использовать вложенные меню в учебной программе Java-магазин (см. главу 17); внешний вид вложенного меню показан на рис. 8-1.

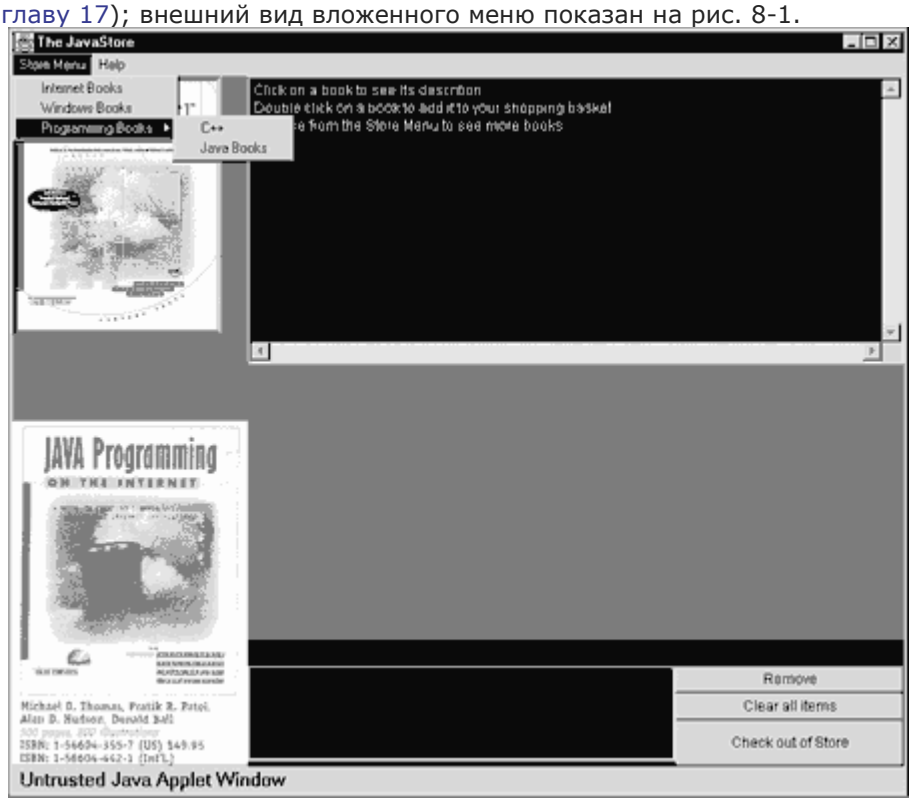

Рис. 8.1.

Обработка событий меню

Когда пользователь выбирает одну из команд в меню, происходит генерация соответствующего события. Реакция программы на это событие, как и на любое другое, обеспечивается установкой соответствующего обработчика. Так, в примере апплета в разделе "Добавление меню: CardLayout" в этой главе мы будем пользоваться следующей проверкой:
if ((evt.id == Event.ACTION_EVENT) && (evt.target==m))

Этот оператор проверяет, является ли перехваченное событие событием действия (action event) и является ли целью (target) данного события объект Menu с именем m. После этого мы можем выяснить, какая именно команда была выбрана, заглянув в параметр события:
(String)evt.arg

Этот аргумент можно теперь сравнить по очереди со строками, содержащими текст всех команд данного меню, чтобы выяснить, какая из команд была выбрана пользователем.

Шрифты

Во многих примерах в этой главе нам с вами придется устанавливать шрифт для вывода информации. Поэтому пора познакомиться с тем, какие возможности в этом отношении предоставляет AWT. Основной из используемых методов - метод setFont - принадлежит классу Component, а не классу Font. Благодаря этому мы можем не только изменять шрифт по отдельности для каждого компонента, но и пользоваться для групп связанных компонентов наследованием шрифтовых параметров от общего объекта-родителя. Конструктор нового шрифта выглядит так:
new Font(String name, int style, int size);

Здесь name представляет собой название шрифта, size - его размер в пунктах, а параметр style может принимать одно из следующих значений: PLAIN, BOLD или ITALIC. Такое создание экземпляра шрифта необходимо, чтобы можно было установить этот шрифт для какого-то из компонентов. Созданный экземпляр шрифта можно использовать неоднократно. Однако прежде,

чем заказывать какой-то шрифт, нужно убедиться, что он присутствует на платформе пользователя и Web-броузер может его применить (см. врезку "Получение списка шрифтов"). Имея список шрифтов, апплет сможет принимать решение о том, какими шрифтами пользоваться на данной платформе.

Метрики шрифтов

Для каждого шрифта апплет может получить определенную метрическую информацию, или метрики - в частности, ширину и высоту любого символа. Зная метрики шрифта, можно выбирать размеры для других компонентов и самого контейнера, обеспечивая правильное расположение текстовых надписей по отношению к другим элементам. Например, метод stringWidth класса FontMertics используется для получения ответа на вопрос "Сколько места по горизонтали будет занимать заданная в качестве параметра строка, набранная указанным шрифтом?":

```
Font MyFont = new Font("Helvetica",Font.PLAIN, 12);
FontMetrics MyFontMetrics = new FontMetrics( MyFont );
int pixelSizeOfString = MyFontMetrics.stringWidth("Hello");
```

В этом примере выясняется, сколько места занимает строка "Hello", набранная шрифтом MyFont, который определен как Helvetica полужирного начертания кегля 12 пунктов. Переменная pixelSizeOfString после выполнения этих операторов будет содержать число пикселов, необходимое для вывода этой строки на экран.

Получение списка шрифтов

Чтобы получить список всех шрифтов, доступных броузеру Web на данной платформе, используйте класс AWT под названием Toolkit:

```
Toolkit Tools = new Toolkit();
FontListString[] = Tools.getFontList();
```

В этом примере список доступных шрифтов помещается в массив строковых переменных. После этого в массиве можно осуществлять поиск и, таким образом, выбирать наиболее подходящий шрифт для экранного вывода.

Менеджеры размещения

Менеджеры размещения - незаменимый инструмент во всех случаях, когда число компонентов интерфейса, используемых в апплете, больше трех. С помощью этих менеджеров можно легко и быстро обеспечить нужное расположение элементов экрана относительно друг друга и включающего их контейнера. Этот механизм позволяет с легкостью решать такие задачи, как, например, динамическое изменение расположения компонентов в зависимости от меняющейся величины окна. Задание абсолютных координат для каждого компонента нельзя назвать хорошим решением по нескольким причинам: во-первых, это чрезвычайно трудоемко, а во-вторых, изменение положения одного компонента потребует пересчета координат всех остальных. JDK включает в себя несколько готовых менеджеров размещения, которые пригодны для построения интерфейсов в большинстве апплетов.

На момент написания этой книги различными третьими фирмами подготавливается несколько интегрированных сред разработки (integrated development environments, IDE), предназначенных для создания Java-программ и, в частности, для более удобной работы с интерфейсом апплетов. Эти среды будут включать в себя визуальные средства для разработки пользовательского интерфейса, которые возьмут на себя большую часть работ по конфигурированию менеджеров размещения. Пока же этим конфигурированием приходится заниматься автору любой программы, которая явным образом пользуется менеджерами размещения.

К простейшим менеджерам размещения относятся FlowLayout и GridLayout, к более сложным - BorderLayout, CardLayout и GridBagLayout. По умолчанию в окне апплета и во всех создаваемых панелях используется менеджер FlowLayout, а во фреймах и диалоговых окнах - менеджер BorderLayout.

Ниже приведено описание всех этих менеджеров размещения и обзор их использования в примерах, с которыми мы будем работать во второй половине этой главы.

СОВЕТ Весьма полезным менеджером размещения является PackerLayout, разработанный Дэроном Мейером (Daeron Meyer), но, к сожалению, не входящий в состав JDK. Этот менеджер должен показаться знакомым тем, кто работал с TCL/TK. Гибкость и богатство возможностей этого

менеджера заставили нас использовать его в некоторых из примеров апплетов в этой книге. Вы найдете этот менеджер размещения на диске CD-ROM, прилагаемом к книге, а также на странице Online Companion.

FlowLayout

Вы уже знакомы с результатами деятельности простейшего из менеджеров размещения - FlowLayout. Этот менеджер работал как менеджер по умолчанию в апплете пересчета денежных сумм, который мы писали в предыдущей главе. Принцип действия этого менеджера и в самом деле очень прост и сводится к следующему: каждый новый добавляемый компонент помещается в текущий горизонтальный ряд, если в этом ряду еще есть место, а если нет - то компонент смещается вниз и начинает следующий горизонтальный ряд. Таким образом, каждый ряд, кроме последнего, содержит столько компонентов, сколько помещается в него при текущей ширине контейнера. Содержимое каждого ряда центрируется, то есть отбивается влево и вправо от краев контейнера. Такой алгоритм позволяет добиться вполне профессионально выглядящих результатов, если предварительно рассчитать размер области, выделяемой для вашего апплета на HTML-странице с помощью атрибутов HEIGHT и WIDTH тега <APPLET>. Например, если вы знаете точные размеры кнопок, вы можете задать для апплета ширину так, чтобы кнопки располагались по вертикали одна под другой. Однако часто попытки добиться красивого расположения с помощью этого менеджера требуют слишком больших усилий, которые, в конечном счете, не окупаются, так как почти любое изменение размеров контейнера разрушает тщательно выверенную структуру (так, столбец кнопок в нашем примере может превратиться в несколько строк кнопок).

BorderLayout

С этим менеджером размещения мы познакомимся в примерах апплетов, приведенных ниже в этой главе. Этот менеджер требует при добавлении нового компонента указывать дополнительный параметр, который может принимать одно из следующих значений: South ("Юг"), North ("Север"), West ("Запад"), East ("Восток") и Center ("Центр"). Первые четыре параметра заставляют менеджер BorderLayout относить добавляемые компоненты к соответствующему краю контейнера - нижнему, верхнему, левому или правому. Параметр Center позволяет указать, что данный компонент может занимать все оставшееся в контейнере свободное место. Таким образом, элементы, добавляемые с параметром Center, будут изменять свой размер, заполняя место, не занятое другими компонентами.

Менеджер BorderLayout применяется чаще всего, так как он довольно практичен, прост в использовании и предоставляет средства для размещения экранных элементов, которых хватает для большинства случаев. BorderLayout способен учитывать разницу в размерах отдельных компонентов и пытается максимально экономно использовать пространство контейнера - области в окне броузера, фрейма или диалогового окна.

GridLayout

Менеджер GridLayout особенно полезен для размещения графических элементов и позволяет с легкостью достичь ровного, единообразного размещения компонентов. Этот менеджер создает решетку, состоящую из квадратов одинакового размера, в каждом из которых располагается один компонент. Мы будем использовать этот менеджер для построения шахматной доски в учебной программе в главе 18. Каждое шахматное поле на доске, созданной с помощью этого менеджера, содержит холст (Canvas), который, в свою очередь, может содержать изображение шахматной фигуры. При использовании этого менеджера новые компоненты добавляются в направлении слева направо и сверху вниз.

CardLayout

Этот менеджер размещения позволяет изменять набор компонентов, выводимых на экран, прямо во время работы апплета. Менеджер CardLayout особенно удобен при работе с панелями.

Менеджер CardLayout позволяет динамически управлять выводом на экран компонентов, добавленных в контейнер. Компоненты должны быть при этом ассоциированы с особыми методами, определенными в классе, - методами next, previous и show. Программист, таким образом, может завести любое количество компонентов, а затем, вызывая эти методы, выводить их на экран последовательно один за другим или в произвольном порядке. В разделе "Динамическая смена компонентов: CardLayout" в этой главе мы будем использовать менеджер

CardLayout для переключения между двумя панелями, содержащими компоненты нашего апплета пересчета денежных сумм. В одной из панелей будут расположены поле, отображающее текущий коэффициент пересчета, и полоса прокрутки, позволяющая изменять это значение, а в другой панели - поля для ввода и вывода денежных сумм. Эти панели будут выводиться на экран не одновременно, а по очереди.

GridBagLayout

Самым сложным из менеджеров размещения является GridBagLayout. В нем используется наиболее совершенный алгоритм реагирования на изменение размеров контейнера, и он позволяет реализовывать сложный интерфейс, в котором контейнер содержит много компонентов различных размеров, некоторые из которых должны находиться в одном и том же точно заданном положении относительно других. Когда вы используете этот менеджер, вы должны задавать параметры расположения для каждого компонента с помощью метода setConstraints. Удобнее всего создать для каждого компонента экземпляр класса gridBagConstraints, что позволит изменять расположение этого компонента независимо от других. Экземпляры gridBagConstraints будут использоваться тем контейнером, для которого задан менеджер размещения GridBagLayout, хотя в примере апплета ниже в этой главе (см. раздел "Точное размещение: GridBagLayout") мы будем пользоваться для разных компонентов одним и тем же экземпляром gridBagConstraints. Вот как выглядит список ограничений на размещение, которые можно задавать при использовании этого менеджера:

- gridx, gridy: позволяет создать ячейку для размещения компонента с указанием ее координат, так что ячейка, расположенная в левом верхнем углу контейнера, будет иметь координаты gridx=0, gridy=0. С помощью значения GridBagConstraints.RELATIVE (которое является значением по умолчанию) вы сможете указать размещение нового компонента не относительно левого верхнего угла контейнера, а относительно компонента, который был добавлен к контейнеру непосредственно перед данным: вплотную к нему справа (для gridx) или снизу (для gridy).
- gridwidth, gridheight: эти параметры позволяют задавать количество ячеек в ряду (gridwidth) или в столбце (gridheight) в области, отведенной под данный компонент. Значения по умолчанию равны 1. Значение GridBagContraints.REMAINDER указывает, что добавляемый компонент должен быть последним в ряду (для gridwidth) или в столбце (для gridheight). Значение GridBagContraints.RELATIVE указывает, что данный компонент должен располагаться после последнего в своем ряду (для gridwidth) или столбце (для gridheight).
- fill: это значение используется в тех случаях, когда размер компонента меньше, чем отводимое под него пространство, и позволяет указать, масштабировать ли компонент, и если да, то как. Этот параметр может принимать значение GridBagContraints.NONE (по умолчанию), GridBagContraints.HORIZONTAL (что означает "растянуть компонент по горизонтали насколько возможно, но не менять его высоту"), GridBagContraints.VERTICAL ("растянуть компонент по вертикали насколько возможно, но не менять его ширину") и GridBagContraints.BOTH ("растянуть компонент по вертикали и горизонтали насколько возможно").
- ipadx, ipady: эти параметры позволяют задавать внутренние отступы, то есть величину, добавляемую к минимальному размеру компонента. Это значит, что ширина компонента не может быть меньше, чем его минимальная натуральная ширина плюс ipadx*2 пикселов (умножение на два вызвано тем, что отступы добавляются к обеим сторонам компонента). Аналогично, высота компонента не может быть меньше, чем его минимальная натуральная высота плюс ipady*2 пикселов.
- insets: этот параметр указывает внешний отступ для компонента, то есть минимальное расстояние между компонентом и границами отведенной для него области.
- anchor: этот параметр используется тогда, когда компонент по размерам меньше отводимой для него области, и позволяет указать размещение компонента внутри этой области. Допустимы следующие значения:
  GridBagContraints.CENTER (центр, по умолчанию)
  GridBagContraints.NORTH (север)
  GridBagContraints.NORTHEAST (северо-восток)
  GridBagContraints.EAST (восток)
  GridBagContraints.SOUTHEAST (юго-восток)
  GridBagContraints.SOUTH (юг)
  GridBagContraints.SOUTHWEST (юго-запад)
  GridBagContraints.WEST (запад)
  GridBagContraints.NORTHWEST (северо-запад)

- weightx, weighty: эта два параметра используются для указания характера распределения пустого пространства и важны для точного задания поведения компонента при изменении размера контейнера. Если ни для одного из компонентов в данном ряду значение weightx не отличается от 0 (или для компонентов в данном столбце - значение weighty), то все компоненты этого ряда (или столбца) будут расположены вплотную друг к другу и отцентрированы внутри своего контейнера. Иными словами, нулевое значение какого-то из этих параметров заставляет менеджер размещения сгонять все пустое пространство по соответствующей координате к краям контейнера.

Выбор менеджера размещения

Теперь вам предстоит решить, какой менеджер размещения лучше всего использовать для нашего апплета. В табл. 8-1 собраны краткие формулировки основных рекомендаций по применению каждого из рассмотренных менеджеров. Можно дать и один общий совет: попробуйте применить несколько разных менеджеров, чтобы на практике проверить применимость каждого из них к вашему случаю.

Таблица 8-1. Выбор менеджера размещения

Что нам нужно	Что мы должны использовать
Быстро разместить все компоненты, не обращая особого внимания на совершенство композиции	FlowLayout
Быстро и по возможности красиво расположить все компоненты	BorderLayout
Разместить компоненты, имеющие одинаковый размер	GridLayout
Разместить компоненты, некоторые из которых имеют одинаковый размер	Используйте GridLayout в отдельной панели, собрав в нее компоненты, имеющие одинаковый размер, а все остальные компоненты вынесите за пределы этой панели
Выводить компоненты на экран по мере необходимости, CardLayout Отображать некоторые компоненты постоянно, а некоторые - по мере необходимости	Используйте CardLayout в панели, предназначенной для вывода компонентов по мере необходимости, а постоянно отображаемые компоненты вынесите за пределы этой панели
Иметь как можно больше контроля над расположением компонентов, а также обеспечить разумную реакцию на изменение размеров контейнера	GridBagLayout

Имейте в виду, что в отдельной панели вполне можно установить менеджер размещения, отличающийся от менеджеров соседних панелей и менеджера всего контейнера, в который входит панель. Например, если в большой панели используется GridBagLayout, то для вложенной в нее панели меньших размеров вполне можно выбрать CardLayout.

Когда создается новый контейнер, ему присваивается менеджер размещения по умолчанию. Чтобы изменить менеджер размещения, действующий в данном контейнере, пользуйтесь выражением:

```
setLayout(new BorderLayout());
```

Если вы имеете несколко панелей, то при задании менеджера размещения нужно указать, в какой панели он будет действовать:

```
input_panel.setLayout(new BorderLayout());
```

Добавление компонентов к контейнеру, для которого установлен отличающийся от стандартного менеджер размещения, имеет только одно отличие от обычного, а именно - вам может понадобиться передавать методу add какие-то параметры расположения:

```
LeftPanel.add("North", ratioField);
```

При использовании менеджера GridBagLayout параметры передаются в специальном классе под названием gridBagConstraints, с помощью которого можно установить для каждого компонента значения определенных параметров расположения. Как правило, компоненты добавляются один за другим в пределах одного горизонтального ряда, пока в этом ряду остается свободное место; следующий добавляемый компонент начинает следующий ряд. Этому правилу в каком-то смысле подчиняется даже BorderLayout - когда вы добавляете несколько компонентов с одним и тем же параметром расположения (например, "South"), они размещаются в своей "части света" по тому же закону.

Выяснение размера для текущего расположения

Разработав и отладив расположение своих элементов интерфейса, вы можете выяснить, какой размер будет иметь окно с компонентами, размещенными таким образом. Для получения этой информации можно использовать любой из следующих методов, возвращающих размер контейнера:

- preferredLayoutSize: public abstract Dimension preferredLayoutSize(Container parent). Этот метод вычисляет естественные ширину и высоту для указанной панели с учетом остальных компонентов в родительском контейнере.
- minimumLayoutSize: public abstract Dimension minimumLayoutSize(Container parent). Этот метод вычисляет минимальные ширину и высоту для указанной панели с учетом остальных компонентов в родительском контейнере.

Примеры

Итак, мы с вами познакомились с несколькими классами API, используемыми при создании более сложного пользовательского интерфейса. Теперь мы рассмотрим примеры кода, в которых используются эти классы и методики работы с ними, о которых мы говорили выше. Большинство наших примеров будут представлять собой развитие апплета пересчета денежных сумм, с которым мы начали работать в предыдущей главе. Таким образом, вы получите представление о том, сколь интересным и разнообразным может быть интерфейс даже такого простого апплета.

Дизайн с использованием фреймов: FlowLayout

Давайте попробуем использовать в апплете пересчета денежных сумм интерфейс, построенный с помощью фреймов. Оказывается, те же самые функции можно реализовать с помощью этого нового интерфейса, введя в исходный текст лишь очень незначительные изменения. Главное изменение состоит в том, что весь апплет теперь будет находиться в отдельном фрейме, а не в области окна броузера Web. В этом примере мы будем пользоваться менеджером FlowLayout, поскольку для нас сейчас главное - быстро получить результат, не тратя много усилий на точное задание расположения компонентов. Эта версия нашего апплета показана на рис. 8-2.

Рис. 8.2.

Пример 8-1a. Апплет, использующий фреймы.

```
import java.awt.*;
import java.applet.Applet;
public class Window1 extends Applet {
public void init(){
                new Frame1();
        }
} // конец Window1
```

Прежде всего нам нужно импортировать библиотеки, в которых находятся нужные нам классы. Обратите внимание, что класс Frame входит в пакет AWT, который импортируется в первой же строке нашей программы. Помните, что имя файла с исходным текстом апплета должно совпадать с именем класса апплета, объявление которого содержится в вышеприведенном примере (Window1). Первое, что делает наш апплет (как и те примеры, с которыми вы познакомились в предыдущей главе), - это вызов метода init. В примере 8-1a создается новый экземпляр класса

Frame1, определение которого приведено ниже. Расширяя этот класс, мы можем определять другие классы, включая в них соответствующее содержимое (в том числе код для обработки событий, происходящих во фрейме). Таким образом можно значительно упростить обработку событий в большом апплете, так как при этом события низкого уровня могут перехватываться и обрабатываться локально, то есть в пределах своей панели, фрейма или диалогового окна.

Пример 8-1b. Апплет, использующий фреймы.

```
class Frame1 extends Frame {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
        Scrollbar ratioScrollBar = new Scrollbar( Scrollbar.HORIZONTAL, 150,
25, 50, 250 );
// объявляем переменные и компоненты AWT,
// которые будут использоваться в Frame1
public Frame1() {
        setLayout(new FlowLayout());
        setFont(new Font("Helvetica", Font.PLAIN, 16));
        setBackground(Color.gray);
        field1.setEditable(true);
        field2.setEditable(false);
        add(ratioField);
        add(ratioScrollBar);
        add(field1);
        add(field2);
        ratioField.setText("1.50");
        resize(field1.preferredSize());
        resize(field2.preferredSize());
        resize(300,250);
        pack();
        show();
} // конец Frame1
```

Поскольку в данном случае мы добавляем новое содержимое к стандартному классу фрейма, мы используем ключевое слово extends. Возможно, вам не совсем понятно, зачем это нужно делать, если мы уже определили класс Frame1. Вспомните, однако, что каждый апплет мы начинаем с объявления "...AppletName extends Applet". Как и в том случае, здесь мы создаем специализированный конструктор. Таким образом, мы получаем свой собственный компонент - в данном случае фрейм, - который затем можно использовать многократно и на основе которого можно создавать другие классы.

Оставшаяся часть примера почти идентична последнему примеру предыдущей главы. Метод setLayout задает менеджер размещения, используемый в созданном фрейме. Метод setFont устанавливает шрифт для вывода текста во фрейме. Аналогичным образом можно установить шрифт для ввода любого из компонентов. После того как мы создали класс Frame1, все компоненты, которые мы будем создавать в нашем апплете, будут попадать во фрейм Frame1, а не в область апплета в броузере Web. Установив размеры фрейма 300 на 250 пикселов, мы вызываем метод pack, который пытается как можно более тесно расположить компоненты в контейнере в пределах ограничений, налагаемых менеджером размещения. Наконец, мы вызываем метод show для вывода фрейма на экран.

Пример 8-1c. Апплет, использующий фреймы.

```
public void convert() {
                float currency1, currency2;
                String InputString = field1.getText();
                field1.setText("");
                currency1 = Float.valueOf(InputString).floatValue();
                currency2 = conversion_ratio * currency1;
                String OutputString = "$" + InputString + " = " + "#" +
Float.toString(currency2) + "\n";
                field2.appendText(OutputString);
        } // конец convert
public boolean handleEvent(Event evt) {
        if (evt.target == ratioScrollBar)
```

```
                {
                        int in;
                        in = ratioScrollBar.getValue();
                        conversion_ratio = in/100f;
ratioField.setText(Float.toString(conversion_ratio));
        }
        if (evt.target == field1)
                {
                        char c=(char)evt.key;
                        if (c == '\n')
                                {
                                convert();
                                }
                        }
                return false;
}
} // конец апплета
```

Метод convert в этом примере ничем не отличается от одноименного метода в примере апплета, с которым мы работали в предыдущей главе. То же самое можно сказать и о методе handleEvent, что особенно интересно, если учесть, что теперь этому методу приходится работать с фреймом. Это объясняется тем, что создаваемый фрейм является расширением и код, отвечающий за обработчику событий, добавлен непосредственно в свойства фрейма. При этом переопределяется только метод handleEvent, принадлежащий классу Frame, а одноименный метод в других классах (например, в базовом классе апплета Window1) остается нетронутым.

Теперь вы уже должны иметь четкое представление о том, как строится апплет, создающий свое собственное окно. Как уже упоминалось выше, существенной частью интерфейса апплетов, использующих много различных элементов интерфейса, являются панели. Использование панелей совместно с менеджерами размещения облегчает визуальную организацию интерфейса апплета. Пример использования панелей вы найдете ниже в этой главе, а сейчас давайте познакомимся с элементом, наиболее близким по функциям к фреймам, - с диалоговым окном.

Диалоговый апплет: BorderLayout

На рис. 8-3 показан апплет, пользующийся простейшим диалоговым окном с одной кнопкой. Этот пример поможет нам разобраться в том, как с помощью диалоговых окон привлечь внимание пользователя и заставить его отреагировать на вопрос или сообщение. В этом примере мы не будем делать еще одну версию апплета пересчета денежных сумм, поскольку диалоговые окна обычно используются для одноразовых запросов, а не для интерактивной работы. Здесь мы будем пользоваться менеджером BorderLayout, который позволит разместить компоненты интерфейса в соответствующих местах окна: текстовая надпись должна быть вверху, инструкция пользователю - в середине, а кнопка Continue - внизу.

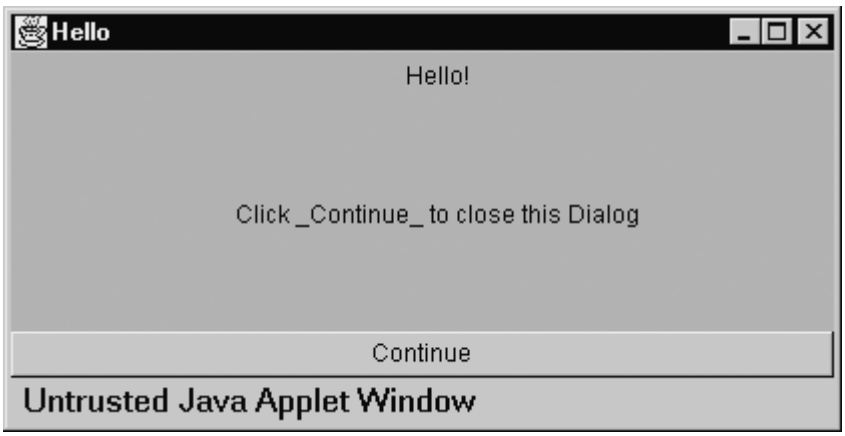

Рис. 8.3.

Пример 8-2а. Апплет с диалоговым окном.

```
import java.awt.*;
import java.applet.Applet;
public class Show_Dialog extends Applet {
        Frame f;
```

```
        Intro_Dialog Hello_Dialog;
public void init(){
        f = new Frame();
        f.resize(50,50);
        f.pack();
Hello_Dialog = new Intro_Dialog(f);
add(new Button("Show Me!")));
}
public boolean action (Event evn, Object obj)
                {
                        if (obj == "Show Me!"_
                                Hello_Dialog.show();
                                return true;
                        }
                                return false;
                }
} // конец апплета Show_Dialog
```

Зачем в этом примере определяется класс Frame? Причина проста: каждое диалоговое окно должно иметь в качестве родителя фрейм. Intro_Dialog - это новый класс, определенный как расширение класса Dialog, а Hello_Dialog - экземпляр этого класса.

СОВЕТ Помните, что любое диалоговое окно должно иметь фрейм в качестве родителя. Поэтому вам придется объявить фрейм, даже если он не выполняет никаких других функций.

В следующем фрагменте кода создается экземпляр класса Frame, хотя выводиться на экран созданный фрейм не будет. Инициализация размера и упаковка фрейма (методом pack) - это не более чем меры безопасности, поскольку, если фрейм останется неинициализированным, а диалоговое окно будет на него ссылаться, может произойти исключение NullPointerException.

Затем мы создаем новый экземпляр класса Intro_Dialog и передаем только что созданный фрейм, делая его родителем созданного диалогового окна Hello_Dialog. Обратите внимание, что кнопка в этом случае добавляется к апплету, а не к фрейму или диалоговому окну, так как эта кнопка используется для вызова на экран диалогового окна. Обработка событий, закодированная выше, относится только к самому апплету, так как все события, происходящие в диалоговом окне, будут обрабатываться с помощью кода, включенного прямо в объявление класса, приведенное ниже. Пока что мы ждем нажатия кнопки "Show Me!" и, перехватив это событие, выводим на экран Hello_Dialog.

Пример 8-2b. Апплет с диалоговым окном, использующий экземпляр фрейма.

```
class Intro_Dialog extends Dialog {
public Intro_Dialog(Frame parent) {
        super(parent, "Hello", true);
        setLayout(new BorderLayout());
        setFont(new Font("Helvetica", Font.PLAIN, 12));
                setBackground(Color.gray);
                add("North", new Label("Hello!", Label.CENTER));
                add("Center", new Label("Click _Continue_ to close this
Dialog", Label.CENTER));
                add("South", new Button("Continue"));
resize(250,250);
        }
public boolean handleEvent(Event evt)
        {
                if (evt.id == Event.ACTION_EVENT )
                        {
                                if("Continue".equals(evt.arg))
                                {
                                dispose();
                                return true;
                                }
                        }
                return false;
```

```
        }
} // конец апплета
```

При инициализации диалогового окна ему могут быть переданы несколько параметров. Поскольку мы используем конструктор Intro_Dialog, принадлежащий к классу Intro_Dialog, эти параметры нужно передать в вышестоящий класс Show_Dialog. Оператор setLayout в действительности можно опустить, потому что менеджер BorderLayout и без того является менеджером по умолчанию для фреймов и диалоговых окон. Мы включили эту строку в пример в качестве напоминания о том, что вызов метода add(newComponent) не будет работать, если не указать дополнительный параметр расположения, - например, add("Center", newComponent). Затем мы добавляем две текстовые надписи и кнопку с надписью "Continue". Метод handleEvent ждет нажатия этой кнопки; когда оно происходит, имя кнопки передается в параметре evt.arg. Проверив значение этого параметра, метод закрывает диалоговое окно. Как и в предыдущих примерах, это можно было бы сделать также с помощью обработчика событий mouse_click или action, однако использование handleEvent позволяет собрать обработку всех событий в одном месте, что весьма удобно.

Апплет с панелями: BorderLayout

Панель - это простейший из всех существующих в AWT контейнеров. Панели удобно использовать для группирования надписей, кнопок и других элементов интерфейса апплета, работающего в окне броузера. Однако можно применять панели и в апплетах, создающих собственные фреймы или диалоговые окна. В этом примере мы также будем пользоваться менеджером BorderLayout для размещения компонентов панели.

В этом разделе мы познакомимся с использованием панелей. Однако мы не будем приводить полностью код всего апплета, а ограничимся лишь частью, имеющей отношение к панелям. Сначала мы создадим две панели, в одной из которых будет размещаться полоса прокрутки и текстовое поле для вывода коэффициента пересчета, а в другой - два текстовых поля для ввода и вывода численных значений. Первая панель будет называться LeftPanel, вторая - RightPanel. Как только панели будут добавлены к апплету, все четыре элемента интерфейса станут доступными (рис. 8-4).

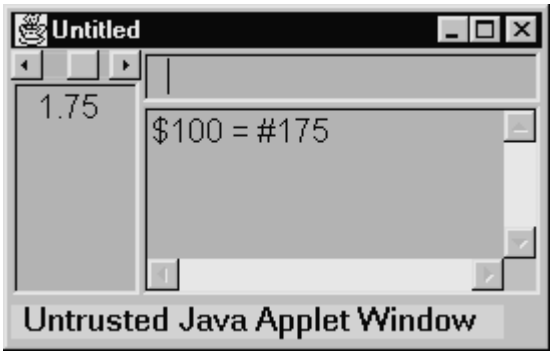

Рис. 8.4.

Пример 8-3а. Апплет, использующий панели.

```
import java.awt.*;
import java.applet.Applet;
public class Panel1 extends Applet {
        public void init() {
                new Frame1();
        }
} // конец Panel1
class Panel_n_Frame extends Panel_n_Frame {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
Scrollbar ratioScrollBar = new Scrollbar( Scrollbar.VERTICAL, 150, 25, 50,
250 );
public Panel_n_Frame() {
        Panel RightPanel = new Panel();
        Panel LeftPanel = new Panel();
        setLayout(new BorderLayout());
```

```
        setFont(new Font("Helvetica", Font.PLAIN, 16));
setBackground(Color.gray);
        field1.setEditable(true);
        field2.setEditable(false);
        LeftPanel.setLayout(new BorderLayout());
        LeftPanel.add("Center", ratioField);
        LeftPanel.add("North", ratioScrollBar);
        RightPanel.setLayout(new BorderLayout());
        RightPanel.add("North", field1);
        RightPanel.add("South", field2);
        ratioField.setText("1.50");
        resize(field1.preferredSize());
        resize(field2.preferredSize());
        resize(300,250);
        add("East", RightPanel);
        add("West", LeftPanel);
        pack();
        show();
} // конец init
```

Здесь для разнообразия мы пользуемся вертикальной полосой прокрутки. Создав панели RightPanel и LeftPanel, мы добавляем к ним компоненты интерфейса. Обратите внимание, что методы setLayout и setFont принадлежат к классу Panel_n_Frame, а не RightPanel и LeftPanel. Если бы интерфейс апплета располагался в окне Web-броузера, а не в отдельном окне, мы бы тем самым меняли свойства той части окна, которая отведена для работы апплета.

Каждая панель может иметь свой менеджер размещения. Как вы уже знаете, менеджер BorderLayout требует указания одного из параметров - North, South, East, West или Center. Аргумент Center, помимо центрирования, имеет также побочный эффект - компонент, указанный с этим параметром, растягивается насколько возможно, заполняя все свободное пространство контейнера. Так, если указать для ratioScrollbar параметр Center вместо West, результат будет довольно забавным - полоса прокрутки увеличится в размерах и займет все свободное место на левой панели. С другой стороны, поле редактирования или холст можно добавлять с этим параметром, так как природа этих элементов допускает свободное масштабирование без искажений. Добавив к панели все необходимые элементы, мы должны будем добавить сами панели в апплет - имеющиеся в панелях компоненты при этом будут добавлены автоматически.

Пример 8-3b. Апплет, использующий панели.

```
public void convert() {
        float currency1, currency2;
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString = "$" + InputString + " = " + "#" +
Float.toString(currency2) + "\n";
        field2.appendText(OutputString);
} // конец convert

public boolean handleEvent(Event evt) {
if (evt.target == ratioScrollBar)
        {
                int in;
                in = ratioScrollBar.getValue();
                conversion_ratio = in/100f;
                ratioField.setText(Float.toString(conversion_ratio));
        }
if (evt.target == field1)
        {
                char c=(char)evt.key;
                if (c == '\n')
                 {
                 convert();
                 }
        }
```

```
        if (evt.id == Event.WINDOW_DESTROY) {
                dispose();
                return true;
        }
        return false;
        }
} // конец апплета
```

Постойте-ка, что означает это WINDOW_DESTROY? Дело в том, что мы обязательно должны обработать событие, генерируемое, когда пользователь щелкает по системной кнопке закрытия окна (или, что то же самое, фрейма). В первом из примеров, где мы знакомились с фреймами, мы этого не делали, но в настоящих апплетах без этого не обойтись. Если вы запустите еще раз апплет из первого примера этой главы и нажмете на кнопку закрытия окна, ничего не произойдет, так как в программе не предусмотрено никаких действий на этот случай. Такой фрейм можно даже свернуть, но уничтожить его полностью можно только одним способом - закрыв запустивший его Web-броузер.

Элементы одинакового размера: GridLayout

Менеджер GridLayout особенно полезен для апплетов, использующих графические изображения. Этот менеджер создает решетку, состоящую из квадратов одинакового размера, в каждом из которых расположен один компонент. Например, с помощью этого менеджера в учебной программе из главы 18 создается шахматная доска, каждое поле которой содержит холст (Canvas), на котором выводится изображение шахматной фигуры. Использование менеджера GridLayout в апплете пересчета денежных сумм показано в примере 8-4.

Пример 8-4. Апплет, использующий менеджер GridLayout.

```
import java.awt.*;
import java.applet.Applet;
public class Grid_Layout1 extends Applet {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(5);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
Scrollbar ratioScrollBar = new Scrollbar( Scrollbar.HORIZONTAL, 150, 25, 50,
250 );
        public void init() {
                setLayout(new GridLayout(3,3));
setFont(new Font("Helvetica", Font.PLAIN, 12));
                setBackground(Color.gray);
                ratioField.setEditable(false);
                field1.setEditable(true);
                field2.setEditable(false);
                ratioField.setText("1.50");
                field1.resize(field1.preferredSize());
                field2.resize(field2.preferredSize());
                ratioField.resize(15,25);
                ratioScrollBar.resize(ratioScrollBar.minimumSize());
                add(ratioField);
                add(ratioScrollBar);
                add(field1);
                add(field2);
} // конец init
```

Внешний вид апплета показан на рис. 8-5. Обратите внимание, как полоса прокрутки и другие компоненты интерфейса, будучи помещены в прямоугольные ячейки, изменили свои пропорции.

Рис. 8.5.

В этом примере реализована решетка размером две ячейки в высоту и две ячейки в ширину, так как весь пример содержит лишь четыре органа управления. На рис. 8-5 хорошо видно, как этот менеджер размещения поступает в ситуации, когда величина ячейки превышает размеры помещаемого в нее компонента.

После создания решетки она, как и в предыдущем примере, заполняется компонентами. Добавление компонентов при использовании менеджера GridLayout не требует указания никаких дополнительных параметров, так как каждый новый компонент продолжает текущий ряд слева направо, а когда ряд завершен, начинает следующий. Остальная часть кода в этом примере аналогична предыдущим примерам; она содержит метод convert и метод handleEvent, обеспечивающие реакцию на нажатие клавиш. При инициализации менеджера размещения ему передаются два параметра, указывающие размер создаваемой решетки. Если один из этих параметров равен нулю, это значит, что в соответствующем направлении решетка может расти настолько, насколько позволяют ограничения контейнера и размеры компонентов.

СОВЕТ Чтобы решетка могла расти в каком-то из направлений без ограниченя количества ячеек, приравняйте нулю соответствующий параметр инициализации менеджера GridLayout.

Динамическая смена компонентов: CardLayout

Менеджер CardLayout позволяет динамически заменять выводимые на экран компоненты, что особенно удобно делать с панелями, содержащими наборы вложенных компонентов. Пример 8-5 иллюстрирует реализацию такого интерфейса. В этом примере все органы управления, относящиеся к изменению коэффициента пересчета, собраны в одной панели (рис. 8-6), а поля редактирования для ввода и вывода пересчитываемых значений - в другой панели (рис. 8-7). Переключение между этими панелями осуществляется выбором одной из двух позиций из выпадающего списка вверху апплета.

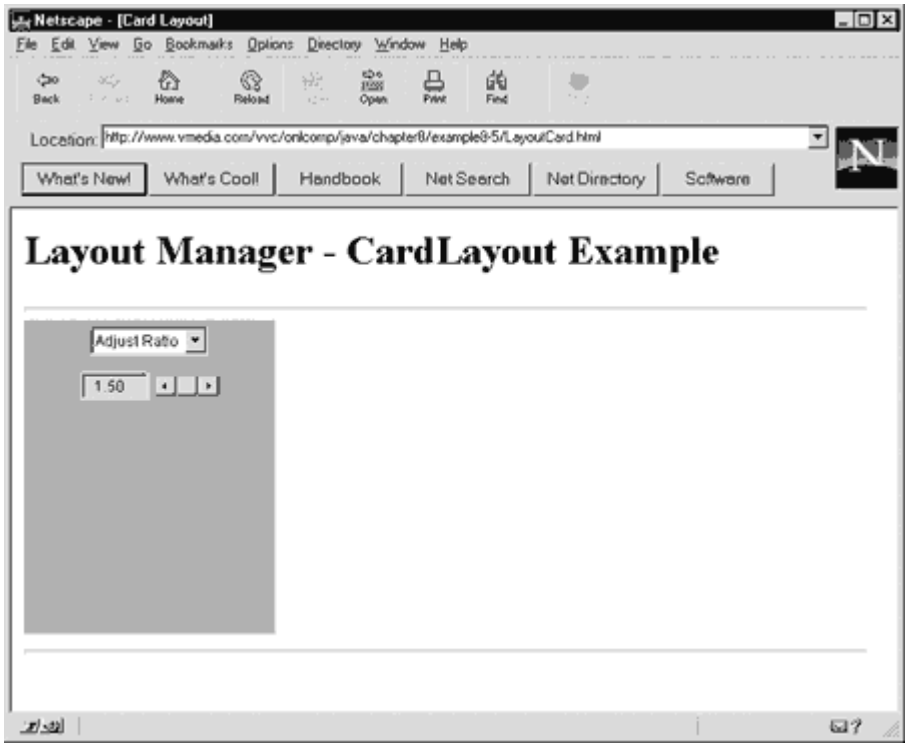

Рис. 8.6.

Пример 8-5а. Апплет, использующий менеджер CardLayout.

```
import java.awt.*;
import java.applet.Applet;
public class LayoutCard extends Applet {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(20);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
Scrollbar ratioScrollBar = new Scrollbar( Scrollbar.HORIZONTAL, 150, 25, 50,
250 );
        Panel cards_panel;
public void init() {
        setLayout(new BorderLayout());
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        setBackground(Color.gray);
        field1.setEditable(true);
        field2.setEditable(false);
        ratioField.setText("1.50");
        field1.resize(field1.preferredSize());
        field2.resize(field2.preferredSize());
ratioField.resize(ratioField.preferredSize());
ratioScrollBar.resize(ratioScrollBar.minimumSize());
```

Панель, в которой будет действовать менеджер размещения CardLayout, необходимо объявить как глобальную, поскольку события, происходящие в этой панели, меняют выводимые части интерфейса и потому не могут быть ограничены вложенными панелями. Оператор setLayout(new BorderLayout()) в вышеприведенном фрагменте устанавливает менеджер размещения для окна апплета:

```
        Panel options_panel = new Panel();
        Choice options = new Choice();
        options.addItem("Adjust Ratio");
        options.addItem("Calculate");
        options_panel.add(options);
        add("North", options_panel);
        cards_panel = new Panel();
        cards_panel.setLayout(new CardLayout());
        Panel ratio_panel = new Panel();
```

```
        Panel calculate_panel = new Panel();
        ratio_panel.add(ratioField);
        ratio_panel.add(ratioScrollBar);
        calculate_panel.add(field1);
        calculate_panel.add(field2);
        cards_panel.add("Adjust Ratio", ratio_panel);
        cards_panel.add("Calculate", calculate_panel);
add("Center", cards_panel);
show();
} // конец init
```

В этом фрагменте в панель под названием options_panel добавляется выпадающий список (Choice). Этот выпадающий список содержит два элемента - "Adjust Ratio" и "Calculate". Затем панель options_panel, содержащая выпадающий список, добавляется в верхнюю часть окна апплета. После этого следует определение панели, в которой будет работать менеджер CardLayout. Как обычно, создается новый экземпляр панели и ему приписывается менеджер размещения CardLayout, а затем создаются еще две панели, которые и будут сменяться на экране при помощи менеджера CardLayout. Эти две панели добавляются в панель cards_panel. Одна из этих панелей содержит поле для вывода коэффициента пересчета и полосу прокрутки для его изменения, а другая - два поля редактирования для ввода исходного значения и вывода результата пересчета. При добавлении этих двух панелей в панель cards_panel им приписываются метки (labels) - текстовые строки "Adjust Ratio" и "Calculate".

После этого панель cards_panel добавляется в окно апплета, которое, таким образом, содержит две панели - cards_panel и options_panel. То, в каком порядке вложенные панели добавлялись в cards_panel, определяет порядок их вывода на экран при вызове методов next или previous. Кроме того, можно вывести любую из имеющихся панелей с помощью метода show, входящего в класс CardLayout, - именно этот метод мы и реализуем ниже в обработчике событий action.

Пример 8-5b. Апплет, использующий менеджер CardLayout.

```
public void convert() {
        float currency1, currency2;
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString =
        "$" + InputString + " = " + "#" + Float.toString(currency2) + "\n";
        field2.appendText(OutputString);
} // конец convert
public synchronized boolean handleEvent(Event evt) {
        if (evt.target == ratioScrollBar)
                {
                int in;
                in = ratioScrollBar.getValue();
                conversion_ratio = in/100f;
                ratioField.setText(Float.toString(conversion_ratio));
                }
        if (evt.target == field1)
                {
                char c=(char)evt.key;
                if (c == '\n')
                        {
                                convert();
                                return true;
                        }
                else { return false; }
                }
        return super.handleEvent(evt);
}
```

При объявлении метода handleEvent нам пришлось использовать ключевое слово synchronized. Это связано с тем, что в том же апплете используется обработчик событий action, и требуется обеспечить, чтобы эти два обработчика событий не вызывались в одно и то же время. Мы могли бы перенести весь код обработки событий в метод action; но в этом примере мы избрали другой

подход, чтобы заодно проиллюстрировать возможность распределить обработку событий между несколькими методами (что позволяет иногда упростить код). Например, можно завести обработчик событий mouseDown, перехватывающий только события, связанные с мышью, а всю остальную обработку событий вынести в метод handleEvent:

```
        public boolean action(Event evt, Object arg) {
                if (evt.target instanceof Choice) {

        ((CardLayout)cards_panel.getLayout()).show(cards_panel,(String)arg);
// * см. объяснение ниже
                        return true;
                }
                return false;
        }
} // конец апплета
```

Строка, после которой стоит комментарий со звездочкой, может показаться слишком запутанной, хотя функция ее в действительности проста: этот оператор обращается к выпадающему списку, выясняет, какой из его элементов выбран в данный момент, и передает этот выбранный элемент непосредственно методу show, где он трактуется как метка той панели, которую нужно вывести в cards_panel. Значение выбранного элемента передается в переменной arg. Помимо метода show, с менеджером CardLayout можно также пользоваться методами next и previous, позволяющими выводить соответственно следующий и предыдущий элемент из тех, что определены в данном контейнере. Этот метод можно, например, использовать при последовательном выводе серии компонентов или для перемещения по цепочке элементов с помощью кнопок типа "Вперед" и "Назад".

Точное расположение: GridBagLayout

Менеджер GridBagLayout - самый сложный и эффективный из менеджеров размещения. В тех случаях, когда ваш интерфейс содержит много компонентов, над расположением которых хочется иметь больше контроля, чем могут обеспечить другие менеджеры размещения, следует использовать GridBagLayout. Предположим, что дизайнер рекомендует для апплета пересчета денежных сумм следующую компоновку: поля вывода коэффициента пересчета и полоса прокрутки для его изменения должны быть расположены на одной горизонтальной линии, ниже по центру - поле ввода исходного значения, а поле для вывода результата должно занимать всю нижнюю часть апплета (рис. 8-8). Такого расположения можно достигнуть только с помощью менеджера GridBagLayout. При этом на расположение каждого компонента сначала налагаются определенные ограничения, и только затем этот компонент добавляется к апплету. Менеджер GridBagLayout располагает компоненты на экране с учетом этих ограничений.

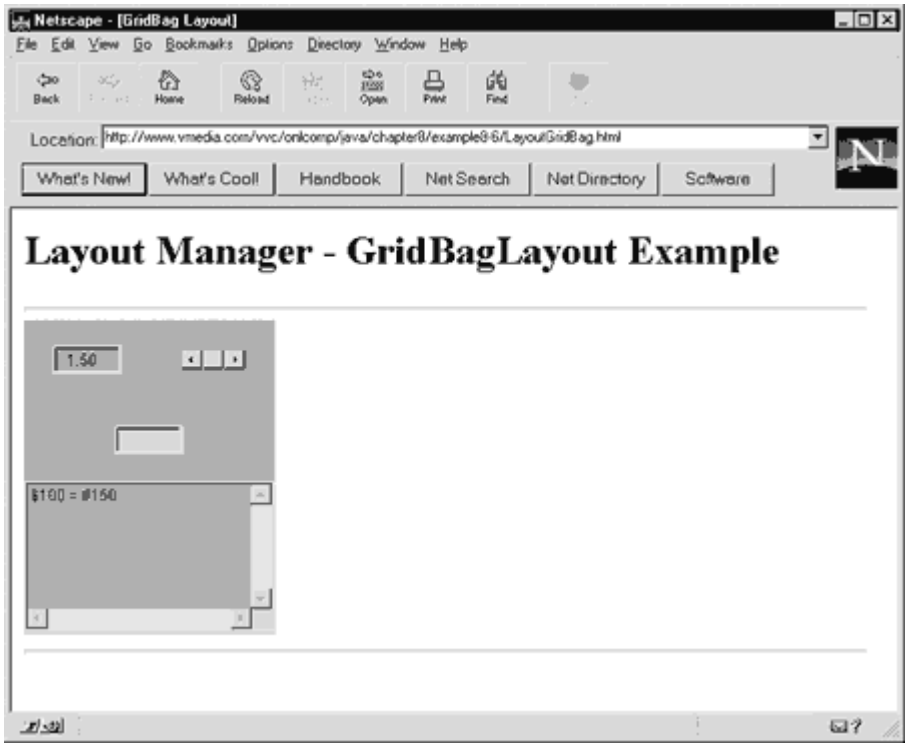

Рис. 8.8.

Пример 8-6. Апплет, использующий менеджер GridBagLayout.

```
public class LayoutGridBag extends Applet {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(5);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
        Scrollbar ratioScrollBar = new Scrollbar( Scrollbar.HORIZONTAL, 150,
25, 50, 250 );
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints Con = new GridBagConstraints();
public void init() {
        setLayout(gridbag);
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        setBackground(Color.gray);
        ratioField.setEditable(false);
        field1.setEditable(true);
        field2.setEditable(false);
        ratioField.setText("1.50");
        field1.resize(field1.preferredSize());
        field2.resize(field2.preferredSize());
        ratioField.resize(ratioField.preferredSize());
        ratioScrollBar.resize(ratioScrollBar.preferredSize());
        Con.weightx=1.0;
        Con.weighty=1.0;
        Con.anchor = GridBagConstraints.CENTER;
        Con.fill = GridBagConstraints.NONE;
        Con.gridwidth=GridBagConstraints.RELATIVE;
        gridbag.setConstraints(ratioField, Con);
        add(ratioField);
Con.gridwidth = GridBagConstraints.REMAINDER;
```

Прежде всего создаются новые экземпляры менеджера размещения GridBagLayout и класса gridBagContraints. В качестве менеджера размещения вызывается GridBag, который является экземпляром GridBagLayout. Если не менять значения weightx и weighty, входящие в класс gridBagContraints (их значения по умолчанию равны нулю), то все компоненты будут тяготеть к центру, тогда как в данном случае нам нужно совсем не это. Значение по умолчанию параметра fill равно BOTH, что означает, что компоненты будут растягиваться по обеим координатам. Здесь, однако, нам не нужно, чтобы полоса прокрутки, поле ratioField и поле ввода растягивались,

поэтому для них параметр fill устанавливается в NONE. Напротив, поле для выходного значения должно быть растянуто насколько возможно, поэтому для него параметру fill присваивается прежнее значение BOTH. Для параметра anchor используется значение CENTER, благодаря чему все компоненты размещаются в центрах отведенных для них областей. Поскольку в момент добавления поле ratioField должно располагаться после последнего элемента в своем ряду, для этого компонента параметр gridwidth устанавливается в RELATIVE. Для полосы прокрутки ratioScroolBar, которая должна быть последней в ряду, значение параметра gridwidth устанавливается равным REMAINDER.

В этом примере один и тот же экземпляр gridBagContraints используется несколько раз, поэтому перед повторным использованием даже тем ограничителям, для которых нас устраивает значение по умолчанию, необходимо явным образом присвоить нужные значения. Наоборот, другие параметры имеют во всех случаях одинаковое значение, и с ними перед повторным использованием экземпляра gridBagContraints ничего делать не нужно. Если бы мы создавали по экземпляру gridBagContraints для каждого компонента, то во всех этих экземплярах пришлось бы устанавливать одни и те же параметры в одни и те же значения. Поэтому более экономный подход - создать один экземпляр этого класса и использовать его для всех компонентов:

```
        gridbag.setConstraints(ratioScrollBar, Con);
        add(ratioScrollBar);
        Con.gridwidth=GridBagConstraints.REMAINDER;
        gridbag.setConstraints(field1, Con);
        add(field1);
        Con.gridy=GridBagConstraints.RELATIVE;
        Con.fill = GridBagConstraints.BOTH;
        Con.gridwidth=GridBagConstraints.REMAINDER;
        Con.gridheight=GridBagConstraints.REMAINDER;
        gridbag.setConstraints(field2, Con);
        add(field2);
} // конец метода init
```

Поле для вывода результата должно занимать все отведенное для него место, поэтому значение fill для него устанавливается равным BOTH, а переменным gridwidth и gridheight присваивается значение REMAINDER. Следующие две строки устанавливают введенные ограничения для поля field2 и добавляют это поле к апплету. Остальная часть кода выглядит так же, как в предыдущих примерах. Теперь вы можете поэкспериментировать со значениями ограничителей, знакомясь с возможностями метода GridBagLayout и совершенствуя интерфейс нашего апплета.

Добавление меню: CardLayout

Давайте вернемся еще раз к примеру, в котором использовался менеджер размещения CardLayout, и заменим в нем выпадающий список на меню (рис. 8-9). Как и прежде, органы управления этого апплета собраны в две панели, отображаемые в окне апплета по очереди. Разобравшись с этим простым примером, вы можете экспериментировать с меню дальше.

Рис. 8.9.

Пример 8-7. Апплет, использующий меню.

```
import java.awt.*;
import java.applet.Applet;
public class Menu1 extends Applet {
public void init(){
new Menu_Frame();
}
```

```
} // конец Menu1
// создаем новый экземпляр класса Menu_Frame
class Menu_Frame extends Frame {
        float conversion_ratio = 1.5f;
        TextField field1 = new TextField(15);
        TextArea field2 = new TextArea(4,20);
        TextField ratioField = new TextField(5);
        Scrollbar ratioScrollBar = new Scrollbar( Scrollbar.HORIZONTAL, 150,
25, 50, 250 );
        Panel cards_panel;
public Menu_Frame() {
        setLayout(new BorderLayout());
        MenuBar Bar = new MenuBar();
        Menu m = new Menu("Operation");
        m.add(new MenuItem("Adjust Ratio"));
        m.add(new MenuItem("Calculate"));
        Bar.add(m);
        setMenuBar(Bar);
```

Первое, что делается в примере 8-7, - создается новое меню. Для этого создается экземпляр класса MenuBar под названием Bar и экземпляр класса Menu под именем m. Затем созданное меню заполняется командами, добавляется к панели Bar, и, наконец, с помощью метода setMenuBar готовая панель добавляется в апплет.

```
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        setBackground(Color.gray);
        ratioField(setEditable(false));
        field1.setEditable(true);
        field2.setEditable(false);
        ratioField.setText("1.50");
        field1.resize(field1.preferredSize());
        field2.resize(field2.preferredSize());
ratioField.resize(ratioField.preferredSize());
ratioScrollBar.resize(ratioScrollBar.preferredSize());
```

Приведенный ниже фрагмент состоит из тех же основных элементов, что и первый пример с менеджером размещения CardLayout. Никаких изменений здесь не требуется; нужно лишь сделать так, чтобы метки панелей совпадали с текстом соответствующих команд меню. На этом совпадении основана процедура смены панелей в методе handleEvent:

```
        cards_panel = new Panel();
        cards_panel.setLayout(new CardLayout());
        Panel ratio_panel = new Panel();
        Panel calculate_panel = new Panel();
        ratio_panel.add(ratioField);
        ratio_panel.add(ratioScrollBar);
        calculate_panel.add(field1);
        calculate_panel.add(field2);
        cards_panel.add("Adjust Ratio", ratio_panel);
        cards_panel.add("Calculate", calculate_panel);
        add("Center", cards_panel);
        resize(250,300);
        pack();
        show();
} // Menu_Frame
public void convert() {
        float currency1, currency2;
        String InputString = field1.getText();
        field1.setText("");
        currency1 = Float.valueOf(InputString).floatValue();
        currency2 = conversion_ratio * currency1;
        String OutputString ="$" + InputString + " = " + "#" +
Float.toString(currency2) + "\n";
```

```
        field2.appendText(OutputString);
} //convert
public boolean handleEvent(Event evt) {
        if (evt.target == ratioScrollBar)
                {
                        int in;
                        in = ratioScrollBar.getValue();
                        conversion_ratio = in/100f;
ratioField.setText(Float.toString(conversion_ratio));
                }
        if (evt.target == field1)
                {
                char c=(char)evt.key;
                if (c == '\n')
                        {
                                convert();
                                return true;
                        }
                else { return false; }
                }
        if ((evt.id == Event.ACTION_EVENT) && (evt.target==m) )
                {
        ((CardLayout)cards_panel.getLayout()).show(cards_panel,(String)evt.arg
);
        return true;
                }
        if (evt.id == Event.WINDOW_DESTROY) {
                dispose();
                return true;
        }
        return false;
}
} // конец Menu_Frame
```

Метод handleEvent обрабатывает событие выбора команды из меню. Этот метод ждет события ACTION_EVENT, пунктом назначения (target) которого является требуемое меню. Если эти условия выполнены, метод проверяет, какая именно команда была выбрана, по значению параметра evt.arg и отображает соответствующую панель в cards_panel. Не забудьте также о необходимости обработки события WINDOW_DESTROY.

Что дальше?

В этой главе мы познакомились с более сложными приемами построения интерфейса пользователя. Применяя методики, рассмотренные в этой главе, и базовые элементы из главы 7, вы сможете создавать апплеты с богатыми, разнообразными интерфейсами. В основе построения интерфейса в Java лежат понятия контейнера и менеджера размещения. Этими же инструментами мы будем пользоваться в учебной части книги, где мы займемся разработкой нескольких полнофункциональных программ, оформленных в виде апплетов.

В следующей главе рассмотрим классы Java, позволяющие работать в апплетах с графической информацией и выводить изображения.

Глава 9

Графика и изображения

В последних двух главах вы изучали создание мощного и привлекательного интерфейса пользователя. Эта глава обучит вас, как выводить и управлять графическими изображениями, которые могут увеличить популярность ваших Java-апплетов и программ.

Класс Graphics, входящий в иерархию java.awt, дает нам множество методов для вывода геометрических форм и размещения текста в области двумерных рисунков (которая связана с видимым объектом Component или Image). Функциональные возможности пакета близки к простому графическому редактору, подобному Microsoft Paintbruish. Пакет java.awt.image, отличный от пакета java.awt и класса java.awt.Image (хотя близко связан с обоими), позволяет взаимодействовать с изображениями на уровне байтов, создавая изображения при помощи заданных программистом алгоритмов. Кроме того, этот пакет предлагает мощные инструментальные средства для обработки уже существующих изображений.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Рисование при помощи класса Graphics

Класс Graphics позволяет рисовать на двумерном холсте с помощью стандартных графических примитивов. Обратите внимание, что вы не можете создавать объект Graphics самостоятельно, так как единственый конструктор определен с модификатором private. Обычно объект Graphics или связан с классом Component, с которым вы работаете, и передан вам, как в случае использования метода paint класса Component, или получен явно методом getGraphics класса Component.

Класс Image также осуществляет метод getGraphics, который возвращает ссылку на объект Graphics, связанный с Image. Эта методика кратко обсуждалась в главе 5, "Апплет в работе".

Рисование контурных объектов

Так как в классе Graphics очень много методов, мы разобьем их на три группы и обсудим поочередно. Первый раздел содержит все методы, связанные с рисованием контурных (незаполненных) рисунков заданным по умолчанию цветом. Эти методы перечислены в табл. 9-1.

Таблица 9-1. Методы рисования класса Graphics для работы с контурными формами

Метод	Описание
drawLine (int, int, int, int)	Выводит линию от позиции, заданной первыми двумя целыми числами (координаты X и Y), до позиции, обозначенной вторыми двумя целыми числами.
drawRect (int, int, int, int)	Выводит прямоугольник. Первые два целых числа указывают верхний левый угол прямоугольника, а последние два целых числа указывают ширину и высоту.
draw3DRect (int,	Выводит подсвеченный трехмерный прямоугольник. Сам прямоугольник

int, int, int, boolean)	задан первыми четырьмя целыми числами, как и в методе drawRect, а булевская переменная указывает, должен ли прямоугольник быть поднят над фоном.
drawRoundRect (int, int, int, int, int, int)	Выводит прямоугольник со скругленными углами, вписанный в нормальный прямоугольник, заданный первыми четырьмя целыми числами. Последние два целых числа указывают ширину и высоту дуги для углов. Ширина и высота дуги определяет диаметр дуги по оси X или Y. Большие значения дают более гладкие закругления.
drawOval (int, int, int, int)	Выводит овал, вписанный в прямоугольник, определенный четырьмя целыми числами.
drawArc (int, int, int, int, int, int)	Выводит дугу, вписанную в прямоугольник, определенный первыми четырьмя целыми числами. Последние два целых числа указывают начальные и конечные углы, измеряемые в градусах. Отсчет углов начинается от центра правой стороны графической области. Положительные значения указывают направление вращения против часовой стрелки, а отрицательные, соответственно, - по часовой стрелке.
drawPolygon (int [], int [], int)	Выводит многоугольник. Целочисленные массивы содержат координаты X и Y для точек, составляющих многоугольник, а целочисленный параметр указывает общее количество точек.
drawPolygon (Polygon)	Выводит многоугольник. Многоугольник задан параметром Polygon.

У методов класса Graphics есть несколько особенностей, которые вы должны знать. Метод draw3DRect выводит прямоугольник, который во всех отношениях идентичен прямоугольнику, получаемому методом drawRect, независимо от того, поднят он или нет. Возможно, что это ошибка, которая будет исправлена в более поздних версиях Java API, а теперь, если вы хотите рисовать трехмерные прямоугольники, вы должны будете вывести их непосредственно. Метод drawArc очень похож на метод drawOval - фактически при вызове drawOval(0, 0, X, Y) и drawArc(0, 0, X, Y, 0, 360) будет получено одно и то же изображение. Использование метода drawArc позволяет запрашивать вывод только некоторой части овала.

Рисование заполненных

Вы можете заполнять краской области экрана различной геометрической формы. Для каждого из контурных методов рисования, перечисленных в табл. 9-1, кроме drawArc и drawLine, есть соответствующий метод заполнения рисунка. Просто замените слово draw на слово fill, и вы получите новую таблицу методов. Подобно методу draw3DRect, метод fill3DRect выводит прямоугольник точно так же, как обычный метод fillRect.

В следующем простом примере мы выведем две линии из противоположных углов экрана, прямоугольник со скругленными углами, центрированный овал и центрированную дугу и заполним центр экрана серым скругленным прямоугольником. Этот пример иллюстрирует использование контурных и заполняющих методов класса Graphics. Снимок экрана показан на рис. 9-1.

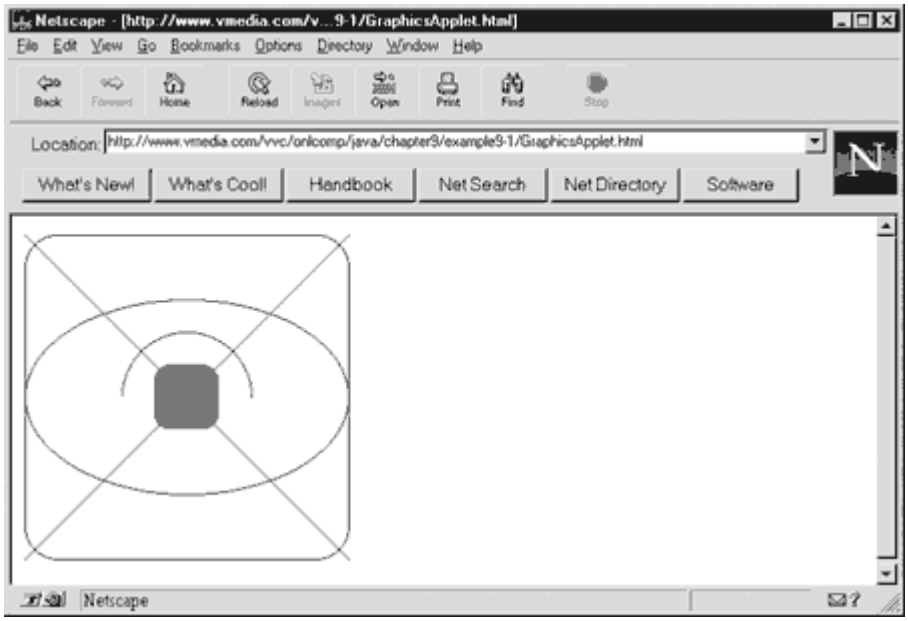

Рис. 9.1.

Пример 9-1. Простой графический апплет.

```
import java.applet.*;
import java.awt.*;
public class GraphicsApplet extends Applet {
        public void init() {
                resize(250,250);
                setBackground(Color.white);
        }
        public void paint(Graphics g) {
                g.setColor(Color.darkGray);
                g.drawLine(0,0,250,250);
                g.drawLine(0,250,250,0);
                g.drawRoundRect(0,0,250,250,50,50);
                g.drawOval(0,50,250,150);
                g.drawArc(75,75,100,100,0,180);
                g.setColor(Color.gray);
                g.fillRoundRect(100,100,50,50,25,25);
        }
}
```

Текст и рисунки

Класс Graphics позволяет выводить некоторые негеометрические объекты, а именно изображения и текст в форме строк, символов и байтовых массивов. Изображения, как мы видели в главе 5, вписываются в прямоугольники на нашем объекте Graphics, но фактически представляют собой сложное изображение, которое было бы трудно или невозможно сделать, используя методы рисования и заполнения, обеспечиваемые классом Graphics.

Методы для вывода текста и изображений перечислены в табл. 9-2. Для них нет дополнительных заполняющих методов.

Таблица 9-2. Методы класса Graphics для вывода текста и изображений

Метод	Описание
drawString(String, int, int)	Выводит данную строку текущим шрифтом в определенном расположении.
drawChars(char[], int, int, int, int)	Преобразует символьный массив, переданный как первый параметр, в строку и выводит ее в месте, определенном последними двумя целыми числами. Первые два целых числа указывают начальное значение индекса для массива и число символов, которые будут преобразованы.
drawBytes(byte[], int, int, int, int)	Выводит массив типа byte тем же самым способом, как и метод drawChars.
drawImage(Image, int,	Выводит данное изображение в месте, определенном двумя целыми

int, ImageObserver)	числами. Параметр lmageObserver указывает класс, который будет наблюдать за данным изображением, чтобы принимать и вывести изображение, когда передача данных закончена. Этот интерфейс будет обсуждаться позже в разделе "Импортирование и создание изображений".
drawImage(Image, int, int, Color, ImageObserver)	Данное изображение выводится в месте, определенном двумя целыми числами, со сплошной заливкой фона заданным цветом.
drawImage(Image, int, int, int, int, ImageObserver)	Изображение помещается в прямоугольник, заданный четырьмя целыми числами. В случае необходимости изображение будет масштабировано, чтобы приспособить его к прямоугольнику.

Изменение класса Graphics

Ряд методов позволяет вам изменять функциональные возможности класса Graphics. Вы можете менять заданный по умолчанию цвет и шрифт, а также копировать и ограничивать области экрана. Эти методы перечислены в табл. 9-3.

Таблица 9-3. Другие методы класса Graphics

Метод	Описание
Graphics create()	Создает новый объект Graphics, который является копией текущего.
Graphics create(int, int, int, int)	Создает новый объект Graphics, который ограничен прямоугольником, заданным четырьмя целыми числами. Любые изменения, сделанные в объекте Graphics, будут отражены в заданной прямоугольной области.
Color getColor()	Возвращает текущий цвет рисунка.
setColor(Color)	Устанавливает текущий цвет рисунка.
setPaintMode()	Устанавливает режим закраски, чтобы всегда красить заданным по умолчанию цветом.
setXORMode(Color)	Устанавливает режим закраски заменой заданного по умолчанию цвета другим цветом. При закрашивании пикселы этого цвета будут заменены на пикселы другого цвета и, в свою очередь, пикселы второго цвета будут заменяться на пикселы первого. Изменение остальных пикселов предсказать нельзя. Однако, если вы закрасите их дважды при активности XORMode, они будут сброшены к своим первоначальным цветам. Этот метод может быть полезен для высвечивания (увеличения яркости) или рисования простых мигающих картинок.
Font getFont()	Возвращает текущий шрифт.
setFont(Font)	Устанавливает текущий шрифт.
FontMetrics getFontMetrics()	Возвращает объект FontMetrics, связанный с текущим шрифтом.
FontMetrics getFontMetrics(Font)	Возвращает объект FontMetrics, связанный с определенным шрифтом.
clipRect(int, int, int, int)	Устанавливает прямоугольник ограничения, заданный четырьмя целыми числами, для рисования в нем объекта Graphics. Когда прямоугольник ограничения установлен, никакое закрашивание не будет выполняться вне этого прямоугольника.
Rectangle getClipRect()	Возвращает прямоугольник ограничения как объект Rectangle.
copyArea(int, int, int, int, int, int)	Копирует прямоугольник, заданный первыми четырьмя целыми числами. Расстояние, на которое нужно переместить его по осям X и Y, задано последними двумя целыми числами.
dispose()	Делает этот объект Graphics непригодным для использования.

Использование класса Image

Хотя методы, предлагаемые классом Graphics, достаточны для создания простых графических изображений, например столбцовых диаграмм или графических меню, они не позволяют вам выполнять сложные графические операции, непосредственно управляя конкретными пикселами. Чтобы сделать это, вы должны использовать класс Image. Объекты Image полезны также тем, что

они обеспечивают удобный непротиворечивый интерфейс для работы с изображениями, импортируемыми из файлов в одном из известных графических стандартов. В этом разделе мы обсудим то, как изображения действительно работают, и асинхронную работу с изображениями, то есть с изображениями, чьи данные недоступны сразу.

Класс Image, часть пакета java.awt, дает несколько методов, при помощи которых можно запрашивать информацию об изображении, например его ширину или высоту. Эти методы перечислены в табл. 9-4.

Таблица 9-4. Методы класса Image

Метод	Описание
Graphics getGraphics()	Возвращает объект Graphics, связанный с изображением.
int getHeight(ImageObserver)	Возвращает высоту изображения, если она известна. Если высота неизвестна, что может быть в случае, если изображение еще не получено, этот метод возвращает -1, и данному lmageObserver будет сообщено, когда высота станет известна.
int getWidth(ImageObserver)	Возвращает ширину изображения, если она известна. Если ширина неизвестна, этот метод возвращает -1, и данному lmageObserver будет сообщено, когда ширина станет известной.
Object getProperty(String, ImageObserver)	Запрашивает значение свойства, заданного параметром String. Например, имя "comments" используется, чтобы сохранить комментарий, содержащий текстовое описание изображения. Если запрошенное свойство не определено для этого изображения, этот метод возвратит объект UndefinedProperty. Если запрошенное свойство не доступно, но имеется (как в случае, если изображение еще не получено), этот метод возвратит пустой указатель, и данному lmageObserver будет сообщено, когда свойство станет известно.
ImageProducer getSource()	Возвращает ImageProducer, который произвел пикселы для этого изображения.
Flush()	Очищает все ресурсы, используемые текущим изображением, включая все сохраненные данные о пикселах. Если это изображение используется снова, оно снова должно быть создано или загружено.

ImageObserver и ImageProducer связываются с помощью интерфейса, упомянутого в табл. 9-4, и используются Java API, чтобы позволить программистам работать с изображениями, чьи данные еще не полностью доступны. Мы обсудим эти интерфейсы в разделе "Интерфейсы для асинхронных образов" ниже в этой главе. Вспомните, что класс Component реализует интерфейс ImageObserver так, что всякий раз, когда вы должны передать ImageObserver как параметр, вы можете передавать ссылку на ваш работающий Component.

Импорт изображений

В главе 5, "Апплет в работе", мы говорили о том, как загрузить изображения из сетевого файлового сервера с помощью URL. Одна из проблем, с которой мы столкнулись при загрузке изображений, состояла в том, что фактически изображение не загружалось до тех пор, пока мы сначала не отобразили его с помощью метода drawImage. Это поведение было особенно значимо, когда мы создали мультипликацию, используя изображения, загруженные через URL. Эта проблема важна не только для изображений, загружаемых из сети, - многие изображения создаются с использованием сложных математических формул, для вычисления которых может потребоваться длительное время. Разработчики Java API в фирме Sun решили позволить программистам начинать использовать изображения в программах прежде, чем загрузка или процесс вычисления для данного изображения будут фактически завершены. Это может быть очень полезно, поскольку вам не придется ждать весь пакет из нескольких изображений перед их выводом; вы можете запрашивать их вывод, когда они готовы для вывода, и ваша программа будет продолжать работу без ожидания.

Использование класса MediaTracker

Хотя обычно удобно иметь изображения, доступные для использования прежде, чем они полностью загружены, бывают ситуации, когда желательно приостановить работу программы, пока данные не будут полностью получены. Как мы видели в главе 5, мультипликация, состоящая из ряда изображений, будет выглядеть очень обрывистой и расплывчатой, пока не будут доступны все изображения. Самым лучшим в этом случае было бы, вероятно, ждать

получения всех изображений и только потом начинать мультипликацию.

Класс MediaTracker, входящий в иерархию java.awt, дает превосходный способ выполнения и управления этим процессом. Основная идея MediaTracker состоит в том, что когда изображение создано, оно добавляется к списку объектов, отслеживаемых MediaTracker. Каждому посреднику назначается некоторое значение приоритета, и программист может в любое время приостановить операции, пока все объекты с заданным приоритетом не будут готовы для отображения. Класс MediaTracker в настоящее время поддерживает только изображения, но фирма Sun в обозримом будущем планирует расширить его до проигрывания звуковых фрагментов и, возможно, видеоклипов. Наиболее полезные методы, предлагаемые классом MediaTracker, перечислены в табл. 9-5.

Таблица 9-5. Наиболее полезные методы класса MediaTracker

Метод	Описание
MediaTracker(Component)	Создает новый MediaTracker, который будет посредником для данного компонента.
addImage(Image, int)	Добавляет изображение к посреднику с определенным уровнем приоритета.
addImage(Image, int, int, int)	Добавляет изображение к посреднику с уровнем приоритета, определяемым первым целым числом. Ширина и высота изображения задаются следующими значениями.
boolean checkAll()	Проверяет, чтобы все прослеживаемые посредники завершили загрузку. Если данные уже не загружаются, этот метод не начинает их загрузку.
boolean checkAIl(boolean)	Проверяет, чтобы все прослеживаемые посредники завершили загрузку. Если объекты еще не загружаются и булевский параметр равен true, этот метод начинает их загрузку.
waitForAII()	Ждет, пока все прослеживаемые посредники не будут завершены.
waitForAIl(long)	Ждет, пока все отслеживаемые посредники не будут завершены или пока пройдет определенное число миллисекунд.
WaitForlD(int)	Ждет, пока все посредники с определенным уровнем приоритета не будут завершены.
waitForlD(int, long)	Ждет, пока все посредники с определенным уровнем приоритета не будут завершены или пока пройдет определенное число миллисекунд.

Ниже приведен простой фрагмент кода, который запрашивает изображение через URL и использует MediaTracker для его получения. Обратите внимание, что мы должны наблюдать за InterruptedException при использовании метода MediaTracker waitForAll - мы приостанавливаем выполнение текущего потока и непосредственно или косвенно должны проверять исключения:

```
MediaTracker tracker;
tracker = new MedlaTracker(this);
Image i;
i = getImage(imageURL);
tracker.addImage( i, 1);
try {
        tracker.waitForAll();
} catch (InterruptedException e) {}
```

Создание изображений

Мы можем теперь загружать изображения из сети и гарантировать их своевременную доставку. А что если мы хотим создавать наши собственные изображения? Раз класс Image определен с модификатором abstract, мы не можем создавать изображения непосредственно. Класс Component предоставляет методы createImage, которые могут использоваться для создания новых пустых объектов класса Image. Для вывода графики в эти пустые изображения мы можем использовать метод getGraphics для запроса объекта Graphics.

СОВЕТ Класс Canvas представляет собой удобный подкласс класса Component. В отличие от большинства других используемых подклассов Component, Canvas не расширяет контейнер. Canvas не может содержать другие компоненты, но, так как класс Image не пропадает из класса

Container, он может быть легко выведен в Canvas. Как правило, класс Canvas используется почти исключительно для вывода изображений.

Класс Component также дает нам некоторые методы для проверки состояния наших изображений и для подготовки их к выводу, хотя, вероятно, проще использовать MediaTracker, чтобы хранить указатели на них. Все эти методы перечислены в табл. 9-6.

Таблица 9-6. Создание изображений с помощью класса Component

Метод	Описание
createImage(int, int)	Создает новое пустое изображение определенной ширины и высоты.
createImage(ImageProducer)	Запрашивает изображение из данного ImageProducer. Этот интерфейс описан ниже в разделе "Интерфейсы для асинхронных изображений" этой главы.
boolean prepareImage(Image, ImageObserver)	Готовит изображение к выводу на экран. Параметр ImageObserver указывает объект, которому посылается состояние изображения; это непосредственно объект Component. Этот метод возвращает булевское значение, готово или нет изображение для вывода.
boolean prepareImage(Image, int, int, ImageObserver)	Готовит к выводу на экран изображение с шириной и высотой, определенными целочисленными параметрами. Параметр ImageObserver работает таким же образом, как в предыдущем методе.

Самый простой способ создания изображений для вашего собственного проекта состоит в том, чтобы создать новый пустой объект Image и использовать метод getGraphics класса Image для непосредственного рисования изображения с помощью методов paint класса Graphics (перечисленных в табл. 9-1 и 9-2). Как мы уже говорили выше, это все еще не дает нам возможности иметь дело с изображением на уровне пикселов. Чтобы работать на уровне пикселов, нам нужно хотя бы поверхностное знакомство с несколькими интерфейсами, разработанными для работы с асинхронными изображениями.

Интерфейсы для асинхронных изображений

Класс java.awt.Image предоставляет три интерфейса - ImageProducer, ImageConsumer и ImageObserver, - которые обеспечивают программистов непротиворечивой средой для создания и обработки данных Image при анимации в реальном масштабе времени. Один из методов createImage, обеспечиваемых классом Component, основан на ImageProducer и передает ему начальные графические данные для Image. Методы ImageProducer для рисования и подготовки изображений требуют, чтобы ImageConsumer послал им данные о пикселах изображения. ImageObserver наблюдает за процессом и получает модификации от ImageProducer, когда новые данные становятся доступными.

Некоторые реализации этих интерфейсов обеспечиваются в Java API. Мы кратко опишем интерфейсы и затем перейдем к их конкретным реализациям в следующем разделе. Среднему программисту никогда не понадобится реализовывать эти интерфейсы, но важно понять то, что фактически происходит, чтобы эффективно использовать изображения.

Интерфейс ImageProducer

Интерфейс ImageProducer должен быть реализован классами, которые могут производить данные для Image. ImageProducer может использоваться как параметр для метода createImage класса Component. Этот интерфейс ответственен за предоставление данных для объекта Image. Интерфейс ImageProducer состоит из пяти методов, перечисленных в табл. 9-7.

Таблица 9-7. Методы интерфейса ImageProducer

Метод	Описание
addConsumer (ImageConsumer)	Добавляет заданный ImageConsumer к списку потребителей, заинтересованных в получении данных изображения.
boolean isConsumer (ImageConsumer)	Возвращает булевское значение, показывающее,

	зарегистрирован ли данный lmageConsumer для получения данных изображения с этим lmageObserver.
removeConsumer (ImageConsumer)	Удаляет заданный ImageConsumer из списка потребителей, заинтересованных в получении данных изображения.
requestTopDownLeftRightResend (ImageConsumer)	Запрашивает, чтобы изображение снова было послано в порядке следования пикселов слева направо и сверху вниз.
startProduction (lmageConsumer)	Регистрирует, что данный ImageConsumer заинтересован в получении данных, и начинает выводить изображение.

Интерфейс ImageConsumer

Интерфейс ImageProducer разработан для использования совместно с интерфейсом ImageConsumer. Интерфейс ImageConsumer должен быть реализован классами, которые заинтересованы в получении пикселов изображения из ImageProducer.

Нам не нужно задумываться о создании ImageConsumer - его функциональные возможности уже встроены в Java API. Методы, необходимые для использования интерфейса ImageConsumer, перечислены в табл. 9-8. Большинство этих методов предназначено для вызова ImageProducer, когда данные изображения становятся доступными.

Таблица 9-8. Методы интерфейса ImageConsumer

Метод	Описание
imageComplete (int)	ImageProducer вызывает этот метод, когда изображение готово или если произошла ошибка при загрузке изображения. Целочисленный параметр указывает состояние изображения. Возможные состояния изображения перечислены в табл. 9-10.
SetColorModel (ColorModel)	Устанавливает класс ColorModel, используемый изображением. Класс ColorModel описан в разделе "Манипулирование изображениями".
SetDimensions (int, int)	Устанавливает ширину и высоту изображения.
setHints (int)	Устанавливает любые сообщения, доступные из lmageObserver. Они могут включать, например, порядок, в котором пикселы будут получены. Целочисленный параметр содержит побитовое ИЛИ всех требуемых сообщений. Возможные сообщения перечислены в табл. 9-9.
SetProperties (Hashtable)	Устанавливает свойства, связанные с изображением. Иногда комментарии или описание включаются наряду с данными изображения. Используется метод ключей для определения, находится ли что-нибудь в хеш-таблице.
SetPixels (int, int, int, int, ColorModel, byte[], int, int)	Устанавливает пикселы в прямоугольнике, заданном первыми четырьмя целыми числами. ColorModel указывает ColorModel, используемую изображением. Массив типа byte содержит фактические данные пиксела, а последние два целых числа содержат начальное значение индекса и число пикселов в каждой строке (размер просмотра). Пиксел с расположением (X, Y) внутри прямоугольника находится в массиве по адресу, заданному следующей формулой: Y * длина_строки + X (+ начальный индекс, если отличен от нуля).
SetPixels (int, int, int, int, ColorMode, int[], int, int)	Делает то же, что и предыдущий метод, но использует целочисленный массив вместо массива типа byte.

Изображение во время создания может находиться в каком-то из определенных состояний. Кроме того, ImageProducer может выдавать зарегистрированным ImageConsumer сообщения (hints) о генерируемых им пикселах. Методы setHints и imageComplete передают эти состояния и сообщения интерфейсу ImageConsumer в виде целых чисел. В ImageConsumer возможные значения сообщений и состояний определены как статические целые. Они перечислены в табл. 9-9 и 9-10. Если ImageProducer требуется передать больше одного из этих сообщений и состояний изображения, он возвратит побитовое ИЛИ соответствующих состояний (более подробно см. врезку "Манипулирование битами").

Таблица 9-9. Сообщения из ImageProducer

Сообщение	Значение
RANDOMPIXELORDER	Пикселы будут переданы без определенного порядка.

TOPDOWNLEFIRIGHT	Пикселы будут переданы в порядке сверху вниз и справа налево.
COMPLETESCANLINES	Пикселы будут переданы в полных растровых строках (горизонтальные строки).
SINGLEPASS	Пикселы будут переданы за один проход, и каждый пиксел будет установлен только однажды. Для некоторых изображений, особенно в формате JPEG, пикселы передаются за несколько проходов, в которых каждый следующий уточняет предыдущий.
SINGLEFRAME	Изображение содержит только один фрейм графических данных, и он уже полный, ImageProducer не будет изменять изображение позже.

Таблица 9-10. Сообщения о состоянии из lmageProducer

Состояние	Значение
IMAGEERROR	При записи данных изображения произошла ошибка.
SINGLEFRAMEDONE	Один фрейм графических данных полон, но последующие фреймы еще не готовы.
STATICIMAGEDONE	Изображение готово, и ImageProducer не будет больше передавать данные для этого изображения.
IMAGEABORTED	Изображение преднамеренно прервано.

Интерфейс ImageObserver

Для тех классов, которые не заинтересованы в получении обязательно всех пикселов, связанных с изображением, но которым требуется информация о продвижении самого процесса создания, фирма Sun обеспечила интерфейс ImageObserver. Этот интерфейс состоит из одного метода imageUpdate и большого количества переменных состояния, определенных в интерфейсе как статические целые. Эти переменные состояния очень похожи по полезности и реализации на переменные состояния и сообщения класса ImageConsumer. Метод интерфейса ImageObserver и его переменные перечислены в табл. 9-11.

Манипулирование битами

Вспомним, что целые числа сохраняются в памяти виртуальной машины Java как последовательность из 32 битов (см. главу 4, "Синтаксис и семантика"). Поразрядное ИЛИ двух целых чисел дает новое целое число, которое имеет значение 1 в каждой позиции, где бит по крайней мере одного из целых чисел имеет значение 1. Например, в памяти виртуальной машины Java целое число 6 представляется как ...0110. Целое число 12 выглядит как ...1100. Поразрядное ИЛИ этих двух целых чисел равно 14: ...1110.

Поразрядное ИЛИ любой комбинации состояний или сообщений гарантирует нам уникальное целочисленное значение. Это происходит потому, что каждое из статических целых чисел, связанных со специфическим состоянием или сообщением, является уникальной степенью двойки, представляемой в памяти как последовательность 32 битов, в которой имеется только один бит со значением 1 (например, 0100).

Поразрядное И двух целых чисел дает целое число, которое имеет значение 1 только в тех позициях, где биты в каждом из целых чисел равны 1. Например, 0110 & 1100 = 0100.

Чтобы выяснить, содержит ли состояние или сообщение интересующую нас информацию, переданную ImageProducer в ImageConsurner, можно использовать поразрядный оператор И.

Немного позже в этой главе мы применим операторы сдвига. Левый оператор поразрядного сдвига обозначается как <<. Он сдвигает левый операнд влево на число битов, определяемых правым операндом. Например, 6 << 1 = 12.

Правый оператор с разрядным сдвигом, >>, работает так же, но сдвигает операнд вправо.

Таблица 9-11. Метод и переменные интерфейса ImageObserver

Метод и переменные	Описание
boolean imageUpdate (Image, int, int, int, int, int)	ImageProducer вызывает этот метод, когда информация, предварительно запрошенная ImageObserver, становится доступной. Параметр Image возвращает ссылку на изображение в запросе, а первый целочисленный параметр содержит поразрядное ИЛИ всех используемых состояний. Последние четыре целых числа обычно интерпретируются как прямоугольник ограничения, но это может быть отменено рядом состояний.

WIDTH	Указывает, что ширина изображения известна и может быть выведена из прямоугольника ограничения.
HEIGHT	Указывает, что высота изображения известна и может быть выведена из прямоугольника ограничения.
PROPERTIES	Свойства изображения известны и могут быть получены через метод getProperty класса Image.
SOMEBITS	Большое количество пикселов (но не все) являются теперь доступными. Гарантируется, что прямоугольник ограничения заполнен.
FRAMEBITS	Полный фрейм многофрейменного изображения теперь доступен. Прямоугольник ограничения игнорируется.
ALLBITS	Изображение является полным, и прямоугольник ограничения игнорируется.
ERROR	Произошла ошибка во время создания изображения.
ABORT	Создание изображения было прервано прежде, чем изображение было закончено. Если при этом не появилось сообщение ERROR, последующее обращение к любым данным изображения завершает его создание.

Подобно другим интерфейсам, описанным здесь, ImageObserver никогда не придется реализовывать большинству программистов. Класс java.awt.Component реализует интерфейс ImageObserver, позволяя ему и всем его потомкам использовать и отображать асинхронные изображения. Всякий раз, когда достаточное количество новых пикселов станет доступным для вывода копии изображения с желаемой или пониженной разрешающей способностью, класс Component запрашивает перерисовку изображения.

Эти интерфейсы могут казаться несколько тяжелыми и неуклюжими для работы, но их преимущества реализованы некоторыми классами, предлагаемыми Java API (в частности, ImageConsumer). Эти классы позволяют декодировать изображения в массив целых чисел и обратно - создавать изображения из целочисленного массива, а также создавать фильтры для изменения изображений.

Манипулирование изображениями

В дополнение к интерфейсам для обработки асинхронных изображений, пакет java.awt.image дает несколько их удобных специфических реализаций. Эти реализации позволяют выполнять с изображениями операции низкого уровня: захват пикселов из изображений с помощью класса PixelGrabber, создание изображений из массивов пикселов с помощью класса MemoryImageSource, фильтрация существующих изображений с помощью класса ImageFilter. Все классы, описанные в этом разделе, являются частью пакета java.awt.image Java API.

Захват пикселов

Класс PixelGrabber - это реализация интерфейса ImageConsumer для захвата пикселов и данных о цвете из изображения или ImageProducer. Можно захватывать все пикселы в данном изображении или только в какой-то прямоугольной части изображения. Конструкторы и другие уникальные методы класса PixelGrabber приведены в табл. 9-12.

Таблица 9-12. Конструкторы и методы класса PixelGrabber

Конструктор / Метод	Описание
PixelGrabber (Image, int, int, int, int, int[], int, int)	Создает новый PixelGrabber, который возвращает пикселы из данного изображения. Первые четыре целочисленных параметра указывают прямоугольник области ограничения, из которой мы хотим получать пикселы. Значения пикселов будут скопированы в целочисленный параметр массива; размер этого массива должен быть равен, по крайней мере, произведению количества элементов по ширине на количество элементов по высоте. Последние два целых числа указывают начальный индекс пиксела в массиве и число пикселов в каждой строке, которые будут помещены в массив (то есть ширину выбранной области).
PixelGrabber (ImageProducer, int, int, int, int, int[], int, int)	Создает новый PixelGrabber, который возвращает пикселы изображения, указанного ImageProducer. Остальная часть параметров идентична параметрам предыдущего конструктора.
grabPixels()	Запрашивает у Image или ImageProducer передачу пикселов. Этот метод

	блокируется, пока данные не получены. Если он был вызван прежде, чем данные прибыли, произойдет исключение InterruptedException.
grabPixels (long)	Запрашивает у Image или ImageProducer передачу пикселов. Этот метод блокируется, пока данные не получены или пока не пройдет число миллисекунд, определенное параметром. Если этот метод был вызван прежде, чем данные прибыли или истекло заданное время, произойдет исключение InterruptedException.
int status()	Возвращает состояние изображения, которое представляет собой поразрядное ИЛИ всех переменных состояния lmageObserver.

Следующий фрагмент кода иллюстрирует использование класса PixelGrabber:

```
PixelGrabber grabber;
int width = myImage.getWidth(this);
int height = myImage.getHeight(this);
int ary[] = new ary[width*height];
grabber = new PixelGrabber(mylmage,0,0,width,height, ary,0 ,width);
try {
        grabber. grabPixels();
} catch (InterruptedException e) {}
int status = grabber.status();
If ((status & ImageObserver.ABORT) || (status & ImageObserver.ERROR)) {
        // произошла ошибка при выборе пикселов изображения
 } else {
        for (int i=0; Kary-length; i++) {
                // делайте что-нибудь с пикселами в массиве
        }
}
```

В этом коде мы используем PixelGrabber, чтобы запросить все пикселы изображения. После того как запрос выполнен, мы проверяем состояние пикселов; если передача была прервана или произошла какая-то ошибка, мы обрабатываем соответствующую ситуацию. В противном случае мы можем считывать пикселы из массива, который мы передали PixelGrabber.

Цветовые модели

Получив массив пикселов, мы должны с ним что-то сделать. Каждый пиксел представляется в этом массиве одиночным целым числом. Необходимо извлечь из этого целого числа все данные, которые нам нужны, чтобы нарисовать или как-то иначе интерпретировать пиксел. Расположение каждого пиксела в изображении может быть выведено из позиции в массиве, и мы используем класс ColorModel, чтобы интерпретировать целое число непосредственно как цвет.

Почти любой цвет может быть представлен как уникальная комбинация оттенков красного, синего и зеленого. Существуют и другие комбинации, основанные на других наборах цветов, но они используются исключительно в компьютерной графике. Почти все цветные мониторы получают каждый пиксел из комбинации красных, синих и зеленых точек. Класс ColorModel не только позволяет нам разбирать целые числа в массиве пикселов и разбивать каждый из них на красную, синюю и зеленую составляющие, но и определять степень прозрачности. Компонент прозрачности управляет выводом пиксела: если пиксел прозрачен, цвет фона преобразуется в отображаемый цвет пиксела. На рис. 9-2 показано наложение частично прозрачного изображения на непрозрачное.

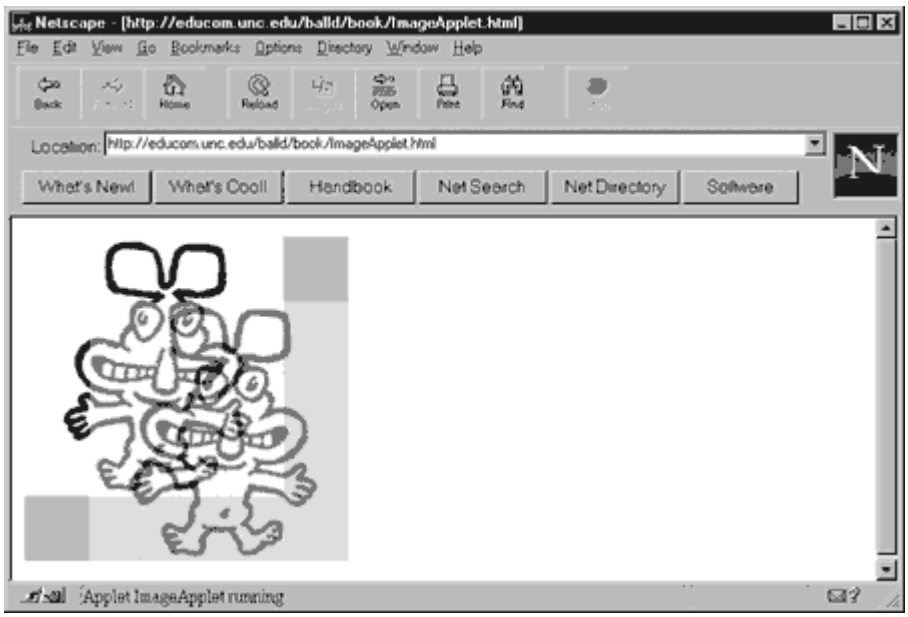

Рис. 9.2.

Класс ColorModel определен с модификатором abstract, хотя два его расширения обеспечиваются в Java API и описаны ниже в этом разделе. В табл. 9-13 приведен список методов класса ColorModel. Обратите внимание, что метод getRGBdefault объявлен с модификатором static и, таким образом, может использоваться для возвращения ссылки времени выполнения на неабстрактный по умолчанию RGB ColorModel.

Таблица 9-13. Методы класса ColorModel

Метод	Описание
ColorModel (int)	Создает цветовую модель, которая поддерживает цвета из определенного числа битов. Число битов непосредственно управляет числом уникальных цветов, возможных в этой модели. Значение восемь битов на пиксел дает 256 различных цветов.
int getAlpha (int)	Возвращает компонент прозрачности пиксела, определенного целочисленным параметром.
int getBlue (int)	Возвращает синюю составляющую пиксела, определенного целочисленным параметром.
int getGreen (int)	Возвращает зеленую составляющую пиксела, определенного целочисленным параметром.
int getRed (int)	Возвращает красную составляющую пиксела, определенного целочисленным параметром.
int getPixelSize()	Возвращает число битов на пиксел.
int getRGB (int)	Возвращает значение пиксела, определенного параметром, в заданной по умолчанию цветовой модели RGB.
static ColorModel getRGBdefault ()	Возвращает заданный по умолчанию ColorModel RGB, используемый пакетом java.awt.image. Эта схема распределяет восемь битов для каждого первичного компонента цветности и прозрачности, который дает 256 градаций красного, синего и зеленого цветов и прозрачности, что дает общее количество 16777216 цветов (не считая оттенков прозрачности).

Есть два базовых пути кодирования красного, синего и зеленого цвета и степени прозрачности, каждый из которых представляет собой целое число из некоторого количества бит в 32-битном целом. При первом способе целочисленное значение пиксела обрабатывается как индекс массива, а красный, синий и зеленый цвета и степень прозрачности берутся из внутренних таблиц палитры. Упорядочивать цвета в таблицах не нужно; соответствие устанавливается создателем модели. IndexColorModel расширяет основной класс ColorModel так, что вы можете создавать новый класс ColorModel, использующий вашу собственную палитру. При построении модели вы создаете массивы компонентов цвета. Когда программист запрашивает цвет из модели, он использует значение пиксела как позицию в массиве и возвращает значение этой позиции из соответствующего массива.

Второй способ извлечения четырех компонентов цвета из 32-разрядного целого числа состоит в том, чтобы выделить части целого числа в разрядные массивы, размеры которых должны в

сумме давать 32. Эта идея проиллюстрирована на рис. 9-3. Каждый разрядный массив содержит информацию о красной, синей и зеленой составляющих и о прозрачности полного цвета. Например, можно выделить 26 битов для красного, 2 бита для синего, 3 бита для зеленого и 1 бит для прозрачности. Согласно этой схеме, мы будем иметь 67108864 возможных оттенков красного, 4 оттенка синего, 8 оттенков зеленого и 2 оттенка прозрачности (полностью непрозрачный или прозрачный). Класс DirectColorModel представляет собой расширение класса ColorModel, который отображает значения пиксела на компоненты цвета, используя эту методику.

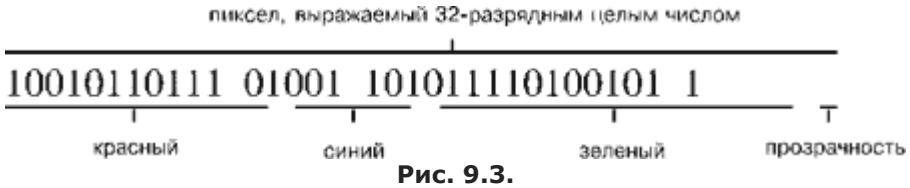

Рис. 9.3.

Теперь массив целочисленных значений пикселов, возвращенных классом PixelGrabber, имеет смысл. ColorModel, используемый классом PixelGrabber, - это заданная по умолчанию модель цвета RGB, и мы можем использовать для нее методы, перечисленные в табл. 9-13, чтобы извлечь из пикселов информацию о цвете. Мы можем добавить следующий код к циклу, приведенному в нашем более раннем примере, чтобы проанализировать конкретные компоненты цвета из массива пикселов:

```
ColorModel cm = ColorModel.getRGBdefault();
int pixel;
for (int l=0; i<ary. length, i++) {
        Pixel = ary[i];
        cm.getRed(pixel);
        cm.getBlue(pixel);
        cm.getGreen(pixel);
        cm.getAlpha(pixel );
}
```

Мы показали, как преобразовать изображения в массивы и как отобразить значения из этих массивов в компоненты цвета с помощью класса ColorModel. Обратное преобразование, превращение массивов в изображения, также легко выполнить с помощью класса MemoryImageSource.

Преобразование массивов в изображения

Класс MemoryImageSource дает способ преобразовывать массивы целых чисел в пикселы для класса Image. Класс MemoryImageSource реализует интерфейс ImageProducer и может использоваться как параметр для метода createImage класса Component. Конструкторы для класса MemoryImageSource приведены в табл. 9-14. Создав новый класс MemoryImageSource, мы можем использовать его для вывода новых изображений с помощью методов createImage класса Component.

Таблица 9-14. Конструкторы класса MemoryImageSource

Конструктор	Описание
MemoryImageSource (int, ColorModel, byte[], int, int)	Создает новый ImageProducer, который строит изображение из значений пикселов в байтовом массиве, передаваемом как параметр. Первые два целых числа указывают требуемую ширину и высоту изображения, а параметр ColorModel задает соотношение между значениями пикселов и компонентами цвета, которые нужно использовать при интерпретации массива. Последние два целых числа указывают начальную позицию в массиве и число байтов, которое занимает каждая строка пикселов в массиве (собственно ширину изобра-жения).
MemoryImageSource (int, int, ColorModel, byte[], int, int, Hashtable)	Работает так же, как предыдущий конструктор. Параметр Hashtable устанавливает свойства изображения.
MemoryImageSource (int, int, ColorModel, int[], int, int)	Создает новый ImageProducer из данного целочисленного массива.

MemoryImageSource(int, int, ColorModel, int[], int, int, Hashtable)	Создает новый ImageProducer из данного целочисленного массива.
MemoryImageSource (int, int, int[], int, int)	Создает новый ImageProducer из данного целочисленного массива, используя заданную по умолчанию цветовую модель RGB.
MemoryImageSource (int, int, ColorModel, int[], int, int, Hashtable)	Создает новый ImageProducer из данного целочисленного массива, используя заданную по умолчанию цветовую модель RGB.

Этот класс лучше всего проиллюстрировать примером. Ниже мы дадим полный апплет, который генерирует из целочисленного массива и выводит изображение. Мы создадим красные составляющие из каждого пиксела, добавляя соответственно отмасштабированные синусы координат X и Y каждого пиксела, аналогично мы создадим синие составляющие каждого пиксела, но уже добавляя косинусы координат X и Y. Окончательные значения каждого пиксела получаются объединением красных и синих составляющих цвета со 100-процентным значением прозрачности в поразрядной операции ИЛИ, сдвигая компоненты, пока они не будут выровнены с соответствующими позициями в заданной по умолчанию схеме цвета RGB. Снимок экрана с готовым изображением показан на рис. 9-4.

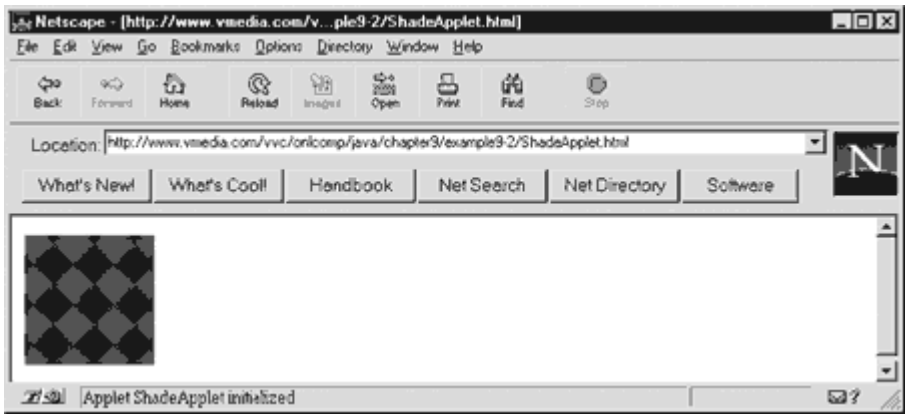

Рис. 9.4.

Пример 9-2. Использование класса MemoryImageSource.

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
public class ShadeApplet extends Applet {
        Image myImage;
        public void init() {
                resize(250,250);
                int ary[] = new int[250*250];
                for (int i=0; i<250; i++) {
                        for (int j=0; j<250; j++) {
                                double x = (16*Math.PI*j/250);
                                double y = (16*Math.PI*i/250);
                                double p = (Math.sin(x)+Math.sin(y)+2)/4;
                                int redvalue = (int)Math.round(p*255);
                                double q = (Math.cos(x)+Math.cos(y)+2)/4;
                                int bluevalue = (int)Math.round(q*255);
                                ary[i*250+j] =
(255<<24)|((redvalue)<<16)|(bluevalue);
                        }
                }
                MemoryImageSource mis;
                mis = new MemoryImageSource(250,250,ary,0,250);
                myImage = createImage(mis);
        }
        public void paint(Graphics g) {
                g.drawImage(myImage,0,0,this);
        }
}
```

Фильтрация изображений

Теперь мы можем создавать изображения непосредственно на уровне битов, без использования графических примитивов, предлагаемых классом Graphics. Мы можем даже управлять изображениями, загруженными из сети; с помощью PixelGrabber мы можем получать значения пикселов из массива. Мы можем провести и обратный процесс, используя класс MemoryImageSource. Java API предоставляет два класса, разработанные специально для этой задачи, - FilteredImageSource и ImageFilter.

Класс FilteredImageSource реализует интерфейс ImageProducer. Он берет ImageProducer и ImageFilter в качестве параметров конструкции. Создавая изображение, он получает данные о пикселах из ImageProducer, определенного в конструкции. Потом изменяет эти данные непротиворечивым способом, продиктованным ImageFilter, и посылает измененные данные обратно через ImageConsumer. Класс ImageFilter реализует интерфейс ImageConsumer. Он не изменяет данные о пикселах. Java API дает два простых расширения, которые позволяют вставлять цветной фильтр или вырезать некоторую область для нового изображения.

Класс RGBImageFilter по умолчанию не выполняет никакой фильтрации цвета, но трансформирует значение пиксела и ColorModel, используемые изображением для заданной по умолчанию модели цвета RGB. Чтобы заставить его фильтровать цвета, мы должны расширить метод filterRGB этого класса для выполнения необходимой нам фильтрации. Рассмотрим пример, который удаляет из изображения весь красный цвет.

Пример 9-3. Фильтр удаления красного цвета.

```
import java.awt.image.*;
public class NoRedFilter extends RGBImageFilter {
        public int filterRGB(int x, int y, int rgb) {
                int alpha = (rgb & 0xff000000)>>24;
                int red = (rgb & 0x00ff0000)>>16;
                int green = (rgb & 0x0000ff00)>>8;
                int blue = (rgb & 0x000000ff);
                return ((alpha<<24)|(green<<8)|(blue));
        }
}
```

Сначала мы расчленяем целую переменную rgb, указывая цвет пиксела в заданной по умолчанию схеме цвета RGB, и затем добавляем прозрачность, зеленую и синюю составляющие обратно в значение RGB, полностью игнорируя красное значение. Обратите внимание, что, так как мы передаем координаты X и Y, мы могли бы изменять цвет фильтрации, основываясь на расположении, но для нашего простого фильтра это ни к чему. Класс RGBColorFilter - удобный фильтр, который вы можете развивать для своих собственных нужд, потому что в случае необходимости он выполняет преобразование к заданной по умолчанию модели цвета RGB.

Класс CropImageFilter максимально прост в использовании; вам не нужно расширять его, чтобы обрезать свои изображения. Взамен вы просто определяете область, которую требуется вырезать при построении фильтра. Конструктор требует четыре целых числа в качестве параметров, определяющих прямоугольник. Вот короткий пример, вырезающий левый верхний угол:

```
FilteredImageSource fis;
CropImageFilter filter;
filter =new CropImageFilter(0,0,10,10);
fis = new FilteredImageSource(myImage.getSource(),filter);
Image myNewImage = createImage(fis);
```

Что дальше?

Классы, описанные в этой главе, позволяют легко и просто генерировать и изменять графические изображения. Класс Graphics позволяет вам выводить изображение с помощью простых графических примитивов, подобных тем, что используются в программах рисования. Расширенные средства создания изображений Java API также позволяют преобразовывать изображения в целочисленные массивы и наоборот, давая точный контроль над изображениями. Фильтрация изображений дает среднему Java-программисту возможность конкурировать с такими программами, как Adobe Photoshop, хотя медлительность используемых при этом вычислений ограничивает полезность этих фильтров до тех пор, пока не станут доступными компиляторы в машинный код.

Следующая глава далеко уходит от проектирования интерфейса пользователя. Глубже копнув в основные правила Java, мы покажем вам подробно, как работают исключения, и исследуем некоторые из проблем, встречающихся при разработке собственных исключений в полном объектно-ориентированном проекте. Кроме того, мы обсудим фундаментальные объекты, из которых получены все классы, и значение, которые они имеют при проектировании собственных иерархий классов.

Глава 10

Структура программы

К настоящему моменту у вас, должно быть, уже сложилось представление о том, как писать мощные апплеты. Вы поэкспериментировали с системой AWT (Abstract Windowing Toolkit) и с анимацией и научились эффективно обрабатывать ввод пользователя. Теперь настало время двинуться дальше, за пределы обыкновенных аплетов, и заставить работать всю остальную часть языка Java.

Конечно, замечательно создавать игрушки и анимированные апплеты на HTML-странице, однако не забывайте о том, что Java - современный объектно-ориентированный язык программирования. Со временем разрабатываемые вами апплеты будут становиться все более сложными, и вам необходимо знать, как создавать простой для восприятия код с высокой способностью к повторному применению в других программах. В этой главе мы изучим способы объединения классов и интерфейсов Java в пакеты (packages), что позволит включать ранее написанные фрагменты кода в новые приложения, и научимся писать устойчивый к ошибочным ситуациям код при помощи механизма обработки исключений.

Сначала мы опишем большой проект с точки зрения объектной ориентации, а затем покажем, каким образом средства Java помогают реализовать его на практике. Наконец, мы обсудим, каким образом изученные ранее средства Java - наследование, интерфейсы и обработка исключений - вписываются в общую картину.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Создание Java-пакетов

Существующие средства интерфейса прикладного программирования (API) Java дают возможность писать приложения практически неограниченной сложности. Разумеется, для того чтобы использовать API по назначению, необходимо написать программу. И вот по мере того, как таких программ становится все больше и больше, у программиста возникает желание использовать в новой разработке фрагменты, уже написанные им ранее для чего-то другого. Раз так, то почему бы не перейти к использованию пакетов?

Одно из основных достоинств объектно-ориентированных языков, к каковым относится и Java, состоит в возможности описать определенную процедуру, или объект, или модель взаимоотношений - одним словом, набор функций, - лишь один раз, оформив ее в виде класса. Далее этот класс можно использовать в других программах безо всяких изменений сколько угодно раз, таким образом освобождаясь от проблемы бесконечного возвращения к старым текстам. В любой новый проект можно без труда вставить ранее написанный объект или интерфейс.

Если классы проектируются с учетом возможного повторного использования, со временем у программиста накапливается целая библиотека классов. В конце концов, такая библиотека начинает играть для проекта ту же, если не большую, роль, что и стандартный интерфейс прикладного программирования.

Прославьтесь в Сети - подарите обществу немного собственного кода

Если у вас уже есть наработанные библиотеки кода, пригодные для использования другими программистами, не жадничайте, сделайте их доступными всем. Взамен вы наверняка получите

массу благодарных электронных посланий; кроме того, кто-нибудь, вероятно, поделится с вами мыслями по поводу возможных усовершенствований. Сотрудничество - воистину то, на чем всегда держался Интернет, и ваша работа может стать частью этого замечательного процесса! На сервере Online Companion есть ссылки на интернетовские архивы с чужими разработками, куда всегда можно привнести что-то свое или позаимствовать уже сделанное другими.

Создание совместимых классов

В главе 6 мы показали, что в классе Object есть несколько методов, как правило, переопределяемых в других классах. Некоторые методы класса Object переопределяются в каждом производном классе и обязательно должны переопределяться в вашем собственном. Что же касается остальных методов, то решение, переопределять их или нет, - стратегически важный для проекта вопрос. Давайте рассмотрим несколько методов и решим, стоит ли переопределять их в каждом конкретном случае. Кроме того, мы рассмотрим последовательность действий, выполняемую в случае, если метод не переопределяется. Начнем с методов, переопределять которые необходимо всегда.

Метод boolean equals(Object o)

Если вы собираетесь сравнивать два объекта одного и того же класса, данный метод переопределяется обязательно. Если он не будет переопределен, объекты этого класса будут равны (метод возвратит true) лишь в одном случае - когда сравниваемые объекты будут на самом деле одним и тем же объектом. То, каким образом переопределять этот метод, в основном зависит от переменных внутри класса. Рассмотрим класс exampleClass. Предположим, что его конструкторы и прочие методы как-нибудь изменяют значения следующих переменных:

```
class exampleClass (
private String s;
private int i;
private int j;
// конструкторы
// методы
}
```

Для такого класса вполне подойдет метод, сравнивающий значения двух его переменных. Его можно было бы превратить в составное булевское выражение, но мы предпочли сравнить обе переменные отдельно:

```
equals(Object obj) {
if (obj == null) return false;
if (!obj instanceof exampleClass)
        return false;
exampleClass ex = (exampleClass)obj;
if (!ex.i != i) return false;
if (!ex.j != j) return false;
return true;
}
```

Первое, что мы делаем, - проверяем, не передан ли методу нулевой указатель вместо самого объекта. Второй оператор проверяет, тот ли тип (exampleClass) у переданного для сравнения объекта. И если тип переданного объекта не совпадает с типом объекта, которому принадлежит описываемый метод, этот оператор возвращает false. Теперь, когда мы знаем, что сравниваются сопоставимые по типу объекты, мы приводим объект obj к типу exampleClass. Далее следуют три строки, операторы которых сравнивают значения переменных двух объектов. Если хотя бы одна переменная не равна другой, метод возвращает false. Обратите внимание на то, что мы имеем доступ к внутренним переменным благодаря тому, что действуем в рамках определения класса, которому они принадлежат.

В данном примере для того, чтобы объекты оказались равными, необходимо, чтобы все переменные были равны между собой. В другой ситуации выполнять эти условия не обязательно. Первый вопрос, который нужно задать себе, приступая к написанию метода для сравнения двух объектов: "Какими свойствами должны обладать два объекта этого класса, чтобы их можно было считать равными?"

Метод String toString()

Проектирование метода toString не представляет сложности - он должен возвращать строку (тип String), отображающую текущее состояние объекта. Его основное назначение - отладка программы при помощи метода System.out.println. В следующем примере описывается класс box (прямоугольник), а метод toString возвращает его высоту и ширину.

```
class myBox{
        int width = 0;
        int height = 0;
        // конструкторы, методы
String toString() {
        return "width = - + width + ", height = - + height;
}
```

Метод int hashCode()

Этот метод, так же как и два только что описанных, должен переопределяться всегда. Если этого не сделать, класс будет невозможно использовать совместно с классом java.util.Hashtable, рассмотренным в главе 6. Для того, чтобы научиться конструировать метод, и для того, чтобы понять, зачем он переопределяется, давайте рассмотрим подробнее структуру хеш-таблиц.

Быстрота поиска - одно из основных достоинств хеш-таблиц. Хеш-таблица никогда не просматривает содержимое всех ключевых полей при поиске. Чтобы найти компонент в таблице, состоящей из N пар "ключ-значение", необходимо произвести всего N операций сравнения. В Java это означает, что необходимо N раз вызвать метод сравнения. Если число N достаточно велико, извлечение элемента из таблицы может занять у системы ощутимое количество времени.

Вместо этого хеш-таблица производит лишь одну операцию сравнения. Это становится возможным благодаря особому принципу хранения ее элементов. Каждая пара "ключ-значение" помещается в корзину. В одной корзине могут находиться несколько пар, но, как правило, в корзине хранится только одна такая пара. Когда хеш-таблице передан ключ, по которому нужно вернуть значение, она спрашивает себя: "В какую корзину я должна поместить этот ключ?" Таблица заглядывает в подходящую корзину и сравнивает переданный ключ с ключами, находящимися в корзине. Чем меньше ключей в корзине, тем эффективнее поиск.

Как хеш-таблица решает, в какую корзину поместить пару "ключ-значение"? Для этого используется специальный алгоритм - хеш-функция (hash function). Ее задача - разместить пары "ключ-значение" по максимально большому количеству корзин и таким образом упростить дальнейший поиск, сведя количество ключей в одной корзине к минимуму. Как правило, при этом соблюдается принцип: "корзин больше, чем пар ключ-значение".

Здесь на сцену выступает метод hashCode. В качестве аргументов хеш-функции, входящей в состав java.util.Hashtable, передаются целые числа. Целые числа необходимы потому, что в одной и той же хеш-таблице могут храниться ключи разных типов. Теперь, когда у вас есть представление о назначении метода hashCode, давайте посмотрим на возможный вариант его корректной реализации. Предположим, что firstObject и secondObject - две отдельные реализации одного и того же класса.

- Если firstObject.equals(secondObject) равно true, целые значения, возвращаемые firstObject.hashCode и secondObject.hashCode, должны быть равны.
- Учитывая это ограничение, метод hashCode должен возвращать как можно более уникальное значение. Чтобы понять почему, давайте рассмотрим наипростейший метод hashCode, удовлетворяющий приведенному выше ограничению:
  int hashCode() {return 1;}

Класс, у которого hashCode определен таким образом, будет работать корректно с любой хеш-таблицей Java. Однако каждый его экземпляр, являющийся ключом хеш-таблицы, будет находиться в одной и той же корзине, а следовательно, метод сравнения будет вызываться для всех ключей. Выходит, что для повышения эффективности поиска следовало бы потрудиться чуть-чуть побольше. В идеальном случае следует разработать такую процедуру, которая бы возвращала уникальное значение hashCode, одинаковое для каждого случая равенства объектов одного и того же класса.

Однако идеальное редко достижимо на практике. Можно только посоветовать не проводить массу бессонных ночей за разработкой идеального метода hashCode, удовлетворяющего абсолютно всем критериям - как правило, это невозможно. Поскольку hashCode возвращает целое число, существует только 4294967296 его возможных значений. Среди всех переменных типа String существует 9223372936854775897 возможных значений только для строки длиной

всего лишь в два символа! Теперь, когда мы знаем, что получить оптимальный хеш-код для типа String невозможно, давайте посмотрим, чего добились разработчики типа String. Их главной целью было обеспечить получение как можно более уникального значения:

```
// метод hashCode класса java.lang.hashCode
public int hashCode() {
        int h = 0;
        int off = offset;
        char val[] = value;
        int len = count;
if (len < 16) {
                for (int i = len; i > 0; i -) {
        h = (h * 37) + val[off ++];
                }
} else {
                // берем лишь некоторые символы
                int skip = len / 8;
                for (int i = len; i > 0; i -= skip, off += skip) {
        h = (h * 39) + val[off];
                }
}
return h;
}
```

СОВЕТ Исходный текст класса String так же, как и текст остальных классов API, находится в JDK. Иногда бывает интересно и полезно заглянуть туда, чтобы познакомиться с точкой зрения разработчиков Java по тому или иному поводу.

Метод Object clone()

Возможно, вы помните из главы 2, что знак операции присваивания = не делает копий объектов. Вместо этого, если одним из его операндов является переменная-экземпляр, одному и тому же объекту присваиваются обе переменные. Предположим, что у нас есть простой связанный список, представленный классом, который содержит целое число и объект - данные списка. Метод setNext позволяет создать структуру данных, приведенную на рис. 10-1.

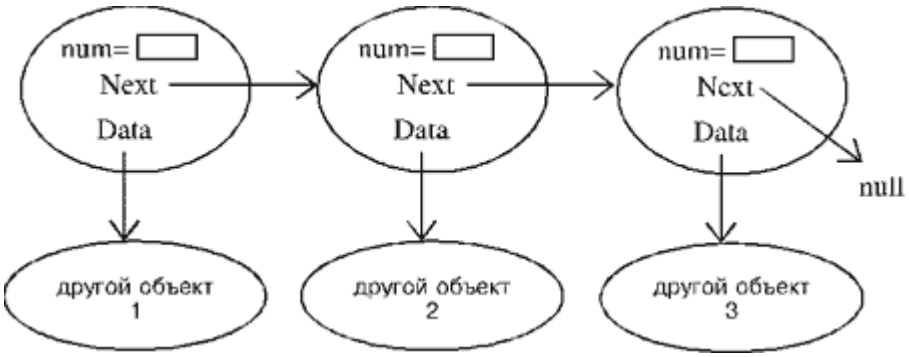

Рис. 10.1.

Обратите внимание на то, что мы переопределили методы класса Object, о которых говорилось выше:

```
public class Node {
        private Object Data = null;
        private Node Next = null;
        private int num;
public void setData(Object obj) {
        Data = obj;}
public void setNum(int someNum) {
        num = someNum;}
public void setNext(Node node) {
```

```
        Next = node;}
public Object getData() {return Data;}
public Node getNextNode() {
        return Next;}
public boolean equals(Object obj) {
        if (!(obj instanceof Node)) return false;
        Node N = (Node) obj;
        return((N.num == num) && Data.equals(N.Data));
}
public int hashCode() {
        return Data.hashCode() * num;}
public String toString() {
        String S = "Data = ";
        S = S + Data.toString();
        S = S + ", num = - + num;
        return S;
}
```

Если мы сделаем следующее:

```
Node A = new Node();
Node B = A;
Integer I = new Integer(32);
B.setNum(16);
B.setData(32);
```

мы не получим два различных экземпляра linkedListNode, но всего лишь один, на который ссылаются переменные A и B. Это значит, что
System.out.println(A);
выдаст результат "data=32, num=16". (Как вы помните, метод System.out. println, получив объект Object в качестве аргумента, вызывает метод toString и выводит результат в виде строки.) А что если нам действительно нужна копия экземпляра? Тут на сцену выходит метод clone класса Object. Он копирует экземпляр, включая текущие состояния всех его переменных. Мы можем сконструировать подкласс класса Node, который допускается клонировать:

```
public class cloneableNode extends Node implements Cloneable {
public Object clone() {
        try {
                return super.clone();
}
        catch (CloneNotSupportedException e) {
                // этого не должно случиться, поскольку
                // мы описали класс как "implements Cloneable"
                throw new InternalError();
        }
}
```

Все, что мы здесь делаем, - это вызываем метод clone класса Object. Поскольку он генерирует исключение CloneNotSupportedException, мы вынуждены его обработать. Попробуем применить наш метод клонирования после того, как создан экземпляр класса cloneableNode:

```
cloneableNode A = new cloneableNode();
A.setNum(16);
Integer I = new Integer(32);
A.setData(I);
cloneableNode B = (cloneableNode) A.clone();
```

Теперь переменные A и B относятся к разным объектам, как показано на рис. 10-2. Однако заметьте, что переменные A.data и B.data остаются теми же. Дело в том, что мы сделали так называемую неполную (shallow) копию экземпляра - его переменные-ссылки на самом деле не копировались. Теперь предположим, что у нас есть связанный список с большим количеством объектов cloneableNode, и мы хотим воспроизвести (реплицировать) его. Нам понадобится следующее определение метода clone:

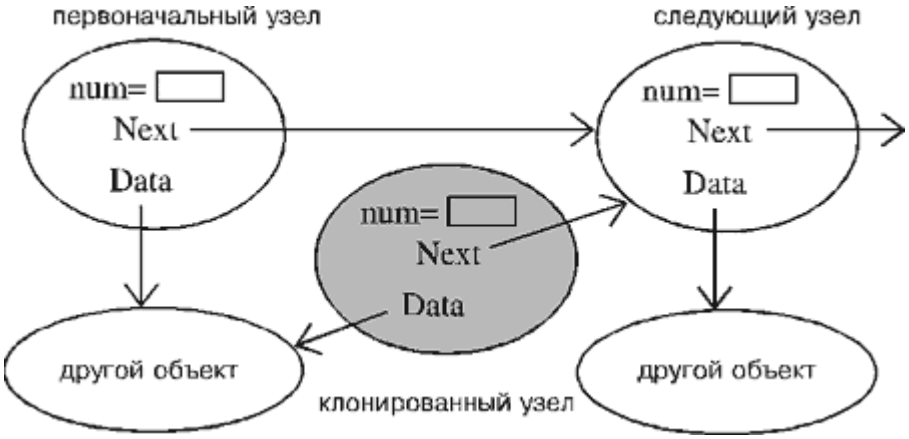

Рис. 10.2.

```
class List extends cloneableNode implements Cloneable {
public Object clone() {
        try {
                List newList = super.clone();
                cloneableNode oldNode = (cloneableNode) this;
                cloneableNode newNode = (cloneableNode) newList;
                while (oldNode.getNextNode() != null) {
                        oldNode = oldNode.getNextNode();
                        newNode.setNextoldNode.clone());
                        newNode = newNode.getNext();
                }
        return newList;
        } catch (CloneNotSupportedException e) {
        throw new InternalError();
}
```

Поскольку метод clone объявлен с модификатором protected, мы не можем скопировать данные. Для того чтобы сделать это, метод необходимо объявить с модификатором public. На этапе разработки программисту нужно решить, должны ли классы, не входящие в состав пакета, иметь возможность клонировать объекты класса. Как правило, необходимость в этом отсутствует.

Рассмотрим класс Graphics, который мы будем обсуждать в последней главе. Экземпляр класса Graphics, в рамках которого можно рисовать, привязан к оболочке времени выполнения программы. На самом деле он представляет собой просто область экрана, на которую выводится рисунок. Клонирование объекта Graphics вовсе не приведет к ожидаемому эффекту - появлению нового экрана. По этой причине нам запрещено клонировать объекты класса Graphics. Точно так же, если вы разрабатываете внутренние по отношению к большой программе классы, вряд ли кому-нибудь пригодится возможность клонировать их, если только не клонируется вся программа целиком.

Кроме того, нам может пригодиться исключение CloneNotSupportedException. Предположим, вы создали класс, умеющий себя клонировать:

```
public class happilyCloning implements Cloneable {
        public Object Clone() {
        try {
                return super.Clone();
        }
catch (CloneNotSupportedException e) {
throw (new InternalError());
        }
}
// что-нибудь еще
}
```

Далее вы решили создать подкласс, который нельзя клонировать кому-либо другому. Поскольку суперкласс этого класса клонируется, не существует иного способа запретить клонирование подкласса, если только не переопределить метод clone. Переопределить его можно таким образом, чтобы всегда возвращалось значение null. В этом случае на вас посыплются проклятия тех, кто будет пытаться клонировать класс и получать бесконечные исключения

NullPointerException. Вместо этого некрасивого способа мы предлагаем генерировать исключение CloneNotSupportedException:

```
public class DontCloneMe {
public Object Clone() throws CloneNotSupportedException {
        throw (new CloneNotSupportedException());
        }
// еще что-нибудь
}
```

Метод void finalize()

Как только экземпляры определенного класса перестают быть нужными программе, для них вызывается метод finalize. Он эквивалентен часто вызываемому для освобождения занятых ресурсов деструктору из языка C++; с его помощью, например, освобождаются выделенные ранее программе блоки памяти. В отличие от C++ для освобождения памяти в Java производится автоматическая сборка мусора (мы уже говорили об этом в главе 2). Если вы программировали на C++, вам, возможно, приходилось писать деструкторы, освобождающие память, почти для каждого класса программы. В Java необходимости в этом нет, поэтому метод finalize используется сравнительно редко.

Давайте вернемся к рассмотренному выше классу - связанному списку. Если выполнить следующую инструкцию, где список curList уже существует:

curList = curList.clone();

то не нужно беспокоиться по поводу памяти, занятой предыдущим экземпляром curList, - сборщик мусора самостоятельно распознает появление неиспользуемой области памяти и позаботится обо всем необходимом.

Таким образом, нам не нужно беспокоиться об освобождении памяти в системе и писать соответствующий метод finalize. Скорее он может пригодиться, когда программа имеет дело с файловой системой или сетевыми ресурсами. До сих пор мы не рассматривали работу с файлами или с сетью (апплеты в любом случае не могут работать с файлами). Поэтому сейчас мы не можем привести пример достойного использования метода finalize. Если все-таки желание увидеть finalize в действии непреодолимо, взгляните на исходный текст FileInputStream.java из пакета java.io. Исходный текст всех классов Java является частью JDK.

Создание повторно используемых компонентов

Итак, мы познакомились с методами класса Object и с тем, как правильно их переопределять. Пришла пора перейти к более общей концепции объектно-ориентированного программирования и поговорить о компонентах, готовых к повторному использованию.

Создание таких компонентов (назовем их "универсальными") является скорее искусством, нежели методикой, которую можно выучить. До того как приступить к написанию кода, необходимо тщательно изучить проблему, которую он призван разрешить. В Java существует множество вещей, облегчающих создание универсальных компонентов, однако сам по себе язык никак не может помочь вам писать эффективнее. Соблюдение хорошего тона в программировании начинается еще до собственно создания программы. Каждый кусочек программы или метод должен быть придирчиво оценен в контексте всего проекта в целом.

Концепция универсальности предполагает, что при написании программы вы учитываете перспективы ее использования в будущем. Ваша цель - создать легко поддающийся структурированной упаковке универсальный код, но сперва нужно решить, что именно стоит помещать в пакеты и как это лучше сделать.

Множество программистов попадают в ловушки, расставленные сложным синтаксисом объектов и вообще объектной ориентацией. Важно помнить, что объектная ориентация дает более интуитивный, по сравнению с другими подходами, способ решения проблемы, однако компьютеры, к сожалению, не столь хорошо умеют решать задачи интуитивно, как мы, люди. Когда-то давно программы писались на языке ассемблера. Можем ли мы написать многопоточный пользовательский интерфейс на этом языке? Конечно! Но сам по себе язык ассемблера не имеет механизмов, которые легко позволили бы описать на нем проблемы, свойственные проекту.

Эти рассуждения можно отнести и к процедурным языкам программирования. В процедурных языках есть средства, позволяющие описывать универсальные последовательности действий, но не универсальные части программы. И это работает замечательно до тех пор, пока программа производит математические расчеты, но отказывает, как только вы попытаетесь решить задачу масштабов реального мира. Рассмотрим проблему управления авиационными перевозками. Вкратце она формулируется так: нам не хотелось бы, чтобы самолеты сталкивались друг с другом во время взлетов и посадок и занимали бы все доступное на аэродроме пространство.

Проблема состоит из следующих компонентов: самолеты, их маршруты, места для парковки, взлетно-посадочные полосы (ВПП), башни слежения за обстановкой в воздухе и ангары. В объектно-ориентированном языке проблему можно представить в виде набора объектов, тогда как в процедурных языках мы ограничены лишь описаниями возможных действий.

Разумеется, вышеописанную проблему можно решить и с помощью процедурного языка, но в этом случае все наше внимание сфокусируется на нем самом. В объектно-ориентированном языке мы освобождаемся от изучения его нюансов и все внимание посвящаем собственно решаемой проблеме. Ключ к написанию эффективной программы не в том, что вы напишете, а в том, правильно ли вы обдумаете проблему еще до начала написания кода. Такая перестановка акцентов свойственна опытным программистам, уже изучившим все особенности того или иного языка. Так что давайте познакомимся с тем, как нужно рассматривать программу для того, чтобы она стала эффективнее.

Что это за проблема?

Вопрос кажется вполне очевидным; тем более удивляет, насколько часто на него не дается ответа. Ответ чаще всего зависит от языка программирования: как я опишу свое решение в терминах этого конкретного языка? Естественно, в конце концов проблема будет описана, но первые ваши мысли должны быть просто упражнением в ее решении. Необходимо подумать, каким образом проблема вписывается в контекст языка на самом высоком уровне. Например, если речь идет о переносимости Java-программы на разные компьютерные платформы, необходимо выяснить, на всех ли из них существуют средства для ее запуска. На этой стадии разработки необходимо избегать ненужных подробностей, а описание проблемы должно быть понятно даже неспециалисту.

Каковы составляющие проблемы и как они взаимодействуют между собой?

Это критическая стадия разделения проблемы на составляющие и победы над ней. Проблема обдумывается в терминах абстрактных, а не привязанных к конкретному языку объектов. Хорошо было бы сделать небольшую схему или чертеж, позволяющий представить проблему наглядным образом, наподобие приведенного на рис. 10-3.

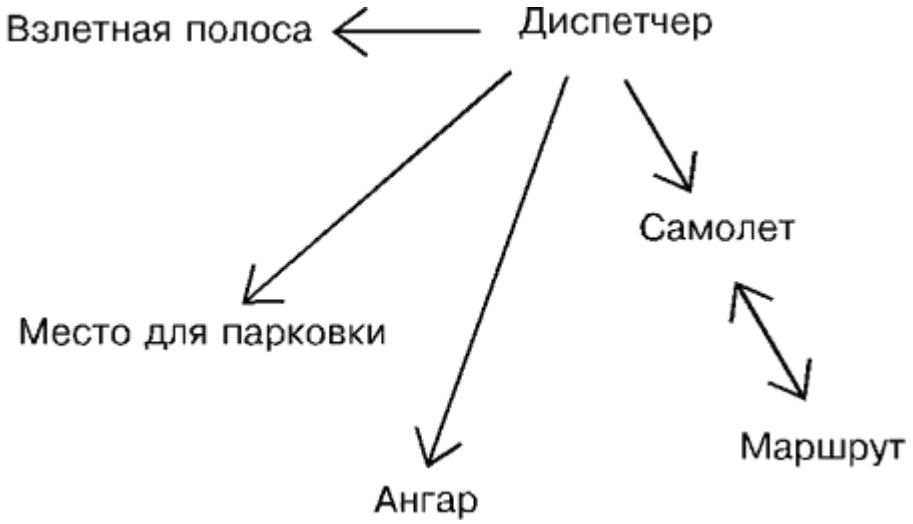

Рис. 10.3.

Задача рисунка - схематично наметить пути взаимодействия элементов, объектов, из которых состоит решаемая проблема, и определить место каждого компонента в общем процессе. Повторим, на этой стадии еще рано думать об объектах со всеми их внутренними данными и методами. Достаточно составить список свойств, как это сделано в табл. 10-1. Для текущей стадии решения проблемы этого оказывается достаточно.

Таблица 10-1. Свойства элементов системы управления воздушными перевозками

Самолет	Имеет определенное положение и траекторию. Следует по некоторому маршруту. Имеет время прибытия и отправления.
Маршрут	Может пересекаться с другими маршрутами. На него оказывают влияние погодные условия.

Ангар	Имеет заданную вместимость и координаты. Может быть заполнен или не заполнен.
ВПП	Имеет определенную длину. Может быть занята либо свободна. Некоторые типы самолетов не могут использовать данную ВПП. Возможность использования ВПП самолетами зависит от погодных условий.
Место для парковки	Имеет заданную вместимость и координаты. На вместимость оказывают влияние погодные условия.
Башня слежения	Необходима для слежения за обстановкой на аэродроме и в воздухе. Необходима для слежения за погодными условиями.

Чем данный проект похож на другие?

До того как приступить к решению проблемы, ее необходимо разделить на области, каждая из которых содержала бы более мелкую отдельную проблему. Проект необходимо структурировать так же, как структурируются большинство программ. Только таким образом достигается универсальность решений - проблемы, составляющие проект, тоже должны оставаться универсальными.

Структурирование не дает запутаться окончательно в массе различных проблем. Например, большинство программ обладают пользовательским интерфейсом. Если вам придет в голову немного его усовершенствовать, добавив, например, новый тип диалогового окна, скорее всего вам захочется использовать его повторно в последующих разработках. Другими словами, хорошо спроектированный пользовательский интерфейс должен быть в известной степени изолирован от окружения конкретного проекта, чтобы его перенос в другую среду не приводил к бесчисленным переделкам и модификациям. То же самое относится и к низкоуровневой работе с сетью или с системой хранения данных. Если вы приложили значительные усилия, разрабатывая какой-нибудь экзотический протокол, вряд ли вам захочется проделывать всю ту же работу заново в новом проекте.

Как обобщить отдельные компоненты?

Отвечая на предыдущий вопрос, мы изолировали пользовательский интерфейс и подсистему работы с сетью от остальной части проекта. Теперь настала пора взглянуть на оставшиеся компоненты и подумать, каким образом их можно сделать универсальными. Ангары, например, весьма похожи на места парковки, а самолет, в сущности, является типом транспортного средства. Размышляя о том, что требуется программе для реализации пользовательского интерфейса, работы с сетью и хранения данных, необходимо подумать о том, могут ли эти компонеты найти применение и в каком-нибудь другом проекте.

Тонкость состоит в том, чтобы вычленить компоненты, которые бы стали универсальными, пригодными для повторного использования. И, приступив затем к написанию классов и интерфейсов, вы действительно получите универсальный код, который можно смело помещать в библиотеку. Что касается рассматриваемого нами примера, то мы конструируем абстрактный класс "транспортное средство", а самолет делаем его подклассом.

Как подогнать исходный текст под объектную ориентацию Java?

Описанная выше методология - один из способов начать писать универсальные компоненты. Чрезвычайно важно пройти через каждую ее стадию. Помните, что крепко подумать необходимо еще до того, как приступать к написанию собственно программы. Теперь мы готовы к этому и приступаем. Каждый из больших компонентов необходимо разбить на более мелкие составляющие - несколько классов и интерфейсов.

На нескольких следующих страницах мы рассмотрим свойства Java, позволяющие описывать реальную проблему в терминах языка программирования. Мы покажем, в каком случае вместо подклассов эффективнее использовать интерфейсы и как правильно пользоваться разнообразными модификаторами методов. Полное понимание всех этих свойств позволит вам самостоятельно создать собственную библиотеку универсальных компонентов Java.

Не затеряйтесь в диаграммах!

Конструируя собственные классы, стоит периодически задавать себе пару вопросов: "Что может пойти не так, как ожидается?" и "Не делаю ли я лишней работы?" Первый вопрос должен привести вас к созданию собственной библиотеки исключений, а второй - сократить чересчур большое количество абстрактных классов и интерфейсов. Очарование универсальности программных компонентов часто приводит к чрезмерному увлечению абстракцией и тому, что никто, кроме вас самих, никогда не сможет воспользоваться этой универсальной сетью.

Вычерчивая диаграмму очередного класса или интерфейса, имейте это в виду.

Превращение проекта в работающий код

И вот наступает момент, когда уже пора переходить от диаграмм к составлению реальной программы. Как мы уже говорили в главе 2, пакет представляет собой контейнер, в который помещаются классы, интерфейсы и исключения. Для того чтобы обозначить, что объект относится к определенному пакету, используется ключевое слово package вместе с присвоенным пакету именем:

```
package somePackage;
public class someClass {
// определение класса
}
```

Если исходный код размещается где-то в иерархии каталогов, в качестве разделителей имен этих каталогов используются точки:

```
package myPackages.someOtherPackage;
public class someOtherClass {
// определение класса
}
```

В любом случае название пакета должно строго соответствовать имени каталога, присвоенного переменной окружения CLASSPATH, о которой мы говорили в главе 2. Точки в названии пакета разделяют имена подкаталогов каталога в переменной CLASSPATH. Для чего предназначены пакеты? Одна из причин - синтаксическая. Компилятору Java нужен способ находить исходный текст программы, а вы не хотите, чтобы все программы находились в одном и том же каталоге. Другая, самая важная причина состоит в наличии модификатора protected. До сих пор мы использовали его не слишком часто. Взамен мы определяли все методы и переменные, к которым можно было обращаться только из объекта, с модификатором private, а все остальные компоненты с модификатором public. Модификатор protected также делает компонент публичным, но в пределах того пакета, которому он принадлежит.

Зачем нам это нужно? Иногда требуется, чтобы определенный метод был доступен лишь некоторым объектам, но не всем. Давайте рассмотрим вышеприведенный пример по управлению воздушными сообщениями. Нашему самолету необходим метод, позволяющий найти свободную ВПП для посадки, но информация к нему может поступить только из диспетчерской. Метод setRunway, принадлежащий классу самолетов, объявляется с модификатором protected и помещается в тот же пакет, в котором описана авиационная диспетчерская. В этот пакет нельзя помещать некоторые классы, например класс "взлетно-посадочная полоса". Представляете, что случится, если этот класс по своему усмотрению вдруг обратится к методу setRunway с разрешением садиться на занятую полосу!

Вы можете сказать: "Все эти классы отнюдь не живут собственной жизнью. Ведь их запрограммировал я, и кому как не мне знать, что от класса "взлетно-посадочная полоса" не может последовать обращения к методу setRunway класса "самолет". Да, в случае простого проекта сказанное вами будет правдой. Но когда дело доходит до крупных проектов, некоторые из модулей действительно начинают жить в определенном смысле собственной жизнью. Как только закончена первичная реализация проекта, кто-нибудь может прийти и попытаться сделать что-то вступающее в полное противоречие с фундаментальными принципами, по которым система проектировалась. При этом совершенно не имеет значения, хорошо ли система была документирована. Лучшая документация в этом случае - это просто запретить компилятору обращаться к методу из класса, из которого таких обращений не было предусмотрено. Модификатор protected имеет смысл применять даже в сравнительно маленьких проектах, над которыми будете работать только вы. Программистам свойственно забывать через некоторое время о том, как была спроектирована программа, и, обратившись к собственному тексту через пару месяцев, вы на самом деле превращаетесь в того "другого" программиста, способного внести полный хаос в отлично задуманную и замечательно реализованную программу.

Те же рассуждения применимы и к модификатору final. Он не относится к концепции пакетов, однако с тем же успехом препятствует ненужной самодеятельности других людей, сопровождающих вашу программу. Как было рассмотрено в главе 4, модификатор final обозначает, что класс, метод или переменная не могут иметь подклассов. С одной стороны, это препятствует конструированию универсальных компонентов, но, с другой стороны, иногда нам вовсе не хочется, чтобы другие люди получили возможность поменять класс, метод или

переменную путем создания подклассов на основе нашего "конечного" класса. Давайте рассмотрим следующий пример, касающийся класса "взлетно-посадочная полоса":

```
class Runway {
        private static final phisicalLocation;
        private static final phisicalLength;
        private static final phisicalWidth;
// другие переменные
Runway(int i, int j, int k) {
        physicalLocation = i;
        physicalLength = j;
        physicalWidth = k;
}
// другие конструкторы
public final int getPhysicalLocation() {
        return physicalLocation;
}
public final int getPhysicalLength() {
        return physicalLength;
}
public final int getPhysicalWidth() {
        return physicalWidth;
}
// другие методы
}
```

В отличие от физических размеров взлетно-посадочной полосы, которые не меняются во времени, длина пробега самолета сильно зависит от состояния погодных условий. Таким образом, физические размеры ВПП (переменные и методы, имеющие к ним отношение) объявляются с модификатором final и не подлежат изменениям со стороны других методов.

Как и модификатор protected, модификатор final позволяет защитить разработку программы, воплощенную в готовом коде, от внешних некорректных на нее посягательств. Применение этих модификаторов облегчает дальнейшее сопровождение программы, а также делает ее компонеты более универсальными.

Техника приведения типов объектов

Одним из способов повторного использования класса является создание его подкласса. При этом необходимо дописать лишь несколько отсутствующих у класса-родителя методов и получить класс, полностью готовый к работе. Недостающие методы потомка уже описаны в классе-родителе. Проектируя большую систему, необходимо учитывать преимущества Java, относящиеся к приведению типов объектов.

Давайте еще раз обратимся к классу "самолет". Когда мы говорили о способах обобщения модулей - составляющих программы, мы предложили сделать этот класс подклассом более общего типа "транспортное средство" (vehicle). Класс "транспортное средство" должен обладать следующими характеристиками:

```
abstract class Vehicle {
        String vehicleType();
        Vector getLocation();
        Vector getTrajectory();
        Path getPath();
        float currentFuelAmount();
        // возвращает значение от 0 до 1
        }
```

Метод getPath возвращает абстрактный объект Path:

```
abstract class Path {
        String pathType();
        Vector startingPoint();
        Vector endingPoint();
        int Length();
```

```
        }
```

Все, что нами было написано до этого момента, представляет скорее лишь академический интерес. Разумеется, ваши коллеги высоко оценят объектно-ориентированный стиль программирования, однако нам необходимо сделать что-то практически полезное. Преимущества станут очевидными, как только мы создадим подклассы наших абстрактных классов для разных типов транспортных средств (Vehicle) и их маршрутов (Path). Далее, экземплярами этих подклассов можно будет управлять при помощи универсальных методов, описанных в классе-родителе "транспортное средство". То же самое можно сказать и об интерфейсах, подклассы которых образуются аналогичным образом.

Проверка кода на устойчивость

Программы необходимо снабжать средствами защиты от разнообразных ошибок, увы, иногда случающихся при исполнении. Пользователи не всегда вводят правильные данные, а программа не всегда получает все требуемые для нормальной работы ресурсы от системы. В более древних языках программирования защита от ошибок реализовывалась в виде многочисленных и трудночитаемых блоков вида if-then-else. В языке Java используется механизм обработки исключений (исключительных ситуаций) - он здорово помогает, эффективно защищая программу от возникновения ошибочных ситуаций, однако не является магической панацеей от всех бед. Вам все равно придется самостоятельно выяснять места в программе, способные привести к ошибке, и писать соответствующий код для их обработки. С другой стороны, обработка исключений делает исходный текст проще, очевиднее и удобнее в сопровождении.

СОВЕТ Механизм обработки исключений в Java эквивалентен тому же механизму в C++. Однако к нему добавлен блок finally, выполняющийся после любого блока catch.

Нам уже неоднократно приходилось пользоваться механизмом обработки исключений. Многие методы и конструкторы из интерфейса прикладного программирования (API) генерируют исключения. Исключение генерируется в том случае, если внутри данного метода происходит что-то неожиданное, а блок catch решает, что с этой неожиданностью делать. Поскольку нам уже приходилось иметь дело с исключениями, начнем сразу с того, как сделать эффективнее их обработку. Когда вы узнаете, как пишутся методы и конструкторы, генерирующие исключения, вы научитесь писать их самостоятельно. Завершим мы наш экскурс написанием собственных подклассов-исключений, позволяющих максимально приблизить обработку ошибочных ситуаций к конкретным требованиям проекта.

Перехват исключений

Впервые с обработкой исключений вы встретились в главе 5, в процессе загрузки по сети графических файлов. Наверное, вы помните, что конструктор URL генерирует исключение, если не может преобразовать переменную-аргумент типа String в формат URL. Давайте посмотрим, что тут можно сделать:

```
public class someApplet extends Applet {
Image I;
// другие описания
public void init() {
String imageName = getParam("image");
try {
        URL imageURL = new URL(imageName);
        // он может сгенерировать исключение MalformedURLException
        I = getImage(imageURL);
        // инициализируем остальные части апплета
        }
catch (MalformedURLException e) {
        String err = e.getMessage();
        showStatus("Couldn't get - +imageName + ":" + e.getMessage);
        stop();
        }
```

```
// другие методы
}
```

Вот простой пример, демонстрирующий использование исключений. Блок try определяет то, что мы намереваемся делать, если все в порядке, тогда как блок catch - то, что необходимо сделать, если в блоке try появится исключение MalformedURLException.

Обработка исключений - важная составляющая правил корректного обращения с любым методом или конструктором программы. Но что такое корректная обработка исключения? Синтаксис нашего примера настолько прост, что не дает полноценного ответа на этот вопрос. Мы должны познакомиться с возможностями обработки исключений в полном масштабе. А именно, необходимо рассмотреть следующие вопросы:

- Исключения являются полноправными классами языка Java. Раз так, то в них существуют методы, вызывая которые можно извлечь более подробную информацию о происходящих в системе событиях.
- Некоторые исключения должны обрабатываться явным образом, тогда как другие - нет. Первые называются "исключениями времени выполнения - (RuntimeException) и должны обрабатываться в любом случае.
- В паре с одним блоком try могут работать несколько блоков catch. Это значит, что в пределах одного блока try можно генерировать несколько различных исключений. На каждое такое исключение может приходиться соответствующий ему блок catch.
- Совместно с блоками try и catch можно применять блоки finally.

Изучая тонкости методов обработки исключений, мы будем продолжать разработку нашего простого примера. К концу исследования у вас сформируется четкое представление о том, как правильно получить доступ к ресурсам, основываясь на информации из принимаемой HTML-страницы. Кроме того, вы научитесь применять концептуальные средства, предназначенные для повышения эффективности использования методов и конструкторов, генерирующих исключения.

Исключения в качестве объектов

Наш блок catch включал следующий оператор:
showStatus(e.getMessage());

Здесь подразумевается, что e - это объект, один из членов которого, метод getMessage, возвращает значение типа String. Разумеется, все эти предположения абсолютно правильны. Все исключения, которые можно захватить, являются подклассами класса Throwable, находящегося в пакете java.lang. Поскольку Throwable всегда является суперклассом всякого захватываемого исключения, для анализа ситуации можно пользоваться любым из его методов, определенных с модификатором private. Эти методы перечислены в табл. 10-2.

Таблица 10-2. Методы класса Throwable, определенные с модификатором private

Метод	Описание
String getMessage()	Возвращает детализированное сообщение. Если объект Thowable сконструирован при помощи строки String,, она выводится на экран.
void printStackTrace()	Печатает содержимое стека трассировки. Содержимое стека - это все методы,, прерванные в момент генерации исключения. String toString(), Возвращает описание.
Throwable fillInStackTrace()	Заполняет содержимое стека трассировки. Полезен в случае,, если обрабатываемое исключение генерируется повторно.

Методы класса Throwable могут быть переопределены подклассом Exception, и к ним, разумеется, могут быть добавлены дополнительные методы. В документации API исключения описаны на соответствующих Web-страницах. Пример такой страницы приведен на рис. 10-4.

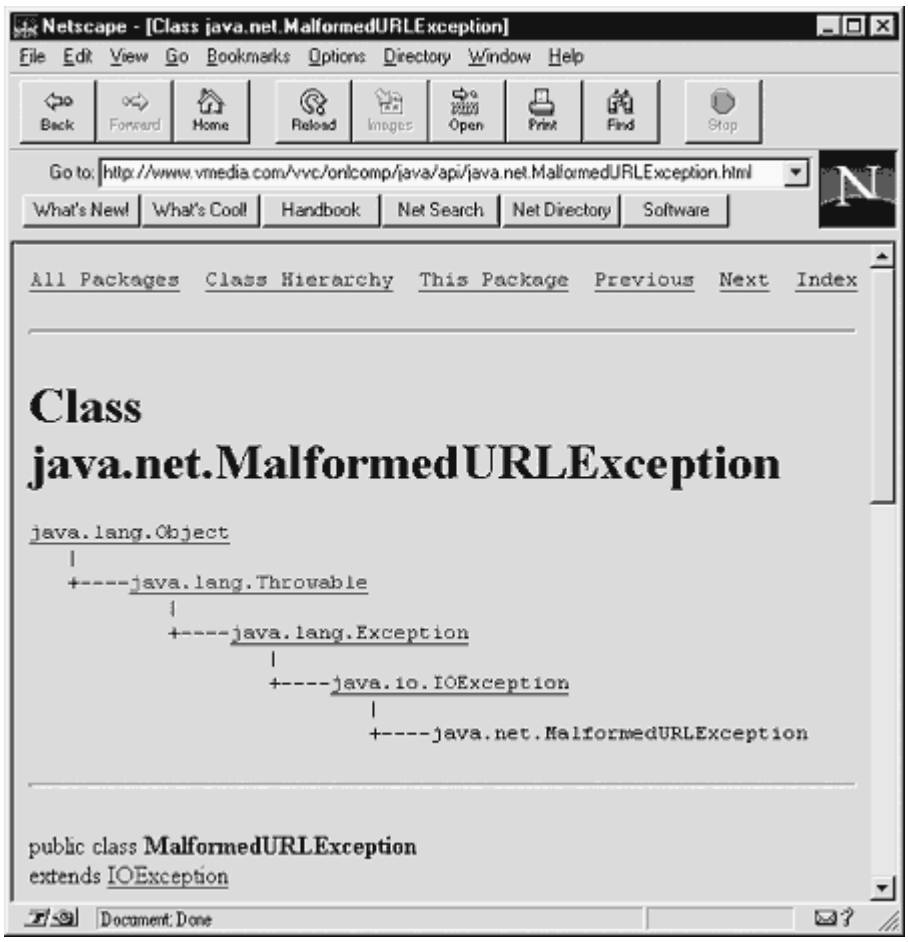

Рис. 10.4.

СОВЕТ Не стоит беспокоиться о том, как поступать с различными объектами, передаваемыми при генерации исключений. Ни метод, ни коструктор просто не имеют права сгенерировать объект-исключение, например String. Дело в том, что String не является подклассом класса Throwable. Единственный тип объекта, который допустимо передавать при генерации исключения, - это тип, являющийся подклассом Throwable.

Различные типы исключений

До настоящего момента мы относились к исключениям как к данностям, которые можно захватывать с помощью операторов, входящих в блоки catch. Теперь вы знаете, что исключения входят в общую иерархию, основанием которой является класс Throwable. Кроме класса Exception, от основания Throwable отходят еще две ветви специального назначения: класс Error и класс RuntimeException. Три этих класса и отношения между ними показаны на рис. 10-5.

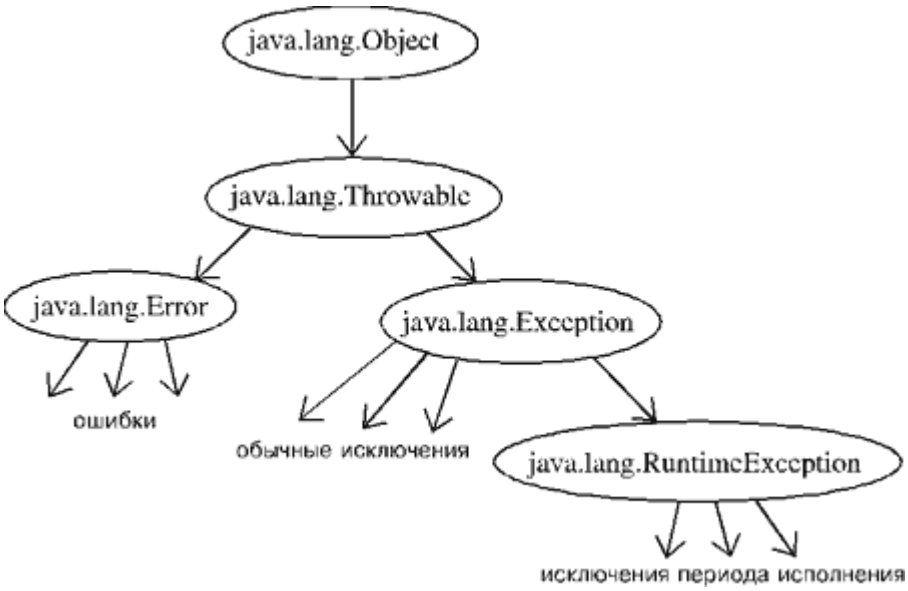

Рис. 10.5.

Три группы, объединенные общим родителем, классом Throwable, отличаются друг от друга функционально и стилистически. Нам как программистам важны скорее их функциональные особенности. Члены второй группы, называемые просто исключениями (Exception), должны обязательно обрабатываться в тех случаях, когда их генерация возможна. Это значит, что данный фрагмент кода не будет компилироваться из-за того, что исключение MalformedURLException не обрабатывается:

```
URL U = new URL("http://www.vmedia.com/?.?");
Image I = getImage(U);
//компилироваться не будет - нет блоков try-catch!
```

В отличие от просто исключений, ошибки (Error) и исключения времени выполнения (Runtime Exception) не нужно обрабатывать явным образом. Если бы мы были обязаны делать это, исходный текст программы превратился бы в непроходимые дебри. Например, если бы мы были должны захватывать исключения типа OutOfMemoryError ("недостаточно памяти"), это приходилось бы делать после каждого оператора, генерирующего любой новый объект! То же самое относится и к исключению NullPointerException ("нулевой указатель"), генерируемому при попытке доступа к нестатическому члену какого-либо объекта до того, как он был реализован. Следующий фрагмент кода демонстрирует один из способов генерации этого исключения:

```
String S;
S.toLowerCase();
// будет сгенерировано исключение NullPointerException
```

Если бы мы были обязаны обрабатывать исключение NullPointerException, это приходилось бы делать каждый раз при обращении к любому объекту. Это было бы не менее болезненно, чем обязательная обработка OutOfMemoryError.

Основное различие между исключениями времени выполнения и ошибками состоит в типах возникающих в системе сбоев, которые они представляют. Ошибки представляют собой ситуации, в которых ваша программа вряд ли может чем-нибудь помочь. Например, вряд ли программа сможет справиться с ошибкой типа "недостаточно памяти". Поэтому в обработке ошибок программой немного смысла. Лучшее, что можно сделать в этой ситуации, - корректно завершить работу приложения.

Исключения времени выполнения - случай сбоев в работе, с которыми наверняка можно самостоятельно справиться. Их, в отличие от ошибок, иногда даже необходимо обрабатывать. Помните, как в главе 6 нам приходилось преобразовывать значения типа String в тип int? В этом случае мы можем расставить блоки try-catch следующим образом:

```
String S="20";
try {
int i = Integer.parseInt(S);
}
```

```
catch (NumberFormatException e) {
        System.out.println("Couldn't convert - + S);
}
```

Мы не обязаны обрабатывать исключение времени выполнения NumberFormatException. Но если значение строки S появляется в результате пользовательского ввода, обработать его совершенно необходимо. Если пользователь ошибется, а возникнувшее при этом исключение не будет обработано, то программа, не подозревая ничего дурного, продолжит работу, что приведет к ее аварийному останову через некоторое время.

Обработка суперкласса класса Exception

До настоящего момента нам приходилось обрабатывать именно те исключения, которые могли быть сгенерированы в определенном месте программы. Но поскольку все исключения подчинены общим правилам наследования классов в Java, обрабатывать можно не только отдельные исключения, но также их суперклассы - вплоть до основного класса Object. Мы можем обработать сам класс Throwable - в этом случае в ловушку попадет все, что только может генерироваться. (К сожалению, в этом случае у нас не будет возможности понять, что именно произошло в системе.) С другой стороны, если мы имеем дело с весьма расширенной иерархией исключений, удобнее было бы установить обработку одного суперкласса из этой иерархии, нежели обрабатывать каждое исключение в отдельности.

Несколько блоков catch в одной конструкции

Итак, система обработки исключений заставляет нас внимательно относиться к возможным неожиданностям в работе приложения. К сожалению, на примере 10-1 нам не удалось толком продемонстрировать, чем же обработка исключений Java лучше обыкновенной конструкции if-then-else. В нем на один блок try приходился один блок catch. На самом деле блоков catch, ассоциированных с блоком try, может быть сколько угодно. Это значит, что операторы блока try исполняются при нормальном ходе выполнения программы.

Программисту, изучающему ваш код, не составит особого труда понять, что в нем пытаются сделать. Любой программист по достоинству оценит возможность отделить код, выполняющийся в нормальной ситуации, от кода, выполняющегося при возникновении ошибок.

Давайте рассмотрим пример, демонстрирующий эти достоинства. Раньше мы просто загружали единственный графический файл в апплет. Теперь усложним задачу: предположим, что Web-дизайнер хочет загружать несколько файлов, указывая время, в течение которого каждый из них должен отображаться на экране, - в общем, создать анимацию. Если один из указанных файлов недоступен в момент выполнения апплета, он заменяется на специальное изображение, говорящее дизайнеру об отсутствии данного файла на сервере, так же как специальный значок на HTML-странице говорит об отсутствии соответствующего изображения автору страницы. Если время отображения не указано, используется время по умолчанию. Нашу анимационную задачу решает следующее приложение:

```
class anumApplet extends Applet {
Image imgs[];
Integer pauses[];
// другие описания
public void init() {
Vector imgVec = new Vector();
Vector pauseVec = new Vector();
Image deadImage = createDeadImage();
Integer defaultPause = new Integer(100);
int indx = 1;
String imgName = getParam("image"+indx);
while (imgName != null) {
        try {
                URL imgURL = new URL (getDocumentBase, imgName);
                        // здесь может возникнуть MalformedURLException
                        imgVec.add(getImage(imgURL));
                String pauseParam = getParam ("pause" + indx);
                        Integer thisPause(pauseParam);
                        // здесь может возникнуть NumberFormatException
                        pauseVec.add(thisPause);
}
```

```
catch (MalformedURLException e) {
        showStatus(e.getMessage());
        System.out.println(e.getMessage());
        imgVec.add(deadImage);
}
catch (NumberFormatException e) {
                pauseVec.add(defaultPause);
}
        finally {indx = indx + 1;}
}
imgs = new Image[imgVec.size()];
imgVec.copyInto(imgs);
pauseVec.copyInto(pauses);
}
```

До тех пор пока все идет нормально, выполняются все операторы блока try и в заключение - операторы блока finally. Если другой человек заглянет в этот текст, он без труда поймет, что происходит в программе, а заглянув в блоки catch, выяснит, как вы намереваетесь поступить с каждой из возможных ошибок.

Давайте помотрим, насколько затруднилось бы чтение исходного текста в случае, если бы в Java не было механизма обработки исключений. Сейчас мы рассмотрим небольшой текст, являющийся псевдокодом, а не настоящей программой; поскольку некоторые исключения должны обрабатываться в обязательном порядке, следующий фрагмент компилироваться не будет. Для целей нашего исследования мы придумали несколько не существующих на самом деле методов, без которых было бы не обойтись, если бы в Java отсутствовал механизм обработки исключений. Во-первых, это метод validURL (возвращает boolean), равный true, если URL корректен. Во-вторых, это validIntString, возвращающий true, если String представляет собой целое число:

```
// Java-псевдокод
// Скомпилировать его невозможно!
public void init() {
Vector imgVec = new Vector();
Vector pauseVec = new Vector();
Image deadImage = createDeadImage();
Integer defaultPause = new Integer(100);
int indx = 1;
String imgName = getParam("image"+indx);
while (imgName != null) {
        if (validURL(imgName)) {
                URL imgURL = new URL(imgName);
                imgVec.add(getImage(imgURL));
                String pauseParam("pause"+indx);
                if (validIntString(pauseParam)) {
                        Integer thisPause(pauseParam);
                pauseVec.add(thisPause);
}
                else pauseVec.add(defaultPause);
else {
        showStatus(imgName+" invalid");
System.out.println(imgName+" invalid");
}
indx ++;}
}
```

Обратите внимание, как сложно разобраться в происходящем в программе, когда не случается никаких ошибок. Кроме того, здесь нет никакого блока try, куда можно было бы заглянуть в случае необходимости. К тому же фрагмент текста программы, реализующий те же функции, что и рассмотренный ранее, вырос в размерах.

СОВЕТ Мы уже говорили о том, что существует возможность обрабатывать суперкласс исключений. Если вы попытаетесь поставить два блока catch, один из которых захватывает суперкласс, а другой - собственно исключение, компилятор заявит, что второй блок catch

недостижим (unreachable) в процессе выполнения программы, если только он не захватывает исключение, генерируемое оболочкой времени выполнения.

Генерация исключений

Мы знаем, как пользоваться методами, генерирующими исключения. Теперь можно приступить к написанию собственных методов этого типа. Используемый при этом синтаксис вполне очевиден. Давайте рассмотрим следующий пример:

```
class sampleConvert {
public static byte intToByte(int I) throws Ecxeption {
if (I > 127 || I < -128) throw Exception();
// здесь не нужен оператор else
// управление автоматически передается блоку catch
return (byte)I;
}
// другие методы
}
```

Если мы генерируем исключение, не относящееся к ошибкам или исключениям времени выполнения, его нужно объявить в соответствующем разделе метода. Вскоре мы увидим, что метод способен генерировать более одного исключения. Для того чтобы показать, что метод генерирует исключение, мы используем ключевое слово throw. Что если метод, вызвавший наш метод, не способен обработать исключение самостоятельно? Он может перехватить его и отправить дальше, к методу, из которого сам был вызван. После оператора throw обязательно должен идти экземпляр подкласса Throwable. Разумеется, мы можем использовать любой доступный для нас конструктор.

Сейчас мы приступим к разработке класса, в процессе которой получим хороший урок на тему "как обрабатывать исключения". В главе 4 было показано, что численные примитивы всегда могут приводиться друг к другу. Даже если мы преобразуем тип int к типу byte и выходим за границы диапазона возможных значений типа byte, преобразование не приведет к ошибке - мы просто получим бессмысленный результат. Но что если нам нужно знать, произошла ли ошибка в процессе преобразования или нет? Лучший способ достичь этого - создать специальный класс Convert, содержимым которого являлись бы методы, предназначенные для проверки такой ситуации. Если преобразование типов приводит к потере информации, метод бы генерировал исключение. Для каждого вида преобразования мы разработаем соответствующий метод, который будет генерировать исключение, как только преобразование типов будет приводить к потере информации. Вот один из таких методов:

```
public static byte toByte(int I) throws Exception {
String mesg = "toByte: ";
if (I > 127) {
        mesg = mesg + I + - is less than byte range";
        throw new Exception(mesg);}
if (I > 128) {
        mesg = mesg + I + - is less than byte range";
        throw new Exception(mesg);}
return (byte)I;}
```

С одной стороны, наш метод - неплохой образец того, как надо обрабатывать исключения. Если бы мы не могли генерировать исключения, у нас не было бы способа сообщить о том, что значение interger вышло за допустимый диапазон.

Однако наш простейший пример абсолютно непригоден для практического применения. Если что-то идет не так, как хотелось бы, наши программы должны знать, почему. Иначе люди, которые, возможно, будут изучать ваш код в дальнейшем, не поймут, что за ошибка произошла в программе. Кроме того, как вы помните, на каждый тип исключения должен приходиться единственный оператор catch. Это значит, что кто-либо, использующий ваш метод, может оказаться не в состоянии отличить один экземпляр исключения от экземпляра исключения другого метода.

Это проблема семантического характера, поскольку в экземпляре класса Exception немного смысла. Нам нужно написать собственные подклассы класса Exception и снабдить их значимыми именами. Так как каждый метод в нашем классе-преобразователе должен генерировать свое собственное исключение, мы разработаем иерархию исключений, как показано на рис. 10-6.

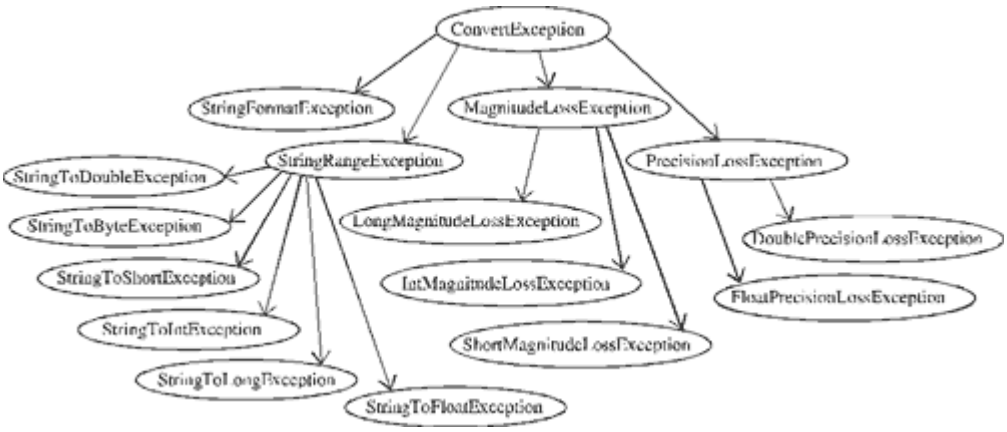

Рис. 10.6.

Написать программу, следующую этой иерархии, очень просто. Все, что необходимо сделать, - это добавить конструктор для различных типов исключений, входящих в части иерархий MagnitudeLoss и PrecisionLoss. Помните, что, как и с любыми классами, каждое исключение должно помещаться в отдельный файл:

Пример 10-1а. Исключения преобразования типов.

```
public class ConvertException extends Exception {
        public ConvertException(String S) {
                super("Couldn't convert: "+S);}
}
public class StringFormatException extends ConvertException {
        public StringFormatException(String S) {
                super("Bad String Format: "+S);}
}
public class MagnitudeLossException extends ConvertException {
        public MagnitudeLossException(String S) {
                super("magnitude loss: - + S);}
}
public class MagnitudeLossException(double d) {
        this(""+d);}
}
public class LongMagnitudeLossException extends MagnitudeLossException {
        public LongMagnitudeLossException(long l){
                super("long "+l);}
}
public class IntMagnitudeLossException extends MagnitudeLossException {
        public IntMagnitudeLossException(int i) {
                super("int "+i);}
                }
public class ShortMagnitudeLossException extends MagnitudeLossException {
        public ShortMagnitudeLossException(short s) {
                super("short - + s);}
        }
public class PrecisionLossException extends ConvertException {
        public PrecisionLossException(String S) {
                super("Precision loss: "+S);}
}
public class DoublePrecisionLossException extends PrecisionLossException {
        public DoublePrecisionLossException(double d) {
                super("double: - + d);}
}
public class FloatPrecisionLossException extends PrecisionLossException {
        public FloatPrecisionLossException(float f) {
                super("float: - + f);}
}
```

Создание иерархии исключений совершенно отличается от создания обычной объектно-ориентированной иерархии. Подклассы Exception выполняют мало полезной работы - только передают информацию о сбое в программе. Большинство исключений, определенных в API, не определяют никаких дополнительных методов или конструкторов, кроме принадлежащих классу Throwable.

Самое важное, что касается подклассов Exception, - это их имена. Глядя на операторы catch, программист должен понимать, о какой именно ошибке идет речь. Пример удачного имени исключения - MalformedURLException ("неправильно сформированный URL"). Для того чтобы понять, в каком случае оно генерируется, нет нужды заглядывать в документацию.

Кроме того, исключения должны быть правильно сгруппированы. Вы, вероятно, помните, как мы запускали javadoc в главе 2. Поскольку исключения - всего лишь классы, javadoc можно запускать и с ними. Ваш подкласс класса Exception будет иметь собственную Web-страницу, на которой разместится ссылка на Web-страницу класса Exception, и те, кто будет пользоваться вашим кодом в дальнейшем, получат в руки исчерпывающее гипертекстовое руководство-путеводитель по пользовательским исключениям.

Теперь, построив иерархию исключений, мы можем приступить к разработке класса Convert. Полный текст класса Convert находится на приложенном к книге диске CD-ROM. Здесь мы приведем лишь методы, преобразующие в тип byte, а также еще один метод, используемый методом toByte(String S).

Пример 10-1b. Класс Convert.

```
public class Convert {
public static byte toByte(short s) throws ShortMagnitudeLossException {
        byte b=(byte)s;
        if (b==s) return b;
        else throw(new ShortMagnitudeLossException(s));}
public static byte toByte(int i) throws IntMagnitudeLossException{
        byte b=(byte)i;
        if(i==b) return b;
        else throw (new IntMagnitudeLossException(i));}
public static byte toByte(long l) throws LongMagnitudeLossException{
        byte b=(byte)l;
        if(l==b) return b;
        else throw (new LongMagnitudeLossException(l));}
public static byte toByte(String S) throws
        StringFormatException, StringToByteException {
        try {
        double d=toDouble(S);
        byte b=(byte)d;
        if (b==d) return b;
        else throw (new StringToByteException(d));}
        catch (StringFormatException e) {
                throw (e);}
        }
public static byte toByte(float f) throws MagnitudeLossException,
FloatPrecisionLossException {
        if(f>127 || f< -128)
                throw (new MagnitudeLossException(f));
        byte b=(byte)f;
        if (b==f) return b;
        else throw (new FloatPrecisionLossException(f));}
public static byte toByte(double d) throws MagnitudeLossException,
DoublePrecisionLossException {
        if(d>127 || d< -128)
                throw (new MagnitudeLossException(d));
        byte b=(byte)d;
        if (b==d) return b;
        else throw (new DoublePrecisionLossException(d));}
//*** toDouble
public static double toDouble(String S) throws StringFormatException {
        S=S.trim();
        try {
        double d=Double.valueOf(S).doubleValue();
        return d;}
```

```
        catch (NumberFormatException e) {
                throw (new StringFormatException(S));}
}
```

Информация об объектах при выполнении программы

Поскольку Java - язык динамический, мы можем получать информацию об объектах "на лету". Для этого используется класс Class - java.lang.Class (рис. 10-7). Этот класс необходим для того, чтобы можно было выяснить, что за тип у объекта, с которым мы работаем, какие интерфейсы в нем применены и ряд других характеристик.

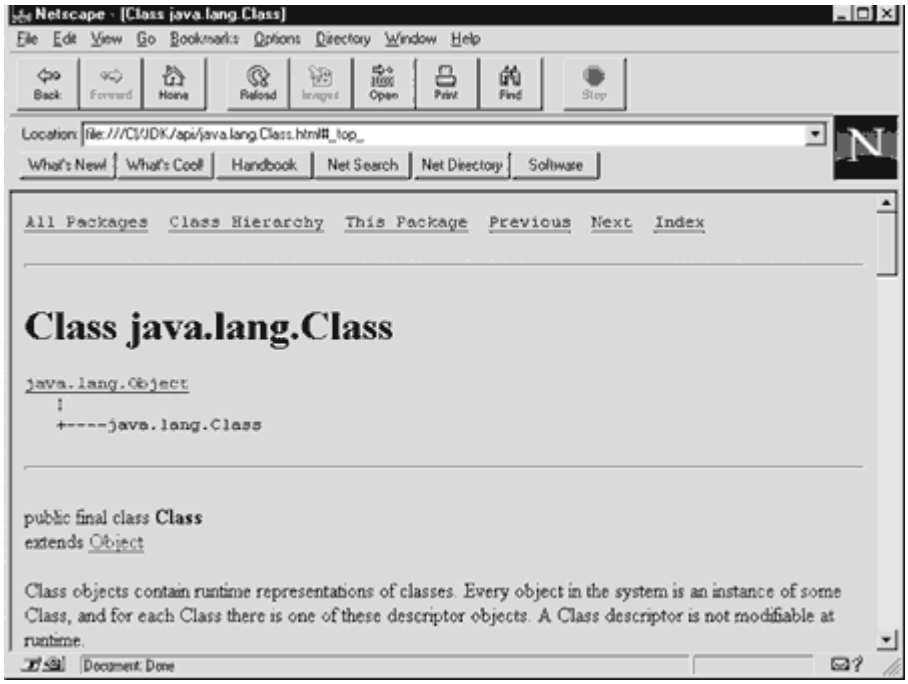

Рис. 10.7.

Он редко применяется в небольших системах, когда мы точно знаем, экземпляры какого типа там попадаются. В больших и сложных системах мы можем использовать этот класс для того, чтобы разбирать, с каким экземпляром имеем дело. В табл. 10-3 перечислены методы, определенные в классе Class. Возвращая объект Class, они однозначно описывают определенный класс или интерфейс Java.

Таблица 10-3. Методы класса Class

Метод	Назначение и генерируемые исключения
static Class forName (String className)	Получив className,, данный метод возвращает экземпляр Class,, свойственный данному классу. Генерируемое исключение: ClassNotFoundException.
String getName()	Возвращает имя данного класса,, связанное с данным экземпляром. Не генерирует исключений.
String toString()	См. выше. Кроме того,, добавляется интерфейс,, если это - интерфейс,, или класс,, если это - класс. Не генерирует исключений.
boolean isInterface()	Возвращает true,, если объект является интерфейсом. Не генерирует исключений.
Class[] getInterfaces()	Возвращает экземпляры Class,, соответствующие интерфейсам,, которыми владеет соответствующий класс или массив нулевой длины,, если у класса нет ни одного интерфейса. Не генерирует исключений.
Class getSuperclass()	Возвращает экземпляр Class суперкласса. Не генерирует исключений.
Object newInstance()	Создает новый экземпляр соответствующего класса. Генерирует исключения: InstantiationException,, IllegalAccessException.
ClassLoader getClassLoader()	Возвращает загрузчик класса. Не генерирует исключений.

Метод newInstance используется, если нужно создавать экземпляры класса "на лету". Экземпляр создается следующим образом (предполагается, что S - имя существующего подкласса Path):

```
try {
        Path p = Class.forName(S).newInstance();
}
catch (InstantiationException e) {
        System.out.println(S+" is not a valid subclass of "+Path);
}
catch (IllegalAccessException e) {
        System.out.println(S+" is not allowed to be accessed");
}
catch (ClassNotFoundException e) {
        System.out.prinln(S+" wasn't found");
}
```

Что дальше?

Мы дали краткое введение в принципы построения программ и постарались показать возможности языка Java, связанные с написанием повторно используемого кода и объектной ориентации. Для того чтобы сделать программирование на Java еще мощнее, мы показали способы повышения устойчивости кода. В следующей главе будут рассмотрены способы построения многопоточных приложений.

Глава 11

Многопотоковость

Мы уже кратко говорили об использовании потоков в языке Java. Java обрабатывает потоки очень изящным способом: это первый популярный язык, который включает потоки в спецификацию языка. Java, как мощный сетевой язык, должен был с самого начала уметь выполнять многопотоковые операции. Включение потоков делает Java уникальным языком разработки программ для Интернет.

Почему потоки так полезны? Вообразите, что вы создаете программу электронной таблицы и хотите, чтобы результаты ввода пользователя обрабатывались в фоновом режиме. В нормальной, однопотоковой программе, чтобы получить такой результат, вы должны были бы использовать некоторые довольно запутанные приемы - например, цикл, который проверяет ввод с клавиатуры и, если ввода нет, проделывает некоторые вычисления. Это решение не слишком изящно, но оно могло бы работать.

Но ваше решение становится все менее четким и ясным, когда позже, при разработке проекта, кто-то запрашивает фоновый режим для подпрограммы печати, а еще кто-то просит об автосохранении. Теперь вам приходится все больше возиться с этими дополнительными рутинными операциями, которые нужно выполнять между нажатиями клавиш. Ваш цикл ввода с клавиатуры быстро становится запутанным и беспорядочным, и все это потому, что у вас есть только один поток выполнения.

Конечно, можно было бы использовать чью-нибудь библиотеку, которая обеспечивает поддержку множества одновременно выполняемых потоков. Это хотя и возможно, но далеко не оптимально. Всякий раз при изменениях в рабочем проекте вам, возможно, придется сталкиваться со все новыми библиотеками поддержки многопотоковости и изучать новые API. Разработчики Java поняли это и выполнили поддержку потоков в спецификации языка. Используя потоки стандарта Java, мы можем решить наши проблемы с программой электронной таблицы намного проще. Каждая новая задача выполняется как отдельный поток. Нам уже не нужно корпеть над каждой новой особенностью. Поддержка потоков в языке помогает и созданию встроенного сборщика мусора.

Эта глава сосредоточена на том, как Java осуществляет поддержку потоков, и на проблемах программ, использующих множество потоков. Если вы еще мало работали с многопотоковостью, эта глава, кроме того, может изменить ваши представления о программировании вообще. Переход от одной большой программы к программе, составленной из маленьких потоков, имеет радикальное значение. На этом пути встречаются западни, которых надо избегать, и концепции, которые надо осмыслить. Так давайте погрузимся в то, как Java создает и выполняет множественные потоки.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Создание потоков при помощи класса Thread

Создание нового потока в Java - простейшая операция. Все, что вы должны сделать, - это расширить класс java.lang.Thread и заменить метод run. Каждый экземпляр этого нового класса будет выполнен как отдельный поток. Всего несколькими строками Java-кода вы можете создавать программы со многими потоками выполнения. Если вы когда-либо пытались сфабриковать фальшивые потоки, вы оцените простоту реализации потоков в Java.

В качестве первого упражнения давайте создадим поток outputThread для вывода некоторого текста. Этот поток отображает три числа и затем завершается.

Пример 11-1. Простой поток.

```
class outputThread extends Thread {
```

```
        outputThread(String name) {
                super(name);
}
        public void run() {
                for(int i=0; i < 3; i++) {
                        System.out.println(getName());
                        Thread.yield();
                 }
         }
}
class runThreads {
        public static void main(String argv[]) {
                outputThread t1 = new outputThread("Thread 1");
                outputThread t2 = new outputThread("Thread 2");
                t1.start();
                t2.start();
        }
}
```

На выходе код генерирует следующее:

```
Thread I
Thread 2
Thread I
Thread 2
Thread I
Thread 2
```

Обратите внимание, что в этой программе мы создаем два потока с помощью двух экземпляров класса outputThread. Затем мы вызываем метод start для каждого потока. Этот метод создает новый поток и затем вызывает наш замененный метод run. Вы уже создали программу с несколькими потоками! И это вовсе не так уж страшно или сложно - фактически в этой книге вы уже использовали потоки.

Помните апплеты для нашей Web-страницы? Мы расширили java.applet.Applet. Правда, это только один поток. Но каждый апплет на странице сделан при помощи одного или нескольких потоков; кроме того, оболочка времени выполнения Java создает некоторое количество потоков для себя. Вы уже знакомы с потоком сборки мусора, который освобождает неиспользуемую память. Как вы можете видеть, потоки - неотъемлемая часть Java.

Различия вывода

Некоторые читатели могут быть удивлены, что вывод на их компьютере несколько отличается от приведенного выше примера. Если вы видите весь вывод первого потока, а затем вывод второго потока, у вас многозадачная система, которая не выполняет квантование времени. Мы обсудим это в разделе "Планирование потоков" ниже в этой главе.

Создание потоков при помощи интерфейса Runnable

Что если бы мы не захотели расширять класс Thread в примере, показанном выше? Возможно, у нас уже есть класс, функциональные возможности которого нас вполне устраивают, и мы только хотим, чтобы он выполнялся как отдельный поток. Ответ прост: используйте интерфейс Runnable. Интерфейсы Java мы обсуждали в главе 3, "Объектная ориентация в Java". Интерфейсы дают хороший способ определить набор стандартных функциональных возможностей для выполняемого класса. Интерфейс Runnable реализует один метод - run, который очень похож на работу класса Thread.

Представьте, что у нас есть класс outputClass, который мы хотим переделать в поток. Все, что мы должны сделать, - это реализовать интерфейс Runnable, создавая разделяемый метод run.

Пример 11-2. Использование интерфейса Runnable.

```
class outputClass implements Runnable {
        String name;
        outputClass(String s) {
                name = s;
```

```
        }
        public void run() {
                for(int i=0; i < 3; i++) {
                        System.out.println(name);
                        Thread.yield();
                }
        }
}
class runThreads {
        public static void main(String argv[]) {
                outputClass out1 = new outputClass("Thread 1");
                outputClass out2 = new outputClass("Thread 2");
                Thread t1 = new Thread(out1);
                Thread t2 = new Thread(out2);
                t1.start();
                t2.start();
        }
}
```

Этот пример по функциям эквивалентен примеру 11-1, но выполнен по-другому. Здесь мы создаем два экземпляра класса outputClass. Это могли бы быть любые классы, но они должны реализовывать интерфейс Runnable. Затем мы создаем два новых потока и передаем их созданным экземплярам outputClass. После этого мы начинаем выполнение потоков как обычно.

Различие между двумя примерами - в обращении конструктора к классу Thread. В первом примере мы вызвали конструктор Thread(String), во втором - Thread(Runnable). Мы могли бы также вызвать конструктор Thread(Runnable, String). Класс Thread имеет следующие конструкторы:

- Thread()
- Thread (String)
- Thread(Runnable)
- Thread (String, Runnable)
- Thread (ThreadGroup, String)
- Thread (ThreadGroup, Runnable)
- Thread(ThreadGroup, Runnable, String)

Мы обсудим ThreadGroup немного позже. Сейчас давайте исследуем, как управлять потоками. Создание потоков - только часть дела; главное - научиться управлять ими после того, как они порождены.

Управление потоками

До сих пор мы выполняли потоки, которые заканчиваются сами собой. Они выполняют задачу и завершаются. А как вы остановите поток, когда Java-программа заканчивает свою работу? Это важный вопрос, и нам понадобятся некоторые общие знания, чтобы на него ответить.

Возвратимся к программе, состоящей только из одного потока. У нас был основной цикл, в котором выполнялся некоторый код. Когда программа должна была закончить работу, мы выходили из цикла и завершали программу. В Java мы получаем то же самое, но с небольшим отличием. Программа не заканчивается до окончания работы всех потоков. Так, если у нас есть поток, который никогда не будет закончен, то и наша программа никогда не завершится.

Каждый поток может пребывать в одном из четырех состояний: создание, выполнение, ожидание или завершение. Только что созданный поток еще не выполняется - он ждет запуска. После этого поток может быть запущен методом start или остановлен, если его переведут в состояние завершения. Потоки в этом состоянии закончили выполнение, то есть это последнее состояние, в котором они могут пребывать. Как только поток достигает этого состояния, он не может быть снова запущен. И когда все потоки в виртуальной машине Java придут в состояние завершения, программа закончит свою работу.

Все запущенные на текущее время потоки находятся в состоянии выполнения. Процессор разделяет время между потоками (как именно Java выделяет, распределяет время процессора, мы обсудим ниже).

В этом состоянии каждый поток доступен для выполнения, но в любой момент времени на процессоре системы может действительно выполняться только одна программа.

Потоки попадают в состояние ожидания, если их выполнение было прервано. Поток может

быть прерван несколькими способами. Он может быть приостановлен для ожидания некоторых ресурсов системы или по требованию о приостановке. Из этого состояния поток может быть возвращен к состоянию выполнения или переведен в выполненное состояние методом stop. В табл. 11-1 приведены методы, управляющие выполнением потоков.

Таблица 11-1. Методы управления потоками

Метод	Описание	Исходное состояние	Новое состояние
start()	Начинает выполнение потока	Создание	Выполнение
stop()	Заканчивает выполнение потока	Создание, выполнение	Завершение
sleep(long)	Пауза на некоторое число миллисекунд	Выполнение	Ожидание
sleep(long,int)	Пауза на некоторое число наносекунд	Выполнение	Ожидание
suspend()	Приостанавливает выполнение	Выполнение	Ожидание
resume()	Продолжает выполнение	Ожидание	Выполнение
yield()	Явно уступает управление	Выполнение	Выполнение

Методы, приведенные в табл. 11-1, не всегда доступны; большинство из них работает, когда поток выполняется. Если вы используете какой-то метод в несоответствующем состоянии - например, если вы попробуете приостановить завершенный поток, - будет сгенерировано исключение IllegalThreadStateException. Во время разработки вы должны знать, в каком состоянии находится поток. Если вам нужна программа для определения состояний потоков, используйте метод isAlive: результат true означает, что поток выполняется или ожидает. Никакого способа определить разницу между выполнением и ожиданием нет.

Вы все ближе подходите к использованию потоков: вы уже можете создавать потоки и управлять их выполнением. О чем мы еще не говорили - это о том, как потоки взаимодействует друг с другом.

Планирование потоков

Порядок, в котором потоки будут выполняться, и количество времени, которое они получат от процессора, - главные вопросы для разработчика. Каждый поток должен разделять процессорное время с другими. Мы не хотим, чтобы один поток монополизировал целую систему. Планирование потоков близко связано с двумя понятиями: выгрузка и квантование времени. Давайте исследуем каждое понятие подробнее.

Система, которая имеет дело со множеством выполняющихся потоков, может быть или приоритетная, или неприоритетная. Приоритетные системы гарантируют, что в любое время будет выполняться поток с самым высоким приоритетом. Всем потокам в системе приписывается приоритет. Переменная класса Thread.NORM_Priority содержит значение приоритета потока по умолчанию. В классе Thread есть методы setPriority и getPriority для установки и определения приоритета. Используя метод setPriority, можно изменять важность потока для виртуальной машины Java. Этот метод в качестве аргумента получает некоторое целое число в диапазоне допустимых значений, заданном двумя переменными класса Thread.MIN_PRIORITY и Thread.MAX_PRIORITY.

Виртуальная машина Java является приоритетной, то есть выполняться всегда будет поток с самым высоким приоритетом. Давайте изменим приоритет потока в Java-машине на самый высокий. После этого поток получит все процессорное время, пока не закончится или не будет переведен в состояние ожидания - возможно, командой sleep. Но что при этом произойдет с потоками, имеющими тот же самый приоритет? Давайте рассмотрим пример, чтобы увидеть поведение Java.

Пример 11-3. Планирование.

```
        class outputThread extends Thread {
        outputThread(String name) {
                super(name);
        }
        public void run() {
                for(int i=0; i < 3; i++) {
                        System.out.println(getName());
                }
        }
}
class runThreads {
        public static void main(String argv[]) {
```

```
                outputThread t1 = new outputThread("Thread 1");
                outputThread t2 = new outputThread("Thread 2");
                outputThread t3 = new outputThread("Thread 3");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

Создавая этот код, мы вообще-то ожидаем, что каждый поток того же самого приоритета должен выполняться в течение некоторого времени и затем уступать управление другому потоку. Как можно увидеть из вывода этого примера, выполненного нашей программой на машине с Windows 95, мы получаем то, что и ожидали:

```
Thread 1
Thread 2
Thread 3
Thread 1
Thread 2
Thread 3
Thread 1
Thread 2
Thread 3
```

Но посмотрим, что мы получим, выполнив тот же самый код на рабочей станции Sun:

```
Thread I
Thread I
Thread I
Thread 2
Thread 2
Thread 2
Thread 3
Thread 3
Thread 3
```

Результаты совершенно различны! Что случилось с платформонезависимостью? Добро пожаловать в теневой мир многопотоковости. Не все машины созданы одинаково, и операционная система воздействует на порядок выполнения потоков. Различие заключается в концепции, называемой квантованием времени.

Как выполняются потоки одного и того же приоритета, в спецификации Java не описано. Казалось бы, что потоки должны использовать процессор совместно, но это не всегда так. Порядок их выполнения определен основной операционной системой и аппаратными средствами. Очевидно, на системах с одним процессором мы не можем ожидать одновременного выполнения больше чем одного потока. Операционная система обслуживает потоки при помощи планировщика, который и определяет порядок выполнения.

Давайте поближе взглянем на то, что же случилось, когда мы запустили нашу программу на машине Sun. Первый поток выполнялся до завершения, потом так делал поток 2 и, наконец, поток 3. Но если первый поток никогда не закончится, потоки 2 и 3 никогда не будут выполняться. Конечно, это вызывает некоторые проблемы. Поток с самым высоким приоритетом будет выполняться всегда, в то время как потоки более низкого приоритета, в зависимости от операционной системы, возможно, выполниться не смогут. Если нам нужна некоторая упорядоченость или даже просто последовательное выполнение потоков, мы должны будем проделать некоторую работу.

Метод yield предлагает простой обходной путь решения проблемы квантования времени. Если используется метод yield, поток добровольно уступит процессор, что даст другим потокам возможность тоже получить время процессора. Добровольная передача управления особенно важна в длинных циклах. Давайте перепишем предыдущую программу с использованием метода yield.

Пример 11-4. Метод yield.

```
class outputThread extends Thread {
        outputThread(String name) {
                super(name);
```

```
        }
        public void run() {
                for(int i=0; i < 3; i++) {
                        System.out.println(getName());
                        Thread.yield();
                }
        }
}
class runThreads {
        public static void main(String argv[]) {
                outputThread t1 = new outputThread("Thread 1");
                outputThread t2 = new outputThread("Thread 2");
                outputThread t3 = new outputThread("Thread 3");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

Этот вариант потребовал только одного изменения. Мы добавили команду yield внутрь основного цикла потока. Хотя это дает нам то, что мы хотим, у такого решения есть несколько недостатков. Метод yield затрачивает ресурсы системы. В случае одного потока использование yield - это пустая трата времени. Не забудьте также, что некоторые системы уже обеспечивают для нас разделение времени. Другой недостаток состоит в том, что мы должны явно уступить управление, - но как узнать, где и как часто это делать?

Так как некоторые системы выполняют квантование времени, а другие этого не делают, мы могли бы разработать простой тест, чтобы определить способности системы. Следующий код определяет, действительно ли система обеспечивает квантование времени.

Пример 11-5a. Тестирующий поток testThread.

```
class testThread extends Thread {
        protected int val=0;
        public void run() {
                while(true) {
                        val++;
                }
        }
        public int getVal() {
                return val;
        }
}
```

Этот первый класс - только поток теста. Он просто увеличивает некоторую переменную. В системе с квантованием времени все текущие потоки должны иметь равные значения. Системы без квантования времени, вероятно, закончат выполнение с одним потоком, получающим значительно большее количество времени, чем другие. Следующий код определяет, выполняет ли система квантование времени.

Пример 11-5b. Определение наличия квантования времени.

```
class isFair extends Thread {
        boolean fair=false;
        boolean determined=false;
        public boolean isFair() {
                if (determined) return fair;
                start();
                while(!determined) {
                        // ожидание, пока значение равно determined
                        try {
                                sleep(1500);
                        }
                        catch (InterruptedException e) {
                        }
                }
```

```
                return fair;
}
public void run() {
        testThread t1 = new testThread();
        testThread t2 = new testThread();
        setPriority(MAX_PRIORITY);
        t1.start();
        t2.start();
        try {
                sleep(500);
        }
        catch (InterruptedException e) {
        }
        t1.stop();
        t2.stop();
        if (t1.getVal() > 2 * t2.getVal()) {
                fair = false;
        }
        else {
                fair = true;
        }
                determined = true;
        }
}
```

Класс isFair - основная часть этого примера. В нем есть метод isFair, который может вызываться, чтобы определить, имеет ли система возможности квантования времени. Несколько миллисекунд он управляет двумя потоками testThreads. Затем он останавливает потоки и проверяет результат. Если один поток получил времени вдвое больше, чем другой, мы назовем это разделение несправедливым. Вообще, этот пример действует по принципу "все или ничего"; один поток, обычно thread1, получает все процессорное время. Это будет несправедливое разделение времени.

Пример 11-5c. Тест справедливости разделения времени.

```
class test {
        public static void main(String argv[]) {
                isFair tThread = new isFair();
                if (tThread.isFair()) {
                        System.out.println("System has time slicing");
                }
                else {
                        System.out.println("System does not time slice");
                }
        }
}
```

Заключительный компонент - собственно программа теста. Вы могли бы использовать что-то подобное в своих программах, чтобы исследовать возможности системы. На системах квантования времени вы можете спокойно продолжать программирование и не волноваться относительно проблем справедливости. На других системах вы должны рассчитывать только на себя, чтобы гарантировать, что ваши потоки получат нужное им процессорное время.

Поместить метод yeld в своем коде нетрудно, но можно просто забыть это сделать. Есть другие способы решения той же задачи. Поток с самым высоким приоритетом в системе всегда выполняется, если не находится в режиме ожидания. Мы можем использовать это, чтобы подражать квантованию времени. В примере 11-6 создается поток slicerThread, чей приоритет установлен в MAX_PRIORITY. Его задача состоит только в том, чтобы ничего не делать. Когда он бездействует, ему больше не требуется планирование. Это означает, что наши потоки с нормальным приоритетом получат возможность выполниться. Каждый раз, когда slicerThread получает управление и засыпает, планировщик выбирает новый поток для выполнения. Он выбирает их циклически, так что каждый нормальный поток получит возможность выполниться, как показано в следующем примере.

Пример 11-6. Квантование времени.

```
        class outputThread extends Thread {
```

```
                outputThread(String name) {
                        super(name);
                }
        public void run() {
                for(int i=0; i < 3; i++) {
                        System.out.println(getName());
                        Thread.yield();
                }
        }
}
class slicerThread extends Thread {
        slicerThread() {
                setPriority(Thread.MAX_PRIORITY);
        }
                public void run() {
                        while(true) {
                                try {
                                Thread.sleep(10);
                                }
                                catch (InterruptedException ignore) {
                        }
                }
        }
}
class runThreads {
                public static void main(String argv[]) {
                        slicerThread ts = new slicerThread();
                        outputThread t1 = new outputThread("Thread 1");
                        outputThread t2 = new outputThread("Thread 2");
                        outputThread t3 = new outputThread("Thread 3");
                        t1.start();
                        t2.start();
                t3.start();
                ts.start();
        }
}
```

Относительно этих примеров нужно сделать несколько комментариев. Для начала скажем, что это решение некрасиво. Хотя мы и выполнили поставленную задачу, но расплачиваемся за это ресурсами системы. Можно попытаться использовать меньшее количество ресурсов, сделав бездействие более длинным. Только при этом каждый поток будет выполняться соответственно дольше. Периоды бездействия нужно сохранить довольно короткими, иначе вывод не будет плавным. Представьте поток, по одному выдающий символы на экран. Если символы будут перемежаться значительными паузами, мы получим не самый идеальный вариант программы.

С точки зрения эффективности лучше поместить в коде оператор yeld. Ни один метод не дает идеального решения, но что-то требуется, чтобы потоки разделяли процессорное время. Не думайте, что только потому, что ваша система выполняет квантование времени, остальные тоже это делают. Java - один из наиболее переносимых языков, но вы можете сделать его менее переносимым, игнорируя проблемы планирования.

Вы можете спросить, почему разработчики Java поступили так. Самый лучший ответ - они должны были сделать это. Планирование лучше всего оставить операционной системе. Если Java возьмется планировать потоки, это будет расточительно и очень неэффективно. В большинстве современных процессоров есть поддержка многозадачного режима на уровне аппаратных средств. Независимо от того, насколько хорош код программы, он не может сравниться с аппаратной реализацией. Под Solaris фирма Sun обеспечивает полную поддержку приоритетных потоков, и когда-нибудь Java сможет воспользоваться этим преимуществом основной OS.

Группирование потоков

Управление несколькими потоками не вызывает никаких реальных проблем, но предположите, что вам нужно следить за сотнями потоков. Необходимость циклически проверять каждый поток была бы утомительна. Вы могли бы столкнуться с этой ситуацией при написании сервера Интернет - некоторые программы обрабатывают тысячи параллельных сеансов. Давайте предположим, что вам нужно произвести некоторое сопровождение системы, так что вы должны

закончить все сеансы. У вас есть два выбора: создать цикл, проверяющий каждый поток, и останавливать потоки явно или использовать группы потоков.

Потоки группируются иерархически. Каждая группа может содержать неограниченное число потоков. Вы можете обращаться к каждому потоку и выполнять операции типа suspend и stop с целыми группами потоков. Давайте создадим несколько групп потоков:

```
ThreadGroup parent = new ThreadGroup( "parent");
ThreadGroup child = new ThreadGroup ( parent, "child");
```

Этот фрагмент кода показывает два пути, которыми может быть создана группа потоков. Первый метод создает ThreadGroup с некоторым именем. Второй метод создает ThreadGroup с родительской группой и некоторым именем. Родительская группа выбирает потоки, которыми она может командовать.

Создав объекты ThreadGroup, мы можем добавлять к ним потоки. Помните конструкторы, имеющие дело с ThreadGroup? Мы можем использовать их, чтобы добавить потоки к группе. Фактически это единственный механизм, который мы можем использовать, - группа потоков не может быть изменена после того, как была создана.

```
Thread t1 = new Thread(parent);
Thread t2 = new Thread ( child, "t2");
```

Теперь, когда у нас есть некоторые потоки в различных группах, что мы с ними можем делать? Наиболее полезные методы, используемые с группами потоков, - suspend, resume и stop. Каждый поток в ThreadGroup будет иметь соответствующий вызываемый метод. ThreadGroup воздействует и на потомков. Используя эти методы, мы можем легко выполнить операции с большим числом потоков.

Группы потоков имеют другие функции, которые, прежде всего, используются виртуальной машиной Java и имеют дело с межпотоковой защитой. В основном Java-машина должна предохранять наши потоки от вмешательства своих собственных потоков. Порожденнный поток в ThreadGroup не может управлять потоком в родительской группе. Вам, вероятно, и не будет нужно такое их поведение. Документации на этот счет в настоящее время мало, так что лучше рассмотреть исходный текст. Это обычно используется реализацией Java-машины для защиты своих потоков от потоков пользователя.

Последняя часть завершит проблему многопотоковости. До сих пор наши потоки вообще не чувствовали других потоков в системе. Мы должны были беспокоиться относительно совместного использования времени, но у потоков не было общих данных. Это академический материал, обычно изучаемый в курсе информатики. Мы можем дать вам краткий обзор, но характер этой книги не позволяет исследовать эту тему с большими подробностями. Мы предлагаем вам провести некоторое самостоятельное исследование. Проверьте книгу на диалоговой обработке запросов или, возможно, на операционных системах. Некоторые темы, которые могут быть вам интересны, - синхронизация, семафоры, взаимная блокировка и условия гонок.

Синхронизация потоков

Давайте на момент представим, что вы имеете докторскую степень в информатике. Вы потратили много лет, изучая сложности программирования. Вы, вероятно, изучали дисциплины диалоговой обработки запросов или параллельного программирования. Вы обучались многозадачному режиму и провели много времени, изучая последние алгоритмы обработки связанных с ним проблем.

Теперь давайте возвратимся в реальный мир. Возможно, у вас докторская степень в программировании или вы волшебник в области параллельного программирования. Если так - можете пропустить этот раздел. Если нет, мы обучим вас основам и, вероятно, ответим на большинство ваших вопросов.

Наличие многих потоков, выполняющихся в программе, может вызывать некоторые проблемы. Если потоки работают независимо, это не создает никакой проблемы, но так случается не всегда. Многопотоковость иногда требует, чтобы один поток связался с другим, обычно через общедоступный ресурс. Это - то место, где и начинаются сложности.

Мониторы: защита общедоступной переменной

Используем простой пример, чтобы проиллюстрировать некоторые из проблем, с которыми мы можем столкнуться при работе с несколькими потоками. Представьте себе вокзал с несколькими путями разных направлений, но у которого есть только один путь, обслуживающий область

погрузки. Где-нибудь по основной линии поезда должны выезжать на путь, обслуживающий область погрузки, чтобы подойти к станции. Предположите, что у нас есть две ветки, ведущие к вокзалу, которые в конечном счете объединяются в одну. В месте, где они объединяются, перед нами стоит проблема - если два поезда попробуют подойти к вокзалу в одно и то же время, они столкнутся. В реальной жизни это столкновение могло бы стоить жизни; в наших программах пересечение потоков, вероятно, означает только возникновение ошибки. Но ошибки раздражают, и даже они могут стоить жизни.

Мы написали простую программу - тренажер поезда, чтобы проиллюстрировать эту ситуацию. У нас есть два поезда, одновременно входящие на станцию. Если никто не вмешается, случится беда. Предложим первые реализации, в которых такого вмешательства нет.

Пример 11-7а. Вокзал: никакого вмешательства.

```
class Display {
        int switch1Loc=15;
        int switch2Loc=10;
        int begin1, begin2;
        int end1, end2;
                public void showLoc(String train, int seq, int len) {
                        if (train.compareTo("train1") == 0) {
                                begin1=seq;
                                end1=seq + len - 1;
                                if (seq > switch1Loc) {
                                System.out.println("train1 near switch @
"+seq);
                                }
                                else if (seq + len > switch2Loc) {
                                System.out.println("train1 @ " + begin1);
                        }
                }
                if (train.compareTo("train2") == 0) {
                        begin2=seq;
                        end2=seq + len - 1;
                        if (seq > switch1Loc) {
                                System.out.println("train2 near switch @
"+seq);
                        }
                        else if (seq + len > switch2Loc) {
                                System.out.println("train2 @ " + begin2);
                        }
                }
                // проверка на пересечение
                if ((begin1 <= switch1Loc && end1 >= switch2Loc) &&
                        (begin2 <= switch1Loc && end2 >= switch2Loc) &&
                        (begin1 <= end2) && (begin1 >= begin2)) {
                        System.out.println("CRASH @ " + seq);
                        System.exit(-1);
                }
        }
}
class train1 extends Thread {
        int seq=20;
        int switch1Loc=15;
        int switch2Loc=15;
        int trainLen=3;
Display display;
train1(String name, Display display) {
        super(name);
        this.display = display;
}
        public void step() {
                seq--;
        }
        public void run() {
                while(seq > 0) {
```

```
                        step();
                        display.showLoc(getName(),seq,trainLen);
                        yield();
                }
                System.out.println(getName() + " finished");
        }
}
```

Класс train1 - наша первая попытка создать объект "поезд". Каждый поезд обслуживается собственным потоком. Выполненный метод вызовет два метода обеспечения step и showLoc. Метод step используется, чтобы переместиться на некоторое расстояние вперед. В нашем примере мы сделали вокзал пять шагов в длину. На любой стороне мы контролируем действия поезда на пять шагов.

Класс Display используется, чтобы отобразить движение наших поездов. Он обнаружит столкновение, если поезда находятся в одном и том же месте. Эту проверку он делает только в то время, когда поезда находятся на станции. Для удобства мы включили код для класса Display.

Пример 11-7b. Код модели вокзала.

```
class testTrains1 {
        public static void main(String argv[]) {
                Display display = new Display();
                train1 t1 = new train1("train1",display);
                train1 t2 = new train1("train2",display);
                t1.start();
                t2.start();
        }
}
```

Код теста прост, он создает два новых поезда и начинает их передвижение. Вот вывод нашей первой попытки:

```
train1 near switch @ 14
train2 near switch @ 14
train1 near switch @ 13
train2 near switch @ 13
train1 near switch @ 12
train2 near switch @ 12
train1 near switch @ 11
train2 near switch @ 11
train1 @ 10
train2 @ 10
CRASH @ 10
```

Заголовки газет кричат о крушении, и программист уволен за некомпетентность. Адвокаты угрожают исками. Компания должна немедленно исправить программы.

Возможно, кто-то вспомнит, что в том курсе по операционным системам, который он когда-то изучал, профессор упоминал что-то о мониторах. Монитор (monitor) - это просто поток, который следит за некоторой переменной. В нашем случае используется переменная-переключатель. В каждый момент времени только один поезд может находиться в этом положении переключателя. Нам нужно затормозить поезд, когда другой поезд проезжает через опасное место.

Java обеспечивает механизм защиты переменных в программе. Проблема сводится к двум потокам, пытающимся выполнить некоторую операцию одновременно. Нам надо, чтобы один поток ждал, пока другой закончится. Операция могла бы включать несколько команд. Эти команды называются критическим разделом. Критический раздел - это та часть кода, которая должна быть защищена, чтобы система не потерпела неудачу. В нашем случае с поездами критический раздел находится в методе step. Поезду нельзя позволять въезжать на путь в то время, когда тот уже занят.

Можно создать класс trainSwitch, который защитит наш переключатель. В нем будут два метода - lock (блокировка) и unlock (разблокировка). Въезжая на опасный путь, поезд должен вызывать lock. Как только поезд оставляет путь, он вызывает unlock. Все другие поезда перед въездом на этот путь должны ждать, пока он не будет разблокирован.

В Java критические разделы кода помечаются ключевым словом synchronyzed. Любой блок кода можно пометить как синхронизированный, но обычно так помечаются методы. Пометка части метода как синхронизированого вообще-то является примером плохого программирования.

Программисты должны видеть области, которые могут вызывать проблемы многопотоковости; если вы не отмечаете метод как синхронизированый, людям понадобится код, чтобы выяснить, что же происходит. По этой причине мы используем ключевое слово synchronyzed только на уровне метода.

Итак, к чему приводит пометка метода как синхронизированного? С каждым классом связан один монитор. Когда вызывается синхронизированный метод, он проверяет монитор. Если класс уже блокирован, подпрограмма вызова будет ждать. Это означает, что в один момент времени только один поток будет находиться в критическом блоке. Давайте посмотрим, как это реализуется.

Пример 11-7c. Класс trainSwitch.

```
class trainSwitch extends Thread {
        protected boolean inUse=false;
        public synchronized void lock() {
                while (inUse) {
                try wait();
                catch (InterruptedException e);
        }
        inUse = true;
}
        public synchronized void unlock() {
                inUse = false;
                notify();
        }
}
```

В классе trainSwitch есть два синхронизированных метода, lock и unlock, которые защищают переменную inUse этого класса. Мы хотим удостовериться, что в любой момент времени только один поток использует переключатель. Состояние переключателя задается в булевской переменной. Когда поезд использует переключатель, inUse устанавливается в true. Любым другим поездам, желающим использовать переключатель, придется ждать, пока булевская переменная не будет сброшена в false.

В примере 11-7b использовалось два новых метода, wait и notify. Они служат двум целям. Первая их задача - разрешать сложную ситуацию. Что случилось, если бы в вышеприведенном примере оператор wait отсутствовал? Подпрограмма блокировки находилась бы в цикле и занимала монитор для класса. И тогда никто бы не смог вызвать метод разблокировки, которому также нужен монитор.

Методы wait и notify используются, чтобы решить эту проблему. Метод wait заставляет поток ожидать некоторого события. Тем временем монитор освобождается. Это позволяет выполнять другие подпрограммы, которым нужен монитор класса. Когда управление возвращается из метода wait, монитор также восстанавливается. Остальная часть критического раздела все еще защищена.

СОВЕТ Методы wait и notify не являются частью класса Thread. Фактически они входят в класс java.lang.Object. Метод wait() ждет неопределенное количество времени, wait(long) ждет некоторое число миллисекунд и wait(long, int) ждет некоторое число миллисекунд плюс некоторое число наносекунд.

Давайте рассмотрим остальную часть кода для нашего примера.

Пример 11-7d. Исправленная модель вокзала.

```
class train2 extends Thread {
        int seq=15;
        int switch1Loc=10;
        int switch2Loc=5;
        int trainLen=3;
        Display display;
        trainSwitch ts;
                train2(String name, Display display, trainSwitch ts) {
                        super(name);
                        this.display = display;
                        this.ts = ts;
```

```
                }
                public void step() {
                        if (seq == switch1Loc + 1) {
                                // заняли пути
                                System.out.println("Locking Switch: " +
getName());
                                ts.lock();
                        }
                        else if (seq + trainLen == switch1Loc) {
                                // освободили пути
                                System.out.println("Unlocking Switch:
"+getName());
                                ts.unlock();
                        }
                seq--;
        }
        public void run() {
                while(seq > 0) {
                        step();
                        display.showLoc(getName(),seq,trainLen);
                        yield();
                }
                System.out.println(getName() + " safe");
        }
}
```

В этом последнем фрагменте кода мы изменили метод step, используя разработанный нами класс trainSwitch. Когда мы попадаем на переключатель, мы запрашиваем его блокировку. Получив блокировку, мы можем въезжать на пути. Когда конец поезда освободил путь, мы можем разблокировать переключатель. Эти операции гарантируют, что только один поезд находится на пути в один момент времени. Ниже приведен вывод исправленной программы. Обратите внимание, что оба поезда проезжают через переключатель без аварии:

```
trainl near switch @ 14
traln2 near switch @ 14
tralnl near switch @ 13
train2 near switch @ 13
train1 near switch @ 12
train2 near switch @ 12
train1 near switch @ 11
train2 near switch @ 11
Locking Switch: trainl
trainl @ 10
Locking Switch: train2
train1 @ 9
traln1 @ 8
train1 @ 7
Unlocking Switch: trainl
trainl @ 6
traln2 @ 10
trainl @ 5
train2 @ 9
trainl @ 4
train2 @ 8
trainl @ 3
train2 @ 7
Unlocking Switch: train2
train2 @ 6
train2 6 5
train2 6 4
train2 @ 3
trainl safe
train2 safe
```

К настоящему времени вы должны понимать основную идею мониторов. В следующих разделах мы рассмотрим другой тип проблем.

Семафоры: защита других общедоступных ресурсов

В предшествующих примерах мы использовали мониторы для защиты переменных. Давайте расширим это понятие до любого общего ресурса. Мы хотели бы защитить любой ресурс системы, который может разделяться несколькими потоками. Примером такого общедоступного ресурса является файл. Два потока, записывающие в один файл сразу, генерируют garbageHaving; ситуация, когда один поток читает файл, в то время как другой поток в него записывает, тоже опасна. Нам нужен некоторый способ защиты целостности нашего файла.

Для защиты файлов можно применить концепции, подобные тем, что мы использовали для защиты переключателя поезда в нашем последнем примере. Этот тип защиты называется исключительной блокировкой. Только один поток может читать или изменять значение в один момент времени. Если мы имеем дело с файлами, особенно с файлами информационных баз данных, мы вообще делаем большее количество операций чтения, чем записи. Мы можем использовать этот факт, чтобы улучшить наш код.

Предположим, что у нас есть база данных служащих. Она содержит информацию типа адресов и номеров телефона. Эти значения иногда изменяются, но гораздо чаще они считываются. Нам нужно защитить данные от искажений, но мы хотим, чтобы программа была настолько эффективна, насколько это возможно. Вместо только исключительных блокировок мы используем блокировку чтения и блокировку записи. Мы можем иметь столько читателей, сколько требуется, но когда кто-то хочет записать что-то в файл, он должен получить исключительную блокировку. Эта концепция носит название семафора (semaphore).

Что нужно для создания этого семафора? Мы хотим позволить потокам свободно читать файл, но предусматриваем, что когда некоторый поток хочет изменить информацию, он должен стать единственным работающим с базой данных потоком.

Наш семафор имеет четыре состояния. В первом, пустом, ни один поток не читает и не записывает информацию. Мы можем принимать запросы и на чтение и на запись, и они могут обслуживаться немедленно. Второе состояние - состояние чтения. Здесь у нас есть некоторое количество потоков, читающих из базы данных. Мы подсчитываем число читателей, и если это число становится равным нулю, мы возвращаемся к пустому состоянию. Запрос на запись должен ждать. Мы можем перейти в состояние записи только из пустого состояния. Все потоки-читатели должны быть завершены, и никакие другие потоки не могут записывать в файл. Любые просьбы о чтении или записи должны ждать завершения этого потока. Любой поток в состоянии ожидания должен ждать завершения потока, находящегося в состоянии записи, чтобы выйти из этого состояния. Когда запрос на запись завершается, мы возвращаемся к пустому состоянию. Когда сообщение notify послано, ждущий поток может обслуживаться. Ниже приведен код, обеспечивающий этот семафор.

Пример 11-8а. Семафор.

```
class Semaphore {
        final static int EMPTY = 0;
        final static int READING = 1;
        final static int WRITING = 2;
        protected int state=EMPTY;
        protected int readCnt=0;
        public synchronized void readLock() {
                if (state == EMPTY) {
                        state = READING;
                }
                else if (state == READING) {
                }
                else if (state == WRITING) {
                        while(state == WRITING) {
                                try wait();
                                catch (InterruptedException e);
                        }
                        state = READING;
                }
                readCnt++;
                return;
        }
        public synchronized void writeLock() {
```

```
                if (state == EMPTY) {
                        state = WRITING;
                }
                else {
                        while(state != EMPTY) {
                                try wait();
                                catch (InterruptedException e);
                        }
                }
        }
        public synchronized void readUnlock() {
                readCnt--;
                if (readCnt == 0) {
                        state = EMPTY;
                        notify();
                }
        }
        public synchronized void writeUnlock() {
                state = EMPTY;
                notify();
        }
}
```

Класс Semaphore реализует семафор, который мы описали. Он использует методы, объявленные с ключевым словом synchronized, и методы wait и notify. Этот класс может использоваться для защиты общедоступного ресурса. Мы можем теперь использовать этот класс, чтобы защитить наш доступ к файлу. Давайте протестируем наш семафор с помощью нескольких читающих и пишущих потоков.

Пример 11-8b. Тест семафора.

```
class process extends Thread {
        String op;
        Semaphore sem;
process(String name, String op, Semaphore sem) {
        super(name);
        this.op = op;
        this.sem = sem;
        start();
}
        public void run() {
                if (op.compareTo("read") == 0) {
                        System.out.println("Trying to get readLock: " +
getName());
                        sem.readLock();
                        System.out.println("Read op: " + getName());
                        try sleep((int)Math.random() * 50);
                        catch (InterruptedException e);
                        System.out.println("Unlocking readLock: " + getName());
                        sem.readUnlock();
                }
                else if (op.compareTo("write") == 0) {
                        System.out.println("Trying to get writeLock: " +
getName());
                        sem.writeLock();
                        System.out.println("Write op: " + getName());
                        try sleep((int)Math.random() * 50);
                        catch (InterruptedException e);
                        System.out.println("Unlocking writeLock: " +
getName());
                        sem.writeUnlock();
                }
        }
}
class testSem {
```

```
        public static void main(String argv[]) {
                Semaphore lock = new Semaphore();
                new process("1", "read", lock);
                new process("2", "read", lock);
                new process("3", "write", lock);
                new process("4", "read", lock);
        }
}
```

Класс testSem запускает четыре потока, которые хотят или читать, или писать в общедоступный файл. Класс Semaphore нужен, чтобы этот множественный доступ не разрушил файл. Вот вывод программы:

```
Trying to get readLock: I
Read op: I
Trying to get readLock: 2
Read op: 2
Trying to get writeLock: 3
Trying to get readLock: 4
Read op: 4
Unlocking readLock: I
Unlocking readLock: 2
Unlocking readLock: 4
Write op: 3
Unlocking writeLock: 3
```

В примере 11-8 у нас три читающих потока и один записывающий. Читающие потоки начинают выполняться перед записывающим, так что тот, прежде чем писать в файл, должен ждать, пока читатели не завершат выполнение. Обратите внимание, что к базе данных могут обращаться сразу несколько читателей, но когда кто-то хочет писать в файл, он должен получить исключительный доступ к файлу. Наш пример иллюстрирует эффективный способ реализации семафора.

Если вы интересуетесь семафорами или вам необходимо выполнить блокировку файла, вам, возможно, понадобится некоторое дальнейшее изучение диалоговой обработки запросов. Наш семафор работает, но ведет к некоторым проблемам. К примеру, он отдает предпочтение потокам-читателям. Поток записи должен ждать завершения всех читающих потоков прежде, чем получить управление. Если во время этого ожидания появятся новые читатели, они могут надолго задержать процесс внесения новой информации в базу данных. Лучше было бы останавливать любые новые запросы на чтение, когда кто-то ожидает записи.

Семафоры и управление ресурсами - сложные темы. Большинству программистов не придется иметь с ними дела. Но вы можете сопоставить эти проблемы с проблемами множественных потоков.

Предотвращение тупиков

Вы можете теперь синхронизировать работу множества потоков и удостовериться, что они сохраняют общедоступные ресурсы. Вы начали экспериментировать со множественными потоками - и вдруг система виснет. Ответа нет. Вы повторно выполняете код, и он работает прекрасно. Двумя неделями позже зависание происходит снова. Вы стали жертвой проблемы тупика (deadlock).

Тупик - понятие, которое легко объяснить и трудно избежать. Вообразите, что вы идете по тротуару. Кто-то находится на вашем пути, так что вы отклоняетесь влево. Стараясь избегнуть вас, встречный прохожий отклоняется вправо. И вы снова стоите лицом к лицу. Теперь вы отклоняетесь вправо, тот идет влево... и так до бесконечности. Такая ситуация и называется тупиком.

Выход из этого затруднительного положения довольно прост для людей - в конце концов кто-то идет в другую сторону. Скажем, вы идете влево и встречный пешеход идет влево - и вы расходитесь. Если бы мы могли вставить в компьютер человеческий мозг, который будет посредником в блокирующих компьютер ситуациях, у нас был бы выход. К сожалению, современное состояние техники пока не достигло такого уровня.

Но как это влияет на программирование на Java? Замените людей на тротуаре потоками. Скажем, у нас есть два защищенных ресурса file1 и file2. Предположим, что для завершения задачи поток нуждается в обоих ресурсах. Первый поток захватывает file1 для себя. В то же самое время второй поток захватывает file2. Первый поток теперь пробует захватить file2, но не

может его получить, приостанавливает свое выполнение и ждет. Второй поток пробует захватить file1 и также ждет. Итак, у нас есть два потока, ождающие ресурсы, которые они никогда не смогут получить. Так как никакой поток не сможет получить оба файла, нужные для завершения работы, они будут ждать неопределенно долго. Это классический случай тупика.

Мы можем обрабатывать тупик двумя способами. Как известно, "болезнь легче предотвратить, чем лечить". Мы должны сделать все, что можно, чтобы избежать этой ситуации. Нам нужно разработать наши потоки так, чтобы знать, как они захватывают спорные ресурсы. Давайте проектировать каждый поток, чтобы он сначала добивался file1. Первый поток захватил бы файл, и второй поток будет ждать. Первый поток может затем получить file2, проделать необходимые операции и освободить оба ресурса. В этом случае решение просто.

Но некоторых тупиков избежать намного сложнее. Иногда настолько сложнее, что лучше иметь дело с тупиком, различными способами пробуя его обнаружить. Мы могли бы завести поток, который бы наблюдал за другими потоками; если ему кажется, что никакого прогресса в выполнении задачи нет, он пробует определить проблему. Фиксирование проблемы вообще вынуждает один или несколько потоков уступить защищенные ресурсы. Недавно освобожденные ресурсы могут позволить другим потокам закончить операцию и уступить ресурсы, что может вывести программу из тупика.

Если все это выглядит сложно, то только потому, что это действительно сложно. Если вы собираетесь экспериментировать с многократными ресурсами, приготовьтесь почитать некоторые учебники. Позвольте предложить один совет: избегайте необходимости блокировать многократные ресурсы. Это то место, откуда начинаются проблемы. Чем большее количество ресурсов требуется каждому потоку, тем сложнее становится проблема.

Переменные volatile

Заключительная тема этой главы - модификатор переменных volatile, с которым большинству программистов сталкиваться не придется. Давайте прочитаем официальное определение языка Java и затем попробуем понять его смысл: "переменная, объявленная с модификатором volatile, как известно, изменяется асинхронно. Компилятор предупрежден, что использовать такие переменные надо более тщательно". Чтобы понять использование наречия "тщательно" в этом контексте, у вас должно быть хорошее понимание виртуальной машины Java. Это в основном означает, что переменная будет переопределена для каждой ссылки. Java причудливо кэширует переменные для многократных потоков, и модификатор сообщает Java-машине прекратить это.

Что такое переменная volatile? Вообразите, что у вас есть переменная, которая реально в памяти отсутствует, но фактически ее значение изменяется. Простейший пример - сигнал модема о наличии несущей в линии. Этот сигнал подается, когда вы соединены. Так как эта переменная изменяется внешним источником, она рассматривается как volatile. Это значение может изменяться между двумя операциями, которые могли бы вызывать ошибки в некоторых ситуациях, так что мы хотим, чтобы компилятор перезагружал значение каждый раз, когда мы обращаемся к переменной volatile.

Если вы не используете собственные (native) методы и не обращаетесь к аппаратным регистраторам или другим значениям изменяющихся данных, вам, вероятно, никогда не придется определять переменные как volatile. Java-код не может непосредственно обращаться к участку памяти, так что эта специфическая потребность в volatile-переменной будет появляться довольно редко.

Что дальше?

Теперь вы должны чувствовать себя вполне уверенно, работая с потоками в Java. Мы рассказали о некоторых наиболее сложных проблемах, возникающих при программировании на Java. В следующей главе мы обсудим, как создать Java-программы, которые преодолеют некоторые из ограничений апплетов. Хотя программы, которые мы разработаем, не так переносимы, как апплеты, вы сможете обратиться к файловой системе и интегрировать в свои программы код на языке C.

ситуациях, у нас был бы выход. К сожалению, современное состояние техники пока не достигло такого уровня.

Глава 12

Программирование за рамками модели апплета

До сих пор в центре нашего внимания было написание апплетов. Однако необходимость реализации апплетов на хост-машине накладывает на них некоторые ограничения. В этой главе мы рассмотрим возможность работы за рамками модели апплета.

Это можно сделать двумя способами. Наиболее простой путь - писать отдельные Java-приложения. Они похожи на другие программы на вашем компьютере и имеют доступ к файловой системе. Более передовым путем представляется использование собственных (native) методов. Собственный метод позволяет интегрировать динамические библиотеки (DLL), являющиеся платформозависимыми. С помощью DLL для усовершенствования Java-программ можно использовать созданные раньше библиотеки C. Поскольку Netscape Navigator 2.0 позволяет связываться с DLL во время выполнения программы, DLL могут также применяться для усовершенствования апплетов. Разумеется, DLL должны быть помещены на хост-машину до того, как апплет будет загружен. Но можно ожидать, что многие не поленятся запросить наши DLL, так же как люди не ленятся получать традиционное программное обеспечение.

СОВЕТ Ко времени издания этой книги, JDK для Macintosh позволяет создавать апплеты, но не отдельные приложения. Если вы работаете на Macintosh, эта глава вам не подходит. По мере развития программных сред мы будем информировать вас о последних достижениях через страницу Online Companion.

От апплетов к самостоятельным приложениям

Зная теперь, что нужно для построения пакетов Java, мы можем использовать одну и ту же программу для создания разных апплетов. Кроме того, мы можем использовать наш пакет с Java-приложениями, для запуска которого не требуется Web-броузера. Мы уже разрабатывали графические интерфейсы пользователя (Graphical User Interface, GUI) для апплетов в соответствии с описанием в главе 7, "Пользовательский интерфейс". Теперь посмотрим, как создать такие же GUI в отдельных Java-приложениях и как использовать в этих новых приложениях уже созданные апплеты.

Приложения, которые мы здесь будем разрабатывать, являются более традиционными программами. Мы избегнем ограничений модели апплета, но зато не воспользуемся ее достоинствами. Пользователи будут вынуждены скачивать или как-то еще доставать наши приложения, и при этом могут возникнуть опасения что-то испортить в компьютере.

При рассмотрении вопроса о том, что создавать - апплет или приложение, - приходится учитывать проблему обеспечения безопасности информации. Но прежде чем мы перейдем к этой теме, давайте обсудим, как заставить работать ваши GUI без помощи оболочки времени выполнения.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Основы графических Java-приложений

Все, что вам нужно для создания отдельного приложения Java, - это иметь метод main, определенный в классе. Когда вы запустите компилятор Java на обработку этого класса, он вызовет метод main. Именно это мы и делали в нашем простом примере "Hello, Java!" в главе 2, "Основы программирования на Java". Разумеется, этот пример не был очень красивым - мы не применяли ничего из того, что обсуждалось в главе 8, "Еще об интерфейсе пользователя", и к тому же не могли пользоваться мышью.

Совсем нетрудно сделать наши отдельные приложения такими же дружественными и иллюстративными, как те апплеты, что мы создавали в главе 8. Все, что нам нужно, - это сделать фрейм либо непосредственно в нашем методе main, либо в одном из методов, вызываемых им. Программа, приведенная ниже, использует Frame2, разработанный в главе 8. Но Frame2 не находится в апплете, а является частью нашего отдельного приложения, sampleStand:

```
import java.awt.*
import Frame2;
class standAlone {
        public static void main(String S[]) {
                Frame2 f2 = new Frame2();
                f2.show();
        }
}
```

В главах 7 и 8 мы рассматривали Abstract Windowing Toolkit (AWT). Ваших знаний об AWT уже почти достаточно для того, чтобы написать отдельное графическое приложение. Создав подкласс в классе Applet, мы можем получить графические и звуковые данные из Интернет. К сожалению, невозможно получить и проиграть звуковые фрагменты для нашего отдельного приложения, не реализовав сам метод main. Графические данные мы можем получить из Интернет с помощью класса Toolkit.

Класс Toolkit - это абстрактный класс, который, вообще говоря, служит связкой между AWT и локальной системой управления окнами. Большинство методов, использующихся в этом классе, вряд ли могут нам пригодиться, потому что они применяются только при связывании AWT с конкретной платформой. Тем не менее для нас важно то, что этот класс определяет метод getImage, который работает точно так же, как метод, к которому мы уже привыкли. Чтобы получить изображения из нашего отдельного приложения, можно воспользоваться следующей программой внутри любого подкласса frame (здесь U обозначает URL):

```
Toolkit T = getToolkit();
Image I = T.getImage(U);
```

Теперь, за исключением проигрывания звуковых фрагментов, мы можем делать с нашими отдельными приложениями все то, что можно делать и с апплетами. Поскольку наше Java-приложение не является апплетом с сомнительной репутацией, мы можем также обращаться к локальной файловой системе. Например, можно воспользоваться загруженным методом getImage для переноса графических данных прямо из файловой системы путем передачи их строки (String) с описанием пути файла:

```
String S = myImage.gif;
Toolkit T = getToolkit();
Image Internet = T.getImage(S);
```

Как и в предыдущем примере с методом getToolkit, мы должны делать это в компонентном подклассе. Наш метод getImage ищет файл myImage.gif в текущем каталоге локальной файловой системы.

Класс FileDialog пакета java.awt создает экземпляр в диалоговом окне, который позволяет пользователю просматривать файловую систему. Этот класс закрыт для апплетов; им совершенно незачем иметь доступ к файловой системе хоста, на который они загружены. Но поскольку наше отдельное приложение достаточно надежно, чтобы установить его на хост-компьютере, мы можем теперь программировать, используя класс FileDialog. Его методы и конструкторы приведены в табл. 12-1.

Таблица 12-1. Класс FileDialog

Элемент	Описание
final static int LOAD	Переменная режима, задающая файловому диалогу режим чтения.

final static int SAVE	Переменная режима, задающая файловому диалогу режим сохранения файлов.
FileDialog(Frame parent, String title)	Создает экземпляр; режим по умолчанию.
FileDialog(Frame parent, String title, int mode)	То же, что FileDialog(Frame parent, String title), но с определенным режимом.
int getMode()	Получает режим данного диалога.
void setDirectory(String dir)	Задает каталог диалога.
void setFile(String file)	При вызове до начала изображения диалога задает файл по умолчанию для диалога.
String getFile()	Получает имя определенного файла.
String getDirectory()	Получает имя каталога диалога.
String paramString()	Переопределяет paramString в Dialog
FilenameFilter setFilenameFilter()	Устанавливает фильтр имени файла.
FilenameFilter getFilenameFilter()	Получает фильтр имени файла.

Пользоваться классом FileDialog очень просто, его единственная задача - снабдить пользователя стандартным диалоговым окном для просмотра файлов на определенной платформе, как показано на рис. 12-1. Когда эта задача выполнена, применяется метод getFile, чтобы получить имя файла, и метод getDirectory, чтобы получить путь к файлу. Фактически файловый диалог не касается непосредственно системы файлов; он только делает доступным то, что выбрал пользователь.

Рис. 12.1.

Внешний вид и операции диалогового окна можно менять несколькими способами. Когда мы конструируем файловый диалог, мы должны задать для него какой-то заголовок. Кроме того, мы можем решить, нужна ли нам в диалоговом окне кнопка Save или кнопка Open; для этого нужно установить соответствующий режим в конструкторе FileDialog(Frame parent, String title, int mode). Заметьте, что эти изменения носят косметический характер, потому что на самом деле класс FileDialog не пытается изменить файлы. Разумеется, гораздо менее дружественным пользователю решением было бы установить нужный режим в FileDialof.LOAD, а потом заново переписывать файл. Поскольку по умолчанию делается именно так, если мы действительно хотим создавать файл или каталог с внесенными пользователем изменениями, мы должны явно задавать нужный режим в FileDialog.SAVE. Установка режима в FileDialog.SAVE избавит пользователя от необходимости переписывать файл. Если пользователь выберет уже существующий файл, диалоговое окно само создаст предупреждающее диалоговое окно - так что наше приложение может об этом не заботиться. Следующие два метода показывают, как пользоваться классом FileDialog в каждом из двух режимов. Поскольку мы пропускаем эти методы через их фрейм-родитель, они могут находиться в любом удобном для нас классе:

```
public String fileToWrite(Frame parent, String title) {
        FileDialog fD = new FileDialog(parent, title, FileDialo.SAVE);
        if (!parent.isVisible()) parent.show();
        fd.show();
        // выполнение будет остановлено до тех пор, пока
        // пользователь не сделает выбор
        String path = fD.getDirectory() + fd.getFile();
        return path;}
public String fileToOpen(Frame parent, String title) {
        FileDialog fD = new FileDialog(parent, title);
        // поскольку режим по умолчанию - SAVE,
        // его не нужно устанавливать
        f (!parent.isVisible()) parent.show();
        fd.show();
        // выполнение будет остановлено до тех пор, пока
        // пользователь не сделает выбор
        String path = fd.getDirectory() + fd.getFile();
        return path;}
```

Обратите внимание на проверку того, виден ли фрейм-родитель: как и при работе со всеми диалоговыми окнами, фрейм-родитель должен быть активным перед тем, как мы попытаемся его показать. Поскольку FileDialog - это модальное диалоговое окно, то после его появления на экране выполнение будет остановлено, а затем снова продолжено после того, как пользователь сделает выбор.

Кроме установки режима мы еще можем изменить поведение диалогового окна, задав FilenameFilter перед тем, как оно появится на экране. Реализовав в каком-то классе интерфейс, а затем передав этот класс классу FileDialog вместе с setFilenameFilter, мы можем ограничить выбор тех файлов, которые FileDialog предоставляет пользователю. Обычно FilenameFilter используют для того, чтобы предоставлять только некоторые типы файлов, например HTML-файлы, или чтобы выдавать файлы только после некоторой даты. Для реализации FilenameFilter нам нужно только переопределить один метод, как показано ниже:

```
class FilenameFilterWrapper implements FilenameFilter {
//переменные, конструкторы
        public boolean accept(File dir, String name) {
        // решайте, принять или нет
}
```

Как видите, здесь мы уже использовали объект File. После того как мы выясним, что такое класс File, вы поймете, как использовать FilenameFilter для фильтрации файлов в соответствии с фрагментами имен файлов, возможностями чтения и записи файла и датой последней модификации.

Доступ к файловой системе

Как мы только что видели, файловый диалог можно применять для того, чтобы разрешить пользователям просматривать файловую систему. Но это не поможет нам получить реальный доступ к тому файлу, который выбрал пользователь. Чтобы сделать это, нужно воспользоваться двумя классами из пакета java.io - File и RandomAccessFile.

Потоки файлов и файловая система

Если вы когда-нибудь смотрели в on-line режиме документацию компании Online Companion, вы, возможно, заметили классы FileInputStream и FileOutputStream. Чтобы ими воспользоваться, нужно знать основные свойства их суперклассов - соответственно InputStream и OutputStream. Об этом мы будем говорить в главе 13, "Работа с сетью на уровне сокетов и потоков".

Учтите, однако, что названия этих классов не совсем точно отражают их сущность. Класс File - это абстракция, означающая как файлы, так и каталоги. Именно этот класс используется для таких задач управления файлами, как проверка существования данного файла, уничтожение файлов, создание новых каталогов, навигация вверх и вниз по структуре каталогов. С другой стороны, класс RandomAccessFile используется для того, чтобы реально читать и записывать файлы. Можно представить себе класс File как морской флот, а класс RandomAccessFile как морскую пехоту - первый является транспортным средством, а вторая выполняет работу по транспортировке.

Класс File

Рассмотрим для начала класс File и его очень важные статические переменные. Статические переменные полезны при разрешении очень простого варианта файловой системы - различных разделителей файлов и пути файлов. Методы и конструкторы этого класса считаются согласованными с переменными типа String, прикрепленными к конвенциям файловой системы хоста, поэтому при создании имен файлов важно использовать эти переменные. В табл. 12-2 приводятся разделители файлов, пути файлов и их значения для UNIX и для Windows 95/NT.

Таблица 12-2. Разделители файлов и пути файлов

Переменная	Смысл	Значение в UNIX	Значение в Windows 95/NT
String separator	Разделитель файлов для системы хоста	/	\
String pathSeparator	Разделитель пути файла для	:	;

	системы хоста		
char separatorChar	Разделитель файлов как символ	/	\
char pathSeparatorChar	Разделитель пути файла как символ	:	;

Хотя символьные представления являются доступными, надежнее использовать представления типа String, перечисленные вверху таблицы. Класс FileDialog, упомянутый выше, гарантированно вернет системе правильное представление. Помня об этих переменных, можно создать экземпляр объекта File с помощью конструкторов, перечисленных в табл. 12-3. Как уже упоминалось выше, создаваемый нами объект File может относиться как к каталогу, так и к обыкновенному файлу.

Таблица 12-3. Конструкторы класса File

Конструктор	Описание
File(String path)	Создает объект файла на основе пути файла; может быть каталогом или файлом.
File(String dir, String fileName)	Создает объект файла, представляющий имя файла (fileName) в каталоге dir.
File(File dir, String fileName)	Создает объект файла, представляющий имя файла (fileName) в каталоге dir.

Вас, возможно, удивляет, почему один и тот же класс применяется и для каталогов, и для файлов. Дело в том, что на уровне операционной системы каталог часто представляет собой специальный файл, содержащий информацию о тех объектах, которые мы привыкли понимать как "нормальные" файлы. Кроме этой программистской детали, файлы и каталоги имеют много практических общих свойств, как показывают методы в табл. 12-4. Эти методы одинаково применимы как к каталогам, так и к файлам.

Таблица 12-4. Общие методы класса File

Метод	Описание
boolean exists()	Возвращает true, если файл или каталог существует.
String getPath()	Возвращает путь, с которым был построен объект file.
boolean isAbsolute()	Возвращает true, если создан абсолютный путь.
String getAbsolutePath()	Возвращает абсолютный путь.
String getParent()	Возвращает абсолютный путь каталога Parent; возвращает ноль, если достигнут верх иерархии или если Parent недоступен.
public boolean delete()	Пытается уничтожить файл или каталог; возвращает true в случае успеха.
public boolean renameTo(File dest)	Пытается переименовать файл или каталог в dest; возвращает true в случае успеха.
public boolean equals(Object o)	Сравнивает с другими объектами Java на совпадение.
public int hashCode	Разрешает хранение файлов в java.util.Hashtable.

Несмотря на известную красоту подхода, связанного со складыванием файлов и каталогов в одну корзину, между ними все же существуют некоторые различия. Табл. 12-5 и 12-6 показывают оставшиеся методы, которые лучше применять соответственно к файлам или к каталогам.

Таблица 12-5. Методы класса File для каталогов

Метод	Описание
public boolean mkdir()	Пытается создать каталог; возвращает true в случае успеха.
public String[] list()	Перечисляет файлы в каталогах, за исключением текущего и содержащего его каталога.
public String[] list(FilenameFilter filter)	Перечисляет файлы в каталогах, применяющих входной фильтр.

Таблица 12-6. Методы класса File для файлов

Метод	Описание
boolean canWrite()	Возвращает true, если записываемый файл существует.
boolean canRead()	Возвращает true, если читаемый файл существует.
long lastModified()	Возвращает приписанное системой время модификации.
long length()	Возвращает длину файла.

Обычно класс FileDialog используется для того, чтобы спросить пользователя, с каким файлом или каталогом в файловой системе он хотел бы работать. Если программа позволяет пользователю оперировать с файловой системой - например, создавать каталоги, уничтожать каталоги или файлы, перемещать каталоги или файлы, - в ней создается класс File и применяются соответствующие методы для работы с файловой системой. Если же нас интересуют данные в каком-то определенном файле, нужно воспользоваться классом File для того, чтобы убедиться, что этот файл не испорчен, а затем создать класс RandomAccessFile для работы с данными. Подобные предварительные операции должны также включать проверку того, что файл можно читать и/или записывать. В некоторых случаях нас может заинтересовать также дата последнего изменения файла.

Класс RandomAccessFile

Несмотря на то, что мы уже можем получить основную информацию о файле - например, когда он был изменен, насколько он большой, - мы все еще не можем его читать или записывать. Для этого нам понадобится класс RandomAccessFile. Этот класс позволяет считывать данные из файла прямо в массивы, строковые переменные и переменные примитивных типов Java. Данные в файл можно писать в обратном порядке.

Класс RandomAccessFile можно создать с помощью либо объекта File, либо переменной типа String, в которой описывается путь к файлу. Сделав это, мы должны сказать, хотим ли мы только читать файл или читать и записывать в него. Файл открывается на чтение и/или запись после реализации объекта RandomAccessFile. Открыв файл, можно читать или писать данные одним из простейших способов - либо байт за байтом, либо строку за строкой. Основные методы и конструкторы для работы с файлами описаны в табл. 12-7. Так же, как все методы класса File, каждый метод вызывает IOException.

Таблица 12-7. Простые методы ввода/вывода класса File

Метод	Описание
RandomAccessFile(String path,String mode)	Конструирует класс RandomAccessFile на основе пути с определенным режимом; "r" только для чтения, "rw" для чтения/записи.
RandomAccessFile(File f, String mode)	Конструирует класс RandomAccessFile на основе объекта File с определенным режимом.
void close()	Закрывает файл.
public long getFilePointer()	Возвращает расположение указателя файла; расстояние в байтах от начала файла.
public int read()	Читает байт данных; если конец файла, возвращает -1.
public String readLine()	Возвращает строку, начиная с текущего положения указателя файла и кончая символами '\n' в конце файла.
public int read(byte b[])	Читает файл в массив байтов, возвращая число прочитанных байтов.
public int read(byte b[], int shift, int len)	Смещает указатель файла и читает len байтов.
piblic int skipBytes(int n)	Смещает указатель файла на n байтов вперед по файлу или к концу файла.
public void seek(long pos)	Устанавливает указатель файла на pos байтов от начала файла.
public void write(int b)	Записывает int в файл, сначала приведя к байту.
public void write(byte b[])	Записывает массив байтов.

Машинозависимые методы

Прежде всего необходимо уточнить, для чего нельзя использовать интегрированные библиотеки C. Мы не можем интегрировать какую-то библиотеку C в апплет и послать этот

апплет по Интернет, чтобы он брал из библиотеки на стороне клиента все, что ему нужно. Если бы такая возможность была, мы рассказали бы о ней в этой книге гораздо раньше!

Библиотека C должна находиться там, где реально выполняется программа. В действительности мы закладываем текст программы на C в класс Java, а затем через класс Java вызываем функции C. Функции C составляют динамическую библиотеку (DLL) и доступны с помощью машинозависимых или собственных (native) методов.

Следите за меняющимися стандартами!

На следующих страницах мы опишем интерфейс машинозависимого метода для JDK 1.0. Однако он будет меняться в последующих версиях языка. Не ждите, что собственные библиотеки, которые вы создадите сегодня, будут работать с будущими версиями Java. Обращайтесь в Online Companion за информацией о текущем состоянии интерфейса машинозависимого метода.

Когда нужны машинозависимые библиотеки

При написании отдельных приложений желательно использовать DLL. Поскольку C - необыкновенно популярный язык, часто оказывается, что то, что вы собираетесь сделать, уже делается с помощью библиотек C. Благодаря этому вам не нужно переписывать все на Java - достаточно обратиться к DLL.

Программисты, пишущие апплеты, тоже могут воспользоваться DLL, но в меньшей степени, чем при написании отдельных приложений. Поскольку язык C не является платформонезависимым, DLL нельзя пересылать через Интернет; иначе они рано или поздно попадут на платформу, на которой их нельзя будет выполнить. Кроме того, при использовании DLL возникают проблемы, связанные с обеспечением безопасности апплетов, поскольку при создании языка C не было предусмотрено обеспечение безопасности информации на распределенных компьютерах.

Тем не менее некоторые Web-броузеры, поддерживающие Java, позволяют апплету пользоваться любыми библиотеками, которые были ранее инсталлированы на стороне клиента. Что это значит для программиста, пишущего апплеты? Если вы занимаетесь в своей организации интрасетями, вы можете проследить за тем, чтобы простая библиотека C была установлена на стороне клиента, и убедиться в том, что она не содержит вирусов и что ею невозможно злоупотребить. Тогда вы можете позволить своим апплетам некоторые вещи, которые вы не хотели бы позволять всем апплетам (например, доступ к файловой системе).

Если ваша любимая игрушка - сжимающее программное обеспечение, вы можете распространять CD, содержащие DLL, или написать программу скачивания этого программного обеспечения из Интернет. После проверки вашей библиотеки DLL на вирусы ее можно будет инсталлировать, чтобы поддерживающий Java Web-броузер мог ее найти. Ваша библиотека не будет испытывать ограничения основной модели апплета, и в то же время выиграет от соединения с поддерживающим Java Web-броузером. Например, вы можете использовать DLL для того, чтобы связать свою программу электронной таблицы прямо с апплетом, связанным с сервером биржевых цен. Или вы можете сделать игры через Интернет быстрее за счет того, что на стороне клиента будет создана графика, доступная через вашу DLL. Тогда вы сможете пользоваться сетевыми свойствами Java чисто для общения, а не для скачивания весомых ресурсов.

Соединяться или нет: решение вашего Web-броузера

При обсуждении машинозависимых библиотек мы предполагаем, что поддерживающий Java Web-броузер позволяет библиотеке DLL соединяться с оболочкой времени выполнения. Фирма Netscape решила, что это можно позволить Web-броузерам, только, к сожалению, к моменту написания этого текста Netscape не сообщила, каким образом разработчики могут соединяться с DLL. Для получения текущей информации обращайтесь по следующему адресу: http://www/vmedia.com/olc/java/updates/native.html

Итак, DLL могут быть полезны для развития приложений Netscape. Прежде чем перейти ко всем хитростям построения DLL, нам нужно понять, почему DLL важны для оболочки времени выполнения Java. Java - платформонезависимый язык, но он все же требует какой-то связи с операционной системой. Такую связь дают DLL, соединенные с API. DLL играют важную роль, решая, что именно позволено сделать некоторым разделам Java-программы в данной среде. Поскольку DLL не являются платформонезависимыми, они играют ключевую роль при переносе Java как единого целого на новые платформы. Ниже в этой главе, при изучении вопросов создания пакетов Java, мы посмотрим, как построение API позволяет облегчить такой перенос в разные оболочки времени выполнения.

Объяснение машинозависимых методов

Java является платформонезависимым языком, но программа, написанная на Java, реально имеет дело с платформой, на которой она запускается в процессе выполнения. Простая Java-программа может рисовать рисунки на многих разных платформах, хотя все платформы совершенно по-разному обращаются с графикой. Для всех операций ввода-вывода низкого уровня, включая ввод с клавиатуры и с помощью мыши, вывод графики и звука и сетевые операции, Java-программа должна выполнять платформозависимые обращения к операционной системе. Выполнение этих обращений производится через динамические библиотеки (DLL), являющиеся частью оболочки времени выполнения.

Язык Java полностью изолирует свои платформозависимые обращения с помощью ключевого слова native. Если вы используете ключевое слово native, это означает, что функции соответствующего метода написаны на другом языке, а для того, чтобы этот метод был вызван, DLL должна быть загружена в исполняющую систему. В настоящее время единственным "другим языком", на котором может быть написана DLL, является C. Поскольку C++ полностью включает в себя C, технически возможно использовать для написания DLL язык C++. Но, к сожалению, для того чтобы склеить вашу программу на C++ с программой на Java, вам все равно придется использовать просто C. Когда мы действительно перейдем к созданию DLL, эти проблемы станут очевидны.

Во многих отношениях собственные (native) методы в точности похожи на другие методы. Как ни странно, это сходство включает возможность воспринимать объекты Java как параметры, возврат объектов Java и выдачу исключений. Однако есть и два очень заметных отличия. Во-первых, собственные методы не имеют тела, и компилятор запрещает им иметь его:
public native String getUserName();

Собственные методы получают свои функции от одной определенной DLL, загруженной в исполняющую систему, что и составляет второе отличие. DLL должна быть загружена, когда класс, содержащий собственный метод, реализован. Теперь мы определили собственный класс-упаковщик (wrapper class), который загружает DLL, называемый Profile. Цель этого класса - позволить апплетам читать файл на машине клиента, содержащей информацию о пользователе. Мы создали два класса исключения, NoProfileException и ProfileWriteUnallowedException, написанные на Java. Эти классы определены на диске CD-ROM.

Пример 12-1. Класс LocalProfile.

```
import java.awt.Color;
class LocalProfile {
        private int CfilePointer=0;
        public LocalProfile() throws UnsatisfiedLinkError, NoProfileException
{
                try {System.loadLibrary(profile);
                        }
                catch (UnsatisfiedLinkError e) {
                        throw(e);}
                // openProfile должен генерировать исключение
                // непосредственно, но это будет сделано в более
                // простом примере, когда мы будем писать DLL
                CfilePointer=openProfile();
                if (CfilePointer==0)
                        throw new NoProfileException();
                }
        private native int openProfile();
        public native void setPublicKey(byte pubKey[]);
        public native String getAttrib(String key);
        // читает String из файла Profile на хосте
        public native Color getFavoriteColor();
        // просмотрев файл Profile, создает
        // экземпляр любимого цвета пользователя
        public synchronized native void setAttrib(String key,
        String value) throws ProfileWriteUnAllowedException;
        // если разрешено, пишет в файл Profile на хосте
}
```

Когда этот класс реализован, загружается DLL "Profile". Любая DLL доступна только через собственные методы, описанные в ее классе-упаковщике, и может быть загружена только до того, как эти методы вызваны. Каждый собственный метод соответствует функции C,

содержащейся в DLL. При вызове собственного метода вместо него вызывается эта функция C. Разумеется, DLL может содержать столько функций, сколько угодно, но они не могут быть вызваны непосредственно. Тем не менее любая функция из DLL может как создавать новые объекты Java, так и манипулировать с текущими - включая вызов общих методов. Как можно понять из программы, приведенной в примере, эта способность распространяется также на выдачу исключений.

Смысл собственных методов в том, что они действуют так же, как любые другие методы Java. То же самое в целом относится к классам, содержащим методы Java, хотя здесь возникают некоторые нюансы. В основном они связаны с необходимостью убедиться в том, что библиотека загружается. В нашем примере это особенно важно. Если мы напишем апплет, включающий наш класс, и он попадет к клиенту, у которого нет DLL, мы захотим, чтобы наш апплет восстановился. Для этого мы загрузим апплет в конструктор и заставим его выдать UnsatisfiedLinkError.

Требование загрузить библиотеку вызывает также проблемы со статическими методами. В любом другом случае все наши методы были бы определены как статические. Но если мы зададим все методы статическими, конструктор не будет вызван и библиотека не будет загружена. Это можно обойти, загрузив библиотеку в статический блок:

```
class LocalProfile {
// объявления переменных
        static {
                try {System.loadLibrary(profile);}
                        catch (UnsatisfiedLinkError e) {
                        // обработка ошибки
                        }
                // статические методы
}
```

Однако загрузка библиотеки в статический блок затруднит восстановление после UnsatisfiedLinkError. Во многих случаях это не важно. Но если цель вашего апплета или приложения в высокоскоростной передаче графики и для этого вы используете собственную программу, вы скорее всего не удержитесь на поверхности, если не сможете загрузить библиотеку. В таком случае вежливо удалиться - пожалуй, лучший выход из создавшегося положения.

Подготовка библиотеки C

Определив методы, используемые в собственном классе-упаковщике, мы можем начать компилировать программу на Java. Поскольку библиотека загружена в исполняющую систему, компилятор не должен ничего знать о самой библиотеке. Конечно, прежде чем использовать DLL, мы должны написать и откомпилировать ее таким образом, чтобы ее можно было загрузить.

Однако прежде, чем мы всем этим займемся, необходимо сделать одно предостережение: создание DLL, связанной интерфейсом с Java-программой, очень похоже на игру в Твистер. Если вы относитесь к людям, считающим препроцессор C запутанным и красивым, этот раздел утолит вашу жажду познания тайн файлов java/include/*.h. В противном случае вам придется воспринимать этот раздел просто как рецепт. Главной составляющей является программа javah. Javah - это средство, которое создает пару специальных файлов, определяющих наши собственные методы и типы данных Java. Прежде чем Java начнет совершать свои магические действия, нужно откомпилировать класс, содержащий собственные методы, - это будет нашим первым шагом. Для компиляции мы используем javac просто как обычный класс.

Нашим вторым шагом будет использование javah для создания файла-заголовка для нашего класса. Для класса LocaProfile командный запрос будет выглядеть так:
javah LocalProfile

Javah смотрит файл LocalProfile.class и выдает определения C для собственных методов и для других элементов в нашем классе LocalProfile. Они помещаются в файл с расширением .h; в данном случае LocalProfile.h. То, что мы создали на этом этапе, понадобится нам, когда мы будем писать нашу DLL. Теперь, когда у нас есть файл-заголовок, нам нужно создать файл с расширением .c, который будет заложен в нашу библиотеку. Это файл представляет собой исполняющую систему, связывающую DLL с Java-программой. Создадим этот файл с помощью следующей команды:
javah -stubs LocalProfile

Эта команда создает файл, называемый LocalProfile.c, - файл, который может показаться уродливым или очаровательным в зависимости от вашего подхода к программированию. В любом случае, больше мы никогда не увидим этот файл. В отличие от LocalProfile.h он не содержит информации, которая нам впоследствии понадобится.

На этом заканчивается первая часть нашей игры в Твистер; теперь мы можем действительно начать писать DLL. Вот обзор магических заклинаний, которые мы должны прочесть, прежде чем начнем программировать на C:

- С помощью javac откомпилируйте файл .java, содержащий собственные методы.
- С помощью javah создайте файл .h, который выполняет определения для собственного класса-упаковщика.
- С помощью javah -stubs создайте файл .c, который построит связывающую исполняющую систему.

Прежде чем двинуться дальше, давайте изучим javah немного подробнее. Во-первых, ей можно задать несколько аргументов. Если вы написали несколько классов для загрузки одной и той же DLL, скомпилируйте их, а затем перечислите после javah и javah -stubs. Кроме того, может быть полезно добавить несколько командных строк с нужными вам опциями. Любые комбинации опций, описанных в табл. 12-8, должны предшествовать именам классов Java.

Таблица 12-8. Опции javah

Опция	Описание
-o outputfile	Помещает выходные данные в outputfile вместо файла по умолчанию.
-d directory	Сохраняет файлы в directory, а не в текущем каталоге.
-td tempdirectory	Сохраняет временные файлы в tempdirectory вместо определенного по умолчанию /temp.
-classpath classpath	Ищет файлы с расширением .class из API в каталоге classpath, а не в /usr/local/java, или в переменной окружения CLASSPATH, если таковая задана.
-verbose	Указывает Java печатать на экран подробности действий.

Теперь мы готовы к тому, чтобы действительно создать нашу динамическую библиотеку. Но прежде чем начнем писать ее функции, нужно определить некоторые формальности. Во-первых, нужно назвать наш .c файл. Мы не можем назвать его LocalProfile.c, потому что javah -stubs уже создала файл с таким именем. Имя выбирается произвольно, но, чтобы не запутаться окончательно, назовем файл profile.c. Следующим шагом будет сделать наши препроцессорные определения, как показано ниже:

```
#include <StubPreamble.h>
#include <avaString.h>
#include "LocalProfile.h"
```

Эти файлы-заголовки должны перевести структуры C в типы Java. Заголовок javaString.h дает нам некоторые специальные функции для обработки Java. Затем мы должны включить те файлы-заголовки C, которые понадобятся нашей DLL для данной системы. Такие файлы варьируются от библиотеки к библиотеке и от платформы к платформе. Вот файлы, которые нам нужны для нашей библиотеки profile:

```
#include <sys/types.h>
#include <sys/param.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
```

Следующий список перечисляет шаги, которые мы уже проделали:

1. Написать и откомпилировать класс Java.
2. Выполнить javah на откомпилированном классе.
3. Выполнить javah -stubs на откомпилированном классе.
4. Выполнить собственные методы на C, которые будут рассмотрены в следующем разделе.

Выполнение собственных методов на C

Теперь, когда устранены формальности, мы можем начать выполнение собственных функций. Поскольку собственные методы должны работать так же, как и другие методы, главная трудность заключается в том, чтобы заставить функции C действовать, как внутренние методы, в нашем

собственном классе-упаковщике. Многие тайны, связанные с тем, как это сделать, можно открыть, если посмотреть на файл LocalProfile.h.

Пример 12-2. Файл LocalProfile.h.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class LocalProfile */
#ifndef _Included_LocalProfile
#define _Included_LocalProfile
typedef struct ClassLocalProfile {
        long CfilePointer;
} ClassLocalProfile;
HandleTo(LocalProfile);
extern long LocalProfile_openProfile(struct HLocalProfile *);
extern void LocalProfile_setPublicKey(struct HLocalProfile *,HArrayOfByte *);
struct Hjava_lang_String;
extern struct Hjava_lang_String
*LocalProfile_getAttrib(struct HLocalProfile *,struct Hjava_lang_String *);
extern struct Hjava_awt_Color
*LocalProfile_getFavoriteColor(struct HLocalProfile *);
extern struct Hjava_awt_String
*LocalProfile_setAttrib(struct HLocalProfile *,struct Hjava_lang_String
*,struct Hjava_lang_String *);
#endif
```

Первая структура (struct) - это представление класса Java на C. Как вы помните, собственные методы ведут себя так же, как другие методы, и потому могут обращаться к частным элементам класса, который их содержит. Структура "ClassLocalProfile" делает это возможным. Заметьте, что эта структура передается всем нашим функциям в добавление к тем параметрам, которые содержат эквивалентный собственный метод Java. Это гарантирует правильное скрытие данных и в то же время позволяет нашим функциям C быть истинными элементами нашего класса.

За структурой следуют определения функций C для наших собственных методов. Попросту говоря, написать DLL означает определить эти функции. Но для того чтобы это сделать, нужно преодолеть различия между Java и C:

- Типы Java должны быть конвертированы в правильные типы C.
- Мы должны иметь возможность обращаться к другим элементам класса из DLL.
- Нам нужны механизмы перевода массивов Java в C.
- Мы должны иметь возможность создавать объекты Java из DLL и обращаться к методам общих элементов и переменным в существующих объектах Java.
- Нам нужно выдавать исключения из DLL.

В нашем примере все это делается. Давайте начнем с простейших функций и постепенно перейдем к самым трудным для выполнения на C. Два наших собственных метода, определенных с модификатором private, validProfile и openProfile, наиболее просты для выполнения, потому что для них нужно знать только, как переводить примитивные типы Java в типы C. В табл. 12-9 показано, как делать этот перевод:

Таблица 12-9. Перевод примитивных типов Java в типы C

Примитивный тип Java	Эквивалент в C
boolean	long
char	unicode
byte	char
short	short
int	long
long	int64_t
float	float
double	double

Поняв, как переводить примитивные типы из Java в C, мы можем теперь выполнить небольшую работу с помощью LocalProfile_openProfile. Этот метод просто открывает файл, описанный в переменной окружения APPLETPROFILE:

```
long LocalProfile_openProfile(struct HLocalProfile *this) {
char profileName[1024];
strcpy(profileName, getenv(APPLETPROFILE));
return (long) fopen(profileName, r);}
```

Как вы помните, этот метод, определенный с модификатором private, используется для того, чтобы присвоить значение переменной CFilePointer. Поскольку этот файл потребуется и другим нашим собственным методам, имеет смысл сохранить его как часть объекта. Когда нужно будет к нему обратиться, мы сделаем это с помощью указателя struct HLocalProfile, передаваемом при любом выполнении собственного метода. При компиляции собственного класса-упаковщика генерируется структура struct. Ее имя начинается с "H" и заканчивается именем класса.

Не беспокойтесь, нам не придется использовать эту структуру для того, чтобы добраться до элементов экземпляра собственного упаковщика. Для этого достаточно реализовать функцию _dynamic_method. Ее можно использовать как для экземпляра собственного упаковщика, так и для других экземпляров (вкратце мы это обсудим ниже). Сейчас нас больше всего заботит доступ к элементам данных экземпляра собственного упаковщика. Для того чтобы обратиться к переменной CFilePointer в LocalProfile, воспользуемся приведенным ниже внешним макросом.

Пример 12-3. Простая реализация собственного метода.

```
long LocalProfile_getAttrib(struct HLocalProfile *this, HJava_lang_String
*key) {
FILE *profile = (FILE *)unhand(this)->CFilePointer;
/* Остальная часть функции*/
}
```

Для того чтобы действительно реализовать остальные функции, мы должны уметь обращаться с массивами и строками Java в C. Это объясняется в следующем разделе.

Массивы и строки Java в C

В нашей библиотеке нам часто будут нужны массивы и строки, и файл java/include/*.h содержит необходимые средства для работы с ними. Начнем со строк и с функции LocalProfile_getAttrib. Эта функция будет рассматривать наш файл profile как хеш-таблицу. Она будет искать строку вида key=value, ключом в которой является переданная нам строка. Если функция не сможет найти ключ, она вернет ноль. В противном случае она вернет строку, стоящую сразу после знака = и заканчивающуюся перед первым встреченным пробелом.

Чтобы это сделать, нужно прежде всего задать символьный массив, который описывает нашу строку Java. Если мы найдем искомое значение, его нужно будет перевести из символьного массива в структуре Hjava_lang_String, который оболочка времени выполнения конвертирует в значение типа String. Функции, делающие это, описаны в табл. 12-10, там же приведены еще некоторые доступные нам функции.

Таблица 12-10. Функции Java для работы со строками

Функция	Описание
void javaString2CString(Hjava_lang_String *jString, char *buf, int size)	Конвертирует jString в символьный массив buf; последний параметр задает длину.
char *makeCString(Hjava_lang_String *jString)	Возвращает параметр массива на основе jString, который числится иполняющей системой как мусор.
char *allocCString(Hjava_lang_String *jString)	Аналогичен makeCString за исключением того, что используется malloc и вы отвечаете за перераспределение памяти.
int javaStringLength(Hjava_lang_String *jString)	Возвращает длину jString.
Hjava_lang_String *makeJavaString (char *s, int size)	Делает представление строки Java на C на основе массива s; последний параметр задает длину.
void javaString2unicode (Hjava_lang_String *jString, unicode *buf,int, size)	Конвертирует jString в массив buf формата unicode; последний параметр задает размер.

Теперь мы можем написать функцию LocalProfile_getAttrib.

Пример 12-4. Использование строк Java в C.

```
struct Hjava_lang_String * LocalProfile_getAttrib(struct HlocalProfile *this,
Hjava_lang_String *key) {
char Ckey[1024];
char Cvalue[1024];
Hjava_lang_String *Jvalue;
FILE *profile = (FILE *)unhand(this)->CfilePointer;
int keyLength = javaStringLength(key);
javaString2CString(key, Ckey, keyLength);
strcat(Ckey, "=%s");
rewind(profile);
if (fscanf(profile, Ckey, Cvalue) != EOF) {
return makeJavaString(Cvalue, strlen(Cvalue));
}
else {
        return NULL;}
}
```

В примере 12-4 мы смогли преодолеть различия между строками Java и C. Теперь пойдем дальше и посмотрим на функции, которые делают массивы Java и C совместимыми. В то время как элементы массивов C представляют собой просто упорядоченный набор указателей, массивы Java являются упорядоченными наборами данных особого типа. Мы не можем просто создать массив C и ожидать, что наша программа на Java поймет, что с ним делать. Мы должны определить, какой тип массива Java мы пытаемся описать в нашей библиотеке. В табл. 12-11 приведена такая информация для простых типов. Объяснения по поводу объектов будут даны, когда мы поймем, как их создавать. Во второй колонке представлена заранее определенная структура, описывающая тип массива. Как и раньше, для того чтобы обратиться к части данных массива, мы можем использовать внешний макрос. В третьей колонке даны константы, описывающие этот тип. Для правильного распределения места константа передается макросу ArrayAlloc.

Таблица 12-11. Простые массивы Java на C

Массив Java	Представление на C	Обозначение типа
boolean[]	HArrayOfInt *	T_BOOLEAN
byte[]	HArrayOfByte *	T_BYTE
char[]	HArrayOfChar *	T_CHAR
short[]	HArrayOfShort *	T_SHORT
int[]	HArrayOfInt *	T_INT
long[]	HArrayOfLong *	T_LONG
float[]	HArrayOfFloat *	T_FLOAT
double[]	HArrayOfDouble *	T_DOUBLE
Object	HArrayOfObject *	T_CLASS

Когда мы хотим создать массив Java, мы используем функцию ArrayAlloc, зависящую от двух параметров. Первый параметр определяет тип массива, который описывается в нашей второй колонке. Второй параметр определяет требуемый размер массива. Затем мы снова применяем внешний макрос для того, чтобы реально обратиться к части данных массива. Следующий раздел программы создает массив из десяти байтов и приписывает ему значения от 0 до 9. Как вы помните, байт Java является символом на C, и, следовательно, мы можем заместить байт символом в приведенном ниже фрагменте:

```
HArrayOfByte *bytes=ArrayAlloc(T_BYTE,10);
for (int i=0;i;i++) {
unhand(bytes)->data[i]=(char)i;}
```

Теперь можно перейти к написанию нашей функции LocalProfile_getPubKey(). Воспользуемся переменной окружения APPLETPUBKEY для того, чтобы узнать, где находится файл, и библиотечной функцией stat для выяснения размера. Если файл найти не удается, эта функция возвращает нуль.

Пример 12-5. Массивы Java на C.

```
HArrayOfByte *LocalProfile_getPubKey(struct HLocalProfile *this) {
        int i;
        FILE *pubKeyFile=NULL;
         char pubKeyFileName[1024];
         char *buf=NULL;
         HArrayOfByte *pubKey=NULL;
         int counter=0;
         strcpy(pubKeyFileName,getenv("APPLETPUBKEY"));
         pubKeyFile=fopen(pubKeyFileName,"r");
        if (pubKeyFile==NULL) return NULL;
        pubKey=(HArrayOfByte *)ArrayAlloc(T_BYTE,256);
                buf=unhand(pubKey)-body;
        while (!feof(pubKeyFile) && counter) {
        buf[counter] = (char)fgetc(pubKeyFile);
        counter++;}
return pubKey;}
```

Создание и обработка объектов Java

Получив представление о том, как обращаться с простыми типами, мы можем теперь начать создание объектов Java внутри наших библиотек. Это самый запутанный этап написания DLL, потому что именно здесь сильнее всего выявляются различия между Java и C. Разрешить эти проблемы можно с помощью структур и функций данных, приведенных в табл. 12-12 и 12-13 соответственно.

Таблица 12-12. Вспомогательные структуры Java

Структура	Описание
ExecEnv	Оболочка выполнения. NULL задает текущую оболочку, обычно как раз она и нужна.
ClassClass	Структура данных, содержащая информацию о классе Java.
Signature	Строка C, описывающая, какие параметры имеет данный конструктор/метод.

Таблица 12-13. Вспомогательные функции Java

Функция	Описание
Hobject *execute_java_constructor(ExecEnv *E, char *className,ClassClass *cb, char *signature	Создает экземпляр className с параметром set, задаваемым сигнатурой.
ClassClass *FindClass(ExecEnv, char *className,bool_t resolve)	Возвращает представление C класса Java className.
long execute_java_static_method(ExecEnv *E, ClassClass *cb,char *methodName, char signature,...)	Реализует статический метод methodName класса Java, описанного с помощью cb.
long execute_java_dynamic_method(ExecEnv *E, HObject *obj,char *methodName,char signature)	Реализует динамический метод объекта Java obj.

Как можно заметить, то, что очень легко на Java, требует большой работы на C. Из табл. 12-13 видно, что необходимо вызывать различные функции для конструкторов, статических методов и для динамических методов.

Разумеется, язык Java также заставляет нас различать типы функций. Но в Java это делается гораздо изящнее, так же как и просмотр оболочкой времени выполнения Java определений класса прежде его создания, что является еще одним способом проверки корректности действий программиста. Когда мы работаем с библиотеками DLL, мы должны иметь дело с FindClass и с функциями execute_java_constructor.

Чтобы заставить наши параметры передаваться правильно, придется совершить некоторые неочевидные действия. Поскольку методы могут перезагружаться, нужно будет явно описать набор параметров, который нам понадобится передавать. Кроме того, необходимо описать типы параметров и возвращаемые значения, чтобы построить соответствующие представления C. Вот тут-то нам и пригодятся строки сигнатур (signature). Наши вспомогательные функции определяют по сигнатуре, что можно ожидать в качестве параметров и что должно быть возвращено.

Сигнатура состоит из ключей, определенных в табл. 12-13, а также внешних и внутренних скобок, содержащих параметры.

Таблица 12-14. Ключи для сигнатур Java

Ключ	Тип Java
Z	boolean
B	byte
C	char
F	float
S	short
V	void
[	массив любого типа
L*;	любой объект Java; * заменяется на classname

Синтаксис сигнатуры имеет следующий вид:
(Parameter Keys) Return Key

Например, метод, не содержащий параметров и ничего не возвращающий, будет иметь сигнатуру ()V. В табл. 12-15 приведены несколько примеров использования сигнатур. Заметьте, что вам не нужно определять тип возвращаемого значения для конструкторов. Будем считать, что класс someClass находится в нашем рабочем каталоге.

Таблица 12-15. Примеры сигнатур

Конструктор/метод	Сигнатура
Hashtable	()
Integer(int j)	(I)
Integer(String S)	(Ljava/lang/String;)
boolean S.beginsWith(String S1, int len)	(Ljava/lang/String;I)Z
someClass(Hashtable H)	(Ljava/util/Hashtable;)
someClass sC.reverse(someClass s)	(LsomeClass;)LsomeClass;

Получив представление о том, как работают сигнатуры, попробуем воспользоваться динамическим методом класса String. Следующая функция производит сравнение со String с помощью функции compareTo из DLL. Будем считать, что "sample" - это имя нашего собственного класса-упаковщика.

```
long sample_CbeginsWith(sample *this,Hjava_lang_String
*S1,Hjava_lang_String *S2,long len) {
char sig[]="(Ljava/lang/String;I)Z";
return execute_java_dynamic_method((ExecEnv
*)NULL,(HObject *) S1,
"beginsWith@,sig,S2,len);}
```

Отметим некоторые особенности, связанные с нашими параметрами для выполнения метода execute_java_dynamic_method. Во-первых, мы передаем NULL в качестве первого параметра. При передаче NULL метод execute_java_dynamic_method будет использовать текущий поток в качестве оболочки выполнения почти всегда, когда это будет нужно. Нашим следующим параметром будет объект, который нужно использовать. При использовании S1 этот собственный метод равносилен S1.compareTo(S2). Следующим параметром является имя метода. Если метод не существует или является статическим, эта функция выдаст исключение. Далее идет сигнатура, которую мы определяем по табл. 12-15. Нашим следующим аргументом будет параметр, который нужно передать, в данном случае S2. Если вы - квалифицированный программист на C, вы заметили по табл. 12-13, что метод execute_java_dynamic_method - функция с переменным числом аргументов. Поскольку мы передаем S2 и len, мы просто приписываем их в конце. Естественно, они должны идти в том же порядке, что и соответствующие ключи в сигнатуре.

ExecEnv и оболочка выполнения

Если вам захочется, вы можете сменить оболочку выполнения и ваш метод будет исполнен не на текущем потоке. Прежде чем вы начнете это делать, помните, что вы имеете доступ к данному потоку Java как к объекту. Это может немного упростить вашу жизнь. Просто обратитесь к

потоку, передав ваш собственный метод как параметр.

Мы передаем параметры конструкторам, динамическим методам и статическим методам одним и тем же способом - создаем сигнатуру и прикрепляем в конце реальные параметры. Конечно, с помощью конструкторов и статических методов мы не можем создать представление объекта, потому что ни одним объект не был реализован. Поэтому методам execute_java_constructor и execute_java_static_method нужна информация о классах.

Структура данных ClassClass содержит информацию о том, какие элементы содержатся в данном классе, и об их атрибутах. Функция FindCLass перенесет эти данные для использования в структуру ClassClass. Применим класс java.awt.Color следующим образом:

```
ClassClass *cb;
cb=FindClass((ExecEnv *)NULL,"java/awt/Color",TRUE);
```

Здесь мы снова устанавливаем значение ExecEnv в нуль. Последний параметр позволяет функции FindClass разрешить этот класс со всеми суперклассами, которые он может иметь. Нет смысла этого не делать. Если мы присвоим классу значение false, мы можем не получить все наши элементы.

Когда у нас есть указатель ClassClass для какого-то класса, мы можем применить методы execute_java_static_method и execute_java_constructor. Ниже показано, как с их помощью преобразуется переменная типа String в int:

```
long sample_strToInt(Hsample *this, Hjava_lang_String *S) {
ClassClass *cb;
char sig[]="(Ljava/lang/String;)I";
cb=FindClass((ExecEnv *)NULL,"java/lang/String",TRUE);
return (long)
        execute_java_static_method((ExecEnv *)NULL,
        "parseInt",sig,S);}
```

Теперь мы можем вернуться к библиотеке DLL, которую мы строим, и реализовать собственный метод favoriteColor. Этот метод возвращает объект Color, а это значит, что мы должны создать его в LocalProfile_favoriteColor. Конструктор, которым мы будем пользоваться, выглядит следующим образом:
Color(int red, int green, int blue)

Пример 12-6. Построение объекта Java на C.

```
struct Hjava_awt_Color *
LocalProfile_favoriteColor(HLocalProfile *this) {
        Hjava_lang_String *RedColorParam;
        Hjava_lang_String *BlueColorParam;
        Hjava_lang_String *GreenColorParam;
        long red; long green; long blue;
        char sig[]="(III)LHjava_awt_Color";
        char intSig[]="(LHjava_lang_String)I";
        ClassClass *Colorcb;
        ClassClass *Integercb;
Colorcb=FindClass((ExecEnv *)NULL,"java/lang/Color",TRUE);
Integercb=FindClass((ExecEnv *)NULL,"java/lang/Integer",TRUE);
RedColorParam=LocalProfile_getAttrib(this,makeJavaString("red",3));
BlueColorParam=LocalProfile_getAttrib(this,makeJavaString("blue",4));
GreenColorParam=LocalProfile_getAttrib(this,makeJavaString("green",5));
red=(long) execute_java_static_method(
        (ExecEnv *)NULL, Integercb, "parseInt", intSig, RedColorParam);
blue=(long) execute_java_static_method(
        (ExecEnv *)NULL, Integercb, "parseInt", intSig, BlueColorParam);
green=(long) execute_java_static_method(
(ExecEnv *)NULL, Integercb, "parseInt", intSig, GreenColorParam);
return (struct Hjava_awt_Color *)
execute_java_constructor((ExecEnv *)NULL,
"Color", Colorcb, sig, red, green, blue);
}
```

Выше мы построили объект и сразу же вернули его. При желании мы могли бы вызвать динамические методы объекта, присвоив его структуре Hjava_lang_Color и затем передав ее методу execute_java_dynamic_method.

Конструкторы и сборка мусора

Сборка мусора осуществляется внутри DLL. Когда вы создаете экземпляр с помощью execute_java_constructor, вам нужно привести его и присвоить соответствующему указателю структуры. Это требование применимо также и к массивам. Если вы используете не ArrayAlloc, а массивы C, сборщик мусора не знает, что он должен сохранять распределенную память.

Выдача исключений

Мы обсудили уже почти все, что позволяет собственным методам вести себя так же, как другие методы. Поняв, как выдавать исключения, вы сможете бесконечно интегрировать свои собственные методы в программу на Java.

Для того чтобы выдавать исключения из DLL, будем использовать функцию SignalError. Она содержит следующие аргументы:

void SignalError(ExecEnv *e, char *ExceptionName, char *description);

Теперь мы можем написать последнюю функцию нашего класса LocalProfile - setAttrib. Этот метод должен читать атрибут "write" из файла profile и проверять, разрешена ли запись. Если запись запрещена, метод выдает исключение.

Пример 12-7. Выдача исключений из DLL.

```
void LocalProfile_setAttrib(struct HLocalProfile *this,
Hjava_lang_String * key, Hjava_lang_String * value
Hjava_lang_String *writeAllowed;
writeAllowed=LocalProfile_getAttrib(this,
makeJavaString("write",5));
if (strcmp("yes",makeCString(writeAllowed))!=0) {
SignalError(0, "profileWriteUnAllowedException", NULL); }
else {writeToProfile(this,key,value);}
}
```

Компиляция и использование DLL

Мы уже написали всю программу для собственных методов; теперь пора ее откомпилировать. Для начала покажем, как это делать в Windows 95/NT. Для компиляции DLL вам понадобится Visual C++ версии 2.0 или выше. Просто перейдите в командную строку DOS и введите следующее:

C: cl LocalProfile.c profile.c -Feprofile.dll -MD -LD javai.lib

Затем добавьте местоположение profile.dll в переменной окружения LIB.

Пользователи Solaris должны ввести следующее (если Java находится в /usr/local/java):

cc -G -I/usr/local/java/include -I/usr/local/java/include/solaris LocalProfile.c profile.c -o libprofile.so

---

СОВЕТ Третий аргумент должен быть приспособлен к различным объектам UNIX. Подробное описание того, как это сделать, вы найдете в документации к вашему JDK.

Когда библиотека DLL откомпилирована, вы готовы к тому, чтобы использовать класс LocalProfile в своей программе. Если вы хотите использовать его с апплетами, класс и библиотека DLL должны быть внутри CLASSPATH на локальной машине. Помните, что возможности связаться с DLL будут зависеть от того, разрешает ли это поддерживающий Java Web-броузер.

Что дальше?

В этой главе мы рассмотрели вопрос о том, как применять Java для доступа к локальной файловой системе и как использовать собственную (native) программу. Отходя от модели апплета, мы можем уменьшить нашу зависимость от Web при развитии Интернет и приложений интранет. В четвертом разделе мы увидим, как можно использовать возможности Интернет во всей их полноте.

Глава 13

Работа с сетью на уровне сокетов и потоков

До сих пор мы обсуждали основные возможности работы с самим языком Java, пакеты, включенные в Java API, работу с графикой и окнами, а также основные свойства потоков в Java. Теперь пора немного расширить наш кругозор и обсудить пакет java.net, который позволит вам сделать свои апплеты и программы на Java открытыми для взаимодействия по сети с другими компьютерами.

Интерес к Java в большой степени объясняется тем, что два свободно доступных Web-броузера, Netscape и Hotjava, могут загружать и выполнять Java-апплеты по Интернет. Апплеты очень тесно связаны с Интернет и вообще с сетями. Поэтому совсем не удивительно, что Java API включает в себя большой пакет сетевых классов. Разумеется, поскольку апплеты не могут загружать данные из файлов локального компьютера, они должны хранить данные в сети и возвращать их оттуда. Сетевые классы, содержащиеся в API, можно разделить на две основные группы: классы, занимающиеся сокетами, и классы, занимающиеся URL. Мы обсудим использование сокетов в этой главе, а использование URL - в главе 14, "Связь по сети с помощью URL".

Сокеты (sockets) не реализуют метод передачи данных - они создают экземпляры более общих классов ввода/вывода, называемых потоками (threads), которые выполняют эту работу. Сокеты можно сравнить с телефонными проводами. Продолжая эту аналогию, можно сравнить выходные потоки с голосом, а входные потоки - с ухом. Сокет переносит данные (ваш голос) по сети; потоки занимаются тем, что закладывают данные в сокет и выдают их наружу. Потоки полезны в своем деле, хотя если вы будете их использовать при программировании апплетов, то скорее всего в комбинации с сокетом или при взаимодействии между апплетами.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Сокеты

Идея сокета неразрывно связана с TCP/IP - протоколом, использующимся в Интернет. По существу, сокет является непрерывной связью данных между двумя хостами сети. Он определяется сетевым адресом конечного компьютера (endpoint), а также портом на каждом хосте. Компьютеры в сети направляют приходящие из сети потоки данных в специальные принимающие программы, присваивая каждой программе отдельный номер - порт программы. Аналогично, когда генерируются выходные данные, программе, инициирующей передачу данных, присваивается номер порта для транзакции. Удаленный компьютер не может ответить на ввод данных. В TCP/IP резервируются определенные номера портов для специальных протоколов - например, 25 для SMTP и 80 для HTTP. Все номера портов, меньшие 1024, зарезервированы на каждом хосте для системного администратора.

Java API реализует класс для осуществления взаимодействия сокетов - java.net.Socket. Следующий фрагмент программы использует самый простой конструктор:

```
try {
         // создание связи между сокетами
         Socet s = new Socket("www.vmedia.com",25)
         /*
                эта часть программы взаимодействует с сокетом
        */
         // закрытие связи между сокетами
         s.close();
} catch (UnknownHostException e) {
         // хост неизвестен
} catch (IOException e) {
         // ошибка ввода/вывода во время связи
}
```

Таблица 13-1. Конструкторы класса Socket

Конструктор	Описание
Socket(String, int)	Имя хоста и порт для связи.
Socket(String, int, boolean)	Имя хоста, порт и булевский указатель сокета: для потоков (true) или для датаграмм (false).
Socket(InetAddress, int)	Интернетовский адрес и порт для связи.
Socket (InetAddress, int, boolean)	Интернетовский адрес, порт и булевский указатель сокета: для потоков (true) или для датаграмм (false).

При создании сокета можно указать, с каким хостом соединяться: либо с помощью переменной типа String, содержащей имя хоста, либо заданием специального класса, java.net.InetAddress. Какая разница между этими двумя способами? Для того чтобы полностью это понять, вам понадобятся некоторые знания о TCP/IP.

Каждому хосту в сети, осуществляющей связь по протоколу TCP/IP, в том числе Интернет, присвоен уникальный численный идентификатор, называемый IP-адресом. IP-адрес состоит из четырех байтов и в общем случае представляется восьмеричным числом с точками - например, 127.0.0.1. Хотя IP-адреса очень удобны для общения между собой компьютеров, людям запоминать их и регулярно использовать очень трудно. Чтобы облегчить использование ресурсов сети, в Интернет используется система DNS (Domain Name System). Человек задает имя хоста, и его компьютер запрашивает местный DNS-сервер, который определяет по данному имени IP-адрес. Класс Socket позволяет задать либо имя хоста в форме строки, либо IP-адрес в форме InetAddress.

СОВЕТ InetAddress для данного сокета можно определить, воспользовавшись методом getInetAddress, определенном на классе Socket. Если вы хотите открыть новое соединение с той же машиной, возможно, немного быстрее будет вместо имени хоста воспользоваться InetAddress, чтобы избежать дополнительного преобразования DNS.

Вас, возможно, удивляет, почему пакет java.net использует класс InetAddress, а не 4-битовый массив, содержащий IP-адрес. Дело в том, что класс InetAddress дополнительно предоставляет методы, которые действуют на IP-адрес; например, метод getHostName выполняет обратное преобразование DNS на классе InetAddress и возвращает имя хоста. InetAddress не содержит общих конструкторов, но зато он дает две статические функции: getByName и getAllByName, которые берут имя хоста и возвращают все InetAddress, относящиеся к этому хосту.

Сокеты и обеспечение безопасности апплета

Netscape Navigator 2.0 не разрешает ненадежным апплетам, загруженным с удаленных серверов, открывать сокеты к любой машине в Интернет. Апплет может открыть только сокеты к хосту, с которого он был загружен. Это свойство не позволяет апплетам генерировать нежелательное или незамеченное перекачивание данных с каждой машины, на которой они запускаются. Надежные апплеты, проверенные цифровой авторизацией, имеют меньше ограничений, чем ненадежные. Они получают возможность взаимодействовать с любым хостом в Интернет. Однако к моменту написания этой книги компания Sun еще не изобрела и не разработала механизма идентификации надежных апплетов, так что все удаленные апплеты пока считаются ненадежными.

Ограничения, связанные с обеспечением безопасности апплетов, не применяются к апплетам, загруженным из каталога, относящегося к локальному классу данного пользователя. Один из способов, с помощью которого пользователи могут обойти это ограничение, - установить апплет в каталогах своих локальных классов. Другой способ - написать сервер, который запускается на том хосте, где находится апплет, и перенаправляет данные, приходящие на определенный порт, по новому назначению. Третий способ - иметь на одной Web-странице множество апплетов, каждый из которых будет находиться на том хосте, с которым должен взаимодействовать апплет. Поскольку апплеты, находящиеся на одной Web-странице, могут взаимодействовать между собой, каждый апплет сможет связаться со всеми хостами других апплетов. Правда, многие операционные системы ограничивают число поддерживаемых соединений, так что последний метод может не сработать.

В табл. 13-12 перечислены методы для класса Socket. Особенно важны методы getInputStream и getOutputStream, потому что их используют для реальной связи с удаленным хостом. Вообще потоки осуществляют перенос данных; в примере 13-1 мы используем их для переноса данных между хостами по сети.

Таблица 13-2. Методы класса Socket

Метод	Описание
close()	Закрывает сокет.
InetAddress getInetAddress()	Возвращает интернетовский адрес компьюте-ра на другом конце сокета.
int getLocalPort()	Возвращает номер локального порта, с кото-рым связан данный сокет.
InputStream getInputStream	Возвращает InputStream, относящийся к данному сокету.

OutputStream getOutputStream()	Возвращает OutputStream, относящийся к данному сокету.
setSocketImplFactory(SocketImplFactory)	Устанавливает фактор реализации сокета для данной системы.

СОВЕТ Если ваша операционная система не UNIX и вы загружаете апплет временного клиента с локального жесткого диска или с Companion CD-ROM, пример 13-1 не будет работать (так же как и другие примеры этой главы), если только у вас случайно не сервер времени. Чтобы посмотреть на этот апплет в работе, сходите на http://www.vmedia.com/vvc/onlcomp/java/chapter13/example1/TimeApplet.htm.

Следующий пример выполняет простой временной клиент. Клиент соединяется с сервером времени на том хосте, откуда был загружен апплет. Сервер времени ждет, пока установится соединение, сообщает текущее время и закрывает связь. Все, что должен сделать наш клиент, - это открыть связь и прочесть данные.

Пример 13-1. Временной клиент.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.io.*;
public class TimeApplet extends Applet {
        private Exception error;
        private String output;
```

В примере, приведенном ниже, мы попытаемся установить связь с удаленным хостом на порту 13 - это порт сервера времени. Если соединение будет установлено, мы попытаемся связать InputStream и Socket. Если мы столкнемся с какой-то исключительной ситуацией (UnknownHostException или IOException), произойдет быстрый возврат. Соединившись с InputStream, мы прочтем строку и закроем связь:

```
public void start() {
        Socket TimeSocket;
        InputStream TimeInput;
        try {
                TimeSocket = new Socket(getCodeBase().getHost(),13,true);
                TimeInput = TimeSocket.getInputStream();
        } catch (Exception e) {
                error = e;
                return;
        }
        output = readString(TimeInput);
        try {
                TimeInput.close();
                TimeSocket.close();
        } catch (IOException e) {}
}
```

Теперь мы готовы к тому, чтобы прочесть выходные данные с сервера времени. Присвоим начальные значения массиву байтов, который должен содержать эти данные, и прочтем информацию из InputStream по одному байту в единицу времени. Когда поток заполнится, метод чтения InputStream вернет -1 (после того, как сокет закроется на удаленном хосте). Для простоты будем считать, что входные данные будут занимать меньше 50 байтов, хотя позже мы покажем вам лучшие способы получать неизвестное количество данных с сервера.

```
private String readString(InputStream in) {
        byte ary[] = new byte[50];
        byte buf;
        int count = 0;
        try {
                buf = (byte)in.read();
                while (buf!=-1) {
```

```
                        ary[count] = buf;
                        count++;
                        buf = (byte)in.read();
                }
        } catch (IOException e) {
                error = e;
                return null;
        }
        return new String(ary,0).trim();
}
```

Метод paint выдаст выходные данные с сервера времени, если все прошло хорошо, или информацию об ошибке, если что-то было не так:

```
        public void paint(Graphics g) {
                g.setColor(Color.white);
                g.fillRect(0,0,499,249);
                g.setColor(Color.black);
                g.drawRect(0,0,499,249);
                if (error!=null) {
                        g.drawString(error.toString(),25,25);
                        g.drawString(error.getMessage(),25,25);
                } else {
                        g.drawString(output,25,25);
                }
        }
}
```

С сокетами особенно легко работать на Java; вся работа сводится к оперированию с входными и выходными данными. С сервером времени легко взаимодействовать потому, что он не требует входных данных. Чтобы показать, как взаимодействовать с более функциональными серверами, нам придется более глубоко исследовать классы потоков. Прежде чем мы перейдем к потокам, остановимся коротко на классах API, позволяющих использовать датаграммы для сетевого общения.

Реализация сокетов

Есть один метод в классе Socket, который мы еще не использовали, - это метод setSocketImplFactory. Сам класс Socket может делать немного больше, чем создавать согласованный интерфейс для связи с удаленными хостами. Каждый Socket содержит частный SocketImpl - другой класс в пакете java.net.package, который реально выполняет работу по соединению и перемещению данных на удаленный хост и обратно. Определенный по умолчанию класс SocketImpl создан для того, чтобы работать с обычными данными по протоколу TCP/IP, но вы можете захотеть воспользоваться другим протоколом, как, например, Apple Talk или IPX.

Можно написать класс, реализующий SocketImplFactory, и интерфейс, позволяющий классу Socket запросить SocketImpl. Новый класс SocketImplFactory создаст классы SockImpls, которые могут передавать данные по протоколам IPX или Apple Talk, но при этом могут использоваться стандартным классом Socket, не требуя от пользователей изучения целого набора новых методов. Такая техника послойного использования уровней абстракции вполне обычна при работе с сетевыми компьютерами.

Несвязываемые датаграммы

Протокол TCP/IP поддерживает также доставку несвязываемых датаграмм и их возврат с помощью UDP (User Datagram Packet, "Пакет пользовательских датаграмм"). Датаграммы UDP, так же как сокеты TCP, позволяют связаться с удаленным хостом по протоколу TCP/IP. В отличие от сокетов TCP, осуществляющих связь с логическим соединением, датаграммы UDP связываются без логического соединения. Если сокет TCP можно сравнить с телефонным звонком, то датаграмму UDP можно сравнить с телеграммой. Доставка датаграммы UDP не гарантирована, и даже если такая датаграмма доставлена, нет гарантии, что она доставлена правильно. Если вы решили применить в своей программе датаграммы, вам придется иметь дело с утерянными или неупорядоченными пакетами.

Из-за ненадежности UDP большинство программистов при написании сетевых приложений

предпочитают работать с сокетами TCP. Тем не менее вы можете решить использовать датаграммы, если захотите написать клиент Java для существующего UDP-сервера, например NFS версии 2. Кроме того, у датаграмм служебный заголовок (overhead) занимает меньше места, чем у сокета TCP, потому что при посылке или получении датаграмм не нужны механизмы ни подтверждения связи (handshaking), ни управления потоком данных (flow control). Кроме того, датаграммы UDP можно использовать на сервере Java, посылающем периодически обновляемую информацию клиентам. Если сервер вместо того, чтобы устанавливать TCP-соединение для каждого клиента, будет просто посылать датаграммы UDP, он будет выполнять меньше работы и, соответственно, работать быстрее.

Два класса в Java ассоциируются с датаграммами UDP - DatagramPacket и DatagramSocket. Класс DatagramPacket используется для посылки или получения датаграмм UDP. Конструкторы и методы для DatagramPacket приводятся в табл. 13-3.

Таблица 13-3. Конструкторы и методы класса DatagramPacket

Конструкторы и методы	Описание
DatagramPacket(byte[], int, InetAddress, int)	Создает пакет для посылки датаграммы. Содержимое массива byte посылается на порт удаленного хоста, заданный вторым целым и InetAddress. Первое целое задает число посылаемых байтов.
InetAddress getAddress()	Возвращает интернетовский адрес пакета.
int getPort()	Возвращает номер порта пакета.
byte[] getData()	Возвращает данные пакета.
int getLength()	Возвращает длину пакета.

Когда вы сконструируете класс DatagramPacket, вам понадобится DatagramSocket, чтобы посылать и получать датаграммы. Эти сокеты привязаны к определенному порту на вашем компьютере, совсем как сокеты TCP. Номера портов, меньшие 1024, зарезервированы для пользователя root на компьютерах с UNIX. Конструкторы и методы для класса DatagramSocket приводятся в табл. 13-4.

Таблица 13-4. Конструкторы и методы класса DatagramSocket

Конструкторы и методы	Описание
DatagramSocket()	Создает сокет UDP на незаданном свободном порту.
DatagramSocket(int)	Создает сокет UDP на заданном порту.
receive(DatagramPacket)	Ждет получения датаграммы и копирует данные в специальный DatagramPacket.
send(datagramPacket)	Посылает DatagramPacket.
getLocalPort()	Возвращает номер локального порта, к которому привязан сокет.
close()	Закрывает сокет.

Мы воспользуемся этими классами для реализации UDP-клиента. Соединимся с UDP эхо-сервером, который на большинстве машин с UNIX находится на порту 7. Эхо-сервер принимает каждую полученную UDP-датаграмму и возвращает ее копию в виде другой UDP-датаграммы туда, откуда первая датаграмма была послана. Вот текст программы нашего эхо-клиента.

Пример 13-2. Апплет датаграммы.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class DatagramApplet extends Applet {
        private InetAddress addr;
        private String message;
        private TextArea output;
        public void init() {
                try {
                        String me = detCodeBase().getHost();
                        addr = InetAddress.getByName(me);
                } catch (Exception e) {
                        handleException(e);
                }
```

```
                message = "abcde";
                output = new TextArea(5,60);
                output.setEditable(false);
                add(output);
                show();
        }
        public void start() {
                byte b[] = new byte[message.length()];
                message.getBytes(0,message.length(),b,0);
                try {
                        DatagramSocket ds
                        ds = new DatagramSocket();
                        DatagramPacket outPacket;
                        outPacket = new DatagramPacket(b,b.length,addr,7);
                        b = new byte[message.length()];
                        DatagramPacket inPacket;
                        inPacket = new DatagramPacket(b,b.length);
                        ds.send(outPacket);
                        output.appendText("Sent: "+message+"\n");
                        ds.receive(inPacket);
                        b = inPacket.getData();
                        output.appendText("Rec'd: "+new String(b,0)+"\n");
                        ds.close();
                } catch (Exception e) {
                        handleException(e);
                }
        }
        public void stop() {
                output.setText("");
        }
        public void handleException(Exception e) {
                System.out.println(e.toString());
                e.printStackTrace();
        }
}
```

Все же несмотря на то, что датаграммы иногда бывают полезны, из-за отсутствия у них служебного заголовка они не могут заменить такой сетевой механизм, поддерживающий надежную связь, как сокеты TCP. Однако, как мы видели раньше, сокеты, выполняемые в Java, тесно связаны с потоками. Чтобы научиться эффективно применять сокеты, необходимо научиться пользоваться потоками.

Потоки

Java API для управления входными и выходными данными программ использует потоки. Поток - это, по сути, метод передачи данных от программы на какое-то внешнее устройство, например в файл или сеть. На одном конце вы помещаете нечто, и оно приходит на другой конец (рис. 13-1). Основными классами потоков являются InputStream и OutputStream, входящие в пакет java.io. В данном разделе мы также рассмотрим богатый пакет расширенных типов потоков, которые значительно упрощают передачу данных по сравнению с использованием непосредственно байтов.

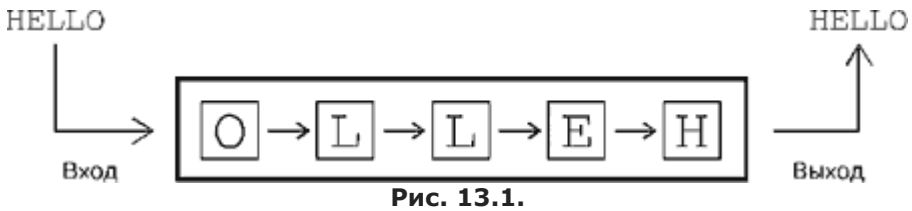

Рис. 13.1.

Входные потоки

При передаче входных данных с удаленного устройства возникают некоторые трудности. Большинство входных потоков реально передается через устройство, которое может работать значительно медленнее, чем использующая их программа. Не самым лучшим (хотя и самым простым) программистским решением было бы ждать медленной или уже закончившейся передачи данных. Скорость связи по сети варьируется в широких пределах, часто при прохождении данных передача зависает на несколько секунд или минут. Класс InputStream построен таким образом, чтобы относительно легко уменьшить подобные задержки.

Класс InputStream определяет метод, возвращающий число байтов, которое можно прочесть из потока без блокировки, то есть за то свободное время, пока приходят новые данные. Методы чтения потока осуществляют блокировку до тех пор, пока не станет доступна хотя бы часть из запрошенных данных. Это означает, что если вы попытаетесь прочесть данные, которые еще не появились, то поток, управляющий проходом данных, дождется, пока хотя бы часть данных придет, после чего продолжит работу. Если данные так и не придут, поток будет ждать до тех пор, пока не закончится время работы сокета. Если достигнут конец потока данных, метод чтения возвращает -1. При возникновении ошибки все методы выдают исключение IOException. Методы класса InputStream приводятся в табл. 13-5.

Таблица 13-5. Методы класса InputStream

Методы	Описание
int available()	Возвращает число байтов, которое можно прочесть в потоке данных без блокировки.
int read()	Возвращает следующий байт из потока данных.
int read(byte[])	Заполняет все свободное место в массиве байтов потока данных и возвращает число реально прочитанных данных.
int read(byte[],int,int)	Заполняет все свободное место в массиве байтов потока данных. Первое целое указывает на смещение, с которого нужно начинать читать, второе целое указывает максимальное число байтов, которые нужно прочесть.
long skip(long n)	Пропускает n байтов входных данных и возвращает число реально пропущенных байтов.
close()	Закрывает поток.
mark(int)	Отмечает текущую позицию в потоке данных. Целый аргумент обозначает минимальное число байтов, которые нужно прочесть до исчезновения этой отметки.
reset()	Возвращает поток данных к последней отмеченной позиции.
boolean markSupported()	Показывает, поддерживает ли данный поток данных отметку и методы сброса.

Выходные потоки

Записать данные в поток гораздо проще, чем получить входные данные. Класс OutputStream состоит из трех методов записи, метода flush, записывающего все выходящие из буфера данные в поток, и метода close, закрывающего поток. Методы записи осуществляют блокировку до тех пор, пока данные реально не запишутся. Эти методы приведены в табл. 13-6. При возникновении ошибки все эти методы выдают IOException.

Таблица 13-6. Методы класса OutputStream

Метод	Описание
close()	Закрывает поток.
flush()	Освобождает выводной буфер от всех данных.
write(int)	Записывает байт в поток.
write(byte[])	Записывает массив байтов в поток.
write(byte[],int,int)	Записывает подмассив данного массива в поток. Первое целое указывает смещение, второе целое указывает длину подмассива.

При использовании класса OutputStream мы можем соединяться с сервером, требующим входные данные. Например, это может быть сервер Finger, работающий на большинстве компьютеров с UNIX. После установления связи клиент сообщает имя пользователя, и сервер выдает в ответ информацию об этом пользователе. Вот текст программы простого апплета клиента Finger.

Пример 13-3. Апплет клиента Finger.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.io.*;
```

Вместо drawString воспользуемся полем TextArea, потому что сервер Finger часто возвращает несколько строк выходной информации. Для простоты зададим имя пользователя, о котором мы запрашиваем информацию, в качестве параметра к Web-странице. Поскольку здесь используется стандартный выходной поток, то перед посылкой мы должны конвертировать строку в массив байтов, хотя ниже в этой главе мы обсудим подклассы класса OutputStream, которые могут записывать переменные типа String непосредственно. Байт EOL используется при синтаксическом анализе выходных данных с сервера Finger для установления конца строки:

```
public class FingerApplet extends Applet {
        private TextArea output;
        private byte user[];
        private byte EOL;
        public void init() {
                String userStr
                userStr = getParameter("USER")+"\n";
                user = new byte[userStr.length()];
                userStr.getBytes(0,userStr.length(),user,0);
                output = new TextArea(10,80);
                output.setEditable(false);
                add(output);
                show();
                String eolStr = new String("\n");
                byte eolAry[] = new byte[1];
                eolStr.getBytes(0,1,eolAry,0);
                EOL = eolAry[0];
        }
```

Раздел программы, приведенный ниже, почти идентичен TimeClientApplet - серверу Finger, находящемуся на порту 79, и нам понадобятся как класс OutputStream, так и класс InputStream. После установления соединения имя разыскиваемого пользователя посылается в виде массива байтов. Ниже приведен еще один метод, readText, копирующий содержимое InputStream в TextArea:

```
public void start() {
        Socket FingerSocket;
        InputStream FingerInput;
        OutputStream FingerOutput;
        try {
                FingerSocket = new Socket(getCodeBase().getHost(),79);
                FingerInput = FingerSocket.getInputStream();
                FingerOutput = FingerSocket.getOutputStream();
                FingerOutput.write(user);
        } catch (Exception e) {
                output.appendText(e.toString()+"\n");
                output.appendText(e.getMessage()+"\n");
                return;
        }
        readText(FingerInput,output);
}
```

Теперь мы готовы к приему данных с сервера Finger. Для хранения каждой строки данных воспользуемся массивом байтов. При достижении символа EOL или при переполнении длины массива (задаваемой числом колонок в поле TextArea) копируем массив байтов в TextArea и создаем новый пустой массив байтов. Это более сложный способ оперирования с неизвестным количеством данных, чем тот, который использовался в первом примере, когда мы просто задали верхнюю границу и создали единичный массив:

```
private void readText(InputStream in, TextArea text) {
        int bufsize = text.getColumns();
        byte ary[] = new byte[bufsize];
        byte buf;
        int count = 0;
        try {
                buf = (byte)in.read();
                while (buf!=-1) {
                        ary[count] = buf;
                        count++;
                        if (buf==EOL || count==bufsize) {
                        String aryStr
                                aryStr = new String(ary,0);
                        text.appendText(aryStr);
                        ary = new byte[80];
                        count = 0;
                        }
                        buf = (byte)in.read();
                }
        } catch (IOException e) {
                text.appendText(e.toString()+"\n");
                text.appendText(e.getMessage()+"\n");
                return;
        }
        text.appendText(new String(ary,0));
}
```

Когда апплет заканчивается, поле TextArea просто очищается для последующего применения:

```
        public void stop() {
                output.setText("");
        }
}
```

Разнообразие потоков

Пакет java.io предоставляет гораздо больше возможностей, чем просто классы InputStream и OutputStream. Оба эти класса содержат по нескольку подклассов в API, позволяющих программистам взаимодействовать с потоками на гораздо более высоком уровне, в результате чего проще выполнять синтаксический анализ и посылать данные. Эти подклассы кратко описаны в табл. 13-7. Если не оговорено специально, каждый входной поток имеет соответствующий выходной поток.

Таблица 13-7. Специализированные подклассы класса InputStream

Подкласс	Описание
FilteredInputStream	Отфильтрованный входной поток байтов.
BufferedInputStream	Входной поток байтов с буфером. Использование буфера может ускорить доступ к потоку, поскольку байты записываются порциями. Этот класс является подклассом класса FilteredInputStream.
FileInputStream	Позволяет потокам соединяться с файлами.
PipedInputStream	В сочетании с PipedOutputStream разрешает направленный в одну сторону поток данных между программными потоками.
LineNumberInputStream	Следит за номерами строк потока. Этот класс является подклассом класса FilterInputStream и не имеет соответствующего OutputStream.
PushbackInputStream	Позволяющий перемотать поток на начало. Этот класс является подклассом класса FilterInputStream и не имеет соответствующего OutputStream.
SequenceInputStream	Позволяет связать вместе несколько потоков InputStream и прочесть данные из них в режиме карусели. Не имеет соответствующего OutputStream.
StringBufferInputStream	Позволяет обращаться с переменными типа String так же, как с

	InputString. Не имеет соответствующего OutputStream.
ByteArrayInputStream	Позволяет обращаться с массивом байтов так же, как с InputString.
PrintStream	Позволяет печатать простые типы данных как текст. Не имеет соответствующего OutputStream.

Большинство специализированных потоков из описанных в табл. 13-7 можно построить над обычными потоками. Например, для создания BufferOutputStream из обычного OutputStream нужно сделать следующее:

```
OutputStream out = mySocket.getOutputStream();
BufferedOutputStream bufferOut;
bufferOut = new BufferedOutputStream(out);
```

Теперь все, что вы напишете с помощью потока bufferOut, будет вписано в начальный OutputStream и при этом будет находиться в буфере.

Потоки данных

Из всего разнообразия потоков двумя наиболее важными являются DataInputStream и DataOutputStream, которые можно построить над обычными потоками InputStream и OutputStream. Эти потоки позволят вам как брать переменные примитивных типов Java (int, String и т. д.) прямо из потока, не заботясь о массиве байтов, так и записывать переменные прямо в поток. Воспользовавшись DataInputStream, мы можем сократить метод readString из апплета клиента времени. Вместо того чтобы считывать данные из потока в массив байтов и затем помещать их в строку, мы можем считывать строку прямо из потока с помощью метода readLine.

Пример 13-4. Апплет клиента времени, использующий DataInputString.

```
private String readString(InputStream in) {
        DataInputStream dataIn;
        dataIn = new DataInputStream(in);
        try {
                return dataIn.readLine();
        } catch (IOException e) {
                error = e;
                return null;
        }
}
```

В табл. 13-8 перечислены новые методы, добавляющиеся с потоком DataInputStream. Все эти методы выдают исключение IOException при возникновении ошибки. Методы readFully выдают java.io.EOException, если встречается EOF до наполнения массива байтов. Эти методы отличаются от методов read(byte[]) и read(byte[],int,int) исходного потока InputStream тем, что они осуществляют блокировку до тех пор, пока не станут доступными все запрошенные данные или пока не встретится EOF. Аналогично, метод skipBytes блокирует до тех пор, пока не будут пропущены все запрошенные байты.

Таблица 13-8. Методы класса DataInputStream

Метод	Описание
boolean readBoolean()	Читает булевскую переменную.
int readByte()	Читает байт (8 бит).
readUnsignedByte()	Читает байт без знака (8 бит).
readShort()	Читает короткое целое (16 бит).
readUnsignedShort()	Читает короткое целое без знака (16 бит).
readChar()	Читает символ (16 бит).
readInt()	Читает целое (32 бита).
readLong()	Читает двойное целое (64 бита).
readFloat()	Читает число с плавающей точкой (32 бита).
readDouble()	Читает число двойной точности с плавающей точкой (64 бита)

readLine()	Читает строку с исключениями \r, \n, \r\n или EOF.
readUTF()	Читает строку в формате UTF.
readFully(byte[])	Считывает байты в специальный массив байтов; блокирует до заполнения массива.
readFully(byte[],int,int)	Считывает байты в специальный массив байтов; блокирует до заполнения подмассива.
skipBytes(int n)	Пропускает n байтов; блокирует, пока не пропущены все байты.

СОВЕТ При использовании readByte и парного ему writeByte не требуется конвертировать байты в сетевой порядок и обратно. Байты записываются и читаются строго в том порядке, в каком они находятся в виртуальной машине Java.

Потоки DataOutputStream работают так же, как и DataInputStream, только в обратном порядке. Можно записать переменные, за исключением строк, прямо в поток, не конвертируя их предварительно в массивы байтов. В табл. 13-9 приводятся новые методы, добавляемые классом DataOutputStream.

Таблица 13-9. Методы класса DataOutputStream

Метод	Описание
writeBoolean(boolean)	Записывает булевскую переменную.
writeByte(int)	Записывает байт (8 бит).
writeChar(int)	Записывает символ (16 бит).
writeShort(short)	Записывает короткое целое (16 бит).
writeInt(int)	Записывает целое (32 бита).
writeLong(long)	Записывает двойное целое (64 бита).
writeFloat(float)	Записывает число с плавающей точкой (32 бита).
writeDouble(double)	Записывает число двойной точности с плавающей точкой (64 бита).
writeBytes(String)	Записывает строку в байтах.
writeChars(String)	Записывает строку в символьной форме.
writeUTF(String)	Записывает строку в формате UTF.

Имейте в виду, что методы DataInputStream не выполняют синтаксический анализ входящего текста на обнаружение символов, похожих на простые типы Java, а методы DataOutputStream не записывают текстовое представление проходящих через них переменных. Эти методы читают и записывают переменные в поток в точности так, как они представлены в виртуальной памяти Java в машине.

Например, когда вы вызываете readInt, он читает 32 бита из потока, после чего возвращает целое, соответствующее этим 32 битам. Он не анализирует строку текстовых данных, чтобы обнаружить нечто, похожее на целое число. Аналогично, метод writeInt потока DataOutputStream не запишет в поток число "10", если вы передаете целое, значение которого не равно 10. Он запишет последовательность из 32 бит, которая при интерпретации как целое число выдаст значение 10. Если вы хотите анализировать синтаксис текста для выявления чисел, вы можете либо считать текстовые данные в строку и воспользоваться StringTokenizer, либо воспользоваться классом StreamTokenizer, который мы обсудим в следующем разделе.

Разбор данных текстового потока

Класс StreamTokenizer, аналогично классу DataInputStream, строится над обычным потоком InputStream. StreamTokenizer отличается от других расширений типов InputStream тем, что он не является непосредственно подклассом класса InputStream, - если вы пользуетесь классом StreamTokenizer, вы не сможете использовать для него обычные методы InputStream. Но зато Stream Tokenizer позволит вам прочитать некоторые простые типы данных Java, а именно строки и числа с плавающей точкой двойной точности, из текстового потока данных. Вам понадобится это свойство для связи с сервером, передающим данные в текстовом виде, например MUD.

Класс StreamTokenizer разделяет поток InputStream на куски, называемые токенами (tokens). Этот класс различает пять видов токенов: слова, числа, символы EOL, символы EOF, обычные символы. Пусть, например, поток содержит следующие данные:
You are user 9 of 100:

StreamTokenizer вернет из этого потока семь токенов. "You", "are", "user" и "of" будут

восприняты как строки, "9" и "100" будут распознаны как числа, а ":" будет возвращено как обычный символ.

Рабочим методом класса StreamTokenizer является метод nextToken. Он читает следующий токен из потока и возвращает целое, указывающее, какой тип токена он нашел. Если токен является словом, числом, EOL или EOF, соответственно целое будет TT_WORD, TT_NUMBER, TT_EOL или TT_EOF. Эти переменные TT_* определены как статические целые в классе StreamTokenizer. Если токен является обычным символом, метод возвращает значение этого символа. Если типом токена является TT_WORD, реальная строка помещается в переменную sval класса StreamTokenizer. Если типом токена является TT_NUMBER, реальное значение типа double помещается в переменную nval класса StreamTokenizer, принадлежащую к этому типу.

По умолчанию класс StreamTokenizer понимает числа как последовательность цифр, возможно, разделенную точкой, заканчивающуюся с каждой стороны пробелом или обычным символом. Слова он понимает двумя способами - как последовательность букв алфавита, заканчивающуюся с каждой стороны пробелом или обычным символом, или как нечто, содержащее внутри одинарные или двойные кавычки.

Существует много способов использовать класс StreamTokenizer. Вы можете задать новый набор символов, распознаваемых как символы слов, как символы пробела или как обычные символы. Можно настроить класс StreamTokenizer так, чтобы он игнорировал комментарии. Можно задать единичный символ как символ комментария, и каждый раз, как StreamTokenizer будет читать этот символ, он будет игнорировать его и все, что идет за ним, до ближайшего символа EOL. Этот класс может также понимать комментарии вида /*...*/ и // ... Можно даже задать новые символы разделения строк. Рассмотрим простой пример. Допустим, ваша входная информация выглядит следующим образом:

```
#Это комментарий
Это группа слов?
:Это целая строка!!!:
```

По умолчанию Stream Tokenizer поймет "#", "?" и "!" как обыкновенные символы, а все остальное - как слова. Допустим, мы создали StreamTokenizer и модифицировали его следующим образом:

```
InputStream in = mySocket.getInputStream();
StreamTokenizer st = new StreamTokenizer(in);
st.commentChar('#');
st.wordChar('?','?');
st.quoteChar(':');
```

Теперь StreamTokenizer полностью проигнорирует первую строку, вернет "слов?" как целое слово и сочтет, что все, что находится между двоеточиями в третьей строке, является строкой. Для лучшего знакомства с методами класса StreamTokenizer рассмотрим табл. 13-10.

Таблица 13-10. Методы класса StreamTokenizer

Метод	Описание
int nextToken()	Читает следующий токен из входного потока и возвращает целое, показывающее тип найденного токена.
commentChar(int)	Задает специальный символ для открывания комментария длиной в одну строку.
quoteChar(int)	Задает специальный символ для разделения строк. nextToken, дойдя до строки, разделенной этим символом, вернет сам символ (а не TT_WORD) и поместит содержимое строки в переменную sval.
whitespaceChars(int,int)	Задает символы из области допустимых значений, которые должны трактоваться как пробел.
wordChars(int,int)	Задает символы из области допустимых значений, которые должны восприниматься как слова.
ordinaryChar(int)	Задает специальный символ, который должен распознаваться как обычный; nextToken, дойдя до этого символа, вернет его значение.
ordinaryChars(int,int)	То же, что ordinaryChar(int), за исключением того, что определена область допустимых значений.
resetSyntax()	Объявляет все символы специальными.
parseNumbers()	Требует, чтобы все числа синтаксически анализировались. Если eolIsSignificant(boolean) имеет значение true, nextToken должен

	вернуть TT_EOL; если false, символы EOL будут пропускаться как пробел.
slashStarComments(boolean)	Если значение равно true, nextToken распознает комментарии в форме /*...*/ и игнорирует их.
lowerCaseMode(boolean)	Если значение равно true, то прежде, чем поместить слова в переменную sval, переводит заглавные буквы в строчные.
pushBack()	Выталкивает последний токен обратно в поток.
int lineno()	Возвращает текущий номер строки.
String toString()	Возвращает текущий токен в формате строки.

Класс StreamTokenizer очень мощный и гибкий. Но пусть вас не пугает широкий спектр его возможностей. Стандартная конфигурация хорошо работает с большинством потоков InputStream. С помощью StreamTokenizer можно научить апплет клиента времени выполнять разбор входных данных при записи их в переменные. Вспомним, что сервер времени возвращает строку в следующей форме:
Sun Jan 21 19:15:01 1996

Мы можем заставить StreamTokenizer разбить эту информацию на две строки, содержащих день и месяц, и на пять значений двойной точности, содержащих число, часы, минуты, секунды и год. Поскольку в новой программе мы использовали класс Vector, нам нужно добавить в начало программы "import.java.util.*". Вот улучшенный метод start и новый метод readTokens.

Пример 13-5. Разбор данных с сервера времени.

```
public void start() {
        Socket TimeSocket;
        StreamTokenizer st;
        try {
                TimeSocket = new Socket(getCodeBase().getHost(),13,true);
                st = new StreamTokenizer(TimeSocket.getInputStream());
        } catch (Exception e) {
                error = e;
                return;
        }
```

Мы хотим проигнорировать колонки в выходных данных сервера времени, поэтому при разборе объявим их пробелами. Чтобы упростить анализ, мы написали метод readTokens, который, беря класс StringTokenizer и вектор из значений типа String и Double, пытается отобрать из потока значения этих типов. Токены должны присутствовать в потоке в том же порядке, в каком они присутствуют в векторе, и разделяться только пробелами. В противном случае произойдет новое исключение. Сейчас мы присвоим начальные значения вектору с тем, чтобы он содержал две строки и пять значений типа Double:

```
        st.whitespaceChars(':',':');
        Vector tokens = new Vector(7);
        tokens.addElement(new String());
        tokens.addElement(new String());
        tokens.addElement(new Double(0));
        tokens.addElement(new Double(0));
        tokens.addElement(new Double(0));
        tokens.addElement(new Double(0));
        tokens.addElement(new Double(0));
        try {
                readTokens(st,tokens);
        } catch (Exception e) {
                error = e;
                return;
        }
        output = "The data is "+tokens.elementAt(0)+", ";
        output = output + tokens.elementAt(1)+" "+tokens.elementAt(2);
        output = output + ", "+tokens.elementAt(6);
        try {
                TimeSocket.close();
        } catch (Exception e) {}
```

```
}
```

Ниже мы приводим новый метод для чтения токенов. Он проходит вектор элемент за элементом, считывая токены из потока и устанавливая соответствующие элементы вектора. Если тип токена, прочитанного из потока, не соответствует типу элемента Object, определенного в векторе, или если вектор содержит тип Object, который мы не можем распознать, выдается следующее исключение:

```
private void readTokens(StreamTokenizer st, Vector v) throws Exception {
        int token;
        int i = 0;
        Enumeration en = v.elements();
        while (en.hasMoreElements()) {
                // это тип объекта, который нам нужен
                Object o = en.nextElement();
                // это следующий токен из потока
                token = st.nextToken();
                // если выходным типом является тип String
                if (o instanceof String) {
                // если токен не является словом
                if (token!=st.TT_WORD) {
                        Exception e;
                        e = new Exception("Unexpected token type: "+token);
                        throw e;
                        // все в порядке, заполняем вектор
                } else {
                        v.setElementAt(st.sval,i);
                }
                // если выходным типом является тип Double
        } else if (o instanceof Double) {
                // если токен не является числом
                if (token!=st.TT_NUMBER) {
                        Exception e;
                        e = new Exception("Unexpected token type: "+token);
                        throw e;
                        // все в порядке, заполняем вектор
                } else {
                        Double d = new Double(st.nval);
                        v.setElementAt(d,i);
                }
        }
        // если выходным типом является недопустимый тип
        else {
                Exception e;
                String type = o.getClass().getName();
                e = new Exception("Cannot parse for "+type);
                throw e;
        }
        i++;
        }
}
```

Взаимодействие InterApplet с каналами

Java API реализует связь потоков данных между различными потоками (threads). Это свойство можно использовать для того, чтобы, не создавая жесткой и сложной системы методов, дать возможность потокам общаться. Вы можете создать пару канальных (piped) потоков данных, PipedInputStream и PipedOutputStream. Все, что будет записано в PipedOutputStream, будет получено на соединенном с ним PipedInputStream. Один поток может построить пару канальных потоков данных и отдать один из них другому потоку. Теперь потоки смогут сообщаться в одном направлении через поток данных. Если потребуется связь в двух направлениях, можно создать другую пару каналов и задать потокам противоположные типы.

Этот метод полезен не только для межпотоковой связи в одном апплете, но он также позволяет

апплетам разделять контекст (например, на Web-странице), чтобы общаться друг с другом. Апплет AppleContext дает метод getApplets, перечисляющий те апплеты, которым он предоставляет свои ресурсы, и метод getApplet, который берет имя апплета и возвращает ссылку на именованный апплет, если таковой существует. Имена апплетам задаются в HTML-документе: <APPLET CODE="..." WIDTH=### HEIGHT=### NAME="...">

Сделав ссылку на апплет, с которым вы хотите пообщаться, вы можете создать пару канальных потоков данных и передать один из них этому апплету с помощью какого-нибудь заранее определенного метода. Это можно осуществить многими способами. Для простоты мы решили создать подкласс класса Applet, ListenApplet (слушающий апплет), действующий в качестве пассивного слушателя. Другой апплет может запросить у него OutputStream, записать туда данные и велеть ему читать данные из потока. ListenApplet берет данные из потока, интерпретирует их как текст и печатает в TextArea. Вот программа для ListenApplet.

Пример 13-6а. Слушающий апплет (Listening Applet).

```
import java.applet.*;
import java.io.*;
import java.awt.*;
import java.util.*;
public class ListenApplet extends Applet {
        // выходное окно
        private TextArea output;
        // содержит InputStream с ключами апплета
        private Hashtable inpipes;
        public void init() {
                inpipes = new Hashtable(1);
                output = new TextArea(10,60);
                output.setEditable(false);
                output.appendText("This Applet listens for master
applets.\n");
                add(output);
        }
```

Апплет может использовать этот метод для того, чтобы OutputStream связался с ним каналом. ListenApplet следит за своими OutputStream, храня хеш-таблицу этих потоков с ключами их соответствующих апплетов. Эта хеш-таблица позволяет осуществлять связь между несколькими апплетами. Апплет, который хочет поговорить с данным апплетом, начинает диалог с обращения к методу, вызывающему ListenApplet. Вызывающий метод создает пару канальных потоков и пропускает PipedOutputStream обратно к говорящему апплету:

```
public PipedOutputStream call(Applet a) throws IOException {
        // на случай, если апплет a уже соединен
        inpipes.remove(a);
        PipedInputStream in
        in = new PipedInputStream();
        PipedOutputStream out
        out = new PipedOutputStream();
        // соединяем каналы друг с другом
        in.connect(out);
        out.connect(in);
        // храним InputStream для дальнейшего использования
        inpipes.put(a,in);
        // возвращаем парный OutputStream
        return out;
}
```

Другие апплеты используют метод зависания, чтобы закрыть канал с этим апплетом. Когда связь разрывается, мы удаляем поток из хеш-таблицы:

```
public void hangup(Applet a) throws IOException {
        PipedInputStream in
        in = (PipedInputStream)inpipes.get(a);
        in.close();
        inpipes.remove(a);
```

```
}
```

Если другой апплет записал данные в поток, наш апплет считывает данные из потока до тех пор, пока они там есть, конвертирует их в тип String и печатает в поле TextArea. Если вам нужно работать с несколькими апплетами, взаимодействующими синхронно, лучше поместить этот метод в отдельный поток. Сначала мы считываем данные в буфер длиной 8. Достигнув границ массива, мы создаем новый массив с удвоенной по сравнению с предыдущим массивом длиной. Этот способ обращения с неизвестным количеством данных с помощью массивов представляется еще более сложным, чем те, что мы рассматривали раньше. Вместо того чтобы просто копировать данные в несколько массивов, как в нашем предыдущем примере, мы используем один массив, но как только он наполнится, мы удвоим его длину:

```
        public void listen(Applet a) throws IOException {
                PipedInputStream in = (PipedInputStream)inpipes.get(a);
                if (in==null) {
                        return;
                }
                int count = 0;
                int arysize = 8;
                byte ary[] = new byte[arysize];
                byte b;
                b = (byte)in.read();
                // если поток не полный
                while (b!=-1) {
                        // если мы уже наполнили наш массив
                        if (count==arysize) {
                                arysize = arysize*2;
                                byte temp[] = new byte[arysize];
                                // эта последняя система вызывается
                                // для копирования массива
                                System.arraycopy(ary,0,temp,0,count);
                                ary = temp;
                        }
                        ary[count] = b;
                        count++;
                        b = (byte)in.read();
                }
                String buffer = new String(ary,0);
                output.appendText("Received: "+buffer);
        }
}
```

Слушающий апплет будет сидеть на Web-странице без дела. Необходимо вызвать второй апплет, чтобы с ним поговорить. Ниже приводится программа для разговаривающего апплета. Заметьте, что, поскольку мы используем расширенные свойства апплета ListenApplet, мы должны его импортировать.

Пример 13-6b. Разговаривающий апплет (Talking Applet).

```
import ListenApplet;
import java.awt.*;
import java.io.*;
import java.applet.*;
public class MasterApplet extends Applet {
        // входное пользовательское окно
        private TextArea output;
        // сигнал посылать входные данные
        private Button send;
        public void init() {
                output = new TextArea(10,60);
                output.setEditable(true);
                send = new Button("Send");
                add(send);
                add(output);
```

```
        }
```

Как только пользователь нажимает на кнопку "Послать", выполняется метод action. Мы находим апплет, с которым хотим поговорить, и посылаем ему содержимое поля TextArea:

```
public boolean action(Event evt, Object o) {
        if (evt.target!=send) {
                return false;
        }
        ListenApplet slave
        slave = (ListenApplet)getAppletContext().getApplet("listener");
        if (slave==null) {
                output.appendText("Found no listening applet.\n");
        } else try {
                sendMessage(output.getText(),slave);
                output.setText("");
        } catch (IOException e) {
                output.appendText(e.toString()+"\n");
        }
        return true;
}
```

В программе, приведенной ниже, мы пытаемся послать строку на выходной (target) апплет. Для начала мы вызываем апплет, чтобы он получил OutputStream. Затем мы конвертируем строку в массив байтов и записываем его в поток, добавив новую строку и символ EOF. Мы просим другой апплет прочесть содержимое потока и затем закрыть связь:

```
public void sendMessage(String s, ListenApplet a) throws IOException {
        PipedOutputStream out;
        out = a.call(this);
        byte b[] = new byte[s.length()];
        s.getBytes(0,s.length(),b,0);
        out.write(b);
        byte NL = '\n';
        out.write(NL);
        byte EOF = -1;
        out.write(EOF);
        a.listen(this);
        a.hangup(this);
        out.close();
        }
}
```

HTML-код для Web-страницы, включающей оба этих апплета, выглядит следующим образом:

```
<APPLET CODE="ListenApplet.class" WIDTH=500 HEIGHT=250
NAME="Listener"></Applet>
<APPLET CODE="MasterApplet.class" WIDTH=500 HEIGHT=250></APPLET>
```

Что дальше?

Метод сетевого общения с помощью сокетов и потоков существует уже много лет; пока это наиболее простой и быстрый способ разговора по сети. URL - это относительно недавнее добавление к постоянно совершенствующимся стандартам сетевого общения. URL дают возможность кодировать всю информацию, которую необходимо идентифицировать, и искать объекты в Интернет с помощью разнообразных протоколов. Обычно тип протокола, имя хоста, путь и имя файла связаны вместе в единую, синтаксически легко анализируемую строку.

Ваш Web-броузер может поддерживать по крайней мере протокол HTTP (HyperText Transfer Protocol, "протокол передачи гипертекста"). Наиболее популярные броузеры поддерживают как HTTP, так и FTP (File Transfer Protocol, "протокол передачи файлов") для передачи объектов, а некоторые поддерживают протокол NNTP (Network News Transfer Protocol, "протокол передачи сетевых новостей") для просмотра групп телеконференций. Все эти протоколы можно

закодировать в URL. Java API работает по умолчанию с некоторыми протоколами и, кроме того, содержит механизм для разработки собственных драйверов URL, которые могут понимать другие протоколы. Кроме того, URL-классы Java позволяют легко искать в сетях не только простые типы данных, но и объекты Java. Мы уже видели один такой пример под названием getImage(URL). В следующей главе мы покажем, как проектировать собственный поиск объектов.

Глава 14

Связь по сети с помощью URL

URL (Uniform Resource Locator, "унифицированная ссылка на ресурс") используется для того, чтобы находить ресурсы в сети Интернет. Разработчики Web-страниц могут делать на своих документах ссылки (links), указывающие на другие Web-страницы или ресурсы Интернет. URL позволяют кодировать местоположение ресурсов сети. Java API включает полный набор классов, помогающих программистам пользоваться URL для доступа к сетевым ресурсам.

Если мы будем использовать URL, а не непосредственно сокеты, мы сможем расширить возможности многократного использования нашей программы. Как будет видно дальше, URL-классы Java позволяют разделить процессы соединения с сетевым ресурсом и интерпретации его данных. Преимущественно URL используются для поиска данных, хотя некоторые из них, например Telnet URL, применяются для того, чтобы получить интерактивный доступ к ресурсам сети. В табл. 14-1 перечислены URL, которые удовлетворяют спецификациям самых общих протоколов. Классы URL, содержащиеся в Java API, предназначены для использования URL при поиске данных, хотя с интерактивными URL работать также легко.

Таблица 14-1. Наиболее распространенные URL

Протокол	URL
HTTP (HyperText Transfer Protocol)	http://www.vmedia.com/
FTP (File Transfer Protocol)	ftp://ftp.vmedia.com/pub
SMTP (Simple Mail Transfer Protocol)	mailto:user@host

News (Usenet)	news://sunsite.unc.edu/
WAIS (Wide Area Index and Search)	wais://sunsite.unc.edu/linux-faq
Telnet	telnet://www.vmedia.com/

СОВЕТ Официальной спецификацией на формат и структуру URL является RFC 1738, доступная на сервере http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.

Netscape Navigator 2.0 наложил несколько весьма жестких ограничений, касающихся использования URL в апплетах. Эти ограничения, делая ненадежные апплеты более безопасными, в то же время запрещают создание с помощью классов URL драйверов протоколов и типов содержимого, не входящих в обеспечение Netscape. Хотя это отчасти ограничивает использование классов URL, Netscape Navigator 2.0 предлагает поддержку для наиболее широко используемых протоколов. И хотя Netscape Navigator 2.0 не поддерживает непосредственно классы URL для любых типов содержимого, мы покажем, как обойти это ограничение для драйверов в разделе этой главы "Типы MIME и класс ContentHandler". Конечно, для поиска изображений и звуковых файлов вы можете по-прежнему использовать методы класса Applet getImage и getAudioClip.

Использование класса URL

Java содержит класс java.net.URL, позволяющий иметь дело с URL, изначально принадлежавшими Java. Кроме того, в Java есть встроенная поддержка типов протоколов и возможности для программистов создавать собственную поддержку для новых протоколов. В главе 5, "Апплет в работе", мы обсуждали абсолютные и относительные конструктры URL. Все четыре конструктора URL приводятся в табл. 14-2.

Таблица 14-2. Конструкторы класса URL

Конструктор	Описание
URL(String,string,int,String)	Создает абсолютный URL из типа протокола, имени хоста, номера порта и расположения каталога.
URL(String,String,String)	Создает абсолютный URL из типа протокола, имени хоста, номера порта и расположения каталога. Номер порта для данного типа протокола считается определенным по умолчанию.
URL(String)	Создает абсолютный URL из синтаксически непроанализированной строки.
URL(URL,String)	Создает абсолютный URL из синтаксически непроанализированной строки, содержащей каталог, относительный для данного URL.

Фирма Sun разработала поддержку URL для очень ограниченного числа протоколов - DOC, FILE и HTTP. Ниже в этой главе, в разделе "Работа с другими протоколами" мы обсудим, как создавать свою поддержку для новых протоколов класса URL. Все общие методы класса URL приведены в табл. 14-3.

Таблица 14-3. Общие методы класса URL

Метод	Описание
boolean equals(URL)	Сравнивает два URL на эквивалентность.
String getFile()	Возвращает абсолютный путь к ресурсам данного хоста.
String getHost()	Возвращает имя хоста.
int getPort()	Возвращает номер порта.
String getProtocol()	Возвращает тип протокола.
String getRef()	Возвращает ссылку на определенное положение внутри файла. Например, http://WWW.vmedia.com/indexhtml#middle содержит ссылку "middle" на отмеченное место внутри файла index.html.
sameFile(URL)	Сравнивает два URL на эквивалентность, игнорируя все ссылки.
String toExternalForm()	Печатает полностью подготовленный URL.
Object getContent()	Восстанавливает содержимое данного URL.
InputStream openStream()	Возвращает InputStream из ресурса.

openConnection()	Возвращает URLConnection для данного URL.
setURLStreamHandleFactory (URLStreamHandleFactory)	Задает URLStreamHandleFactory.

Реализовав класс URL, мы захотим получить доступ к ресурсам, на которые он указывает. Класс URL предлагает для этой цели три основных метода. Наиболее простой заключается в чтении из InputStream, полученного с URL. Это делает метод openStream:

```
InputStream is;
try {
        is = myURL.openStream();
        // читаем данные из is
} catch (IOException e) {}
```

Применение URL в этом случае мало отличается от использования сокетов. Основное отличие состоит в том, что при программировании с помощью URL связь через сокеты автоматически открывается на нужном порту и выполняются все протоколы установки связи (handshaking) и процедур запросов. Например, если вы запрашиваете поток со стандартного HTTP URL, URL открывает сокет на порту 80 на определенной машине и выдает команду GET для определенного положения каталога. InputStream, возвращенный методом openStream, располагается в начале потока байтов данного ресурса. Ниже дается короткий пример чтения непосредственно из InputStream с URL:

```
URL myURL;
try {
        myURL = new URL(getCodeBase(),
                "/index.html");
} catch (MalformedURLException e) {}
try {
        InputStream is = myURL.openStream();
        int i = is.read();
        while (i!=-1) {
                // читаем данные из потока
        }
} catch (IOEXception e) {}
```

Здесь мы создаем URL к Web-странице нашего апплета, открываем InputStream, соединенный с этой страницей, и читаем HTML-файл из InputStream.

Получение содержимого

Метод getContent - приятное свойство класса URL, которое в настоящее время мало используется. Этот метод открывает поток к ресурсу в точности так, как это делает метод openStream, но затем пытается определить тип MIME потока и конвертировать поток в объект Java. Зная тип MIME потока данных, URL может передать поток данных методу, созданному для работы именно с этим типом данных. Этот метод должен выдать нам данные, инкапсулированные в соответствующем типе объекта Java. Например, если мы создали URL, указывающий на изображение в формате GIF, метод getContent должен понять, что поток относится к типу "image/gif", и вернуть экземпляр класса Image. Объект Image будет содержать копию GIF-картинки. Точный механизм, используемый методом getContent, будет объяснен ниже в этой главе в разделе "Типы MIME и класс ContentHandler". Ниже приводится пример использования метода getContent. В этом примере мы создаем URL для основного индекса Web-сервера апплета. Мы вызываем метод getContent, чтобы восстановить ресурс в объекте Java, а затем применяем знак операции instanceof для определения того, какой тип объекта возвращен:

```
URL myURL;
try {
        myURL = new URL(getCodeBase(),
                "/index.html");
} catch (MalformedURLException e) {}
try {
        Object o = myURL.getContent();
```

```
} catch (IOEXception e) {}
if (o instanceof Image) {
        // если o - изображение
} else if (o instanceof String) {
        // если o - строка
} else {
        // по умолчанию
}
```

Спецификация MIME

Спецификация типа MIME (Multipurpose Internet Mail Extensions, "много-целевые почтовые расширения Интернет") впервые была предложена в RFC 1341 для упрощения пересылки двоичных данных по e-mail. Вместо того чтобы просто посылать двоичные данные внутри обычного электронного сообщения и надеяться, что получатель их правильно прочтет, MIME позволяет свести воедино все разнообразие информации о двоичных данных, а именно - метод кодирования данных и тот тип программы, которую использует почтовый клиент для просмотра раскодированных данных.

Спецификация MIME доказала свою полезность и в областях, отличных от e-mail, особенно в передаче файлов по HTTP. Когда HTTP-клиент запрашивает файл, при желании он может запросить MIME-информацию о файле и работать с данными в соответствии с типом содержимого файла. Например, если HTTP-клиент получает файл с типом MIME "image/gif", он, возможно, передаст данные подпрограмме или утилите просмотра изображений.

Если вас интересуют ссылки на Web-страницы и RFC, содержащие информацию о MIME, обращайтесь по адресу http://www.worldtalk.com/web/text/email.html.

Соединение с помощью класса URLConnection

Если вы используете URL, требующий какую-то дополнительную входную информацию, например URL к сценарию CGI, или если вам нужно больше информации о некоем ресурсе, вы можете воспользоваться методом openConnection. Этот метод возвращает объект URLConnection, связанный с URL. Класс URLConnection, содержащийся в Java API, дает абстрактное представление реальной связи между компьютером, на котором находится апплет, и компьютером, содержащим данный ресурс.

Возможно, вам захочется воспользоваться сценарием на CGI в качестве простой внутрисерверной поддержки для своего Java-апплета. Апплеты не могут просто записывать файлы на свой клиент или на свой сервер, но вы легко можете записать безопасный сценарий, и это даст вам доступ на чтение или запись к файлам, находящимся на сервере. Тогда Java-апплеты смогут установить связь по URL с вашим сценарием, что даст вам возможность посылать данные к сетевому ресурсу, чтобы ваш апплет мог передавать свои файловые запросы сценарию через CGI-переменные. Этот сценарий может, в свою очередь, выполнять необходимые операции с файлами и сообщать в выходных данных статус произведенной операции. Эта методика используется в главе 17, "Взаимодействие с CGI: Java-магазин".

Некоторые протоколы, помимо разрешения доступа к сетевым ресурсам, могут выдавать информацию о данном ресурсе, например, тип MIME или дату последнего изменения ресурса. Класс URLConnection содержит достаточный набор методов для доступа к этим данным.

URLConnection можно создать с помощью метода openConnection класса URL или используя требуемый URL в качестве аргумента. Класс URLConnection дает программистам больше информации и лучшее управление ресурсом, определяемым URL. Согласно описанию Java API, этот класс является абстрактным; подклассы, реализованные для определенных протоколов, включены в JDK. Это относится к протоколам HTTP, FILE и DOC (FILE URL применяется для локальных файлов, а DOC URL использован в броузере Hotjava). Netscape Navigator 2.0 позволяет апплетам при работе с URL использовать только HTTP-протокол.

В классе URLConnection определено много общих методов, дающих легкий доступ к информации о ресурсе, находящемся на данном URL. Эти методы приведены в табл. 14-4. Некоторые из них применимы только при использовании HTTP URL и возвращают null или 0 во всех других случаях.

Таблица 14-4. Информационные методы класса URLConnection

Метод	Описание
String getContentEncoding()	Возвращает закодированные выходные данные ресурса или,, если они неизвестны,, null (например,, base64,, 7bit).
String getContentLength()	Возвращает размер ресурса в байтах.
String getContentType()	Возвращает тип MIME ресурса (например,, image/gif).

String getDate()	Возвращает дату посылки ресурса.
String getExpiration()	Возвращает срок годности ресурса.
String getHeaderField(String)	Возвращает поле заголовка с названием данной строки.
String getHeaderField(int)	Возвращает поле заголовка с индексом данного целого.
String getHeaderFieldKey(int)	Возвращает имя поля заголовка с индексом данного целого.
long getHeaderFieldDate(String,long)	Возвращает поле заголовка с названием данной String,, распознанное как дата. Целочисленный аргумент long служит по умолчанию полем заголовка,, если оно не распознано или не найдено.
int getHeaderFieldInt(String,int)	Возвращает поле заголовка с названием данной строки,, распознанное как целое. Целочисленный аргумент служит по умолчанию полем заголовка,, если оно не распознано или не найдено.
long getModifiedSince()	Возвращает поле if modified since.

HTTP и класс URLConnection

Для того чтобы понять, как класс URLConnection работает с простым HTTP URL, рассмотрим кратко внутренние механизмы работы протокола HTTP. Ниже приведена запись типичной сессии HTTP; первая строка является запросом клиента, а все остальное - ответ сервера:

```
GET /index.html HTTP/1.0
HTTP/1.0 200 Document follows
Date: Sun, 10 Mar 1996 03:52:15 GMT
Server: NCSA/1.4
Content-type: text/html
Last-modified: Fri, 08 Mar 1996 20:24:18
Content-lenght: 4611
<html>
. . .
</html>
```

Если мы создаем URLConnection для страницы index.html из приведенного примера, метод getContentType вернет "text/html", а метод getContentLength вернет "4611". Заметим, что определять эти значения с помощью HTTP исключительно легко - сервер возвращает их в заголовке HTTP. Многие другие протоколы не дают этой информации.

Теперь вернемся к рабочему инструменту методов класса URLConnection, к методу getContent. Именно этот метод реально запускается, когда мы вызываем метод getContent на URL. Для начала метод getContent пытается определить тип содержимого рассматриваемого ресурса, изучая либо заголовок HTTP, либо сам поток, либо расширение имени файла. Решив, какой тип MIME использовать, метод getContent запрашивает класс ContentHandler для данного типа MIME и передается в класс ContentHandler.

Типы MIME и класс ContentHandler

Каждый тип MIME может иметь собственный класс ContentHandler. Этот класс ContentHandler в конечном счете обрабатывает данные ресурса, когда мы вызываем getContent на URL или URLConnection. Класс ContentHandler состоит из одного метода getContent, который использует URLConnection как аргумент и возвращает объект, по возможности относящийся к ресурсу. Например, ресурс с MIME-типом "image/gif" будет скорее всего возвращен как Image или какой-то подкласс Image из метода getContent. Класс ContentHandler согласно описаниям Java API является абстрактным; подклассы класса ContentHandler дают реализации для различных типов MIME. Ниже приведен не очень сложный ContentHandler для любых типов "text/*":

```
import java.net.*;
import java.io.*;
public class TextContentHandler extends ContentHandler {
public Object getContent(URLConnection urlc) {
try {
        InputStream is;
        is = urlc.getInputStream();
```

```
        DataInputStream ds;
        ds = new DataInputStream(is);
        int length;
        length = urlc.getContentLength();
        if (length!=-1) {
                 byte b[] = new byte[length];
                 ds.readFully(b);
                String s = new String(b,0);
        } else {
                // длина неизвестна
                String s = "";
                int i = is.read();
                while (i!=-1) {
                        s = s+(char)i;
                         i = is.read();
                 }
        }
                         return s;
                } catch (Exception e) {
                        e.printStackTrace();
                        return null;
                }
        }
}
```

Как URLConnection выбирает подходящий ContentHandler при вызове getContent? Вначале выбирается тип MIME потока и запрашивается ContentHandler для этого типа MIME из ContentHandlerFactory. Согласно описаниям Java API, ContentHandler Factory - это интерфейс. Он состоит из одного метода createContentHandler, который использует строковую переменную в качестве аргумента и возвращает ContentHandler. Строковое значение указывает, каким типом MIME данный ContentHandler может пользоваться. Ниже приведен текст программы для простого ContentHandlerFactory, который может использоваться нашим TextContentHandler:

```
import java.net.*;
import java.io.*;
public class MyContentHandlerFactory
        extends Object
        implements ContentHandlerFactory {
        public ContentHandler
        createContentHandler(String type) {
                if (type.startsWith("text")) {
                        return new TextContentHandler();
                }
                else {
                        return null;
                }
        }
}
```

Класс ContentHandlerFactory

Мы можем задать класс ContentHandlerFactory для нашего URLConnection с помощью метода setContentHandlerFactory. К сожалению, в Netscape Navigator 2.0 апплетам не разрешено задавать какие-либо factory, и потому апплетам запрещено пользоваться этим методом. Трудно понять, почему апплетам запрещено задавать ContentHandlerFactory, потому что все, что делают методы ContentHandler, - это перенос потоков данных в объекты Java. К счастью, мы можем читать из потоков и писать в потоки, связанные с URLConnection, так что это препятствие преодолимо. Более того, при работе с URL мы можем использовать классы ContentHandler и ContentHandlerFactory несколько обходным путем. Рассмотрим следующий апплет.

Пример 14-1. Апплет, использующий класс ContentHandler.

```
import java.applet.*;
import java.net.*;
```

```
import java.awt.*;
import java.io.*;
import TextContentHandler;
public class URLApplet extends Applet {
        private TextArea output;
        private ContentHandler handler;
        public void init() {
                handler = new TextContentHandler();
                output = new TextArea(12,80);
                output.setEditable(false);
                add(output);
                show();
                resize(500,250);
        }
        public void start() {
                try {
                        URL myURL = new URL(getCodeBase(),
                                "/index.html");
                        URLConnection myUC =
                myURL.openConnection();
                        myUC.connect();
                        Object o = handler.getContent(myUC);
                        output.setText((String)o);
                } catch (Exception e) {
                        handleException(e);
                }
        }
        public void stop() {
                output.setText("");
        }
        public void handleException(Throwable t) {
                System.out.println(t.toString());
                t.printStackTrace();
                stop();
        }
}
```

На рис. 14-1 приведен экран этого апплета. Апплет считывает и показывает в TextArea сам текст программы. Не пользуясь методами getContent класса URL и классами URLConnection, вызовем непосредственно метод getContent класса ContentHandler. Мы можем также расширить апплет для использования ContentHadnlerFactory.

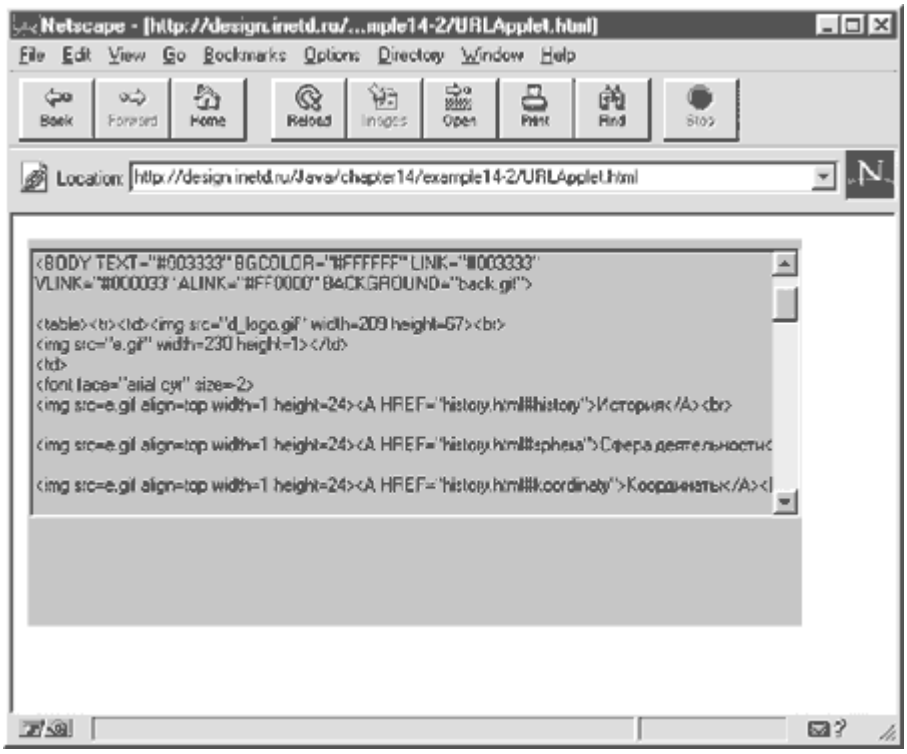

Рис. 14.1.

Пример 14-2. Апплет, использующий класс ContentHandlerFactory.

```
import java.applet.*;
import java.net.*;
import java.awt.*;
import java.io.*;
import MyContentHandlerFactory;
public class URLApplet extends Applet {
        private TextArea output;
        private ContentHandlerFactory factory;
        public void init() {
                factory = new MyContentHandlerFactory();
                output = new TextArea(12,80);
                output.setEditable(false);
                add(output);
                show();
                resize(500,250);
        }
        public void start() {
                try {
                        URL myURL = new URL(getCodeBase(),"/index.html");
                        URLConnection myUC = myURL.openConnection();
                        myUC.connect();
                        String type = myUC.getContentType();
                        ContentHandler handler =
factory.createContentHandler(type);
                        Object o = handler.getContent(myUC);
                        output.setText((String)o);
                } catch (Exception e) {
                        handleException(e);
                }
        }
        public void stop() {
                output.setText("");
        }
        public void handleException(Throwable t) {
                System.out.println("CAUGHT: "+t.toString());
                t.printStackTrace();
                stop();
```

```
        }
}
```

Этот апплет делает примерно то же самое, что и апплет в примере 14-1, за исключением того, что данный апплет демонстрирует использование ContentHandlerFactory. Сначала мы определяем тип MIME ресурса URL, а затем запрашиваем соответствующий этому типу ContentHandler. Сейчас наш ContentHandlerFactory поддерживает только тип MIME "text/*", но впоследствии мы можем создать новые классы ContentHandler и добавить их к factory.

Определенный по умолчанию ContentHandlerFactory, используемый классами URLConnections в Netscape Navigator 2.0, в настоящее время не поддерживает никакие типы MIME. Вызвав метод getContent на URL или URLConnection, мы скорее всего получим InputStream; это означает, что ContentHandlerFactory не знал, что делать с ресурсом. InputStream, возвращенный методом getContent, соединен с ресурсом; программисту предоставляется возможность интерпретировать полученные байты. Со временем непременно появятся другие реализации интерфейса ContentHandlerFactory, открывающие новые возможности для данного аспекта Java API.

Сделайте это сами с помощью потоков

Сейчас в условиях, когда существуют ограничения на установление ContentHandlerFactory для классов URLConnection в апплетах, вы, возможно, захотите читать прямо из потоков, связанных с URLConnection, чтобы получить данные из сетевого ресурса. Класс URLConnection предоставляет для этой цели два метода: getInputStream и getOutputStream. Если вы попытаетесь вызвать один из них на типе URL, который их не поддерживает, вы получите исключение UnknownServiceException. Вот небольшой пример:

```
InputStream is;
OutputStream os;
try {
        is = myConnection.getInputStream();
        os = myConnection.getOutputStream();
        // чтение из is и запись в os
} catch (Exception e) {
        // обработка исключения UnknownServiceException
}
```

Класс URLEncoder

Когда вы записываете данные в OutputStream, соединенный с ресурсом через протокол HTTP, вы должны убедиться в том, что ваши выходные данные могут быть переданы с URL, - некоторые символы считаются протоколом HTTP специальными, они должны быть закодированы перед отправкой. Java API предоставляет класс URLEncoder, который это сделает. Он состоит из единственного метода, encode, который берет строковую переменную и возвращает ее после того, как закодирует в ней все специальные символы.

Настройка класса URLConnection

Для конфигурации класса URLConnection существуют различные варианты настройки. Некоторые из их значений по умолчанию вы можете изменить, после чего все новые классы URLConnection получат новое значение в качестве начального, задаваемого по умолчанию для данной настройки. Методы задания этих настроек описаны в табл. 14-5.

Таблица 14-5. Различные настройки класса URLConnection

Метод	Описание
boolean getAllowUserInteraction()	Возвращает флаг диалога пользователей, показывающий, разрешает ли данный тип URL диалог пользователей.
setAllowUserInteraction(boolean)	Устанавливает флаг диалога пользователей.
boolean getDefaultAllowUserInteraction()	Возвращает значение флага диалога пользователей, задаваемое по умолчанию.
setDefaultAllowUserInteraction()	Задает значение по умолчанию для флага диалога пользователей.
boolean getUseCaches()	Некоторые протоколы разрешают помещение ресурсов в локальный кеш. Данный метод возвращает булевское значение, показывающее статус такого поведения.

setUseCaches(boolean)	Задает настройку кеша для данного URLConnection.
boolean getDefaultUseCaches()	Возвращает значение по умолчанию для настройки кеша.
setDefaultUseCaches(boolean)	Задает значение по умолчанию для настройки кеша.
String getRequestProperty(String)	Возвращает свойство, проиндексированное по данной строке.
setRequestProperty(String,String)	Присваивает второй строке свойство, проиндексированное по первой строке.
String getDefaultRequestProperty(String)	Возвращает определенное по умолчанию свойство, проиндексированное по данной строке.
setDefaultRequestProperty(String,String)	Присваивает второй строке определенное по умолчанию свойство, проиндексированное по первой строке.
boolean getDoInput()	Возвращает булевское значение, показывающее, поддерживает ли данный URLConnection входные данные (input).
setDoInput(boolean)	Задает флаг doinput.
boolean getDoOutput()	Возвращает булевское значение, показывающее, поддерживает ли данный URLConnection выходные данные (output).
setDoOutput(boolean)	Задает флаг DoOutput.

Работа с другими протоколами

Мы можем расширить возможности класса URL для работы с другими протоколами, которые не поставляются компаниями Sun и Netscape. Класс URLStreamHandler, описанный в Java API, используется классом URL при открытии URLConnection. Задача класса URLStreamHandler - ввести необходимые сокеты в компьютер, содержащий данный ресурс, и выполнить требуемое протоколом установление связи. Наравне с классом ContentHandler каждый протокол имеет или может иметь относящийся к нему класс URLStreamHandler. Методы класса URLStreamHandler приведены в табл. 14-6.

Таблица 14-6. Методы класса URLStreamHandler

Метод	Описание
URLConnection openConnection(URL)	Создает URLConnection, связанный с определенным URL.
void parseURL(URL,String, int,int)	Производит разбор данной строки на данном URL. Целые относятся к начальному и конечному индексам URL внутри строки. По умолчанию этот метод проводит разбор URL по стандартной форме Интернет (//hostname/directory).
String toExternalForm(URL)	Возвращает данный URL в стандартном строковом обозначении.
setURL(URL,String,String, int,String,String)	Задает тип протокола, имя хоста, номер порта, расположение каталога и ссылку на данный URL.

Мы можем изменить определенный по умолчанию URLStreamHandler для данного URL с помощью метода setURLStreamHandlerFactory на URL. Этот метод является статическим, поэтому он может быть вызван до создания URL. Метод задает по умолчанию URLStreamHandlerFactory для всех последовательно создаваемых URL. Если мы создаем новый экземпляр URLStreamHandlerFactory, мы захотим изменить его, и, кроме того, мы захотим, чтобы его использовали наши URL для поддержания своих связей. Как и интерфейс ContentHandlerFactory, интерфейс URLStreamHandlerFactory состоит из единственного метода createURLStreamHandler, который использует строковую переменную как аргумент и возвращает URLStreamHandler. Строка показывает, каким протоколом должен пользоваться URLStreamHandler.

Пока нам не разрешается задавать URLStreamHandlerFactory для URL в апплетах, запускаемых в Netscape Navigator 2.0. Опять-таки причины такого ограничения не вполне ясны, поскольку это не позволяет программистам распространять Java-апплеты для общения через другие протоколы, такие как FTP или HTTP-NG. Тем не менее можно создать апплет, общающийся через другие протоколы, с помощью непосредственного программирования сокетов, но вы не сможете использовать URL, описанные в Java API.

Если вы заинтересованы в том, чтобы расширить классы URL для работы с новым протоколом, вам придется реализовать специфические для этого протокола URLConnection и URLStreamHandler, а затем либо создать, либо расширить URLStreamHandlerFactory для

включения ваших новых классов. Пока Netscape не ослабит ограничения на задание factory в апплетах, вы не сможете пользоваться собственными классами URLStreamHandlerFactory в апплетах, если только вы не решите полностью переписать классы URL.

Чем хороши URL

Возможно, вы не вполне понимаете, для чего написана эта глава, ведь большая часть возможностей классов URL не разрешена или не поддерживается при запуске апплетов Java в Netscape Navigator 2.0, являющимся сейчас наиболее популярным Web-броузером. Огромное значение классов URL заключается в том, что они позволяют хранить всю информацию, необходимую для общения с определенным ресурсом, в одном классе. Они создают все сокеты, выполняют установление связи и интерпретацию заголовков, необходимую для поиска ресурсов через HTTP. С появлением классов ContentHandler, которые могут переводить более сложные ресурсы в объекты Java, такие как видеофайлы или файлы баз данных, простота использования URL для поиска ресурсов делает использование URL гораздо более эффективным, чем самостоятельное раскодирование потоков байтов. Возможно, Netscape в конце концов позволит апплетам задавать собственный поток и драйверы обработки содержимого. В противном случае, по-видимому, к определенному по умолчанию ContentHandlerFactory будут добавлены новые ContentHandler в качестве завершения поддержки Java программой Netscape Navigator.

Что дальше?

В этой главе мы описали, как и зачем пользоваться URL при программировании с использованием ресурсов сети. Один из наиболее распространенных способов использования URL в Java - соединение с программами через CGI.

Следующая глава резко отклоняется в сторону от апплетов. В добавление к обсуждению программирования серверов на Java мы рассмотрим некоторые возникающие при этом вопросы. Использование программ CGI в качестве внутренних для Java-апплетов имеет ряд преимуществ - например, совместимость с существующими интерфейсами, а также скорость и простоту развития. Тем не менее CGI имеет много ограничений; наиболее ощутимое - его непостоянство. Мы покажем вам, как писать на языке Java серверы с гораздо более широкими возможностями, чем могла бы предложить даже очень сложная программа CGI.

Глава 15

Разработка серверов на Java

До сих пор основное внимание этой части книги уделялось написанию сетевого клиента в парадигме апплета. Теперь мы сделаем небольшой шаг назад и обратимся к более широкому кругу тем, включающему систему клиент-сервер с точки зрения World Wide Web.

Когда возникает необходимость в централизации обслуживания, например при слежении за данными или при разделении данных, нужно написать независимый сервер. Другое применение сервера может заключаться в строгом контролировании безопасности использования данных. Например, в программе общения, которую мы предлагаем в данной главе, сервер можно использовать для создания ограниченного доступа только для избранных пользователей и для

разрешения сеансов частного общения между пользователями.

В этой главе мы сделаем следующее:

- Реализуем протокол.
- Построим схему потока данных в динамической системе клиент-сервер.
- Построим сервер, написанный на Java, для реализации нашего протокола.
- Построим клиент, основанный на нашем протоколе.

Мы решили создавать протокол специально для проведения асинхронных конференций между двумя или более людьми. Другими словами, мы хотим создать протокол chat (общение). Наш протокол общения мы будем называть Internet Turbo Chat, или ITC.

СОВЕТ Фрагменты кода, приводимые в качестве примеров в этой главе, помещены на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html)

Создание собственного сервера и протокола

Начнем с создания наброска протокола. Желательно проявить гибкость при конструировании протокола, чтобы потом можно было добавлять новые функции; в то же время все должно быть определено достаточно строго для обеспечения устойчивости к ошибкам. Для начала мы хотим выполнять следующие функции высокого уровня:

- задание имен пользователей;
- асинхронные сетевые операции;
- обозначение новых связей;
- создание окна сервера, перечисляющего пользователей на связи.

Необходимость в процедуре "login" очевидна: прежде чем получить доступ к системе общения, пользователь должен ввести свое имя. Затем мы можем передать на сервер введенные этим пользователем сообщения, и при этом нам не придется их изменять за счет кодирования. Воспользуемся упрощенным подходом: ограничим данные, передаваемые по сети, строками. Методы readln и println в DataInputStream и классы PrintStream предлагают хороший способ передачи данных без кодирования/раскодирования.

Протокол, которым мы пользуемся, достаточно прост. Он осуществляет начальную обработку запроса (транзакцию) имени пользователя на клиенте и на сервере и затем отображает сообщения клиента для всех других клиентов, соединенных с сервером. Сервер обрабатывает только первое, что ввел клиент, то есть его пользовательское имя. Это нужно серверу для поддержания списка связей. Сообщения, присылаемые клиентом серверу, форматируются так, чтобы серверу не нужно было проводить разбор входных данных. Это уменьшает запаздывание и повышает эффективность работы сервера. Однако если мы решим выполнять другие функции, например "частное" общение, нам придется заставить сервер анализировать входные данные от клиента с тем, чтобы они были правильно маршрутизированы.

Определение задач сервера

Как уже упоминалось выше, для выполнения специальных функций часто требуется специализированный сервер. В нашем случае нужно выполнить несколько специальных задач - отобразить входную информацию для всех связей и следить за связями на сервере. Мы могли бы еще усложнить программу общения, заставив каждого клиента также служить простым сервером, чтобы можно было осуществить коммуникацию между двумя программами общения. Но поскольку мы хотим иметь возможность одновременного общения более чем двух клиентов, такое усложнение нежелательно.

Кроме того, клиент общения должен быть насколько возможно мал, потому что он будет загружаться вместе с Web-страницей каждому пользователю каждый раз, как тот заходит на эту Web-страницу. Для этого мы можем задать серверу общения дополнительные функции, которые могут выполняться как сервером, так и клиентом, например разбор входных и выходных данных.

Кроме того, мы должны учитывать нагрузку на машину, возникающую при запуске сервера, - если машина медленная, сервер будет медленно выполнять свои функции.

Определение взаимодействия клиент-сервер

Когда требуется синхронное обновление информации, важной задачей является координация работы сервера с несколькими клиентами. Например, если вы хотите обновлять всех соединенных с сервером клиентов, передавая им какие-то специальные данные или команды, вы должны сделать свой сервер устойчивым к ошибкам, чтобы он не "зависал" при попытке передать данные по сети.

В нашей программе общения необходимо, чтобы все клиенты получали данные с сервера одновременно, так чтобы несколько строк, переданных одним пользователем, были вместе, когда их увидят другие пользователи. Мы сделаем это с помощью только одного независимого потока, работающего с выходными данными к клиентам. При этом независимый поток сможет свободно посылать данные всем присоединенным клиентам, не привязываясь к входным данным с какого-то определенного входа. Поскольку мы делаем только одну операцию в потоке, а именно записываем данные клиентам, мы можем быть уверены, что клиент получит данные в то же самое время. Для лучшей координации такого взаимодействия мы должны рассмотреть вопросы, связанные со временем, сложностью и скоростью работы сервера.

Распределение времени

Мы уже говорили о необходимости писать данные на порты клиентов одновременно. Хотя в полной мере это невозможно, наша задача - уменьшить время, необходимое для записи данных всем клиентам, так чтобы у них создавалось впечатление, что данные передаются одновременно. Потоку свободного доступа (free-running), которым мы пользуемся для записи данных присоединенным клиентам, может быть придан высший приоритет по сравнению с потоками, читающими данные с каждого клиента. Мы можем подождать, пока записывающий поток закончит работу, и затем начать чтение данных с присоединенных клиентов. Другой временной аспект, который необходимо иметь в виду, связан с пересылкой данных с клиента на сервер. Если два клиента посылают данные на сервер одновременно, поток, записывающий выходные данные, может оказаться занят. Чтобы избежать пересечения, необходимо воспользоваться синхронизированным блоком программы, что объясняется в разделе ниже.

Структура связей

То обстоятельство, что у нас несколько связей, выводит наш сервер общения на более сложный уровень, чем это кажется поначалу. Мы должны следить за несколькими вещами, в первую очередь - за состоянием каждой связи. Если пользователь прерывает клиент общения, не поставив об этом в известность сервер, мы получим задержанную связь, которая будет функционировать неправильно. Записывающий поток может попытаться писать в несуществующее соединение, и мы будем продолжать пытаться читать на сервере данные из этого соединения. Решение этой проблемы двойное: когда осуществляется операция с уже оборвавшимся соединением - выдать исключение и прекратить связь; кроме того, выполнять профилактическую очистку связей, периодически проверяя их. Такая проверка и исключение связей позволяют высвободить ценные ресурсы системы. В нашем примере профилактическая проверка связей выполняется классом ConnectionWatcher. Он также запускается в собственном потоке, так что он может выполнять свои обязанности, не задерживая выполнение остальных функций.

Мы должны иметь возможность читать с присоединенного клиента, при этом не лишаясь возможности читать из других связей. Если мы воспользуемся циклом, просто выполняющим последовательно readln на каждом присоединенном клиенте, мы можем столкнуться с тем, что будем ждать пользователя, который уже не работает с клиентом общения, а оставил его запущенным и ушел обедать. Для разрешения этой проблемы нужно сделать весь сервер асинхронным - то есть создать читающий поток для каждой связи. При этом индивидуальный поток может ждать столько, сколько необходимо, пока пользователь введет сообщение, не подвесив при этом сервер. Когда пользователь вводит сообщение, мы просто передаем его на запись (то есть на его собственный поток), и сообщение будет записано каждому клиенту. Читающий поток может немедленно возобновить чтение другого сообщения, посланного пользователем. Благодаря тому, что для каждой связи создан собственный читающий поток, нам не нужно ждать, пока на каждой отдельной связи закончится чтение данных, и, таким образом, сервер получается действительно асинхронным. Однако нужно соблюдать осторожность, чтобы не обратиться к разделенному записывающему потоку и к ConnectionWatcher, когда их пытается использовать другой поток. Для этого в блоке программы, к которому могут попытаться

осуществить многократный доступ, сделаны синхронные описания.

Как уже упоминалось выше, поток ServerWrite пишет выходные данные во все связи. Для того чтобы "разбудить" ServerWriter, используется извещающий вызов (notify call). Когда ServerWriter возобновит исполнение, он просмотрит заранее определенные переменные на наличие в них данных, которые нужно послать присоединенным клиентам. Для того чтобы хранить выходные данные и избежать пересечения в случае, если два читающих потока делают извещающий вызов, используется структура данных "первый вошел - первый вышел" (FIFO). Класс FIFO подробно рассматривается в главе 17, "Взаимодействие с CGI: Java-магазин".

Если бы мы проводили разбор входных и выходных данных, нам понадобился бы дополнительный поток, задачей которого была бы обработка запросов из других потоков, их анализ и пересылка данных. Это увеличивает сложность, потому что нам может понадобиться такой дополнительный поток для каждой связи. В нашем сервере общения мы не выполняем разбор данных.

Скорость работы

Поскольку мы должны создать свой поток для каждой связи, число потоков, выдаваемых сервером, может оказаться достаточно большим. В ситуации, когда много потоков борются за два постоянно присутствующих потока ServerWriter и ConnectionWatcher, встает вопрос о том, насколько быстро работает сервер. Этого нельзя не учитывать, если мы ожидаем наличие большого количества связей. Мы должны, насколько возможно, упростить наши два класса с учетом их активного использования. Кроме того, если ожидается большое количество клиентов общения, мы, возможно, захотим удвоить сам сервер, чтобы поделить нагрузку. Тогда нам придется иметь дело с множественными серверами и с прохождением данных через них, а это выходит за рамки обсуждения этой главы.

Построение сервера Java

Здесь мы реализуем протокол, описанный в предыдущем разделе. Начнем с детального обсуждения сервера общения и закончим кратким описанием соответствующего клиента общения. Сервер общения, показанный на рис. 15-1, в противоположность апплету запускается как приложение Java. Парадигма апплета мало подходит для примера этой главы, потому что мы хотим, чтобы сервер имел постоянное местоположение, а также минимальное количество ограничений на то, откуда он может принимать связи. Заметим, что сервер общения полностью содержится в одном классе - chatserver. Единственным исключением является класс FIFO, который мы импортируем.

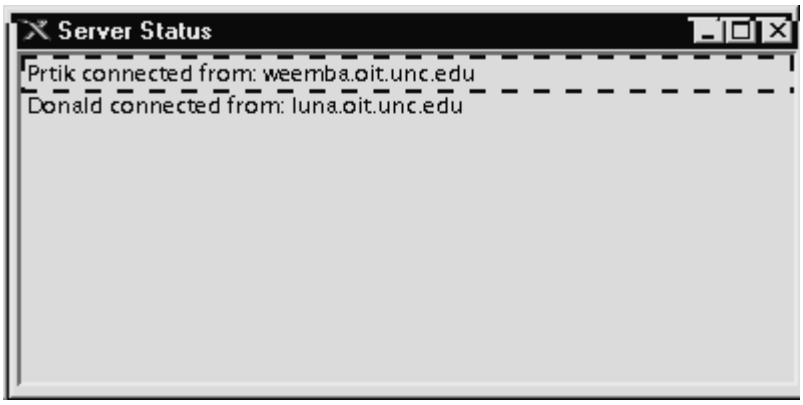

Рис. 15.1.

Для улучшения структуры и работоспособности сервера мы будем широко пользоваться свойством многопотоковости. Мы создадим свой поток для каждого присоединенного клиента и запустим три других потока: поток ServerWriter, записывающий данные в каждый присоединенный клиент; базовый поток chatserver, слушающий, когда новые клиенты пытаются установить связь с сервером; и поток ConnectionWatcher, убирающий прерванные связи. Как мы обсуждали в разделе "Структура связей", нам могут потребоваться еще дополнительные потоки, так что не будем бояться их использовать! Потоки позволяют удобно организовать управление потоками данных и выполнить комплексное взаимодействие, что будет видно при рассмотрении текста программы. Правда, потоки добавляют всей системе дополнительные заголовки сообщений (overhead), но, как правило, пока число связей не очень велико, потоки стоят затраченных ресурсов.

Заметьте, что в программе сервера общения очень много операторов try и catch. Чтобы программа была более устойчивой к ошибкам, важно полностью использовать возможности обработки исключений. Например, при запуске процесса проверки связей мы выполняем обработку исключений. В более сложном сервере исключения играют очень важную роль для стабилизации общих функций. Обработка исключений должна производиться даже в том случае, когда есть слабая вероятность возникновения ошибки. Кроме того, исключения могут использоваться для вызова методов очистки, освобождения системных ресурсов, отладки и жесткого контроля над прохождением потока данных сервера.

Базовым классом является chatserver, являющийся расширением Thread. Базовый класс реализуется как поток, потому что нам нужно запустить слушающий сегмент сокета и при этом не останавливаться для выполнения других операций. Этот поток создает собственный поток для каждой связи и передает ему вновь созданный сокет клиента.

Пример 15-1a. Программа сервера общения.

```
import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;
import FIFO;
public class chatserver extends Thread
        {
        public final static int DEFAULT_PORT = 6001;
        protected int port;
        protected ServerSocket server_port;
        protected ThreadGroup CurrentConnections;
        protected List connection_list;
        protected Vector connections;
        protected ConnectionWatcher watcher;
        protected ServerWriter writer;
        // если произошло исключение, выходим с сообщением об ошибке
        public static void fail(Exception e, String msg) {
                System.err.println(msg + ": " + e);
                System.exit(1);
        }
        // Создаем ServerSocket для прослушивания связей.
        // Начало потока.
        public chatserver(int port) {
                // назовем этот поток
                super("Server");
                if (port == 0) port = DEFAULT_PORT;
                this.port = port;
                try { server_port = new ServerSocket(port); }
                catch (IOException e) fail(e, "Exception creating server
socket");
                // создаем группу потоков для текущих связей
                CurrentConnections = new ThreadGroup("Server Connections");
                // фрейм для представления текущих связей
                Frame f = new Frame("Server Status");
                connection_list = new List();
                f.add("Center", connection_list);
                f.resize(400, 200);
                f.show();
                // вектор для хранения текущих связей
                connections = new Vector();
                // Создаем поток ConnectionWatcher,
                // который ждет отключения других потоков.
                // Он запускается автоматически.
                writer = new ServerWriter(this);
                watcher = new ConnectionWatcher(this, writer);
                // сервер начинает слушать связи
                this.start();
        }
        public void run() {
                try {
```

```
                        while(true) {
                                Socket client_socket = server_port.accept();
                                Connection c = new Connection(client_socket,
CurrentConnections, 3, watcher, writer);
                                // избегаем одновременного доступа
                                synchronized (connections) {
                                connections.addElement(c);
                                connection_list.addItem(c.getInfo());
                        }
                }
        }
        catch (IOException e) fail(e, "Exception while listening for
connections");
}
// запускаем сервер, прослушивающий определенный порт
public static void main(String[] args) {
                int port = 0;
                if (args.length == 1) {
                        try port = Integer.parseInt(args[0]);
                        catch (NumberFormatException e) port = 0;
                }
                new chatserver(port);
        }
}
```

На этом месте наш сервер начал работу и слушает на порту 6001 новые связи. Для каждой новой связи сервер создает свой поток, называемый Connection, и передает ему соответствующие параметры. Мы добавляем новый поток в вектор, содержащий все потоки действующих связей. Этот вектор используется впоследствии для проверки состояния связи. SereverWriter также использует его для записи в присоединенные клиенты. Новое соединение добавляется в список, представленный во фрейме.

Общение с помощью сокетов и работа с потоками ввода/вывода

Класс Connection является потоком, осуществляющим все вводные операции с клиентом. Этот класс передает драйвер выходящего потока классу ServerWriter, потому что мы предназначили этот поток для записи данных присоединенных клиентов. Класс Connection инициализирует входной и выходной потоки присоединением соответствующих входных и выходных компонентов сокета. Кроме того, поскольку именно этот поток первым начинает работу конкретно для данной связи, в этом методе мы получаем начальное имя пользователя. Имя пользователя мы передаем классу chatserver, использующему метод getInfo.

Пример 15-1b. Программа сервера общения.

```
class Connection extends Thread {
        static int numberOfConnections = 0;
        protected Socket client;
        protected ConnectionWatcher watcher;
        protected DataInputStream in;
        protected PrintStream out;
        protected ServerWriter writer;
        public Connection(Socket client_socket, ThreadGroup
CurrentConnections, int priority, ConnectionWatcher watcher, ServerWriter
writer)
        {
// присваиваем потоку имя и номер группы
super(CurrentConnections, "Connection number" + numberOfConnections++);
                this.setPriority(priority);
// задаем приоритет потока
                // локализация параметров
                client = client_socket;
                this.watcher = watcher;
                this.writer = writer;
        try {
```

```
                in = new DataInputStream(client.getInputStream());
                out = new PrintStream(client.getOutputStream());
                writer.OutputStreams.addElement(out);
        }
// Присоединяем потоки данных к входным и выходным потокам данных
// сокета клиента и добавляем outputstream к вектору, содержащему все
// выходные потоки данных, которые использует записывающий поток.
        catch (IOException e) {
                try client.close(); catch (IOException e2) ;
                System.err.println("Exception while getting socket streams: "
+ e);
                        return;
                }
                // запускаем поток на выполнение
                this.start();
        }
        // метод run выполняет цикл по чтению строк до тех пор,
        // пока не прекратит работу из-за обрыва связи
        public void run() {
                String inline;
out.println("Welcome to Internet Turbo Chat");
// посылаем приглашение клиенту
                try {
// выполняем цикл до тех пор, пока связь не порвется
                        while(true) {
                                // чтение строки
                                inline = in.readLine();
                                if (inline == null) break;
// если null - связь порвалась
                        writer.outdata.push(inline);
// сохраняем строку для записывающего потока
synchronized(writer)writer.notify();
// И вызываем записывающий поток. Заметим, что synchronized() применяется для
того,
// чтобы предотвратить одновременное обращение к записывающему потоку
//  двух связей, что представляет собой форму блокировки.
                        }
                }
                catch (IOException e);
// Когда связь прервалась, производим очистку и вызываем watcher,
// чтобы убрать связь из вектора действующих связей и из списка.
// Watcher, кроме того, убирает outputstream из вектора записывающего потока,
// содержащего outputstreams.
                finally {
                        try client.close();
                catch (IOException e2) ;
synchronized (watcher) watcher.notify();
                }
        }
// Этот метод получает имя пользователя на начальной связи, печатает его для
всех
// присоединенных в данный момент клиентов и передает обратно информацию для
// помещения ее в список текущих связей.
        public String getInfo() {
                String inline="";
                try { inline = in.readLine(); }
                catch (IOException e) System.out.println("Caught an
Exception:" +e);
                writer.outdata.push(inline+" has joined chat\n");
                synchronized(writer)writer.notify();
// Опять вызываем записывающий поток для представления сообщения нового
// пользователя. Делайте это синхронно, чтобы не получить множественный
доступ.
return (inline+ " connected from: " + client.getInetAddress().getHostName());
```

```
// возвращаем имя клиента и имя его компьютера для добавления в список связей
        }
}
```

Теперь у нас есть входной читающий поток, определенный и запущенный для каждого присоединенного клиента. Читая новое сообщение клиента, мы вызываем записывающий поток для записи этого сообщения всем остальным клиентам. При этом мы легко осуществляем выход из клиента общения, вызвав watcher и дав ему задание убрать клиента, с которым прервалась связь, из всех векторов хранения.

Работа со многими связями и клиент множественного апплета

Мы рассмотрели два основных рабочих инструмента - потоки ServerWriter и ConnectionWatcher, которые руководят связями и генерируют выходные данные входных сообщений для всех клиентов. Единственная задача потока ServerWriter - ждать, пока его не "позовут" через его извещение (notify), затем взять сообщение, которое нужно послать клиентам, и послать его. ConnectionWatcher побуждается к действию своим собственным извещением, но, кроме того, он просыпается и запускается каждые 10 секунд. Его задача - проверять прочность каждой связи и убирать связи, переставшие функционировать.

Пример 15-1c. Программа сервера общения.

```
class ServerWriter extends Thread {
        chatserver server;
        public Vector OutputStreams;
        public FIFO outdata;
        private String outputline;
// Делаем OutputStreams и outdata
// общими для обеспечения удобства работы; это позволяет получить прямой
доступ,
// чтобы добавлять или убирать соответственно outputstream или данные
сообщения.
        public ServerWriter(chatserver s) {
                super("Server Writer");
// помещаем этот поток в родительский ThreadGroup и даем ему имя
                server = s;
                OutputStreams = new Vector();
                outdata = new FIFO();
                this.start();
// поток начинает работать
        }
        public synchronized void run() {
                while(true) {
// Поток совершает бесконечный цикл, но на самом деле он запускается только
тогда,
// когда условие ожидания задается заново с помощью извещения.
// Опять же, мы делаем это в синхронном блоке, чтобы запереть поток и
предотвратить
// множественный доступ.
                try this.wait(); catch (InterruptedException e)
                        System.out.println("Caught an Interrupted Exception");
        outputline = outdata.pop();
// Получаем сообщение в верхней части FIFO outdata, куда уведомляющий
// метод должен был добавить сообщение.
                synchronized(server.connections) {
// Мы должны еще запереть поток watcher,
// чтобы он не пытался что-нибудь делать до окончания работы программы.
                        for(int i = 0; i < OutputStreams.size(); i++) {
                                PrintStream out;
                                out = (PrintStream)OutputStreams.elementAt(i);
                                out.println(outputline);
// Делаем итерации по outputstreams и печатаем сообщение в каждый
outputstream.
                         }
```

```
                }
                }
        }
}
```

Суперкласс выполняет большую работу, он запускается каждый раз, как приходит новое сообщение от одного из клиентов. Кроме того, он запирает поток ConnectionWatcher: это необходимо делать для того, чтобы убедиться, что выходной поток данных до окончания работы не убран из вектора OutputStreams потока ServerWriter. Когда выясняется, что связь прервалась, поток ConnectionWatcher изменяет вектор OutputStreams потока ServerWriter.

Пример 15-1d. Программа сервера общения.

```
class ConnectionWatcher extends Thread {
        protected chatserver server;
        protected ServerWriter writer;
        protected ConnectionWatcher(chatserver s, ServerWriter writer) {
                super(s.CurrentConnections, "ConnectionWatcher");
// помещаем поток в родительский ThreadGroup и даем ему имя
                server = s;
                this.writer = writer;
                this.start();
        }
        // Этот метод ждет извещения о существующих потоках и очищает списки.
        // Этот метод синхронный, то есть перед запуском он запирает объект
this.
        // Это необходимо для того, чтобы метод мог вызвать wait() на this.
        // Даже если объекты Connection никогда не вызовут notify(), этот
метод включается
        // каждые пять секунд и на всякий случай проверяет все связи.
        // Заметим также, что весь доступ к вектору Vector и к компоненту GUI
List
        // производится тоже внутри синхронного блока. Это предохраняет класс
Server
        // от добавления новой связи, пока мы убираем старую связь.
public synchronized void run() {
// Мы запираем этот поток, когда он запускается,
// чтобы не запретить множественный доступ.
                while(true) {
// поток совершает бесконечный цикл
                        try this.wait(10000);
// Поток "запускается" каждые 20 секунд, чтобы
// убрать возможные несуществующие связи.
                        catch (InterruptedException e){
                                System.out.println("Caught an Interrupted
Exception");
                                }
                synchronized(server.connections) {
// Проходим по каждой связи.
                        for(int i = 0; i < server.connections.size(); i++) {
                                Connection c;
                                c =
(Connection)server.connections.elementAt(i);
                                if (!c.isAlive()) {
// Если связь больше не существует, убираем ее из Vector.
server.connections.removeElementAt(i);
                writer.outdata.push(server.connection_list.getItem(i)+" has
left chat\n");
synchronized(writer)writer.notify();
// Говорим другим клиентам, что данный пользователь вышел.
server.connection_list.delItem(i);
// Наконец, убираем его из списка связей сервера.
                                i--;
// Мы должны уменьшить счетчик, поскольку
// мы только что уменьшили вектор связей на единицу.
```

```
                                    }
                                    }
                            }
                            }
                    }
}
```

На этом мы завершили работу над программой для сервера. Воспользовавшись различными классами, содержащимися в Java API, мы создали вполне ошибкоустойчивый многопотоковый сервер. Этот сервер можно посмотреть в действии, воспользовавшись клиентом общения на http://www.vmedia.com/onlcomp/ java/chapter15/ChatClient.html. Теперь мы перейдем к клиенту общения. При его создании мы также будем использовать многопотоковость для получения асинхронного общения.

СОВЕТ Другой пример выполнения серверов на Java можно посмотреть в главе 19, "Как написать свой собственный сервер: планировщик встреч".

Построение клиента общения

Апплет клиента общения содержится в окне Web-броузера (рис. 15-2). Этот апплет автоматически соединяется с портом 6001 на том хосте, с которого загружается Web-страница. Мы открываем фрейм, чтобы пользователь мог ввести свое имя и представиться остальным пользователям; создаем два независимых потока - один для чтения из сети и второй для записи в сеть. В результате наш клиент получается действительно асинхронным, поскольку пользователь может вводить свое сообщение, в то время как на экране появляются новые сообщения от других пользователей.

Рис. 15.2.

Пример 15-2. Клиент общения

```
import java.io.*;
```

```
import java.net.*;
import java.awt.*;
import java.applet.*;
public class chatclient extends Applet {
public static final int DEFAULT_PORT = 6001;
public Socket socket;
private Thread reader, writer;
public TextArea OutputArea;
public TextField InputArea;
public PrintStream out;
public String Name;
public UserInfoFrame NameFrame;
        // создаем читающий и записывающий потоки и запускаем их
        public void init () {
                OutputArea = new TextArea(20, 45);
                InputArea = new TextField(45);
                NameFrame = new UserInfoFrame(this);
                add( new Label("Internet Turbo Chat"));
                add(OutputArea);
                add( new Label("Type a message below and hit ENTER"));
                add(InputArea);
                resize(400,400);
                        try {
                                socket = new
Socket(getDocumentBase().getHost(), DEFAULT_PORT);
                                reader = new Reader(this, OutputArea);
                        out = new PrintStream(socket.getOutputStream());
// Задаем различные приоритеты, чтобы консоль разделялась эффективно.
                        reader.setPriority(3);
                        reader.start();
                }
                catch (IOException e) System.err.println(e);
        }
public boolean handleEvent(Event evt) {
        if (evt.target == InputArea)
                {
                        char c=(char)evt.key;
                        if (c == '\n')
// Ждем, пока пользователь нажмет клавишу ENTER.
// Это показывает нам, что сообщение готово к отправке.
                {
                        String InLine = InputArea.getText();
                        out.println(Name + "> " + InLine);
                        InputArea.setText("");
// Отправляем сообщение, но добавляем к нему имя пользователя,
// чтобы другие клиенты знали, кто послал сообщение.
                         return true;
                }
        }
else if ( evt.target == NameFrame) {
// Первое введенное имя пользователя передается базовому апплету.
// Мы должны послать это имя на сервер, чтобы оно задало список
пользователей.
                Name = (String)evt.arg;
                out.println(Name);
// посылаем имя пользователя на сервер общения
                return true;
        }
        return false;
}
}
// Читающий метод читает входные данные из сокета
// и обновляет OutputArea, вводя новое сообщение.
class Reader extends Thread {
```

```
        protected chatclient client;
        private TextArea OutputArea;
        public Reader(chatclient c, TextArea OutputArea) {
                super("chatclient Reader");
                this.client = c;
                this.OutputArea = OutputArea;
        }
        public void run() {
                DataInputStream in = null;
                String line;
                try {
                        in = new
DataInputStream(client.socket.getInputStream());
                        while(true) {
// бесконечный цикл
                                line = in.readLine();
                                if (line == null) {
                                OutputArea.setText("Server closed
connection.");
                                break;
// цикл разрывается, когда связь прекратилась
                                }
                                OutputArea.appendText(line+"\n");
// добавляем новое сообщение к OutputArea
                        }
                }
                catch (IOException e) System.out.println("Reader: " + e);
                finally try if (in != null) in.close(); catch (IOException e)
;
                System.exit(0);
        }
}
// Это класс frame, который нужен, чтобы получить имя пользователя.
class UserInfoFrame extends Frame {
public TextField UserNameField;
public Applet parent;
public UserInfoFrame(Applet parent) {
        UserNameField = new TextField(10);
        this.parent=parent;
        add("North", new Label("Please enter your name and hit ENTER"));
        add("South", UserNameField);
        resize(300, 100);
        show();
}
// Передаем введенное имя в форме события, посланного апплету-родителю.
public boolean keyDown( Event evt, int key)
        {
                char c=(char)key;
                if (c == '\n')
                        {
                                Event NewEvent = new Event(this,
Event.ACTION_EVENT,
UserNameField.getText());
                                parent.postEvent(NewEvent);
// генерируем событие на родительском апплете
                                dispose();
// теперь разрушаем фрейм
                        return true;
                        }
                else { return false; }
        } // действие
}
```

Что дальше?

Вы получили полную порцию основного блюда, теперь готовьтесь к десерту! Мы обрисовали в общих чертах и обсудили темы, важные при создании апплетов игр, построении систем клиент-сервер, обладающих полным набором свойств, и почти все промежуточные темы. В части V мы предлагаем четыре учебных апплета, использующих принципы построения, технику и средства, описанные в этой книге. Желаем вам получить удовольствие!

Глава 16

Интерактивная анимация: рекламный апплет

Что вы узнаете из этой главы

Здесь мы создадим интерактивный анимационный апплет высокого уровня. Создание анимаций и интерактивности само по себе является достаточно простой задачей,, но мы хотим написать апплет,, с которым легко будет работать Web-дизайнерам,, не имеющим средств программирования на Java. Рассмотрим вначале,, какие знания нам для этого понадобятся:

- Использование классов URL для доступа к файлу конфигурации.
- Динамическая загрузка удаленных классов с использованием класса Class.
- Создание интерфейса для расширения возможностей конфигурации.
- Трассировка изображений с помощью MediaTracker.
- Применение двойной буферизации для устранения мерцания экрана.
- Распределение событий мыши.

Контракт

Представьте, что рекламное агентство, работающее в on-line, захотело разнообразить свою Web-страницу с помощью апплетов. Особенно оно заинтересовалось интерактивной анимацией, предоставляемой языком Java, и вас попросили создать апплет, который будет создавать анимацию из серии картинок, а также взаимодействовать с пользователем. Например, пользователь должен иметь возможность щелкнуть или переместить мышь над какой-то частью картинки, чтобы загрузить новую страницу или изменить анимацию. При этом, поскольку сотрудники агентства не могут программировать самостоятельно, требуется апплет высокого уровня, в котором задана возможность изменять картинки, составляющие анимацию, и смоделирована реакция апплета на действия пользователя.

Таким образом, у нашего апплета предполагаются две аудитории - пользователи Web, которые будут смотреть апплет, и Web-дизайнеры, которые будут его использовать для своих целей. Однако прежде чем углубиться в программирование, давайте посмотрим, как использовать для достижения желаемых целей то, что мы уже знаем. После этого мы построим и реализуем апплет высокого уровня.

Свойства

Мы знаем, что у нашего апплета будут две аудитории - пользователи Web и Web-дизайнеры из рекламного агентства. Давайте подумаем, какими свойствами должен обладать апплет, чтобы удовлетворить потребностям первой аудитории.

Цель апплета - оживить Web-страницу при передаче рекламного сообщения пользователю Web. Наш апплет - средство передачи сообщения, следовательно, мы должны быть уверены, что апплет правильно работает в качестве такого средства. Первая трудность, с которой мы столкнемся, состоит в том, что апплет должен быть загружен по сети и запущен. Поскольку он является только одним из элементов Web-страницы, люди могут не захотеть ждать, пока он появится перед глазами. Если загрузка будет занимать слишком много времени, человек может уйти со страницы еще до того, как он узнает о существовании апплета. Эта проблема особенно остра для такого апплета, как наш, потому что он должен загружать много картинок, а передача изображений по Интернет требует времени. Поэтому нужно сделать так, чтобы апплет запускался сразу же, еще до того, как загрузятся изображения.

Кроме того, мы должны быть уверены, что анимация изображений будет гладкой. Это достигается с помощью техники двойной буферизации, описанной в главе 5, "Апплет в работе".

Вопросы, как сделать быструю загрузку, гладкую анимацию и простой в использовании интерфейс, относятся к техническим аспектам функционирования апплета в качестве средства передачи сообщения. Теперь нам нужно подумать о том, как облегчить Web-дизайнеру процесс отображения сообщения. Мы не можем влиять на Web-дизайнера в его выборе графики для апплета, но мы можем сделать так, чтобы изображения легко встраивались в апплет.

Как обсуждалось в главе 5, Web-дизайнер может воспользоваться тегом <PARAM> для того, чтобы менять способ работы апплета. Но этого было бы достаточно, если бы мы описывали только серии изображений для анимации. Однако в нашем апплете Web-дизайнеру понадобится описывать не только области внутри каждой картинки, но также процесс взаимодействия изображений с пользователем. И вся эта информация должна содержаться в теге <APPLET>. А если Web-дизайнер захочет использовать ту же конфигурацию для другой Web-страницы, потребуется переносить части текста с одной страницы на другую.

Такой перенос текста может быть очень утомительным, поэтому мы будем хранить конфигурационную информацию в отдельном файле. Тег param-value будет использоваться только для указания на конфигурационный файл. Этот файл будет написан на очень простом языке программирования, который мы разработаем специально для этого апплета. Тут мы, конечно, легко можем увлечься, в результате чего Web-дизайнеру придется потом изучать сложный и разнообразный синтаксис. Но лучше не допустить этого, ведь наш заказчик хочет, чтобы апплет был формируемым и простым для использования.

Мы говорили о двух основных аудиториях, для которых предназначен апплет, - Web-пользователи и Web-дизайнеры. Давайте теперь рассмотрим, что мы - программисты - можем получить от опыта создания апплета. Мы воспользуемся техникой, описанной в главе 10, "Структура программы", чтобы убедиться, что у нас есть компоненты для неоднократного использования, и построим апплет таким образом, чтобы потом в него легко было добавлять новые функции.

План работы

Ну что ж, проблемы, связанные с нашим будущим апплетом, мы уже обсудили, давайте обдумывать решение. Представим себе наш апплет с точки зрения внешней и внутренней сторон. Внешняя сторона апплета - это та часть, которую видит пользователь, а внутренняя - это то, что

использует Web-дизайнер для формирования апплета. Внешняя сторона - это логическая отправная точка, поскольку лишь создав нечто, можно думать о его формировании.

Создание структуры изображения

Мы уже знаем, как обращаться со многими основными элементами нашего изображения, умеем анимировать картинки и осуществлять взаимодействие с мышью. По сути, все, что мы делаем, состоит из этих двух элементов. Когда мы перемещаем мышь или щелкаем мышью по картинке, мы смотрим на области, описанные для данного изображения, и, если мышь находится внутри одной из них, выполняем соответствующее действие.

Давайте на время оставим анимацию и подумаем о том, как выделить интерактивную часть нашего апплета. Первой нашей задачей будет выяснение того, находится ли мышь внутри области, описанной Web-дизайнером. Беглый обзор электронной документации по Abstract Windows Toolkit показывает, что в нашем распоряжении имеется класс Polygon. Кроме того, AWT содержит методы, позволяющие определить, находится ли данная точка внутри фигуры. Так что нам не придется писать сложные алгоритмы для определения местоположения мыши; для этого достаточно создать реализацию классов Polygon и позволить им делать необходимые расчеты.

Следующей задачей будет задание правильной реакции на перемещение или щелкание мыши внутри одной из определенных областей. Одно из решений - создание длинного ряда операторов if-then-else в методах mouseMove и mouseDown. Мы можем при конфигурации апплета загрузить каждую из наших областей и соответствующих ей действий в хеш-таблицу и при каждом действии мыши смотреть в таблицу. Будем считать, что мы используем классы Polygon только для описания областей и поместим их всех в хеш-таблицу под названием areasTable. Тогда метод mouseDown может обходиться с отвечающими на щелчок мыши областями следующим образом:

```
public boolean mouseDown(Event evt, int x, int y) {
        Enumeration e=areasTable.keys();
        while (e.hasMoreElements()) {
                Polygon p=e.getNextElement();
                if (p.inside(x,y)) {
                String S=(String)areasTable((Polygon(p));
                if (S.equals("sound action"))
                        // издаем звук
                if (s.equals("link action"))
                        // связь со страницей
                if (s.equals("redirect action"))
                // меняем анимацию
        // ...и т. д., для всех типов областей
                }
        return true;
}
```

В нашем примере возникает несколько проблем, связанных с эффективностью работы программы, которых можно избежать. Однако, сделав программу более эффективной, мы можем сузить область ее применения. Если мы будем пользоваться методом просмотра таблицы, нам придется дописывать программу каждый раз при возникновении нового типа действия. Вместо этого можно создать базовый класс для активных областей. Тогда наш апплет будет просто говорить активной области делать то, что она должна, и метод mouseDown будет гораздо проще. Поскольку нам больше не нужна хеш-таблица, будем хранить наши области actionArea в векторе aAreas:

```
public boolean mouseDown(Event evt, int x, int y) {
        Enumeration e=aAreas.elements();
        while (e.hasMoreElements()) {
                actionArea A=e.NextElement();
                if (A.inside(x,y))
                        A.doAction();
        }
```

Теперь можно не переписывать апплет каждый раз, когда мы создаем новую активную область. Кроме того, метод mouseDown стал гораздо яснее. Но что если мы захотим создать две одновременные анимации или добавить апплету дополнительные функции? Проблема в том, что мы поместили все функции непосредственно во внутренние методы апплета.

Гораздо лучше будет создать компонент, который затем можно поместить в апплет. Поскольку класс Applet является подклассом класса Container, он автоматически передаст свои события компонентам, которые в нем содержатся. Фактически выполнение нашей работы внутри компонента облегчит обмен интерактивной анимацией с любым апплетом. Это примерно то же самое, что использование Scrollbar или любого другого подкласса класса Component. Но где нам сделать подкласс? Можно сделать подкласс непосредственно в классе Component, но лучше давайте сделаем подкласс в Canvas. Класс Canvas облегчит задачу размещения рисунка, а это важно, если мы хотим, чтобы рисунок можно было перемещать. Назовем этот подкласс класса Canvas actionImageCanvas.

Итак, возникла необходимость в нескольких классах. У нас обязательно будет подкласс класса Applet, но с этого момента он будет играть очень маленькую роль - содержать лишь actionImageCanvas. Кроме того, у нас есть подклассы класса actionArea. Стоит ли помещать подклассы actionArea непосредственно в actionImageCanvas? Для этого придется перенести изображения непосредственно на поверхность рисунка, а это может оказаться неудобно, если мы когда-нибудь захотим что-то создать или поменять в изображениях на стороне клиента.

Лучше создадим класс actionImage, который будет отображать сам себя и следить за actionArea. В результате мы создадим иерархию для нашего внешнего интерфейса, изображенную на рис. 16-1. Заметим, что эта иерархия не является иерархией в терминах наследования, а только в терминах того, как мы структурируем наше отображение. Скорее, это иерархия вложений.

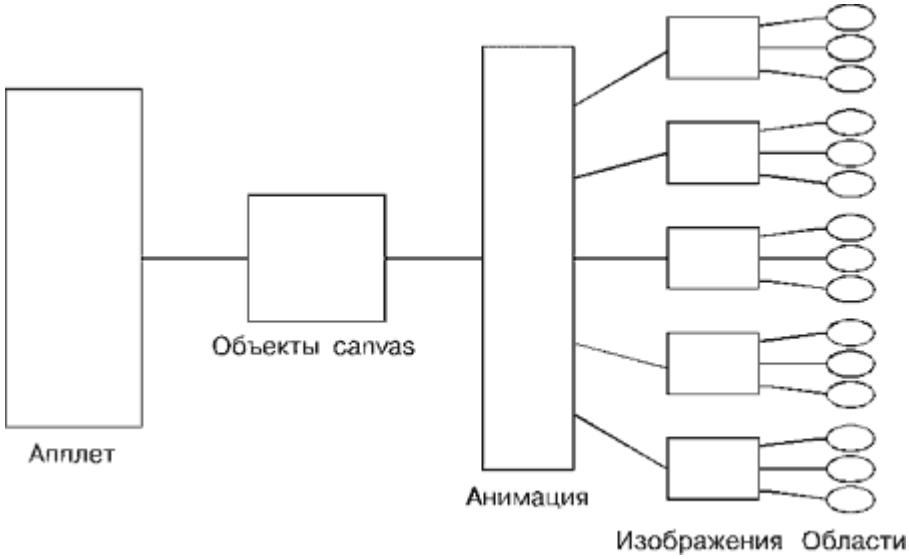

Рис. 16.1.

Теперь посмотрим, как в нашу схему встраивается анимация. Концепция анимации та же, что и для любой другой анимации, за исключением того, что мы не просто отображаем картинку, а должны активизировать определенный объект actionImage. Единственный вопрос при этом заключается в том, какой класс нашей иерархии должен запустить и поддерживать работу потока. Мы смело можем исключить из рассмотрения подкласс апплета Applet - вся суть actionImageCanvas состоит в том, чтобы отделить проект от апплета. Запуск потока из actionImageCanvas создаст сложности, если мы когда-нибудь захотим запустить больше одной анимации внутри одной рабочей области. Таким образом, остается поручить запуск анимации классу actionImage. В табл. 16-1 показано, какие функции приписаны различным классам нашей системы.

Таблица 16-1. Функции классов

Класс	Функция
подкласс класса Applet	Передача события классам actionImageCanvas.
actionImageCanvas	Создание поверхности рисунка для классов actionImage. Проверка диапазонов и конфликтов для множественных анимаций. Передача события классу actionImage.
actionImage	Запуск анимации. Обмен с нужными классами actionArea. Передача события классам actionArea.
actionArea	Выполнение действия.

Конечно, здесь мы передаем события через каждый из наших классов, хотя могли бы все делать внутри апплета. Но та иерархия, которую мы создали, допускает возможность расширения на каждом уровне.

Компоновка структуры изображения

Теперь, когда у нас есть гибкая структура для нашего экранного вывода, пора подумать о том, как все это будет сформировано. Как уже отмечалось, далеко не идеальным решением было бы считывать все необходимые параметры с Web-страницы. Вместо этого мы откроем URL-соединение и прочтем текстовый файл с параметрами. Чтобы не нарушать объектно-ориентированного стиля, создадим класс для работы с такой конфигурацией. Назовем его actionAnimConfigure.

При этом мы должны тщательно следить за тем, чтобы гибкость нашей структуры была доступна Web-дизайнеру. Если мы хотим добавить новый тип класса actionArea, будет удобно не переписывать actionAnimConfigure. В этом деле нашим секретным оружием будут возможности языка Java по динамической загрузке и связыванию, которые мы обсуждали в главе 10, "Структура программы". Имея некоторый базовый класс, мы можем создать экземпляр подкласса с данным именем. Допустим, к примеру, что у нас есть подкласс класса actionArea, называемый linkArea, задача которого - загружать данную страницу. Задав строку "linkArea", мы можем создать экземпляр этого класса, как это показано ниже. Однако в процессе загрузки по сети класса, основанного только на его имени, может возникнуть несколько исключений.

```
try {
String S="linkArea";
actionArea baseArea = (actionArea)Class.forName(S).newInstance();
} catch (ClassNotFoundException e) {
        System.out.println("Couldn't find"+S);
        System.out.println(e.getMessage());}
catch (InstantiationException e) {
        System.out.println("Couldn't create"+S);
        System.out.println(e.getMessage());}
catch (IllegalAccessException e) {
        System.out.println("Couldn't access"+S);
        System.out.println(e.getMessage());}
```

Мы создаем экземпляр linkArea, но он приведен к actionArea. Это означает, что мы можем вызвать только один из методов, определенных в нашем базовом классе actionArea. Но когда этот метод вызывается, фактически активизируется метод в linkArea.

Фокус состоит в том, чтобы прочитать строку, которую дает Web-дизайнер в конфигурационном файле, и воспользоваться ею для создания подклассов нашего базового класса. Мы хотели бы это делать таким образом, чтобы сохранялся высший уровень конфигурационных возможностей. Если впоследствии мы создадим подкласс ActionImageCanvas и добавим ему новые функции, хотелось бы, чтобы Web-дизайнер мог воспользоваться этими функциями, всего лишь изучив пару новых параметров, которые нужно задать в конфигурационном файле. В то же время нам нужно, насколько возможно, не допускать Web-дизайнера к внутренним хитросплетениям программы. В конце концов, это в большой степени наш долг как программистов.

Как уже говорилось выше, наша система будет работать со своим собственным очень простым языком программирования. Рассмотрим для начала требования, налагаемые системой на этот язык. Нам нужно будет иметь возможность выяснять, какой класс необходим Web-дизайнеру, но не забывайте, что просто создание экземпляра класса еще не делает его пригодным для функционирования. Еще нам понадобится способ внесения дополнительной информации в экземпляр. Например, наши классы actionArea должны знать свое расположение на изображении. Кроме того, каждый подкласс класса actionArea должен располагать информацией о собственном действии. Класс linkArea должен знать, например, с какой страницей связываться. Таким образом, наш язык должен иметь возможность передавать объекту незаданное заранее количество конфигурационной информации.

Вдобавок к этому мы должны быть уверены, что передаем информацию нужному объекту. Один фрейм анимации может содержать несколько классов linkArea, и мы должны убедиться в том, что не путаем классы linkArea одного фрейма с linkArea другого фрейма. Таким образом, мы должны иметь возможность дифференцировать конфигурационную информацию, чтобы она попала в нужное место.

В то же время язык должен быть простым. Нам известно, что Web-дизайнер знает HTML и вынужден следить за его постоянно возникающими расширениями. Так что можно создать язык,

по сложности подобный HTML, или использовать HTML в качестве модели.

Попытаемся создать конфигурационный язык по подобию HTML. Мы хотим описать интерактивную анимацию с двумя изображениями. Первая картинка будет содержать два класса ShowDocArea, а вторая будет содержать ShowDocArea и soundArea:

```
<ActionImage=ActionImage>
image=someImage.gif
next=1
<ShowDocArea=ShowDocArea>
doc=http://www.vmedia.com
area=0,0;10,0;10,10;0,10
</ShowDocArea>
<ShowDocArea=ShowDocArea>
doc=http://www.vmedia.com/java
area=10,10;10,20;20,20;20,10
</ShowDocArea>
</ActionImage>
<ActionImage=ActionImage>
image=someOtherImage.gif
next=0
<ShowDocArea=ShowDocArea>
doc=http://www.vmedia.com/java
area=0,0;10,0;10,10;0,10
</ShowDocArea>
<SoundArea=SoundArea>
sound=someSound.au
area=10,10;10,20;20,20;20,10
</SoondArea>
</ActionImage>
```

Позаимствовав стиль языка HTML, мы можем создать несложный язык, удовлетворяющий нашим требованиям и в то же время понятный всем, кто имеет опыт создания Web-страниц. Остается несколько проблем, которые надо разрешить. Во-первых, нельзя делать несколько анимаций внутри одного ActionImageCanvas. Эту проблему можно решить, добавив тег Animation более высокого уровня:

```
<ActionImageAnimation=ActionImageAnimation>
теги ActionImageAnimation...
</ActionImageAnimation>
<ActionImageAnimation=ActionImageAnimation>
теги ActionImageAnimation...
</ActionImageAnimation>
```

Теперь мы знаем, что содержащиеся в тексте теги относятся к разным анимациям. Для того чтобы определить размер рабочей области и допустить возможность создания подкласса ActionImageCanvas с дополнительными функциями, можно применить одну и ту же стратегию. Можно еще разрешить задание по умолчанию некоторых описаний типа во фрейме и в тегах для рабочей области. Окончательно язык будет сформулирован при реализации класса Configure. Пока что мы можем достаточно свободно описать язык, как это сделано ниже. Не забудьте, что каждый тег на уровне ниже анимации может встречаться несколько раз:

```
<ActionImageCanvas=имя подкласса>
параметры=значения
<ActionImageAnimation=имя подкласса>
параметры=значения
<ActionImage=имя подкласса>
параметры=значения
<ActionArea=имя подкласса>
</ActionArea>
</ActionImage>
</ActionImageAnimation>
</ActionImageCanvas>
```

Построив язык для Web-дизайнера, мы должны решить, как будет действовать класс Configure при запуске апплета. В этой ситуации нам потребуются интерфейсы. Мы можем воспользоваться интерфейсом под названием Configurable. Наш класс Configure сможет передать информацию, вызвав методы, определенные в интерфейсе Configurable, как это показано на рис. 16-2.

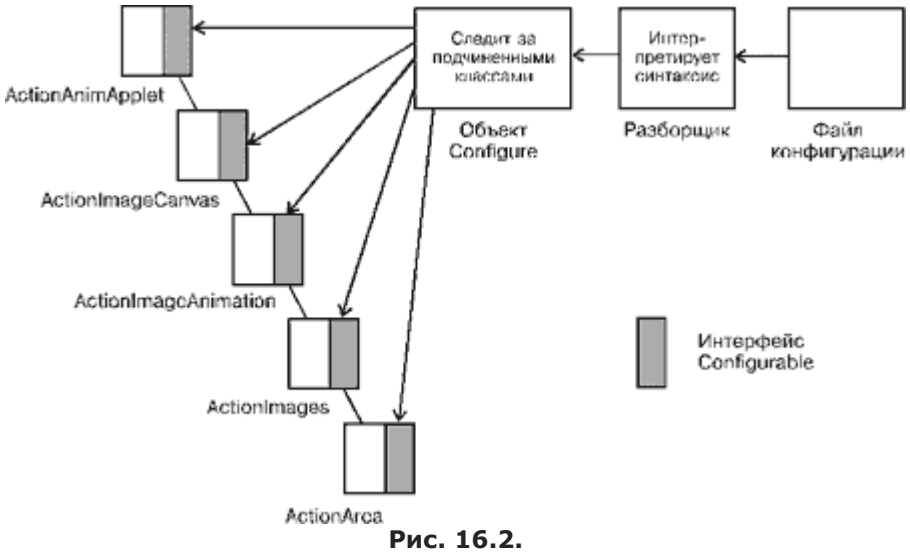

Рис. 16.2.

Реализация

Закончив проектирование, мы можем приступить к написанию программы. Только с чего начать? Можно написать апплет, добиться, чтобы заработала анимация, а потом модифицировать наш проект. Но если подумать о том, сколько различных систем должны взаимодействовать для того, чтобы наша система работала, то получается, что лучше всего будет для начала написать по отдельности каждый модуль. Это означает, что потребуется больше времени, пока программа реально заработает, зато когда уже она заработает, мы окажемся ближе к поставленной цели.

Возможности конфигурации

Для начала нам понадобится интерфейс Configurable для установки системы. Придется подключать модули из нашей иерархии к модулям более высокого уровня. Известно также, что, когда модуль подключен, ему нужно передать конфигурационную информацию. Хорошо было бы дать модулю знать, когда конфигурация будет закончена. Затем модуль может проверить, достаточно ли у него информации для того, чтобы начать действовать. В результате мы имеем три метода для нашего интерфейса Configurable:

```
        Interface Configurable {
        public void attachObject(Object o) throws Exception;
        public boolean setParent(Configurable parent);
        public void configureObject(String param, String value) throws
Exception;
        public void completeConfiguration() throws Exception;
}
```

Заметим, что метод attachObject соответствует экземпляру объекта; это означает, что Configure реализует класс, после чего передает его одному из модулей нашей системы. Заметим также, что последние два метода вызываются из объекта, который должен быть сконфигурирован, а не из того, к которому он подключен.

Мы задали последним двум методам возможность выдавать исключение, но, как говорилось в главе 10, "Структура программы", было бы плохо не дать этим исключениям более описательные имена. Так что давайте зададим иерархию исключений, сопровождающую интерфейс Configurable. Исключение верхнего уровня можно назвать ConfigurableException, чтобы мы знали, где оно возникло. Теперь посмотрим, что может работать неправильно в последних двух методах:

- configureObject может не узнать параметр.
- configureObject может не суметь вычислить значение.

- completeConfiguration может не суметь получить информацию и завершить конфигурацию.

Заданная иерархия исключений для Configurable приводится на рис. 16-3.

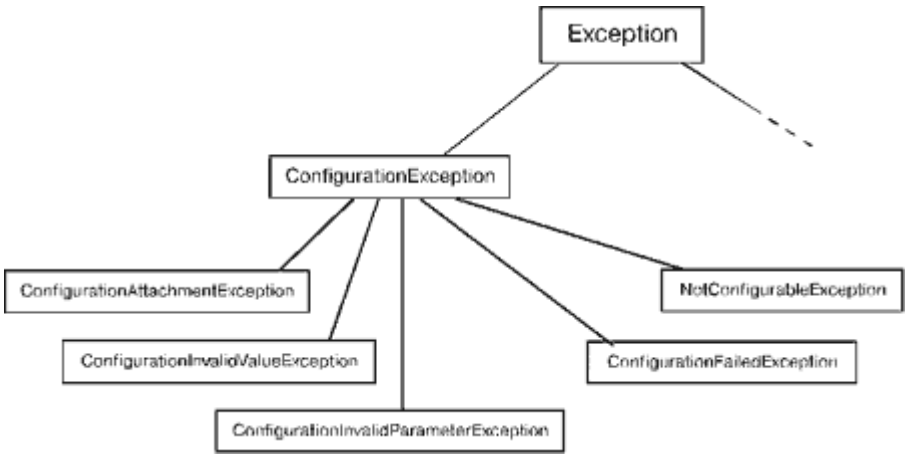

Рис. 16.3.

Теперь мы можем закончить программирование интерфейса Configurable и на этом завершить первую реальную программу нашего проекта. Поскольку она не является специфической, поместим ее в пакет ventana.util:

```
package ventana.util;
public interface Configurable {
        public void attachObject(Object o) throws
                ConfigurationAttachmentException;
        public void setParent(Object o);
        public void configureObject(String param, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException;
        public void completeConfiguration() throws
ConfigurationFailedException;
}
```

Следующим шагом будет создание базового класса Configure. Он может потребоваться нам впоследствии, поэтому напишем общий класс, соответствующий интерфейсу Configurable, и поместим его в ventana.util. Чтобы этот класс оставался общим, будем проводить разбор нашего языка в специальном классе ConFileParser:

```
package ventana.util;
import java.util.*;
public class Configure {
        private ConfFileParser parser;
        private Stack Configurables;
        private Configurable original;
public Configure(ConfFileParser cFP, Configurable startObject) {
                parser=cFP;
                parser.setConfigure(this);
                Configurables=new Stack();
                Configurables.push(startObject);
                }
public boolean startConfiguration() {
        return parser.start();}
protected Configurable currentConfigurable() {
        return (Configurable)Configurables.peek();}
// следующие методы вызываются анализатором
public void attach(String className) throws
        NotConfigurableException,
        ConfigurationAttachmentException,
        ClassNotFoundException,
```

```
        InstantiationException,
        IllegalAccessException{
        // анализатор выясняет, где произошел сбой
        Class thisClass=Class.forName(className);
        // проверяем, действительно ли это Configurable
        Class interfaces[]=thisClass.getInterfaces();
        boolean isConfigurable=false;
        for (int i=0;i<interfaces.length;i++)
                if(interfaces[i].equals(Class.forName
("ventana.util.Configurable")))
                isConfigurable=true;
        if(!isConfigurable)
        throw (new NotConfigurableException(className));
        Object instance=thisClass.newInstance();
        currentConfigurable().attachObject(instance);
        Configurables.push((Configurable)instance);
        }
public void passToCurrent(String parameter, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException
{
        currentConfigurable().configureObject(parameter,value);}
// Может быть вызван непосредственно из анализатора, но тогда целые числа
// сохраняются как целые - лучше, чтобы анализатор не имел доступа к стеку.
public void configureCurrent() throws
ConfigurationFailedException{
currentConfigurable().completeConfiguration();
if (!Configurables.empty())
        Configurables.pop();
else throw
        (new ConfigurationFailedException("empty stack"));
}
}
```

Базовые классы для экранного вывода

Теперь у нас написан интерфейс Configurable и мы можем написать базовые классы для нашего экранного вывода. Наиболее важным этапом здесь будет реализация методов интерфейса Configurable по мере продвижения вниз по иерархии. При этом мы определим также другие методы, которые понадобятся нашим классам.

Начнем с класса базового апплета для нашей системы. Хотя любой класс, реализующий Configurable, сможет провести разбор файлов и сконфигурировать их, апплеты, которые должны запускать ActionImageAnimation, должны создать подкласс в этом классе. Классу ActionImage понадобится метод mediaTrackerHandle, чтобы его экземпляры загружали изображения:

```
package ventana.aia;
import java.awt.*;
import java.applet.*;
import java.util.*;
import java.net.*;
import java.io.*;
import ventana.util.*;
public class ActionAnimApplet extends Applet implements Configurable{
        private boolean configOk=false;
        Vector canvases=new Vector();
        MediaTracker imageLoader;
        public void init() {
                imageLoader=new MediaTracker(this);
                URL docURL=getDocumentBase();
                String confName=getParameter("conf");
                try {
                URL confURL=new URL(docURL,confName);
                        AIAnimParser p=new AIAnimParser(confURL);
```

```
                Configure Conf=new Configure(p,this);
                        configOk=Conf.startConfiguration();
                        if (!configOk) return;
                }
                catch (MalformedURLException e) {
                        showStatus(confName+" invalid URL");
                        System.out.println(confName);
                         System.out.println(e.getMessage());
                                stop();}
                catch (IOException e) {
                        showStatus(confName+" not accessible");
                        System.out.println(e.getMessage());
                        stop();}
                System.out.println("init complete");
        }
        public void paint(Graphics g) {
                paintComponents(g);}
private ActionImageCanvas getCanvas(int i) {
        return (ActionImageCanvas)canvases.elementAt(i);}
        public void start() {
                if (configOk) {
        for (int i=0;i<canvases.size();i++) {
                getCanvas(i).show();
                getCanvas(i).beginAction();}
                        }
        }
public void stop() {
        for (int i=0;i<canvases.size();i++)
                getCanvas(i).stopAction();}
protected MediaTracker mediaTrackerHandle() {
        return imageLoader;}
protected ActionAnimApplet actionAnimAppletHandle() {
        return this;}
// реализация Configurable
public void attachObject(Object o) throws ConfigurationAttachmentException {
         if (!(o instanceof ActionImageCanvas))
        throw(new ConfigurationAttachmentException("not ActionImageCanvas"));
        // проверяем, правильный ли тип объекта мы взяли
        ActionImageCanvas cur=(ActionImageCanvas)o;
        canvases.addElement(cur);
                cur.setParent(this);
         add(cur);}
public void configureObject(String param, String value)
throws ConfigurationInvalidParameterException,
        ConfigurationInvalidValueException {
        throw (new ConfigurationInvalidParameterException
                        ("No Configuration parameters"));
        }
public String toString() {
        String S="ActionAnimApplet: \n";
        for (int i=0;i<canvases.size();i++) {
                S=getCanvas(i).toString();}
        return S;}
public void completeConfiguration() throws
        ConfigurationFailedException {
        for (int i=0;i<canvases.size();i++)
                 getCanvas(i).setParent(this);
        }
public void setParent(Object o) {
        System.out.println("no parent can be set for ActionAnimApplet");}
}
```

Действие подкласса Applet - все запустить. Заметим, что метод paint просто указывает компонентам рисовать самих себя. Это позволяет нам легко включить в апплет другие свойства. Теперь мы можем начать нашу иерархию с нижнего базового класса ActionImageCanvas:

```
package ventana.aia;
import java.util.*;
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
import ventana.util.*;
import ventana.awt.*;
public class ActionImageCanvas extends Canvas implements Configurable{
        private int canvasWidth=200;
        private int canvasHeight=200;
        private Vector Anims=new Vector();
        private Color backGroundColor=Color.lightGray;
        private Vector curActionImages=new Vector();
        private ActionAnimApplet motherApplet;
        private BufferedImageGraphics buffer;
        private Event curEvent;
public void paint(Graphics g) {
        if (buffer.needBuffer()) {
                Image bI=createBackground();
                buffer.setBuffer(bI);}
buffer.paintBuffer(g);}
public synchronized Image createBackground() {
// нужно создать подкласс,
// если вы хотите иметь фон в виде изображений и т. п.
        return createImage(size().width,size().height);}
// разрешаем потоку ActionImageAnimation обновлять canvas
public ActionImage getCurActionImage(int i) {
        return (ActionImage) curActionImages.elementAt(i);}
public synchronized void addToCanvas(ActionImage AI) {
        curActionImages.addElement(AI);
        buffer.addImage(translateActionImage(AI));
}
private PositionedImage translateActionImage(ActionImage AI) {
        Image I=AI.getImage();
        int x=AI.XPos();
        int y=AI.YPos();
        int iWidth=I.getWidth(motherApplet);
        int iHeight=I.getHeight(motherApplet);
        Rectangle r=new Rectangle(x,y,iWidth,iHeight);
        return new PositionedImage(I,r);}
public synchronized boolean removeImageFromCanvas(ActionImage AI) {
        return buffer.removeImage(translateActionImage(AI));
                }
public synchronized void removeAreaFromCanvas(ActionImage AI) {
        curActionImages.removeElement(AI);}
public boolean handleEvent(Event evt) {
        curEvent=evt;
        return super.handleEvent(evt);}
public void updateCanvas() {
        if (curEvent!=null)
        handleEvent(curEvent);
        refreshCanvas();}
public void refreshCanvas() {
        repaint();}
// передаем события текущему ActionImage
private boolean insideActionImage(int x, int y, ActionImage AI) {
        if(x<AI.XPos() || y<AI.YPos())
        return false;
        int imgWidth=AI.getImage().getWidth(motherApplet);
        int imgHeight=AI.getImage().getHeight(motherApplet);
```

```
        if(x>AI.XPos()+imgWidth)
        return false;
        if(x>AI.YPos()+imgHeight)
                return false;
return true;}
private Rectangle AIrect(ActionImage AI) {
        int x=AI.XPos();
        int y=AI.YPos();
        int width=AI.getImage().getWidth(motherApplet);
        int height=AI.getImage().getHeight(motherApplet);
        return new Rectangle(x,y,width,height);}
public boolean mouseMove(Event evt, int x, int y) {
        boolean shouldRefresh=false;
        for (int i=0;i<curActionImages.size();i++) {
                ActionImage someAI=getCurActionImage(i);
if(insideActionImage(x,y,someAI)) {
        int relativeX=x-someAI.XPos();
        int relativeY=y-someAI.YPos();
                Graphics gC=buffer.graphicsContext(AIrect(someAI));
        shouldRefresh=someAI.mouseMove(relativeX,relativeY,gC);
        }
}
if (shouldRefresh) refreshCanvas();
return true;
}
public boolean mouseDown(Event evt, int x, int y) {
        boolean shouldRefresh=false;
        for (int i=0;i<curActionImages.size();i++) {
                ActionImage someAI=getCurActionImage(i);
if(insideActionImage(x,y,someAI)) {
        int relativeX=x-someAI.XPos();
        int relativeY=y-someAI.YPos();
        Graphics gC=buffer.graphicsContext(AIrect(someAI));
        shouldRefresh=someAI.mouseDown(relativeX,relativeY,gC);
        }
}
if (shouldRefresh) refreshCanvas();
return true;
}
public boolean mouseDrag(Event evt, int x, int y) {
        boolean shouldRefresh=false;
        for (int i=0;i<curActionImages.size();i++)   {
                ActionImage someAI=getCurActionImage(i);
        if(insideActionImage(x,y,someAI)) {
                 int relativeX=x-someAI.XPos();
                 int relativeY=y-someAI.YPos();
                 Graphics gC=buffer.graphicsContext(AIrect(someAI));
                 shouldRefresh=someAI.mouseDrag(relativeX,relativeY,gC);
        }
}
        if (shouldRefresh) refreshCanvas();
        return true;
}
public boolean mouseUp(Event evt, int x, int y) {
        boolean shouldRefresh=false;
        for (int i=0;i<curActionImages.size();i++)    {
                ActionImage someAI=getCurActionImage(i);
        if(insideActionImage(x,y,someAI)) {
                int relativeX=x-someAI.XPos();
                int relativeY=y-someAI.YPos();
                Graphics gC=buffer.graphicsContext(AIrect(someAI));
                someAI.mouseUp(relativeX,relativeY,gC);}
        }
if (shouldRefresh) refreshCanvas();
```

```
return true;
}
// реализация Configurable
public void setParent(Object o) {
        motherApplet=(ActionAnimApplet)o;
        setBackground(motherApplet.getBackground());}
public ActionAnimApplet getApplet() {
        return motherApplet;}
public void attachObject(Object o) throws
        ConfigurationAttachmentException {
        if (!(o instanceof ActionImageAnimation))
                throw (new ConfigurationAttachmentException("not
ActionImageAnimation"));
        ActionImageAnimation cur=(ActionImageAnimation)o;
        Anims.addElement(cur);
        cur.setParent(this);
}
public void configureObject(String param, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException {
        param=param.toLowerCase();
        value=value.toLowerCase();
        try {
        if (param.equals("width")) {
                canvasWidth=Integer.parseInt(value);
        return; }
if (param.equals("height")) {
        canvasWidth=Integer.parseInt(value);
        return; }
} catch (NumberFormatException e) {
throw new ConfigurationInvalidValueException("Not a number");}
throw
        (new ConfigurationInvalidParameterException
                        ("Unrecognized: "+param));}
public void completeConfiguration() throws
        ConfigurationFailedException {
                buffer=new
BufferedImageGraphics(motherApplet,getBackground());
        for (int i=0;i<Anims.size();i++) {
                getAnim(i).setParent(this);
                getAnim(i).setApplet(motherApplet);}
        }
public String toString() {
        String S="ActionImageCanvas\n";
        S=S+"bgColor"+backGroundColor.toString()+"\n";
        for (int i=0;i<Anims.size();i++)
                S=S+getAnim(i).toString();
                return S;}
// этот метод запускает анимационные потоки в canvas; вызывается апплетом
protected void beginAction() {
        for (int i=0;i<Anims.size();i++)
                getAnim(i).startAnimation();}
protected void stopAction() {
        for (int i=0;i<Anims.size();i++)
                getAnim(i).stopAnimation();}
private ActionImageAnimation getAnim(int i) {
        return
(ActionImageAnimation)Anims.elementAt(i);}
public Dimension minimumSize() {
        return new Dimension(canvasWidth,canvasHeight);
}
public Dimension preferredSize() {
        return minimumSize();
}
```

Мы снова разрешаем множественные экземпляры следующего члена иерархии ActionImageAnimation. Этот класс фактически рисует фреймы. Для осуществления двойной буферизации воспользуемся отдельным классом BufferedImageGraphics:

```
package ventana.awt;
import java.awt.*;
import java.awt.image.*;
import java.util.*;
public class BufferedImageGraphics {
        private Graphics graphicsBuf;
        private Image imageBuf;
        // используем для выполнения двойной буферизации
        private ImageObserver imageObsv;
        private Vector curImages=new Vector();
        private int originalWidth;
        private int originalHeight;
        private Rectangle cropRect=new Rectangle();
        // область обрезки
        private boolean waitToTouchImages=false;
        // охраняет буфер во время процесса
        // добавления или удаления изображений
        Color backgroundColor;
        public BufferedImageGraphics(ImageObserver observer, Color c) {
                imageObsv=observer;
                backgroundColor=c;
        }
public boolean needBuffer() {
        return (graphicsBuf==null && imageBuf==null);}
public void setBuffer(Image i) {
        imageBuf=i;
        originalWidth=i.getWidth(imageObsv);
        originalHeight=i.getHeight(imageObsv);
        graphicsBuf=imageBuf.getGraphics();
        graphicsBuf.setColor(backgroundColor);
        graphicsBuf.fillRect(0,0,originalWidth,originalHeight);}
public synchronized void addImage(PositionedImage pI) {
        while (waitToTouchImages);
        waitToTouchImages=true;
        // Поскольку мы должны рисовать картинку и убрать обрезку, может
возникнуть
        //  команда сбросить буфер до окончания работы. Это приведет к хаосу.
        Rectangle r=pI.getRect();
        // выясняем, как это влияет на область обрезки
        if (cropRect==null || cropRect.isEmpty()) cropRect=r;
        else
                cropRect=cropRect.union(r);
        curImages.addElement(pI);
        if (graphicsBuf!=null)
                graphicsBuf.drawImage
(pI.getImage(),r.x,r.y,backgroundColor,imageObsv);
        waitToTouchImages=false;}
private PositionedImage getCurPosImage(int i) {
        return (PositionedImage)curImages.elementAt(i);}
public synchronized boolean removeImage(PositionedImage I)
{
        while (waitToTouchImages);
        waitToTouchImages=true;
        int imgIndx=curImages.indexOf(I);
        if (imgIndx==-1) {
        waitToTouchImages=false;
        return false;}
        Rectangle clearReg=getCurPosImage(imgIndx).getRect();
        // область для очистки
        curImages.removeElementAt(imgIndx);
```

```
        graphicsBuf.fillRect
        (clearReg.x,clearReg.y,clearReg.width,clearReg.height);
        // очистка области
        cropRect=new Rectangle();
        // подготовка к сбросу области обрезки
for (int i=0;i<curImages.size();i++) {
        PositionedImage somePI=getCurPosImage(i);
        Image img=somePI.getImage();
        Rectangle someRect=somePI.getRect();
        cropRect.add(someRect);
        int x=someRect.x;
        int y=someRect.y;
        graphicsBuf.drawImage(img,x,y,imageObsv);
        }
waitToTouchImages=false;
return true;}
public Graphics graphicsContext(Rectangle r) {
        cropRect.add(r);
        return graphicsBuf.create(r.x,r.y,r.width,r.height);}
public synchronized void paintBuffer(Graphics g) {
        while (waitToTouchImages);
        waitToTouchImages=true;
        Rectangle curCR=g.getClipRect();
        if (curCR==null) curCR=new
Rectangle(0,0,originalWidth,originalHeight);
        g.clipRect(cropRect.x,cropRect.y,cropRect.width,cropRect.height);
        g.drawImage(imageBuf,0,0,backgroundColor,imageObsv);
        g.clipRect(0,0,originalWidth,originalHeight);
        cropRect=new Rectangle();
        // сброс области обрезки
        waitToTouchImages=false;
}
}
```

Посмотрим теперь, как будут передаваться события. Мы передаем события непосредственно всем ActionImage, которые в данный момент отображены. Поскольку наш ActioImageCanvas является компонентом, содержащимся в апплете, нам не нужно вручную передавать события наружу из апплета - это делается автоматически. А поскольку ActionImage не является подклассом класса Component, передачу должен выполнять ActionImageCanvas. Мы убеждаемся в том, что событие передано нужному ActionImage и что координаты являются относительными.

Почему бы не создать подкласс в Component?

Мы могли бы осуществить передачу событий, создав подкласс ActionImageCanvas в Container и подкласс ActionImage в Component. Однако фактически задача Container - содержать элементы экрана, которые не надо перемещать. Чтобы не бороться с трудностями при создании диспетчера компоновки, который будет отвечать за перемещение изображений, напишем новый класс, который будет уметь обращаться с событиями мыши.

Создадим базовый класс ActionImageAnimation. Этот базовый класс не будет пытаться перемещать картинки, но в нем можно создать подкласс, разрешающий разнообразные перемещения. Например, мы можем создать класс EllipticalActionImageAnimation, который будет перемещать картинки по эллиптической траектории. Как уже говорилось выше, подкласс может также реагировать на конфликты с другими картинками на рабочей области. Кроме того, этот класс может создавать такие эффекты, как соединение двух картинок вместе. Сливая два изображения, метод может просто вызвать метод updateImage класса ActionImageCanvas:

```
package ventana.aia;
import java.util.*;
import java.net.*;
import java.awt.*;
import java.applet.*;
import ventana.util.*;
public class ActionImageAnimation implements Configurable, Runnable {
```

```
        private ActionImage ActionImages[];
        private Thread animator;
        private ActionAnimApplet motherApplet;
        private ActionImageCanvas parentCanvas;
        private boolean checkSequence=false;
        private Vector loadingActionImages=new Vector();
        private int curPause=100;
        private ActionImage curActionImage;
        public void attachObject(Object o) throws
                ConfigurationAttachmentException {
                if (!(o instanceof ActionImage))
                        throw (new ConfigurationAttachmentException
                        ("not an ActionImage"));
                ActionImage cur=(ActionImage)o;
                loadingActionImages.addElement(cur);
                cur.setParent(this);}
public void setParent(Object o) {
        if (o instanceof ActionImageCanvas) {
                parentCanvas=(ActionImageCanvas)o;
                motherApplet=parentCanvas.getApplet();
        }
}
public ActionAnimApplet getApplet() {
        return motherApplet;}
public void checkSequence() {
        checkSequence=true;}
public void configureObject(String param, String value) throws
        ConfigurationInvalidParameterException,
        ConfigurationInvalidValueException { }
public String toString() {
        String S="ActionImageAnimation\n";
        for (int i=0;i<ActionImages.length;i++)
                S=S+ActionImages[i].toString();
                return S;
}
public void completeConfiguration() throws
        ConfigurationFailedException {
        ActionImages=new ActionImage[loadingActionImages.size()];
        loadingActionImages.copyInto(ActionImages);
        if (checkSequence) orderActionImages();
}
public void orderActionImages() {
        ActionImages[ActionImages.length-1].setNext(0);
        for (int i=0;i<ActionImages.length-1;i++)
                ActionImages[i].setNext(i+1);
}
public void startAnimation() {
        if (curActionImage==null) curActionImage=ActionImages[0];
        animator=new Thread(this);
        animator.start();}
public void stopAnimation() {
        animator.stop();}
public void run() {
        while(animator==Thread.currentThread()) {
        MediaTracker tracker=motherApplet.mediaTrackerHandle();
        if ((tracker.statusAll(true) & MediaTracker.ERRORED) !=0) {
                System.out.println("Одно или больше изображений не
загрузились");
                return;}
//if ((tracker.statusAll(true) & MediaTracker.COMPLETE)!=0)
//{
parentCanvas.removeImageFromCanvas(curActionImage);
parentCanvas.removeAreaFromCanvas(curActionImage);
// первый переход совершен
```

```
int indx=curActionImage.next();
curActionImage=ActionImages[indx];
if(!(tracker.checkAll())) {
        if (!(tracker.checkID(curActionImage.priority(),true))) {
                try {
                animator.sleep(10);}
        catch (InterruptedException e) {break;}
                continue;}
        }
else {
        curActionImage.updateValues();
        parentCanvas.addToCanvas(curActionImage);
        parentCanvas.updateCanvas();
try {
        animator.sleep(curActionImage.pause());
                }catch (InterruptedException e) {
        break;}
        }
}
}
private protected ActionImage getNextActionImage(int i) {
        MediaTracker tracker=motherApplet.mediaTrackerHandle();
        int nextAI=curActionImage.next();
        if((tracker.statusAll(true) & MediaTracker.COMPLETE) !=0)
                return ActionImages[nextAI];
        else return curActionImage;}
        public void setApplet(ActionAnimApplet ap) {
        motherApplet=ap;}
}
```

Нам осталось написать только два базовых класса. Первый - класс ActionImage. В самой простой форме он по требованию берет свое изображение и передает события классу ActionImageArea. Как уже говорилось выше, в ActionImage можно создать подкласс, чтобы создавать изображения на стороне клиента и, таким образом, не зависеть от скорости передачи данных по сети:

```
package ventana.aia;
import java.util.*;
import java.net.*;
import java.awt.*;
import java.applet.*;
import ventana.util.*;
public class ActionImage implements Configurable{
        private Image thisImage;
        private URL imageURL;
        private int width=0;
        private int height=0;
        private int x=0;
        private int y=0;
        private int next=-1;
        private int imagePriority=0;
        private int pause=100;
        private ActionImageAnimation animParent;
        private ActionAnimApplet motherApplet;
        private ActionArea actionAreas[];
        private Vector loadingAreas=new Vector();
        public int XPos() {return x;}
        public int YPos() {return y;}
        public Rectangle getBoundingRect() {
                return new Rectangle(x,y,width,height);}
        public int pause() {return pause;}
public void attachObject(Object o) throws
        ConfigurationAttachmentException {
        if(!(o instanceof ActionArea))
```

```
                throw (new ConfigurationAttachmentException
                        ("not action area"));
        ActionArea curActionArea=(ActionArea)o;
        loadingAreas.addElement(curActionArea);
        curActionArea.setParent(this);
}
public void configureObject(String param, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException {
        boolean paramHandled=false;
        param=param.toLowerCase();
        value=value.toLowerCase();
if (param.equals("image")) {
try {
        URL docBase=motherApplet.getDocumentBase();
        imageURL=new URL(docBase,value);
        paramHandled=loadImage(value);}
catch(MalformedURLException e) {
        throw (new ConfigurationInvalidValueException("not a URL: +value));}
}
try {
        if (param.equals("priority")) {
                imagePriority=Integer.parseInt(value);
                paramHandled=true;}
                if (param.equals("width")) {
                width=Integer.parseInt(value);
                paramHandled=true;}
        if (param.equals("height")) {
                height=Integer.parseInt(value);
                paramHandled=true;}
        if (param.equals("x")) {
                x=Integer.parseInt(value);
                paramHandled=true;}
        if (param.equals("y")) {
                y=Integer.parseInt(value);
                paramHandled=true;}
        if (param.equals("next")) {
                next=Integer.parseInt(value);
                paramHandled=true;}
        if (param.equals("pause")) {
                pause=Integer.parseInt(value);
                paramHandled=true;}
        } catch (NumberFormatException e) {
        throw (new ConfigurationInvalidValueException
        (e.getMessage()));}
if (!paramHandled) {
        throw (new     ConfigurationInvalidParameterException
        (param));}
}
public String toString() {
        String S="ActionImage\n";
        S=S+"width="+width+"\n";
        S=S+"height="+height+"\n";
        S=S+"next="+next+"\n";
        for (int i=0;i<actionAreas.length;i++)
                S=S+actionAreas[i].toString();
        return S;}
public void completeConfiguration() throws
        ConfigurationFailedException {
        MediaTracker mT=motherApplet.mediaTrackerHandle();
        actionAreas=new ActionArea[loadingAreas.size()];
        loadingAreas.copyInto(actionAreas);
        if (imageURL==null) throw (new
                ConfigurationFailedException("no image URL"));
```

```
        else
                thisImage=motherApplet.getImage(imageURL);
        if (next==-1) animParent.checkSequence();
        if (width>0 && height>0)
                mT.addImage(thisImage,imagePriority,width,height);
        else {
        if (width>0 && height<=0) throw(new
                ConfigurationFailedException
                ("width specified, but not height"));
        if (height>0 && width<=0) throw(new
                ConfigurationFailedException
                ("height specified, but not width"));
        }
mT.addImage(thisImage,imagePriority);
mT.checkID(imagePriority,true);
// проверяет, что MediaTracker берет нужное изображение
}
```

На этом мы закончили работу с классом ActionArea. Класс ActionImage уже выяснил тип события и вызывает нужное событие, непосредственно работающее с методом:

```
package ventana.aia;
import java.applet.*;
import java.awt.*;
import java.util.*;
import ventana.util.*;
public class ActionArea implements Configurable {
        private Polygon thisArea;
        private protected ActionAnimApplet motherApplet;
        private ActionImage parentActionImage;
        public void setParent(Object o) {
        parentActionImage=(ActionImage)o;
                motherApplet=parentActionImage.getApplet();}
        public void attachObject(Object o) throws
                ConfigurationAttachmentException
        {throw (new ConfigurationAttachmentException("Can't attach"));}
public void configureObject(String param, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException {
        param=param.toLowerCase();
        if(param.equals("area"))
                parseArea(value);
        else throw (new
                ConfigurationInvalidParameterException
                (param));
}
public String toString() {
        String S="ActionArea\n";
        S=S+thisArea.toString();
        return S;}
public void completeConfiguration() throws
        ConfigurationFailedException {
        if (thisArea==null)
                throw (new ConfigurationFailedException
                        ("no area described",false));}
private void parseArea(String value) throws
        ConfigurationInvalidValueException {
        StringTokenizer sT=new StringTokenizer(value,";");
        Vector pairs=new Vector();
        while (sT.hasMoreTokens())
                pairs.addElement(sT.nextToken());
        if (pairs.size()==0) throw (new
                ConfigurationInvalidValueException
```

```
                    ("no pairs found"));
        for (int i=0;i<pairs.size();i++) {
                String thisPair=(String)pairs.elementAt(i);
                StringTokenizer st2=new
                StringTokenizer(thisPair,",");
                if (st2.countTokens()!=2) throw (new
                        ConfigurationInvalidValueException
                        ("invalid pair: "+thisPair));
                 String xAsString=st2.nextToken();
                 String yAsString=st2.nextToken();
                 try {
                        int thisX=Integer.parseInt(xAsString);
                        int thisY=Integer.parseInt(yAsString);
                        if (thisArea==null)
                                thisArea=new Polygon();
                        thisArea.addPoint(thisX,thisY);}
                        catch (NumberFormatException e) {
                        throw (new
                                ConfigurationInvalidValueException
                                ("no numbers in pair: "+thisPair, false));
                }
        }
}
public boolean inside(int x, int y) {
return
        (thisArea!=null?thisArea.inside(x,y):false);}
public boolean mouseMove(int x, int y,Graphics g) {
        motherApplet.showStatus
        ("mouse moved: "+x+","+y);
        return false;
}
public boolean mouseDown(int x, int y,Graphics g) {
        motherApplet.showStatus
        ("mouse down: "+x+","+y);
        return false;}
public boolean mouseUp(int x, int y,Graphics g) {
        motherApplet.showStatus
        ("mouse up: "+x+","+y);
        return false;}
public boolean mouseDrag(int x, int y,Graphics g) {
        motherApplet.showStatus
        ("mouse drag: "+x+","+y);
        return false;}
}
```

Теперь мы заполнили все базовые классы нашей системы. Как уже было сказано, во всех классах можно создать подклассы для включения новых функций - мы написали классы, которые знают друг о друге.

Пора начать компоновать основную систему выполнения. Первым шагом будет создание иерархии исключений. Как вы помните из главы 10, мы просто переопределяем конструктор и передаем исключению сообщение, объясняющее, что случилось. Мы не будем здесь обсуждать текст программы для исключений - вы можете найти эту программу на диске CD-ROM или странице Online Companion. После обработки исключений следующим шагом будет создание анализатора. Затем мы реализуем подклассы ActionArea.

Создание анализатора

Прежде чем мы начнем работать с апплетом, нужно создать анализатор. Как мы говорили при написании класса Configuration, задача анализатора - проверять правильность конфигурационного файла и передавать информацию объекту Configuration. Кроме того, анализатор должен интерпретировать исключения, которые может выдавать объект Configuration, и создавать понятные сообщения.

Благодаря структуре нашего языка анализатор должен просмотреть документ только один раз.

Главное, что здесь требуется, - это перевод тегов в переменные, которые можно передавать объекту Configuration. Кроме того, нам нужно несколько подпрограмм, которые будут выдавать более дружественные для пользователя сообщения, чем сообщения об исключениях. Лучший способ сделать сообщения понятными - это следить за номером строки. Как мы уже говорили, нам еще нужны комментарии и пустые строки:

```
package ventana.aia;
import java.util.*;
import java.net.*;
import java.io.*;
import ventana.util.*;
public class AIAnimParser extends ConfFileParser {
private DataInputStream conf;
private int lineNumber=1;
private Stack tokenStack=new Stack();
private final String packageName="ventana.aia.";
private final String beginClassToken="<";
private final String endClassToken=">";
private final String classNameDescript="type";
private final String closureToken="</";
private final String assignString="=";
private final String commentChar="#";
int hierarchyDepth=0;
final String validClasses[]={"ActionAnimApplet","ActionImageCanvas",
               "ActionImageAnimation","ActionImage","ActionArea"};
public AIAnimParser(URL U) throws IOException{
conf=new DataInputStream(U.openStream());}
public boolean start() {
        try {
                while(conf.available()>0) {
                String curLine=conf.readLine();
                if (!parseLine(curLine)) {
                        messageError("Parsing stopped");
                        return false;}
                lineNumber++;}
        } catch(IOException e) {
                 messageError("Error reading configuration file",e);
        }
        return true;
        }
public boolean parseLine(String line) {
        line=line.trim();
        // вырезанное пустое место
        if (line.startsWith(commentChar) ||
                line.length()==0)
        return true;
        if (line.startsWith(closureToken))
                {return closeCurrentToken(line);
        }
        if (line.startsWith(beginClassToken))
                {return parseToken(line);
        }
        // передаем param/value
        // текущему объекту
        passParam(line);
        return true;
        // Если value не распознано, не прекращаем анализ.
        // Вызов для завершения конфигурации на closureToken вызовет ошибки.
}
public void passParam(String line) {
        StringTokenizer splitter=new StringTokenizer(line,assignString);
        String param=splitter.nextToken();
        String value=splitter.nextToken();
try {
```

```
        ConfigureObject.passToCurrent(param,value);}
        catch (ConfigurationInvalidParameterException e) {
        messageError("invalid parameter",e,curClass());
}
        catch (ConfigurationInvalidValueException e)
        {messageError("invalid value",e,curClass());}
}
public boolean parseToken(String token) {
        int start=beginClassToken.length();
        int end=token.length()- endClassToken.length();
        token=token.substring(start,end);
        StringTokenizer splitter=new StringTokenizer(token,"=");
        String baseClassName=splitter.nextToken();
        String subClassName=splitter.nextToken();
        if (subClassName==null) {
        subClassName=baseClassName;}
        if (baseClassName==null){
        messageError("No class specified");
        return false;}
        else return configureNewClass(baseClassName,subClassName);
}
private boolean configureNewClass(String base, String sub) {
        if (hierarchyDepth==validClasses.length)
                {messageError("Can't attach to "+base);
        return false;}
        String nextBase=validClasses[hierarchyDepth+1];
        if (!nextBase.equals(base))
                {messageError("Can't deal with "+base);
        return false;}
try {
        sub=packageName+sub;
        ConfigureObject.attach(sub);
        hierarchyDepth++;
        return true;
        }catch (ConfigurationAttachmentException e){
        messageError("Attachment not allowed",e,base);
        }catch (IllegalAccessException e) {messageError("access to class not
allowed,",e,base);}
        catch (InstantiationException e) {messageError("class didn't
instnatiate",e,base);}
        catch (NotConfigurableException e) {messageError("invalid class for
AIA",e,base);}
        catch (ClassNotFoundException e) {messageError("class wasn't
found",e,base);}
        return false;}
public boolean closeCurrentToken(String token)
{
        if (token.indexOf(curClass())==-1) {
        messageError(token+" doesn't match "+curClass());
        return false;}
        try {
                ConfigureObject.configureCurrent();}
        catch (ConfigurationFailedException e) {
        if (e.isTerminal()) {
                messageError("configuration not completed",e,curClass());
        return false;}
        else
                messageError("parameter was ignored",e,curClass());
        }
        hierarchyDepth--;
        if (hierarchyDepth<0) {
                messageError("Internal parser error");
        return false;}
        else return true;
```

```
}
private String curClass() {
        return validClasses[hierarchyDepth];}
public void messageError(String mesg) {
        System.out.println("Error at line "+lineNumber+": "+mesg);}
public void messageError(String mesg,Exception e) {
        messageError(mesg);
        System.out.print("details: ");
        System.out.println(e.getMessage());
        System.out.println("\n\n*** Java's error message *****");
        System.out.println("(Non programmers should ignore)");
        e.printStackTrace();
        System.out.println("------------\n\n");}
public void messageError(String mesg, Exception e, String className) {
        System.out.println(className+" reports:");
        messageError(mesg,e);}
}
```

Создание ActionArea

Мы построили анализатор и готовы завершить систему, создав несколько рабочих областей. Все, что нам нужно для этого сделать, - это задать их конфигурацию и действия, которые они должны совершать. Последнее можно сделать, переопределив методы обработки событий мыши в классе ActionArea - mouseDown, mouseUp, mouseDrag и mouseMove. Конфигурация потребует только переопределения метода configureObject. Первый класс, который мы переопределим, просто покажет новую страницу, выдаваемую определенным URL:

```
package ventana.aia;
import java.applet.*;
import java.awt.*;
import java.util.*;
import java.net.*;
import ventana.util.*;
public class ShowDocArea extends ActionArea implements Configurable {
URL doc;
public void configureObject(String param, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException {
        param=param.toLowerCase();
        if(param.equals("doc"))
                parseDocURL(value);
        else
                super.configureObject(param,value);
        }
public void parseDocURL(String value) throws
        ConfigurationInvalidValueException {
        try {
                doc=new URL(value);
        }catch (MalformedURLException e) {
                throw new ConfigurationInvalidValueException(value+" not a
URL");}
}
public String toString() {
        String S="SoundActionArea\n";
        S=S+doc.toString()+"\n";
        return S;}
public void completeConfiguration() throws
        ConfigurationFailedException {
        if (doc==null)
                throw (new ConfigurationFailedException
                        ("no doc described"));}
public boolean mouseUp(int x, int y,Graphics g) {
        motherApplet.getAppletContext().showDocument(doc);
```

```
        motherApplet.showStatus("going to ");
        return false;}
public boolean mouseMove(int x, int y,Graphics g) {
        motherApplet.showStatus("Go to: "+doc.toExternalForm());
        return false;}
}
```

Теперь давайте напишем класс ActionArea для проигрывания звуковых фрагментов:

```
package ventana.aia;
import java.applet.*;
import java.awt.*;
import java.util.*;
import java.net.*;
import ventana.util.*;
public class SoundActionArea extends ActionArea implements Configurable {
AudioClip sound;
public void configureObject(String param, String value) throws
ConfigurationInvalidParameterException,
ConfigurationInvalidValueException {
        param=param.toLowerCase();
        if(param.equals("sound"))
                parseSound(value);
        super.configureObject(param,value);}
        public void parseSound(String value) throws
                ConfigurationInvalidValueException {
        try {
         URL U=new URL(value);
         sound=motherApplet.getAudioClip(U);
         if (sound==null)
                throw new ConfigurationInvalidValueException(value+" not a
valid file");
         }catch (MalformedURLException e) {
         throw new ConfigurationInvalidValueException(value+" not a URL");}
        }
        public String toString() {
                String S="SoundActionArea\n";
                S=S+sound.toString()+"\n";
                return S;}
        public void completeConfiguration() throws
                ConfigurationFailedException {
                if (sound==null)
                        throw (new ConfigurationFailedException
                                ("no sound described"));}
        public boolean mouseMove(int x, int y,Graphics g) {
                sound.play();
                return super.mouseMove(x,y,g);}
}
```

Возможные улучшения

Благодаря выбранному нами способу структурирования апплета мы можем расширить его применение в различных направлениях. Самый простой путь состоит в добавлении новых ActionArea, но можно еще создать подклассы в классе ActionImageAnimation и добавить новые возможности, такие как изменение изображений или поддержка фоновых изображений прямо на рабочей области. Не проделывая дополнительной работы, мы можем получить несколько экземпляров рабочих областей в одном апплете и несколько анимаций внутри рабочей области.

Глава 17

Взаимодействие с CGI: Java-магазин

Что вы узнаете из этой главы

В этой главе мы создадим апплет "Java-магазин" - программу "тележки для покупок", которая позволит посетителям виртуального магазина просматривать и выбирать товары. Вот некоторые вопросы, которые мы будем обсуждать:

- Использование HTTP для получения данных для апплета.
- Связь с программами CGI, выполняющимися на Web-сервере.
- Проектирование удобного интерфейса пользователя.
- Динамическое порождение интерфейса пользователя.
- Создание компонентов многократного использования.

Полный исходный текст для классов, обсуждаемых в этой главе, помещен на диск CD-ROM, прилагаемый к книге. Этим диском могут пользоваться те из читателей, кто работает с Windows 95/NT или Macintosh; пользователи UNIX должны обращаться к Web-странице Online Companion, на которой собраны сопроводительные материалы к этой книге (адрес http://www.vmedia.com/java.html).

Контракт

Представьте, что ваш заказчик хочет улучшить интерактивный книжный магазин. В реализации этого магазина используется притягательная графика, последние приемы HTML и система "тележки для покупок", в которую покупатели могут отбирать приобретаемые книги. Система тележки для покупок следит за тем, что пользователь выбрал при просмотре интерактивного каталога магазина. Просматривая каталог, пользователь может нажать кнопку, чтобы добавить товар к своему списку приобретаемых товаров. Существующая система тележки для покупок выполнена в виде ряда сценариев CGI, которые производят HTML-бланки из интерактивного каталога и обрабатывают выбор товаров для приобретения. Однако штат, обслуживающий этот магазин, жалуется на следующие ограничения HTML-бланков:

- Отсутствие реального взаимодействия с пользователем. Пользователь может только нажать кнопку, чтобы добавить или удалить товар из тележки для покупок, что не так удобно, как хотелось бы.
- Web-сервер сильно загружен постоянными запросами, которые выдает программа тележки для покупок, поскольку пользователь просматривает множество товаров. Каждое действие, производимое пользователем (например, выбор товара для приобретения), должно вызвать программу CGI на Web-сервере, которая, в свою очередь, должна следить за сохранением предыдущего выбора.
- Изменение Web-страницы интерактивного магазина - трудоемкая задача. Появление новых товаров и перемещение товаров с одной страницы на другую требуют добавления и изменения многих связей; по мере роста магазина эти связи все больше усложняются - и у персонала уже не хватает времени на его обслуживание.

Рост интерактивного магазина является обязательным условием нашего заказчика, поэтому текущие ограничения должны быть сняты. Мы предлагаем Java-апплет, который может заняться этими проблемами. Он будет работать с существующим интерактивным магазином, так что переход от магазина, построенного на HTML, к магазину, базирующемуся на Java, будет гладким. Пользователи, Web-броузеры которых не способны выполнять Java-апплеты, все равно смогут получить доступ к товарам компании. Происхождение заказов, предназначенных существующему интерактивному магазину или новому магазину, основанному на Java, будет безразлично штату, который обрабатывает заказы, так что никакого дополнительного обучения не потребуется.

Свойства

Java-апплет позволит пользователю управлять магазином с помощью меню. Каждый пункт меню будет представлять новый отдел магазина. Вложенные меню позволят штату организовывать магазин в подотделы. Для примера, в отделе "Программирование" возможны подотделы "Книги по Java", "Книги по C++" и "Книги по Visual Basic". В Java-магазине будет "корзина для покупок". В дополнение к списку, содержащему текущий выбор пользователя, эта программа позволит пользователю удалять товары из списка приобретения.

В Java-магазине будет и справочное меню, чтобы пользователь мог, к примеру, послать письмо с вопросом к владельцу магазина или получить детализированную упорядоченную информацию. Java-магазин будет состоять из двух основных панелей: левая панель будет содержать изображение товаров в текущих отделах магазина, а правая - детализированное описание товара, который выбран на левой панели.

Выбрав товар, пользователь может нажать на кнопку "Check out of Store", по которой это приобретение будет внесено в список и послано сценарию CGI на Web-сервере. Эта информация может быть упакована так, чтобы она была идентична ныне существующим интерактивным данным, основанным на HTML-бланках. Это позволит нам вызывать ту же самую CGI-программу.

Внутреннее устройство Java-магазина является более сложным, чем вышеупомянутый внешний интерфейс пользователя. Меню организованы динамически - они формируются, основываясь на файлах, постоянно находящихся на Web-сервере. Это позволяет настраивать Java-магазин без какого-либо программирования. Штат магазина может заменять предлагаемые товары, их описания и организацию отделов, изменяя простые текстовые файлы. Java-магазин читает эти файлы и формирует интерфейс пользователя согласно их указаниям.

Конструкция

Сначала мы сделаем грубый набросок интерфейса пользователя. Внешний интерфейс должен быть тщательно разработан, чтобы быть компактным и интуитивно понятным. Мы решили использовать справочное меню и меню, которые позволят пользователю передвигаться между отделами магазина. На экране будут находиться большая панель, содержащая изображения товаров в текущих отделах магазина, панель, в которой можно показывать описание товара, и панель для списка выбранных товаров (то есть корзина для покупок). На рис. 17-1 показан общий вид и размещение элементов, которое мы хотели бы получить.

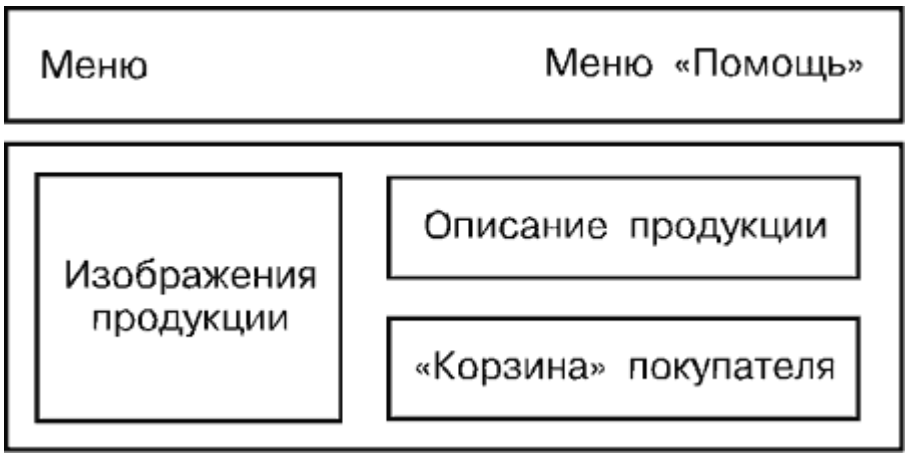

Рис. 17.1.

Мы используем прямое взаимодействие между Web-сервером и апплетом на Java. Как вы знаете, Java-апплеты могут открывать соединения с портами, отличными от порта HTTP Web-сервера. Однако мы хотим добраться и до пользователей, которые должны пройти через firewall, чтобы обратиться к ресурсам Web. Мы используем встроенный HTTP Web-броузера, гарантирующий, что любой обратившийся к Web-странице, из которой вызывается наш Java-магазин, может также обращаться к информации, сгенерированной в Java-магазине (например, к изображению товара и его описанию). Кроме того, мы должны использовать стандартное HTTP-соединение для вызова существующей CGI-программы, используемой сейчас интерактивным магазином. Эту задачу можно решить с помощью двух классов Java API - URL и URLConnection. Мы делаем запросы непосредственно к Web-серверу, чтобы отыскать все данные, которые мы должны сформировать в Java-магазине.

Файлы данных, по которым мы будем формировать интерфейс пользователя, должны быть организованы так, чтобы штат магазина мог легко вносить изменения. Пусть основные пункты меню будут содержаться в файле Store.idx. Этот файл будет включать имена каталогов, сопровождаемые именем меню. В нашей программе этот файл будет выглядеть так:

```
Internet/
Книги по Интернет
Windows/
Книги по Windows
Programming/
Книги по программированию
```

Каталоги будут отделами магазина. Внутри каталога будет файл имя каталога.idx, который будет содержать имена файлов изображений и описаний и уникальное имя, добавляемое в корзину для покупок, когда пользователь выбирает этот товар. Если имя заканчивается на "/", это означает вложенное меню и соответствующий подотдел магазина. Отделы по-прежнему физически представлены как каталоги; в каждом из них будет находиться свой idx-файл. Вот как выглядит типовой idx-файл для отдела "Интернет":

```
Webserver.gif
Webserver.txt,
Книга о Web Сервере
Java.gif
Java.txt
Программирование на Java для Интернет
```

Используя эту структуру каталогов, мы устанавливаем жесткую организацию, которая достаточно проста для того, чтобы ей следовать. Чтобы сделать точные изменения, вам нужно будет только определить, товар из какого отдела должен быть добавлен или удален, перейти в соответствующий каталог и изменить в нем idx-файл. Групповые изменения будут более трудоемкими, но мы идем на этот компромисс, так как малые изменения будут происходить намного более часто, чем удаление всех отделов магазина. Поиск всех idx-файлов выполнен в основном классе Store, а составление интерфейса пользователя из данных, найденных в классе Store, будет генерироваться в классе StoreWindow.

Мы выполним Java-магазин в двух основных классах. Первый называется просто Store (магазин). Этот модуль выбирает данные для формирования интерфейса пользователя. Он также

вызывает класс StoreWindow. StoreWindow - экран, который появляется после того, как данные интерфейса пользователя были найдены. StoreWindow содержит корзину для покупок, изображения товаров, меню магазина и описания товаров. Он также вызывает другие модули, чтобы решить задачи низкого уровня, которые необходимы, чтобы сформировать интерфейс пользователя. На рис. 17-2 показана полная иерархия классов для нашего Java-магазина.

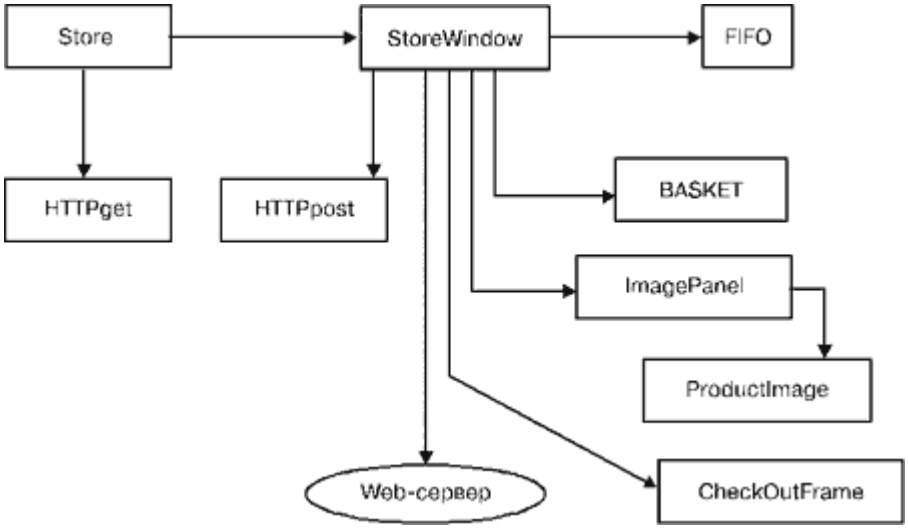

Рис. 17.2.

Мы разделим Java-магазин на несколько ключевых модулей, каждый из которых разработан для выполнения специфической задачи. Задачи мы включим в классы:

- HTTPget и HTTPpost - подпрограммы HTTP низкого уровня, которые используются, чтобы выбрать idx-файлы и файлы описания.
- ProductImage - специальный класс, который выводит изображения товара.
- FIFO - класс, основанный на принципе "первый вошел - первый вышел", который облегчает формирование интерфейса пользователя.
- ImagePanel - класс панели, который отыскивает изображение товара и помещает их в одну панель вместе с поиском описания товара.
- Basket - класс другой панели, который содержит весь список выбранных товаров с их методом.
- Store - основной класс апплета, который отыскивает вышеупомянутые файлы конфигурации.
- StoreWindow - класс окна, который соединяет компоненты и добавляет высокоуровневые функциональные возможности.
- CheckOutFrame - класс, который вызывается, когда пользователь выходит из магазина. Он получает некоторую информацию от пользователя и передает ее CGI-программе через класс HTTPpost.

За исключением основных модулей Store и StoreWindow классы разработаны так, чтобы быть настолько общими, насколько возможно при их многократном использовании. Действительно, класс HTTPget используется и в классе Store, и в классе ImagePanel, а объект PtoductImage - многократно используется в классе ImagePanel.

Нашему апплету не нужны сложные обработчики событий. Мы хотим позволить пользователям выбирать пункты меню, для которых мы должны выбрать метод handleEvent или action, и затем соответственно отвечать. Мы также хотим показать описание товара, когда пользователь щелкает на его изображении. Когда пользователь дважды щелкает на изображении товара, мы добавляем его в корзину для покупок. Как вы можете видеть, у нас есть только два главных класса, обрабатывающих специфические события высокого уровня. Модуль StoreWindow обрабатывает события, перечисленные выше. Класс Basket обрабатывает три кнопки, названия которых соответствуют выполняемым ими операциям: Remove (удаление), Clear (удаление всего) и Check out of Store (выход из магазина). В действительности событие, отвечающее на щелчок пользователя по кнопке Check out of Store, передается родительскому классу StoreWindow, как показано на рис. 17-3.

Рис. 17.3.

Реализация

Давайте начнем нашу реализацию классов с классов-помощников. Когда мы будем обсуждать высокоуровневый класс StoreWindow, вы увидите, что эти классы легко приспособить для совместной работы.

Мы сформируем и проверим каждый класс-помощник прежде, чем начинать работу над классом StoreWindow. Дополнительное время, которое мы будем тратить на строгую проверку каждого модуля, позволит нам быстро и без неожиданностей собрать компоненты воедино. На рис. 17-4 показан Java-магазин, каким его будут видеть покупатели.

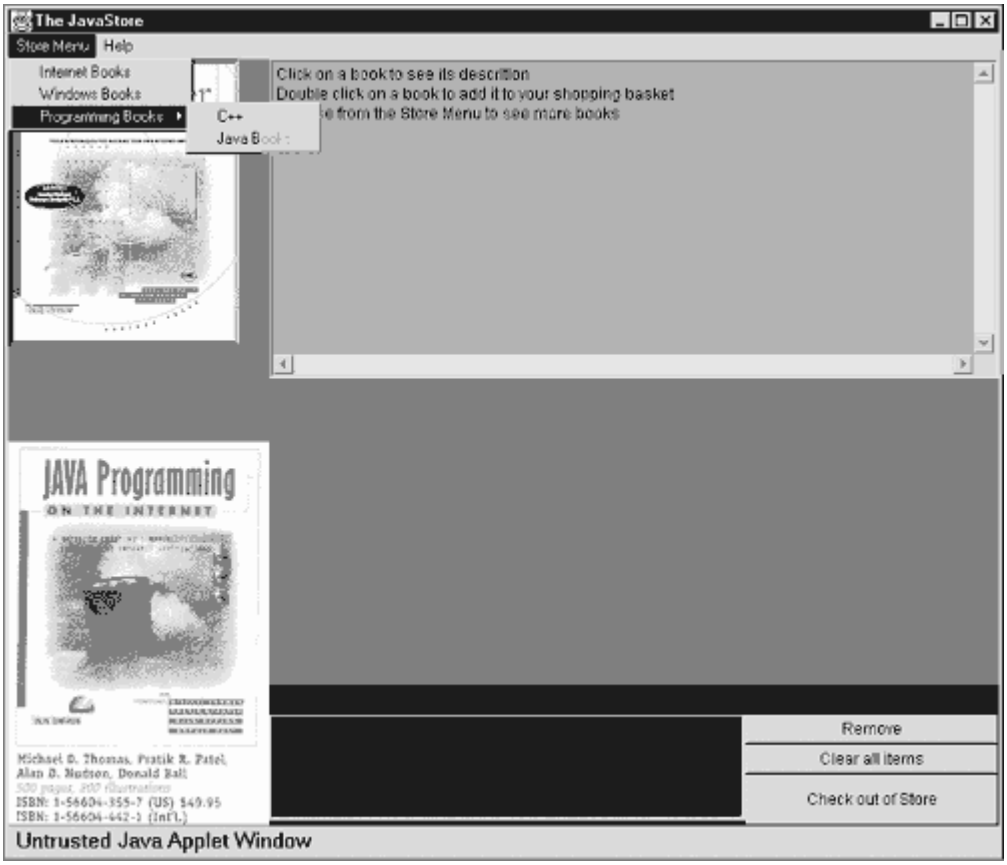

Рис. 17.4.

СОВЕТ В следующем разделе мы предполагаем, что читатель имеет некоторое представление о HTTP, POST, GET и роли CGI-программ. Если вы еще не владеете этими понятиями, проконсультируйтесь по книге "The Web Server Book" (издательство Ventana Press) или странице Online Companion, чтобы получить больше информации по этой теме.

HTTP-запросы

В первую очередь нам нужны модули, которые получают установочные данные магазина, чтобы мы могли сформировать панели товаров и меню. Давайте будем использовать HTTP-запросы, чтобы получить файлы установки .idx. У нас будут два родовых класса - один для выполнения запроса GET и другой для взаимодействия с CGI-программой на Web-сервере, то есть запроса POST. Класс HTTPpost будет обсуждаться ниже. Мы представим вариант класса HTTPget, чтобы показать использование сетевых сокетов для связи с Web-сервером напрямую. Полный исходный текст обоих классов доступен на диске CD-ROM, прилагаемом к книге (для пользователей Windows 95/NT и Macintosh), и на Web-странице Online Companion (для пользователей UNIX).

В Java API есть хороший набор классов, выполняющих работу по созданию URL и соединению с ним: классы URL, URLConnection и URLEncoder. В главе 14, "Связь по сети с помощью URL", об этих классах говорится более подробно. Давайте посмотрим на класс HTTPpost, используемый в Java-магазине:

```
import java.net.*;
import java.io.*;
public class HTTPpost {
        String CGIresults="";
                public HTTPpost(String URLline , String PostData){
                String inputLine;
String PostDataEncoded = "result="+URLEncoder.encode(PostData);
```

Мы устанавливаем HTTPpost, чтобы выбрать два параметра - сначала строковый параметр URLline, который содержит имя CGI-программы, которую мы хотим выполнить на Web-сервере, и полный URL к этой программе. Параметр PostData содержит данные, которые мы хотим передать CGI. Нам нужно удостовериться, что все эти данные передаются Web-серверу в соответствующем формате, так что для кодировки мы используем стандартизированный формат x-www-form-urlencoded с помощью метода URLEncoder.encoder. Мы приписываем это значение к строке "result". CGI будет видеть эти данные как "result=(PostData)", что означает, что значение "result" находится в PostData. Это важно помнить при создании CGI-программы, поскольку мы будем перемещать эту пару значений наружу для обработки. В следующем фрагменте кода устанавливается соединение с Web-сервером:

```
        try {
                URL url = new URL(URLline);
                URLConnection connection = url.openConnection();
                PrintStream outStream = new
                PrintStream(connection.getOutputStream());
                DataInputStream inStream;
```

Сначала мы создаем новый экземпляр из URL, используя параметр URLline. Затем мы создаем новый экземпляр URLConnection, используя объект, возвращаемый методом url.connection. Это дает нам основу для формирования входного и выходного потоков. Мы создаем PrintStream для вывода PostData CGI-программы с помощью URLconnection как канал для фактического соединения. Аналогично ниже мы создаем inStream из экземпляра URLConnection, чтобы получить данные с Web-сервера или CGI-программы:

```
                outStream.println(PostDataEncoded);
                outStream.close();
                inStream = new DataInputStream(connection.getInputStream());
                while (null != (inputLine = inStream.readLine())) {
                        CGIresults = CGIresults + inputLine +"\n";
                }
                inStream.close();
        } catch (MalformedURLException me) {
System.err.println("MalformedURLException: " + me);
        } catch (IOException ioe) {
                System.err.println("IOException: " + ioe);
        }
 }
```

Мы теперь готовы добраться и до бизнеса. Сначала мы посылаем параметр PostData Web-серверу, который передает его CGI-программе. Затем мы начинаем получать данные из CGI, пока

буфер не станет пустым. Эти данные сохраняются в глобальных переменных CGIresults. Мы должны перехватить все исключения, сгенерированные при создании нового URL и вызванные методом URL.openConnection. Ниже приведен метод, который мы вызываем, чтобы возвратить результат HTTPpost:

```
public String results() {
return CGIresults;
};
}
```

Метод results просто возвращает вывод из CGI. Он используется для того, чтобы удостовериться, что CGI-программа получила данные, которые мы ей послали, и правильно их обработала. Мы также можем использовать этот метод, чтобы показать сообщение пользователю в нашем апплете. Кроме того, мы можем передавать URL обратно из CGI-программы и затем вызывать showDocument, чтобы заставить Web-броузер показать страницу. Это может использоваться для того, чтобы послать URL, содержащий страницу "Спасибо за покупку", или страницу, сообщающую об ошибке, или еще что-нибудь подобное.

СОВЕТ Класс HTTPget не принимает параметр POST; взамен ему требуется параметр QUERY LINE. Он также выбирает данные от Web-сервера. Мы хотим использовать его, чтобы получать Store.idx и другие idx-файлы, содержащие структуру магазина. Если бы нам нужно было получить HTML-файл, мы бы увидели все теги HTML. Так как idx-файлы - простые текстовые файлы, нам не нужно волноваться о форматировании HTML-файлов. Теоретически мы могли бы даже получать двоичные данные, но это потребовало бы изменения входного потока, чтобы читать байты вместо строк. Это не особенно полезно, так как у нас уже есть метод getImage.

Ниже представлен другой путь выполнения HTTP-запросов в классе HTTPget. Он почти полностью идентичен классу HTTPpost за исключением того, что для обеспечения связи мы используем непосредственно соединения сокетов. Вы можете пропустить чтение кода, если чувствуете, что вам удобнее работать с HTTPpost, обсужденным выше.

```
import java.awt.*;
import java.net.*;
import java.io.*;
import java.util.*;
public class HTTPget
        {
        private final String CONTENTtype = "application/octet-stream";
        private String RECIEVEdata = "";
        private String home;
        private int port;
        public HTTPget(String URLbase, int URLport) {
        // Приведенный здесь конструктор довольно сложный.
        // Мы посылаем URL в качестве имени машины в URLbase
        // и порт Web-сервера в URLport.
        // Путь к сценарию посылается описываемым ниже методом
        // в параметре Script.
                home = URLbase;
                port = URLport;
                if (port == -1) port = 80;
}
        public void submit(String Script, String PostData)
                {
                Socket sock;
                OutputStream outp;
                InputStream inp;
                DataOutputStream dataout;
                DataInputStream datain;
                String URLline;
                RECIEVEdata = "";
                URLline = Script+"?"+PostData;
```

```
                // создаем сокет клиента
                        try
                        sock = new Socket(home, port);
                catch (Exception e)
                        {
                        System.out.println("Error with HTTP GET "+e);
                }
// получаем выходной поток для связи с сервером
                try
                {
                        outp = sock.getOutputStream();
                        inp = sock.getInputStream();
                }
                catch (Exception e)
                        {
                        RECIEVEdata = e+" (getstream)";
                        try sock.close();
                        catch (IOException ee) ;
                        return;
                        }
                        try
                        {
                                dataout = new DataOutputStream(outp);
                                datain = new DataInputStream(inp);
                }
                catch (Exception e)
                {
                        RECIEVEdata = e+" (Dstream)";
                        try sock.close();
                        catch (IOException ee) ;
                        return;
                }
// посылаем запрос HTTP-серверу и получаем возвращаемые данные
                try
                {
                        dataout.writeBytes("GET " + URLline + " HTTP/1.0\r\n");
                        dataout.writeBytes("Content-type: " + CONTENTtype +
"\r\n");
                        dataout.writeBytes("Content-length: 0 \r\n");
                        dataout.writeBytes("\r\n");
                        dataout.flush();
                        boolean body = false;
                        String line;
                        while ((line = datain.readLine()) != null)
                        {
                                if (body)
                                <_>RECIEVEdata += "\n" + line;
                                else if (line.equals(""))
                                <_>body = true;
                        }
                }
                catch (Exception e)
                {
                        RECIEVEdata = e+" (write)";
                        try
                        sock.close();
                        catch (IOException ee) ;
                        return;
                }
                // закрытие магазина
                try
                {
                        dataout.close();
                        datain.close();
```

```
                }
                catch (IOException e) ;
                try
                sock.close();
                catch (IOException e) ;
        }
        public String Results() {
                return RECIEVEdata;
        }
}
```

Размещение информации о товарах

Класс ProductImage хранит не только изображение товара, но также метку и описание. Класс выполнен как расширение класса Canvas, так что мы можем присваивать ему значение и хранить товар со специфическими данными, содержащимися в самом объекте. Мы используем его в классе ImagePanel, поскольку мы добавляем к нему изображение товара. К описанию и метке можно обращаться непосредственно, создавая ссылку к специфическому ProductImage и затем обращаясь к описанию и метке через эту ссылку. Этот механизм реализован в классе StoreWindow, так что мы не должны использовать класс ImagePanel, чтобы добраться до данных. Кроме того, мы добавляем общие переменные height и width, чтобы позволить ImagePanel правильно устанавливать размер для всех ProductImage, который добавляет его специфическому ImagePanel. Давайте рассмотрим класс ProductImage:

```
import java.awt.*;
public class ProductImage extends Canvas {
private Image piece;
public String label;
public int height;
public int width;
public String Description;
public void setLabel(String s) {
        label = s;
}
public void setImage(Image i) {
        if (piece !=i) {
                piece = i;
                height= piece.getHeight(this);
                width= piece.getWidth(this);
        }
}
```

Мы не затеняем метку, ширину, высоту и описание, но мы хотим затенить часть Image. Методы setLabel и setImage вызываются с соответствующими объектными параметрами, и экземпляр ProductImage хранит их для будущего использования. Описание устанавливается непосредственно ссылкой на переменную Description в экземпляре ProductImage. Этот объект тоже базируется на классе Canvas, так что мы можем добавлять его непосредственно к панели, которая показывает изображение товара:

```
public void paint(Graphics g) {
        Rectangle r = bounds();
        g.setColor(getBackground());
        g.fillRect(0,0,r.width,r.height);
        if (piece!=null) {
                g.drawImage(piece,0,0,this);
        }
}
```

Теперь мы рисуем изображение, переданное нам методом setImage в экземпляре ProductImage, так что оно будет готово к добавлению к панели, которая показывает изображения товаров.

Класс FIFO

Класс FIFO - это расширение класса Vector, доступного в API. Vector представляет собой класс расширяемого массива в пакете java.util. Мы добавляем некоторые методы для соответствия структуре данных FIFO. FIFO используется при формировании интерфейса пользователя, что сильно облегчает отслеживание данных этого интерфейса. Мы можем помещать объекты в FIFO в особом порядке и доставать их обратно в том же самом порядке, используя соответствующие методы push и pop. У нас также есть метод isEmpty, который проверяет, является ли список пустым:

```
import java.util.Vector;
import java.util.*;
public class FIFO extends Vector {
public String pop() {
        String s="";
        // берем первый элемент
        try{ s= (String)firstElement(); }
        catch (NoSuchElementException e )
                {System.out.println("FIFO EMPTY!!!!!");
                s="";
        }
        // и удаляем первый элемент
        try{ removeElement(s);}
        catch(ArrayIndexOutOfBoundsException e)
                {System.out.println("FIFO EMPTY!!!");}
        return s;
}
// добавляем элемент в верх списка
public void push(String s) {
        addElement(s);
}
// проверяем список на пустоту
public boolean empty() {
        return isEmpty();
}
```

Методы, доступные в классе Vector, делают за нас реальную работу. Они генерируют исключения, которые мы должны захватить. Мы определили переменные типа String для элементов, хранящихся в векторе, но с некоторыми небольшими модификациями мы можем заставить FIFO принять любой объект. Мы выбрали тип объекта String, потому что FIFO в нашем апплете используется только со строками.

Получение изображений и описаний

Класс ImagePanel фактически вызывает метод getImage и метод HTTPget, чтобы получить данные о каждом товаре на соответствующей панели. Каждый отдел магазина имеет собственный экземпляр ImagePanel. Мы работам с отделами так, чтобы было легко их изменять, показывая различные панели. Это показано в классе StoreWindow. У нас есть один основной цикл, который повторяется, пока список FIFO, содержащий данные для текущего отдела магазина, не обработан полностью:

```
import java.awt.*;
import java.net.*;
import java.util.*;
import java.io.*;
import ProductImage;
import FIFO;
import HTTPget;
public class ImagePanel extends Panel {
        private ProductImage ImageCanvas;
        private URL ImageURL;
        protected Toolkit Tools;
```

```
        public int width;
        public int height;
        private MediaTracker tracker;
        private Image i;
        public String label;
        private HTTPget DescText;
```

Обратите внимание, что в этом классе мы вызываем MediaTracker. Он гарантирует, что изображения получены прежде, чем выполняется ProductImage.setImage. Выполнение с изображением, которое еще не было получено, вызовет NullPointerException. Toolkit необходим, чтобы можно было вызвать getImage, так как мы непосредственно находимся не в панели апплета, а только во фрейме StoreWindow:

```
public ImagePanel(String ImageURLBase, FIFO FileList, String PanelLabel) {
        String s;
        String Desc;
        Toolkit Tools;
        String Name;
label=PanelLabel;
Tools = getToolkit();
tracker= new MediaTracker(this);
width=0;
height=0;
setLayout(new GridLayout(0,1));
while (!FileList.empty()) {
        s = FileList.pop();
        try {
                ImageURL = new URL(ImageURLBase + s );
}
catch (MalformedURLException e) {
        System.out.println("Error retrieving " + ImageURL);
}
Desc = FileList.pop();
DescText = new HTTPget(ImageURL.getHost(), ImageURL.getPort());
DescText.submit( Desc, "");
// получаем изображение
ImageCanvas = new ProductImage();
i = Tools.getImage(ImageURL);
tracker.addImage(i, 0);
try {
        tracker.waitForID(0);
}
catch (InterruptedException e) {
        return;
}
```

Мы используем менеджер размещения GridLayout с одним столбцом и неопределенным числом строк. Так как в различных отделах магазина может быть различное число товаров, мы выбираем такую реализацию для того, чтобы не ограничивать число товаров, которые могут находиться в отделе магазина. Метод pop для FileList FIFO возвращает первый элемент в списке и удаляет его. Три компонента, которые сгруппированы для каждого товара, - имя файла изображения товара, имя файла описания и фактическое имя - анализируются в классе StoreWindow и передаются, когда создан ImagePanel для каждого отдела магазина.

Сначала выбирается файл изображения. Мы формируем URL из параметра, переданного как основной URL, и добавляем путь и имя файла из FileList. Аналогично, мы используем HTTPget, чтобы отыскать файл описания. Мы создаем новые экземпляры ProductImage, называемые ImageCanvas, в цикле для каждого товара. ProductImage создается для каждого товара в отделе. ProductImage также содержит описание и информацию об имени, даже если это объект Canvas. Это сделано для того, чтобы хранить данные о конкретном товаре в одном объекте. Мы назначаем выбранное изображение объекту I и запускаем апплет, чтобы получить изображение немедленно, используя MediaTracker. Конечно, работая с tracker.waitForID(0), мы должны перехватить исключение InterruptedException.

```
                ImageCanvas.setImage(i);
```

```
                ImageCanvas.setLabel(s);
                ImageCanvas.Description=DescText.Results();
                Name= FileList.pop();
                ImageCanvas.setLabel(Name);
        ImageCanvas.resize(ImageCanvas.width,ImageCanvas.height);
                        add(ImageCanvas);
                        ImageCanvas.show();
                        height += ImageCanvas.height;
                        width += ImageCanvas.width;
                }
        }
} // конец ImagePanel
```

Теперь, когда мы выбрали все данные по нужным товарам с Web-сервера, давайте поместим их на соответствующее место. Мы используем экземпляры класса ProductImage, чтобы содержать эти данные; во фрагменте кода, приведенном выше, для этого неоднократно приписывается значение ImageCanvas. Мы устанавливаем изображение I, отыскиваем вышеупомянутое изображение с помощью метода setImage, используем метод setLabel, чтобы непосредственно установить метку и описание с помощью переменной Description в классе ProductImage (все через ImageCanvas). Метка или имя является последним членом группы из трех элементов, которые мы выталкиваем от FileList FIFO для специфического товара. Мы используем высоту и ширину изображения товара, чтобы изменить размеры ImageCanvas прежде, чем добавить его к панели ImagePanel. Наконец, мы устанавливаем полный размер ImagePanel так, чтобы его можно было использовать в классе StoreWindow. Экземпляры ImagePanel, представляя различные отделы магазина, добавлены как групповой компонент непосредственно к фрейму StoreWindow.

Обработка действий пользователя

Класс Basket - наша реализация "корзины для покупок". В дополнение к списку товаров, которые пользователь выбрал для приобретения, можно удалить товары из корзины. Корзина состоит из четырех основных элементов: списка и трех кнопок. Она расширяет панель так, чтобы ее можно было добавлять непосредственно к фрейму StoreWindow. Мы обрабатываем функции, которые работают непосредственно со списком внутри этого класса (кнопки "Remove" и "Clear all items"), и передаем нажатие кнопки "Check out of Store" родительскому StoreWindow:

```
import java.awt.*;
public class Basket extends Panel {
        private List ItemList;
        private Button Remove;
        private Button Clear;
        public Button Checkout;
public Basket(String Name) {
        Label BasketLabel = new Label(Name, Label.CENTER);
        GridBagConstraints Con = new GridBagConstraints();
        GridBagLayout gridBag = new GridBagLayout();
        ItemList = new List(5, false);
        Remove = new Button("Remove");
        Clear = new Button("Clear all items");
        Checkout = new Button("Check out of Store");
        setLayout(gridBag);
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        setBackground(Color.blue);
        Con.anchor = GridBagConstraints.CENTER;
        Con.fill = GridBagConstraints.NONE;
        Con.weighty = 1.0;
        Con.weightx=0.0;
        Con.gridwidth = GridBagConstraints.REMAINDER;
        gridBag.setConstraints(BasketLabel, Con);
        add(BasketLabel);
        Con.fill = GridBagConstraints.BOTH;
        Con.gridheight=3;
        Con.gridwidth=GridBagConstraints.RELATIVE;
        Con.weighty = 1.0;
```

```
        Con.weightx=3.0;
        gridBag.setConstraints(ItemList, Con);
        add(ItemList);
        Con.weightx=1.0;
        Con.weighty = 0.0;
        Con.gridheight=1;
        Con.gridwidth=GridBagConstraints.REMAINDER;
        Con.fill = GridBagConstraints.HORIZONTAL;
        gridBag.setConstraints(Remove, Con);
        add(Remove);
        gridBag.setConstraints(Clear, Con);
        add(Clear);
        Con.fill = GridBagConstraints.BOTH;
        gridBag.setConstraints(Checkout, Con);
        add(Checkout);
}
```

Здесь мы реализуем три кнопки и список, но затеняем (или объявляем с модификатором private) все из них за исключением кнопки "Check out of Store". Это сделано для того, чтобы к ней мог обратиться родительский класс StoreWindow. Функции, обрабатывающие другие кнопки, расположены в самом классе Basket. Обратите внимание, что список создан так, чтобы позволить выбрать только один экземпляр. Мы используем менеджер размещения GridBagLayout для установки ItemList в режиме использования максимума места на панели. Кнопка "Check out of Store" занимает место, остающееся после того, как добавлены кнопки "Remove" и "Clear all items". Детальное обсуждение менеджера размещения GridBagLayout см. в главе 8, "Еще об интерфейсе пользователя".

```
public void addItem( String ItemName ) {
ItemList.addItem(ItemName);
ItemList.select(ItemList.countItems()-1);
}
public String getItems() {
        String s = "";
        for (int i=0; i<ItemList.countItems(); i++ ) {
                s += ItemList.getItem(i)+"\n";
        }
return s;
}
public boolean action( Event evt, Object obj)
        {
        if ("Remove".equals(obj))
                {
                if (ItemList.getSelectedIndex()!=-1)
                        {
                        ItemList.delItem(ItemList.getSelectedIndex());
                        }
                return true;
                }
        else if ("Clear all items".equals(obj))
                {
                        ItemList.clear();
                        return true;
                }
        return super.action(evt, obj);
        } // конец action
}
```

Далее мы разрабатываем два метода, addItem и getItems, чтобы позволить добавлять элементы к списку и выбирать список. Эти методы используются в StoreWindow, когда пользователь дважды щелкает по изображению товара или хочет покинуть магазин (мы обсудим это позже, когда доберемся до раздела, описывающего класс StoreWindow). Соответствующие обращения добавляются к методу ItemList.

Обработчики событий для кнопок "Remove" и "Clear all items" расположены внутри класса и заменяют метод action. Мы проверяем, не соответствует ли переданный параметр obj метке одной

из этих двух кнопок, чтобы не передать событие родительскому классу через оператор super.action(evt, obj). Это делается, прежде всего, потому, что мы хотим посылать родительскому классу только событие "Check out of Store". ItemList обрабатывает выбор из списка самостоятельно, так что с ним ничего делать не нужно.

Считывание данных о конфигурации и инициализация

Класс Store - основа апплета. Он вызывает класс StoreWindow, который является фреймом вне броузера Web. Магазин вызывается из Web-страницы с тегом <APPLET>. Главная цель класса Store состоит в том, чтобы получать и анализировать файлы конфигурации Java-магазина для использования их классом StoreWindow. Мы выбираем idx-файлы, начиная с файла Store.idx и затем рекурсивно получая остальные требуемые idx-файлы из указанных подкаталогов:

```
import java.awt.*;
import java.applet.Applet;
import StoreWindow;
import java.util.StringTokenizer;
public class Store extends Applet {
        StoreWindow f;
        String ProductDir;
        String CheckOutCGI;
// Мы хотим поместить его в тег <PARAM>!
        String Tree = "";
        public void init(){
                ProductDir = getParameter("index_location");
                CheckOutCGI = getParameter("CGI");
                ParseIndex( ProductDir, "Store.idx");
        f = new StoreWindow(getDocumentBase(), Tree, CheckOutCGI );
}
```

Мы создаем новый экземпляр класса StoreWindow и получаем указатель на расположение первого файла конфигурации Store.idx (из параметра index_location HTML-кода, из которого вызывался апплет). Мы также получаем URL CGI-программы, чтобы выполнить ее, когда пользователь покинет магазин. Путь будет передан методу ParseIndex, который получает и анализирует все idx-файлы. Кроме того, мы передадим "Store.idx" как заданное по умолчанию имя индексного файла:

```
private void ParseIndex ( String ProductDir, String s )
{
        HTTPget Index;
        String next;
        String nextDESCRIPTION;
        String nextNAME;
        String Params = "";
        String ProductIndex = ProductDir + s;
        Index = new HTTPget(getDocumentBase().getHost(),
getDocumentBase().getPort());
        Index.submit( ProductIndex, "");
        // получаем idx-файл с Web-сервера
        StringTokenizer LoadINDEX = new StringTokenizer(Index.Results(),
"\n");
        // обрыв idx-файла по "\n"
        while (LoadINDEX.hasMoreTokens()) {
        // просмотр данного idx-файла, пока в нем что-то есть
                next= LoadINDEX.nextToken();
                // если это не новый каталог, разбираем его и
                // возвращаемся в начало, или вызываем ParseIndex
                // на этом idx-файле
                if (next.endsWith("/")) {
                        nextDESCRIPTION = LoadINDEX.nextToken();
                        Tree += ("Menu:" + nextDESCRIPTION +"\n");
                        // отрезаем остаток после / и добавляем .idx
```

```
                        ParseIndex(ProductDir + next, next.substring(0,
(next.length()-1)) + ".idx");
                        // мы используем рекурсию, так что будьте внимательны!
}
else if (! next.equalsIgnoreCase("<none>") )
                {
                nextDESCRIPTION = LoadINDEX.nextToken();
                nextNAME = LoadINDEX.nextToken();
                Tree+= ("GIF:" + ProductDir + next + "\nTXT:" + ProductDir +
nextDESCRIPTION + "\n"+"NAME:"+nextNAME+"\n");
                }
        else {
                Tree+=("Encountered a " + next+"\n" );
                }
        }
        Tree += "END:MENU\n";
}
```

Мы начинаем с методов URL апплета, getDocumentBase.getHost и getDocumentBase.getPort, чтобы использовать класс HTTPget для получения индексного файла с Web-сервера. Объект Index HTTPget теперь позволяет нам добраться до idx-файла методом Results. HTTPget вызывается неоднократно для различных idx-файлов, пока все они не будут получены.

Затем с помощью StringTokenizer мы разбиваем полученные idx-файлы на части по символам новой строки \n, так как при структурировании idx-файлов мы помещали данные в отдельные строки. Теперь в idx-файле может находиться один из четырех элементов: меню или подменю (тогда мы были бы должны получить другой idx-файл), или имя gif-файла, имя файла описания или идентификатор. Порядок элементов нам известен, так что мы должны искать конечный "/", чтобы определить меню или подменю. Если это - меню или подменю, мы вызываем ParseIndex с путем каталога, который является просто текущим токеном, и добавляем метку, которую мы получаем из следующей строки в idx-файле, разграничивая их отметкой "Menu:". Мы добавляем все это к нашей строковой переменной Tree и передаем StoreWindow, который затем анализирует этот предварительно отформатированный список, чтобы сформировать интерфейс пользователя. Рекурсивное обращение ParseIndex анализирует эти меню, используя те же самые методы ParseIndex, но с другими параметрами. Такое использование рекурсии предохраняет нас от необходимости выполнения вложенных циклов с введением контрольных точек.

Если токен не является каталогом (то есть меню), то он представляет собой товар. Мы ставим отметку, идентифицирующую элемент, добавляемый к Tree, - "GIF:", "TXT:" и "NAME:". Мы будем добавлять эти три части для каждого товара, а отметки позволят нам отладить проблемы с idx-файлами или формированием списка Tree. Мы зацикливаемся внутри каждого idx-файла, пока его обработка не будет завершена, но проводим ветвление несколько раз, когда рекурсивно обрабатываем меню и подменю. Результирующая переменная, строка Tree, может выглядеть запутанной, но мы используем подобный рекурсивный метод в StoreWindow для формирования интерфейса пользователя, и этому методу очень удобно работать с нелинейной формой конфигурации дерева, которую мы здесь создали.

Когда файлы конфигурации проанализированы и помещены в форматированный список Tree, мы создаем новый StoreWindow (getDocumentBase, Tree), которому передаем текущую конфигурацию переменной Tree. Мы также передаем базовый URL так, чтобы можно было использовать его для получения с Web-сервера GIF-файлов изображения и файлов описания для соответствующих товаров.

Объединяем все вместе

Мы соединяем компоненты и добавляем высокоуровневые функциональные возможности в классе StoreWindow. У нас есть предварительно отформатированные данные конфигурации магазина в переменной Tree и все необходимые классы, чтобы начать создавать интерфейс пользователя. Сначала мы устанавливаем соответствующий базовый URL, чтобы получать изображения и файлы описания с Web-сервера. После этого мы создаем требуемые экранные элементы, например корзину для покупок и основанную на менеджере размещения CardLayout панель изображений. Позже, когда мы сформируем интерфейс пользователя и меню, мы отобразим эти панели на экране.

```
import java.awt.*;
import java.net.URL;
import java.util.*;
```

```
class StoreWindow extends Frame {
        Panel ProductPicPanel;
        String Params = "";
        Panel DescriptionPanel;
        String BaseURL;
        Panel MainPanel;
        TextArea DescArea;
        Basket ShoppingBasket;
        String NewPickMessage;
        String CGI;
public StoreWindow(URL BURL, String Tree, String CGIprogram ) {
this.CGI=CGIprogram;
        if (BURL.getPort()==-1) {
                BaseURL = BURL.getProtocol() +"://" + BURL.getHost();
        }
        else
        {
                BaseURL = BURL.getProtocol() +"://" + BURL.getHost()
+":"+BURL.getPort() ;
        }
ShoppingBasket = new Basket("Shopping Basket");
setTitle("The JavaStore");
MainPanel = new Panel();
MainPanel.setLayout(new BorderLayout());
setLayout(new BorderLayout());
DescArea = new TextArea(15, 30);
DescArea.setEditable(false);
DescArea.setBackground(Color.black);
DescArea.setForeground(Color.white);
DescArea.setFont(new Font("Helvetica", Font.PLAIN, 12));
NewPickMessage="Click on a book to see its descrition\nDouble click on a book
to
add it to your shopping basket\nChoose from the Store Menu to see more
books\n";
MainPanel.add("North", DescArea);
MainPanel.add("South", ShoppingBasket);
MenuBar Bar = new MenuBar();
ProductPicPanel = new Panel();
ProductPicPanel.setLayout(new CardLayout());
```

Приведенный выше код главным образом инициализирует фрейм и создает некоторые из нужных нам объектов AWT. Мы устанавливаем менеджеры размещения для основной области и ProductPicPanel. Мы используем CardLayout для ProductPicPanel, так как мы хотим изменять отображаемую панель, когда выбор меню меняется. Мы добавляем полосу меню, а элементы меню мы добавим несколько позже. Далее, мы вызываем метод BuildUI, который формирует меню и добавляет панели ProductPicPanel. Обратите внимание, что мы уже добавили объекты, которые должны быть в MainPanel. Позже мы добавим MainPanel и ProductPlcPanel к фрейму StoreWindow.

```
        Bar.add(BuildUI(Tree));
        Menu Help = new Menu("Help");
        Help.add(new MenuItem("About us"));
        Help.add(new MenuItem("Ordering information"));
        Help.add(new MenuItem("-"));
        Help.add(new MenuItem("Email us"));
        Bar.setHelpMenu(Help);
        Bar.add(Help);
        setMenuBar(Bar);
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        setBackground(Color.gray);
        add("Center", MainPanel);
        add("West", ProductPicPanel);
        add("Center", MainPanel);
        resize(700,500);
        show();
```

```
} // конец StoreWindow
```

BuildUI возвращает меню, которое мы добавляем непосредственно к полосе меню фрейма. Кроме того, мы добавляем меню "Help" и создаем элементы меню. Мы теперь готовы добавить ProductPicPanel и MainPanel к фрейму StoreWindow. Метод BuildUI был вызван в этом месте, так что интерфейс пользователя завершен. Ниже мы вносим в список код для метода BuildUI, а также описываем обработку необходимых событий:

```
public boolean handleEvent(Event evt) {
        if (evt.id == Event.WINDOW_DESTROY) {
                dispose();
                return true;
        }
        if (evt.target instanceof ProductImage) {
                if (evt.id==Event.MOUSE_DOWN) {
                        ProductImage p = (ProductImage)evt.target;
                if (evt.clickCount==2){
                        ShoppingBasket.addItem(p.label);
                        }
                else {
                        DescArea.setText(p.Description);
                        }
                }
                return true;
        }
        return super.handleEvent(evt);
}
public boolean action(Event evt, Object arg) {
        if ( evt.target instanceof MenuItem) {
                ((CardLayout)ProductPicPanel.getLayout()).show
                (ProductPicPanel,(String)arg);
DescArea.setText(NewPickMessage);
return true;
}
if ("Check out of Store".equals(arg))
        {
        System.out.println(ShoppingBasket.detItems());
        dispose();
        SubmitItemList(ShoppingBasket.getItems());
                // вызываем метод, чтобы запустить CGI
                return true;
                }
        return false;
}
private void SubmitItemList(String ItemList) {
// посылаем ItemList CGI-программе
        CheckOutFrame Exit = new CheckOutFrame("Check Out of Java Store",
        ItemList, CGI);
}
```

Метод handleEvent отслеживает два специфических события - одиночное или двойное нажатие мыши на изображении товара. Мы используем параметр clickCount, чтобы сосчитать число щелчков, и затем переходим на соответствующую обработку. Если пользователь дважды щелкает на изображении, чтобы поместить товар в корзину для покупок, мы вызываем Basket.addItem (ItemName) метод. Как вы, вероятно, помните, мы создали метод addItem, который позволяет добавлять элементы к списку выбранных товаров в классе Basket. Если регистрируется одиночное нажатие, мы показываем описание товара, которое выбираем в классе ImagePanel. Мы заканчиваем handleEvent, передавая остающиеся события обратно так, чтобы следующий обработчик событий мог их обработать.

Обработчик событий сначала проверяет взаимодействие пользователя с меню. Если пользователь выбрал новый элемент меню или отдел магазина, мы показываем панель, которая соответствует этому отделу. При этом переключается сообщение команды, которое нужно отобразить в текстовой области DescArea. Вторая часть обработчика событий проверяет событие нажатия кнопки "Check out of Store", которое передано из ShoppingBasket. Мы получаем список

товаров, выбранных пользователем, и передаем его методу SubmitItemList, который вызывает метод CheckOutofFrame и посылает список CGI-программе, расположенной на Web-сервере. После того как пользователь сделал это, нам больше не нужно оставаться в магазине, поэтому мы закрываем фрейм StoreWindow. CGI-программа будет обрабатывать данные.

```
private Menu BuildUI(String Tree) {
        FIFO PanelAdd = new FIFO();
        Menu StoreMenu = new Menu("Store Menu");
        StringTokenizer SplitTree = new StringTokenizer(Tree, "\n");
        ParseTree( SplitTree, StoreMenu, "", PanelAdd);
        return StoreMenu;
} // конец StoreWindow
private void ParseTree(StringTokenizer TreeBranch, Menu MenuIn, String
PreviousToken, FIFO PanelAdd)
        {
        StringTokenizer Leaf;
        Menu subMenu;
        String next;
        // ImagePanel MenuPanel;
        while (TreeBranch.hasMoreTokens()) {
                Leaf = new StringTokenizer(TreeBranch.nextToken(), ":");
                next = Leaf.nextToken();
                // ищем "Menu" и добавляем новый элемент в меню
                if (!PreviousToken.equals("")) {
                if (next.equalsIgnoreCase("Menu")) {
                        subMenu = new Menu(PreviousToken);
                MenuIn.add(subMenu);
                // рекурсивная обработка этого нового подменю
                PreviousToken="";
                MenuIn=subMenu;
        }
        else {
                MenuIn.add(PreviousToken);
                PanelAdd.push(PreviousToken);
        }
}
// ищем "Menu" и добавляем новый элемент в меню
if (next.equalsIgnoreCase("Menu")) {
        ParseTree(TreeBranch, MenuIn, Leaf.nextToken(), PanelAdd);
}
else if (next.equalsIgnoreCase("GIF")) {
        PanelAdd.push(Leaf.nextToken());
        ParseTree(TreeBranch, MenuIn, "", PanelAdd);
}
else if (next.equalsIgnoreCase("TXT")) {
        PanelAdd.push(Leaf.nextToken());
        ParseTree(TreeBranch, MenuIn, "", PanelAdd);
}
else if (next.equalsIgnoreCase("NAME")) {
        PanelAdd.push(Leaf.nextToken());
        ParseTree(TreeBranch, MenuIn, "", PanelAdd);
}
        else if (next.equalsIgnoreCase("END")) {
                if (!PanelAdd.empty())
                        {
                        String PanelName =(String)PanelAdd.pop();
                        Leaf.nextToken();
                        <%-2>ProductPicPanel.add(PanelName, new
ImagePanel(BaseURL, PanelAdd, PanelName));<%0>
                        }
                }
        }
}
} // конец StoreWindow
```

Метод BuildUI в действительности инициализирует метод ParseTree. Мы создаем новый объект FIFO, новое меню. Затем мы используем StringTokenizer с Tree, отделяя токены по символу новой строки. Tree мы создали в классе Store из данных конфигурации, полученных с Web-сервера. Оно было предварительно отформатировано и упорядочено так, чтобы рекурсивный метод ParseTree мог его использовать. Большая часть работы в методе BuildUI заключается в правильном использовании рекурсии с методом ParseTree.

Как мы уже говорили, метод ParseTree работает рекурсивно. Он вызывается, когда мы сталкиваемся с новым меню. Это позволяет нам обрабатывать вложенные меню без использования вложенных циклов. Мы вызываем ParseTree со следующими параметрами: StringTokenizer, содержащий дерево, которое должно быть обработано; родительское меню; строка, содержащая предыдущий токен; и список FIFO, содержащий данные для ImagePanel, который должен быть сформирован с этими данными конфигурации. Родительское меню и предыдущий токен требуются для подменю. Вообще мы следуем этим путем в методе ParseTree: обрабатываем текущий лист и получаем следующий лист, который является меткой для предыдущего, затем вызываем ParseTree с остающейся частью дерева. Мы останавливаем разбор дерева, достигнув его конца с отметкой "END". Во время прохода по дереву мы помещаем элементы в буфер FIFO PanelAdd. Этот FIFO-буфер передается ImagePanel, который использует данные в FIFO, чтобы выбрать изображение и описание товара.

Передача выбора пользователя на Web-сервер

Мы вызываем класс CheckOutFrame в методе SubmitItemList. CheckOutFrame состоит из двух текстовых полей и кнопки "Done", как показано на рис. 17-5. Пользователь заполняет поля с именем и телефонным номером. Эти данные наряду со списком выбранных товаров передаются CGI-программе, выполняющейся на Web-сервере классом HTTPpost. Код CheckOutFrame показан ниже. Обратите внимание, что мы уничтожаем StoreWindow перед открытием CheckOutFrame, чтобы не запутать пользователя, и тем самым мы немедленно привлекаем его внимание. Для размещения компонентов мы снова используем менеджер GridBagLayout. Обработчик событий начинает наблюдать за нажатием кнопки Done, по которой мы берем выбранный список товаров, имя пользователя и номер его телефона и посылаем эти данные CGI-программе:

Рис. 17.5.

```
import java.awt.*;
import HTTPpost;
public class CheckOutFrame extends Frame {
TextField Name;
TextField Phone;
Button Done;
String ItemList;
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints Con = new GridBagConstraints();
Label LName;
Label LPhone;
Label Message1;
```

```
Label Message2;
HTTPpost CGIpost;
String CGI;
public CheckOutFrame( String Title, String ItemList, String CGI )
{
super(Title);
this.ItemList = ItemList;
this.CGI = CGI;
setLayout(gridbag);
Name = new TextField(25);
Phone = new TextField(25);
Done = new Button("Done");
LName = new Label("Your Name");
LPhone= new Label("Phone number");
Message1=new Label("Please enter in the above information and a");
Message2=new Label("sales agent will call to confirm your order");
Con.weightx=.2;
Con.weighty=.2;
Con.anchor = GridBagConstraints.CENTER;
Con.fill = GridBagConstraints.NONE;
Con.gridwidth = GridBagConstraints.REMAINDER;
gridbag.setConstraints(Name, Con);
gridbag.setConstraints(LName, Con);
add(LName);
add(Name);
gridbag.setConstraints(Phone, Con);
gridbag.setConstraints(LPhone, Con);
add(LPhone);
add(Phone);
gridbag.setConstraints(Done, Con);
gridbag.setConstraints(Message1, Con);
gridbag.setConstraints(Message2, Con);
add(Message1);
add(Message2);
add(Done);
pack();
resize(300,300);
show();
}
public boolean action(Event evt, Object arg) {
        if ("Done".equals(arg))
        {
                System.out.println(CGI);
                CGIpost = new HTTPpost(CGI, "NAME: " + Name.getText() +
                "\nPHONE: " + Phone.getText() +"\nPURCHASES:\n" + ItemList +
"\n");
                System.out.println(CGIpost.results());
                dispose();
                return true;
        }
        return false;
}
} // конец CheckOutFrame
```

Когда пользователь нажимает кнопку Done, мы вызываем HTTPpost и передаем URL CGI-программе наряду с данными, которые ввел пользователь. После этого программа завершается. CGI-программа обрабатывает полученные данные, сохраняя их в файле, или отправляет по почте отделу заказов.

Обработка принятых данных при помощи CGI-программы

Мы хотим использовать CGI-программу на сервере Web, чтобы или сохранить информацию о заказе пользователя, или отправить ее по почте в отдел заказов. Не забудьте, что эта программа

определена в теге <PARAM> HTML-документа, из которого вызывается класс Store. Реализация CGI может иметь специфические особенности платформы и в настоящее время не очень хорошо подходит для Java. Чтобы получить больше информации по этой теме, проконсультируйтесь с книгой "The Web Server Book" или с какой-нибудь книгой по программированию CGI. Существует несколько свободно доступных библиотек CGI, которые могут облегчить задачу написания CGI-программы, делающей то, что вам нужно. Если вы не знаете, откуда начать, связи с ресурсами программирования CGI вы можете найти на странице Online Companion.

Возможные улучшения

К Java-магазину можно добавить много интересных особенностей. Используйте то, что вы теперь знаете о потенциале программирования на Java, и поразмышляйте о дополнительных возможностях. Вот некоторые идеи, которые вы можете выполнить как упражнение:

- Поддержка интерфейса "drag and drop". Для этого нужно сделать пиктограмму корзины для покупок так, чтобы пользователь, выбирая товар для приобретения, мог переместить его на пиктограмму, после чего тот будет автоматически добавлен к списку товаров в корзине.
- Дополнительные текстовые поля, в которых пользователь мог бы вводить свой адрес и, возможно, номер кредитной карточки. Это нужно выполнить с особой осторожностью, используя шифрование для защиты информации пользователя. Эта особенность значительно облегчила бы обработку заказов и устранила бы потребность в коммерческом агенте для запроса этой информации.
- Справочные меню. Мы показали, как в нашей программе обрабатывается меню, и выполнить обработчика событий для справочного меню было бы довольно просто.

Глава 18

Взаимодействие с серверами других протоколов:шахматный клиент

Что вы узнаете из этой главы

Здесь мы создадим на языке Java графический клиент для шахматного сервера, подробно рассмотрев следующие вопросы:

- программирование сокетов;
- асинхронная передача данных;
- многопотоковость;
- создание комплексного пользовательского интерфейса.

Контракт

Представьте, что некая недавно образованная компания, работающая на Интернет, решила создать бесплатное высококачественное развлечение для Интернет-сообщества с тем, чтобы привлечь пользователей к своим серверам, и заключила с нами контракт на выполнение этой работы. Мы решили, что для этой задачи подойдет программа игры в шахматы, позволяющая людям играть друг с другом. Мы будем использовать в качестве внутреннего интерфейса шахматный сервер (Chess Internet-Server, CIS), потому что он может выполнять такие функции, как установку связи, подбор игроков, ведение счета, проверку правильности ходов. Однако несколько ограничивает возможности сервера то, что его интерфейс является символьным. На рис. 18-1 изображена текстовая шахматная доска, предоставляемая сервером.

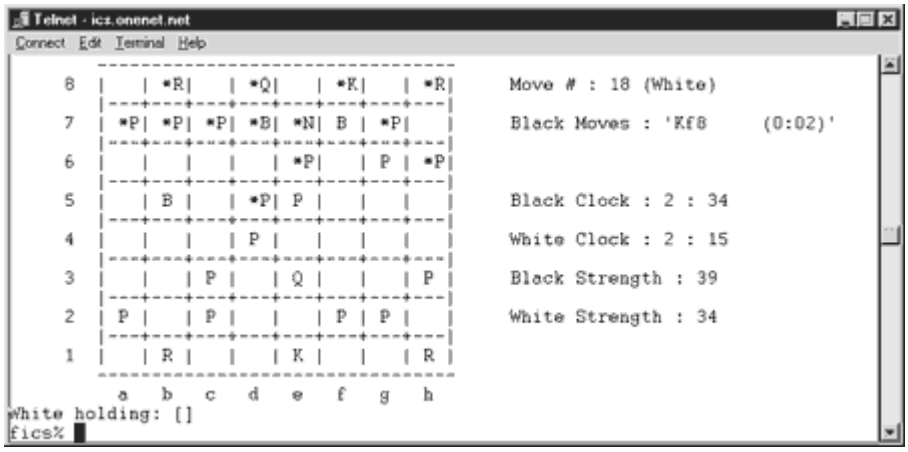

Рис. 18.1.

CIS распространяется бесплатно на условиях публичной лицензии GNU; его можно найти по адресу ftp://chess.onenet.net/pub/chess/Unix.

Существует несколько графических оболочек для работы с шахматным сервером, однако они написаны не для каждой платформы и, кроме того, должны предварительно устанавливаться на машину клиента. Мы решили разработать Java-апплет, который бы служил графической оболочкой для работы с сервером. Программируя графическую оболочку (интерфейс

пользователя) в виде апплета, мы одновременно решаем две задачи: во-первых, клиент может работать на любом компьютере, на котором установлен Java, во-вторых, у пользователей не возникает проблем с его инсталляцией. Ограничения, введенные в Netscape Navigator и связанные с безопасностью апплетов, не мешают нам благодаря тому, что сервер и клиент могут быть установлены на одном и том же компьютере.

Свойства

Наиболее важным свойством, которое добавляет к CIS графический внешний интерфейс, является графическая шахматная доска, позволяющая пользоваться мышью. Нам необходима также внутренняя текстовая строка, где пользователь сможет давать CIS команды, которые не генерируются нашим клиентом. CIS поддерживает ограниченную по времени игру в шахматы, так что мы добавим к шахматной доске динамические часы, отсчитывающие секунды активного игрока. Игрокам разрешается общаться с другими игроками, находящимися в данный момент на сервере, так что мы добавим еще текстовое окно для обеспечения удобства общения. Иногда в дебрях общения теряются входные запросы матча - мы заставим клиента отслеживать их и показывать пользователю в отдельном окне. И в качестве завершающего штриха мы добавим динамически обновляемый список игроков на сервере, получающих текущие запросы матча. Из этого списка наш пользователь сможет выбрать потенциального противника.

Разработка и исполнение

Опишем процесс разработки более или менее хронологически. Начнем с самых фундаментальных задач, выработаем решения, а затем нарастим на полученный скелет мясо в виде функционального CIS-клиента. После этого мы разработаем некоторые расширения нашего апплета и предложим читателю ряд упражнений.

Суть задачи состоит в том, чтобы разработать в апплете графический внешний интерфейс для CIS. Чтобы построить апплет соответствующей сложности, нам придется разбить задачу на несколько самостоятельных кусков, а именно:

- разработку процедуры общения с сервером;
- создание общей шахматной доски;
- создание класса, соединяющего шахматную доску с CIS.

Клиент служит связующим звеном между CIS и пользователем; он должен принимать пожелания пользователя и переводить их на язык команд CIS, и наоборот - принимать выходные данные CIS и переводить их в дружественный для пользователя формат.

Взаимодействие с асинхронным сервером

Из главы 13 вы знаете, как с помощью сокетов устанавливать соединение между апплетом и сервером. Однако мы еще не затрагивали режима асинхронного соединения. Простой сервер, например Finger, обслуживает синхронные потоки данных - он всегда ожидает ввода команды и только после этого передает данные. Некоторые серверы являются асинхронными - они умеют высылать данные в любой подходящий момент времени. К ним относится и наш шахматный сервер. Это значит, что, если один пользователь пошлет данные серверу для другого пользователя, данные будут переданы этому пользователю без всякого предупреждения.

Мы решили разработать специальный приемник данных, работающий в отдельном потоке. Благодаря этому мы будем уверены, что, как бы апплет не был сильно занят в текущий момент, данные, пришедшие из сети, будут приняты. Приемник данных считывает данные в виде значений типа String, получаемых из потока InputStream, разделяя строки при появлении символов EOL или EOF. Давайте опишем интерфейс Listener, позволяющий это делать:

```
package ventana.io;
public interface Listener {
        public void receiveInput(InputStreamHandler i, Object o);
}
```

Этот метод позволяет приемнику данных - классу InputStreamHandler - передавать значения String, считанные из входного потока InputStream, другому объекту с помощью метода receiveInput. Listener способен принимать любые объекты, но в данном случае мы будем передавать через него только значения String. Кроме того, мы возвращаем ссылку на

InputStreamHandler. Она пригодится, если нам понадобится использовать и различать одновременно несколько потоков

```
InputStreamHandler:
package ventana.io;
import java.io.*;
public class InputStreamHandler implements Runnable {<Com private Thread
engine;
        private DataInputStream in;
        private Listener client;
        public InputStreamHandler(InputStream in,
        Listener client) {
                this.in = new DataInputStream(in);
                this.client = client;
                engine = new Thread(this);
                engine.start();
        }
        public void close() {
                if (in!=null) {
                        try {in.close();}
                        catch(IOException e) {}
                }
                engine.stop();
        }
        public void run() {
                if (in == null) {return;}
                        String s;
                try {s = in.readLine();}
                catch (IOException e) {
                        return;
                }}
                while (s != null && engine.isAlive()) {
                        client.receiveInput(this,s);
                        try {s = in.readLine();}
                        catch (IOException e) {s = null;}
                }
        }
}
```

Когда объект InputStreamHandler сконструирован, он создает DataInputStream на базе переданного ему объекта InputStream и запускает поток. Метод run считывает строки из DataInputStream и передает их клиенту Listener с помощью метода receiveInput. Если происходит исключение IOException, InputStreamHandler просто прекращает считывание данных.

Создание шахматной доски

Код, формирующий шахматную доску, никак не связан с кодом, взаимодействующим с сервером. Поэтому создаваемая нами доска с одинаковым успехом может служить как для игры в шахматы, так и, например, в шашки. Наша доска довольно проста - она состоит из клеточек 8 на 8, содержит несколько изображений и реагирует на перемещение и щелчки мышью.

Наша шахматная доска является расширением класса Canvas, который, в свою очередь, является более-менее пригодной для работы реализацией абстрактного класса Component. Мы решили не использовать для доски класс Container по двум причинам. Во-первых, изображения шахматных фигур не принадлежат классу Component и, следовательно, не могут напрямую добавляться к объекту Container. Во-вторых, современная реализация Java API для Microsoft Windows неправильно работает в том случае, если объект Container содержит большое количество объектов Component.

Класс ChessBoard несет ответственность за прорисовку шахматной доски и генерацию ходов. Общие методы класса ChessBoard перечислены в табл. 18-1.

Таблица 18-1. Общие методы класса ChessBoard

Метод	Описание
setImage(Image, String)	Задает квадрат, указанный данной строкой и содержащий данное

	изображение, и перерисовывает квадрат. Строка должна представлять собой шахматное обозначение поля, например "A1".
endGame()	Заканчивает шахматную партию.
setGenerateMoves(boolean)	Указывает, разрешить ли пользователю делать ход.
setOrientation	Ориентация доски - белыми вниз (true) или белыми вверх (false).
boolean getOrientation()	Возвращает текущую ориентацию доски.

Класс ChessBoard содержит двумерный массив изображений. Это и есть шахматные поля. ChessBoard разрешает доступ к массиву полей методом setImage. Метод setImage не определяет индексы массива непосредственно, а позволяет указать в строке с шахматным обозначением, какое поле изменять. Переход от шахматных обозначений к индексам массива осуществляется внутренними средствами, что позволяет выполнять его по-разному, в зависимости от ориентации доски - белыми вниз или белыми вверх. Например, поле "A1" должно быть в левом нижнем углу, если наш пользователь играет белыми, но в правом верхнем углу, если он играет черными.

Когда пользователь щелкает мышью в каком-то шахматном поле, оно становится выделенным и на нем появляется красный кружок, обозначающий выделение. Если пользователь снова щелкает в этом поле, выделение снимается и кружок исчезает. Если пользователь щелкает мышью в другом поле, в то время как первое поле еще выделено, генерируется ход, который передается классу-родителю ChessBoard, после чего с первого поля выделение снимается. Такое поведение может быть запрещено методом setGenerateMoves - чтобы не дать пользователю сделать ход вне очереди.

Вот текст программы для нашего класса ChessBoard:

```
import java.awt.*;
import java.util.*;
public class ChessBoard extends Canvas {
        private Image[][] squares;
        private boolean up = true;
        private boolean generate = false;
        private Point selected;
        private Color dark;
        private Color light;
        public ChessBoard() {
                squares = new Image[8][8];
                dark = Color.gray;
                light = Color.lightGray;
                resize(320,320);
                repaint();
        }
```

Метод EndGame вызывается, когда партия закончена. Он делает все поля темными, чтобы показать пользователю, что партия завершена, а затем перерисовывает доску. Наконец, он пресекает попытки пользователя делать еще ходы:

```
public void endGame() {
        dark = Color.darkGray;
        light = Color.lightGray;
        repaint();
}
public void setGenerateMoves(boolean b) {
        generate = b;
}
public void setOrientation(boolean b) {
        up = b;
}
public boolean getOrientation() {
        return up;
}
```

Метод getSquare используется внутри для перехода от шахматных обозначений к индексам массива. По двум заданным целым числам, указывающим позицию поля в массиве, метод возвращает строку, соответствующую этому полю в шахматных обозначениях:

```
private String getSquare(int x, int y) {
        char ary[] = new char[2];
        if (up) {
                ary[0] = (char)('a'+x);
                ary[1] = (char)('1'+(7-y));
        } else {
                ary[0] = (char)('a'+(7-x));
                ary[1] = (char)('1'+y);
        }
        return new String(ary);
}
```

Класс, ответственный за реальную игру в шахматы, использует метод setImage для того, чтобы вставлять в поля шахматные фигуры. Метод setimage берет изображение - возможно, это изображение фигуры - и строку, указывающую, в какое поле поместить изображение. Если новое изображение отличается от того, что находилось в поле, поле обновляется и перерисовывается. Чтобы удалить фигуру с поля, можно просто поставить в качестве изображения нулевой указатель (null):

```
public void setImage(Image i, String s) {
        int x = (int)(s.charAt(0)-'a');
        int y = (int)(s.charAt(1)-1-'0');
        if (up) {
                y = 7-y;
        } else {
                x = 7-x;
        }
        if (squares[x][y]!=i) {
                squares[x][y] = i;
                paintSquare(x,y,false,getGraphics());
        }
}
public boolean mouseDown(Event evt, int px, int py) {
        if (!generate) {
                return false;
        }
        int x = px/40;
        int y = py/40;
        if (selected==null) {
                selected = new Point(x,y);
                paintSquare(x,y,true,getGraphics());
        } else if (selected.x==x && selected.y==y) {
                selected = null;
                paintSquare(x,y,false,getGraphics());
        } else {
                String move = getSquare(selected.x,selected.y);
                move = move+"-"+getSquare(x,y);
                System.out.println("New move: "+move);
                Event e = new Event(getParent(),Event.ACTION_EVENT,move);
                getParent().deliverEvent(e);
paintSquare(selected.x,selected.y,false,getGraphics());
        selected = null;
        }
        return true;
}
public synchronized Dimension preferredSize() {
        return new Dimension(320,320);
}
public synchronized Dimension minimumSize() {
        return new Dimension(320,320);
}
```

Метод paintSquare отвечает за прорисовку отдельных полей. По заданному массиву индексов он закрашивает поле соответственно светлым или темным цветом (левое нижнее поле всегда светлое) и рисует изображение шахматной фигуры, если она есть на этом поле. Булевский аргумент указывает, надо ли рисовать красный кружок (обозначающий выделение поля):

```
public void paintSquare(int x, int y, boolean b, Graphics g) {
        if ((x%2==0 && y%2==0)||(x%2==1 && y%2==1)) {
                g.setColor(light);
        } else {
                g.setColor(dark);
        }
        g.fillRect(x*40,y*40,40,40);
        if (squares[x][y]!=null) {
                g.drawImage(squares[x][y],x*40,y*40,this);
        }
        if (b) {
                g.setColor(Color.red);
                g.fillOval(x*40+15,y*40+15,10,10);
        }
}
public void paint(Graphics g) {
        for (int y=0; y<8; y++) {
                for (int x=0; x<8; x++) {
                        paintSquare(x,y,false,g);
                }
        }
}
}
```

Связь шахматной доски с CIS

Разработаем апплет, который использует для реализации CIS-клиента как нашу шахматную доску, так и приемник асинхронных данных. Ниже будет представлен текст программы, разбирающей выходные данные CIS и выдающей команды CIS. Если мы хорошо спроектируем наш апплет, это будет единственный раздел, содержащий специфическую для CIS программу.

Шахматную доску не стоит помещать в сам апплет, потому что CIS позволяет нескольким пользователям наблюдать и даже участвовать в нескольких партиях одновременно. Наш клиент не должен ограничивать возможности CIS, а должен, наоборот, расширять их. CIS следит за партиями по номерам, и когда он посылает сообщение с обновленной доской, в это сообщение включается номер партии. Наш основной апплет спроектирован так, чтобы он содержал набор шахматных досок с индексами по номерам партий. Каждая доска соответствует одной партии и находится в собственной рамке.

Установка секундомера

Вместо того чтобы просто показывать, сколько осталось времени, воспользуемся возможностью языка Java работать с потоками и сделаем так, чтобы часы отсчитывали секунды для активного в данный момент игрока. Класс Label дает общее нередактируемое поле выходного текста. Расширим его в класс StopWatch (секундомер), чтобы получить метку, которая показывает текущее время и каждую секунду уменьшает переменную, содержащую величину времени. Общие методы класса StopWatch приводятся в табл. 18-2.

Таблица 18-2. Общие методы класса StopWatch

Метод	Описание
set(int)	Задает и показывает оставшееся время.
start()	Начинает отсчет времени.
stop()	Прекращает отсчет времени.
clear()	Полностью останавливает часы.

Приведем программу для StopWatch:

```
import java.awt.*;
public class StopWatch extends Label implements Runnable {
        private Thread engine;
        private int time=0;
        public StopWatch() {
                super();
                engine = new Thread(this);
                engine.setPriority(3);
        }
        public StopWatch(int i) {
                this();
                set(i);
        }
```

Метод set устанавливает время, оставшееся на секундомере. Он должен быть синхронизирован, потому что разные пары игроков могут попытаться установить часы одновременно. Если время не отрицательно, переустанавливаем выходную метку, чтобы показать время:

```
public synchronized void set(int i) {
        time = 1;
        if (time>0) {
                int min = time/60;
                int sec = time%60;
                if (sec>9) {
                        setText(""+min+":"+sec);
        } else {
                setText(""+min+":0"+sec);
                }
        } else {
                setText("0:00");
        }
}
public void start() {
        if (engine!=null) {
                if (!engine.isAlive()) {
                        engine.start();
                } else {
                        engine.resume();
                }
        }
}
public void stop() {
        if (engine!=null) {
                engine.suspend();
        }
}
        public void clear() {
                if (engine!=null) {
                        engine.stop();
                        engine = null;
                }
        }
        public void run() {
                while (engine!=null && engine.isAlive()) {
                        set(time-1);
                        try {
                                engine.sleep(1000);
                        } catch (InterruptedException e) {}
                }
        }
}
```

Реализация ChessFrame

Теперь мы можем приступить к реализации ChessFrame. В табл. 18-3 приведены общие методы класса ChessFrame.

Таблица 18-3. Методы ChessFrame

Метод	Описание
updateGame(String)	Обновляет состояние кадра для отражения текущего состояния игры.
endGame()	Заканчивает текущую игру.

ChessFrame в первую очередь несет ответственность за разбор сообщений об обновлении игрового поля и выводе соответствующей информации. Сообщение об обновлении по умолчанию было приведено выше на рис. 18-1. CIS дает возможность выбрать из имеющегося набора стилей игровых досок. Стиль, выбранный нами под номером 12, посылает сообщение об обновлении в виде одной строки, содержащей несколько полей, разделенных пробелами:

```
<12> rnbqkbnr pppppppp ---- ----- ---- ---- PPPPPPPP RNBQKBNR
W -1 1 1 1 1 0 1 donald donald 2 0 0 39 39 0 0 1 none (0:00)
none 0
```

Первое поле показывает, что строка содержит сообщение об обновлении игровой доски в формате 12. Родитель ChessFrame будет использовать это поле для того, чтобы определить необходимость вызова метода updateGame. Следующие восемь полей содержат информацию о текущем состоянии доски. Каждой фигуре соответствует определенная буква; белые отмечаются прописными буквами, а черные - строчными. Дефисы обозначают клетки, не содержащие никаких фигур. Остальные поля мы обсудим позже, по мере рассмотрения текста метода updateGame.

Вот исходный текст класса ChessFrame:

```
import ChessBoard;
import ChessClient;
import PackerLayout;
import java.awt.*;
import java.util.*;
import ventana.awt.*;
public class ChessFrame extends Frame {
        private ChessBoard board;
        private ChessClient parent;
        private String user;
        private Panel players;
        private Panel whitePlayer;
        private Panel blackPlayer;
        private Label whiteName;
        private Label blackName;
        private StopWatch whiteTime;
        private StopWatch blackTime;
```

Кадр содержит два главных компонента: сам ChessBoard и панель с информацией об игроках. Эта панель разделена на две панели для игроков, играющих белыми и черными. На каждой из этих панелей выводятся имя игрока и оставшееся ему по секундомеру время.

```
public ChessFrame(ChessClient parent, String user, Font fixed) {
        super("Chess Game");
        this.user = user;
        this.parent = parent;
        setLayout(new PackerLayout());
        whitePlayer = new Panel();
        whitePlayer.setLayout(new PackerLayout());
        whiteName = new Label("-------");
        whiteName.setFont(fixed);
        whiteName.setAlignment(Label.CENTER);
        whiteTime = new StopWatch();
```

```
        whiteTime.setFont(fixed);
        whiteTime.setAlignment(Label.CENTER);
        whitePlayer.add("whitename;fill=x;pady=5", whiteName);
        whitePlayer.add("whitetime;fill=x;pady=5", whiteTime);
        blackPlayer = new Panel();
        blackPlayer.setLayout(new PackerLayout());
        blackName = new Label("-------");
        blackName.setFont(fixed);
        blackName.setAlignment(Label.CENTER);
        blackTime = new stopWatch();
        blackTime.setFont(fixed);
        blackTime.setAlignment(Label.CENTER);
        blackPlayer.add("blackname;fill=x;pady=5", blackName);
        blackPlayer.add("blacktime;fill=x;pady=5", blackTime);
        players = new Panel();
        players.setLayout(new PackerLayout());
        players.add("wplayer;side=bottom;fill=x", whitePlayer);
        players.add("bplayer;side=top;fill=x", blackPlayer);
        add("panel1;side=left", players);
        board = new ChessBoard();
        add("panel2;side=left", board);
        pack();
        resize(size());
        show();
}
```

Данный кадр обрабатывает лишь одно событие - WINDOWS_DESTROY. Любое другое событие перенаправляется к родительскому объекту. Таким образом, события, связанные с ходами игроков, передаются выше и в конце концов попадают на сервер:

```
public boolean handleEvent(Event evt) {
        if (evt.id==Event.WINDOWS_DESTROY) {
                dispose();
                return true;
        } else {
                parent.deliverEvent(evt);
                return true;
        }
}
```

В момент первоначального обновления шахматной доски мы обнаруживаем, какого цвета фигурами играет наш игрок. Если он играет черными, метод flipPlayers поменяет метки игроков местами и перевернет доску:

```
protected void flipPlayers() {
        players.remove(whitePlayer);
        players.remove(blackPlayer);
        players.add("wplayer;side=top;fill=x" whitePlayer);
        players.add("bplayer;side=bottom;fill=x", blackPlayer);
        board.setOrientation(false);
}
```

Следующий метод является сердцевиной объекта ChessFrame - он занимается разбором сообщения об обновлении. Сперва при помощи StringTokenizer мы считываем из этого сообщения восемь строк состояния шахматной доски. Далее, мы определяем, чей был ход (эта информация будет использована немного позже, когда мы будем определять, сколько времени осталось у каждого игрока). Вот фрагмент кода, реализующего этот алгоритм:

```
public void updateGame(String s) {
        System.out.println(s);
        StringTokenizer st = new StringTokenizer(s);
        st.nextToken(); // метка-индикатор 12
        String lines[] = new String[8];
```

```
        for (int i=0; i; i++) {
                lines[i] = st.nextToken();
        }
        String token = st.nextToken(); // Чей это ход (W, B)
        boolean whiteMove = true;
        if (token.equals("B")) {
                whiteMove = false;
        }
        st.nextToken();
        st.nextToken(); // могут ли белые делать короткую рокировку?
        st.nextToken(); // могут ли белые делать длинную рокировку?
        st.nextToken(); // могут ли черные делать короткую рокировку?
        st.nextToken(); // могут ли черные делать длинную рокировку?
        st.nextToken(); // количество ходов
        st.nextToken(); // номер игры
```

Следующие токены содержат имена игроков. Если они не совпадают с именами игроков, указанными в метках, мы предполагаем, что сервер CIS сам знает, что он делает. Если имя нового игрока черными совпадает с именем нашего пользователя, а переменная type не равна двум, мы переворачиваем доску таким образом, что черные оказываются внизу. Переменная type содержит информацию о роли игрока в шахматной партии. Если она равна двум, это значит, что пользователь играет в тренировочном режиме, то есть против самого себя. Если значение переменной равно единице, значит, пользователь играет с реальным противником и в данный момент его очередь ходить. В любом случае нам требуется, чтобы доска генерировала ходы:

```
token = st.nextToken() // имя игрока белыми
if (! WhiteName.getText().equals(token)) {
        whiteName.setText(token);
}
token = st.nextToken(); // имя игрока черными
String type = st.nextToken();
if (! BlackName.getText().equals(token)) {
        blackName.setText(token);
        if (user.equals(token) && ! type.equals("2")) {
                board.setOrientation(false);
                flipPlayers();
                System.out.prinln("Tried to anyway");
        }
        layout();
}
if (type.equals("1") || type.equals("2")) {
        board.setGenerateMoves(true);
} else {
        board.setGenerateMoves(false);
}
token = st.nextToken(); // время начала партии
token = st.nextToken();
token = st.nextToken();
token = st.nextToken();
```

Два следующих токена содержат количество секунд, оставшееся в распоряжении каждого из игроков. StopWatches устанавливается на новое, правильное время, для игрока, который только что сделал ход, отсчет останавливается и, в свою очередь, запускается счетчик другого игрока:

```
token = st.nextToken(); // оставшееся время игрока белыми
try {
        Integer i = new Integer(token);
        whiteTime.set(i.intValue());
} catch (NumberFormatException e) {}
token = st.nextToken(); // оставшееся время игрока черными
try {
        Integer i = new Integer(token);
        blackTime.set(i.intValue());
```

```
} catch (NumberFormatException e) {}
if (whiteMove) {
        blackTime.stop();
        whiteTime.start();
} else {
        whiteTime.stop();
        blackTime.start();
}
```

И наконец, мы приступаем к разбору оставшихся токенов. Для того чтобы процедура обновления шахматной доски происходила незаметно для пользователя, мы выполняем ее в самый последний момент. Считывание каждой строки состояния происходит последовательно, символ за символом. Если очередной считанный символ является дефисом, соответствующий квадрат доски не содержит картинки; в противном случае картинка запрашивается у родительского процесса:

```
for (int i=0; i; i++) {
        String line = lines[i];
        char ary [] = new char[8];
        line.getChar(0, 8, ary, 0);
        for (int j=0; j; j++) {
                char row = (char)(`8'-i);
        char column = (char)(`a'+j);
        Image piece;
        if (ary[j]!='-`) {
                piece = parent.getPieceImage(""+ary[j]);
        } else {
                piece = null;
                }
                board.setImage(piece,""+column+row);
        }
}
```

Написание апплета

Мы создали отдельное окно, в котором расположена игровая доска общего назначения и информация, необходимая игроку. Теперь мы можем приступить к написанию собственно апплета. Мы начнем с малого, добавляя дополнительные функции по мере необходимости. Первая реализация апплета должна уметь соединяться с сервером и предлагать игроку окно, позволяющее выбрать партнера из списка. Как только принимается строка, содержащая в начале <12>, апплет должен передавать ее классу ChessFrame, конструируя последний в случае необходимости. Кроме того, апплет должен различать сигнал сервера о конце игры и передавать эту информацию в ChessFrame. Вот исходный текст апплета ChessClient:

```
import ChessFrame;
import PackerLayout;
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.util.*;
import java.io.*;
import ventana.io.*;
public class ChessClient extends Applet implements Listener {
        private String user;
        private Label title;
        private TextArea output;
        private TextField input;
        private Hashtable games;
        private Hashtable pieceImages;
        privateSocket ChessSocket;
        private InputStream ChessInput;
        private OutputStream ChessOutput;
        private InputStreamHandler ChessInputHandler;
        private MediaTracker tracker;
```

```
        private Font fixed;
        private Font pretty;
```

Первое, что необходимо сделать при инициализации апплета в процедуре init, - установить два типа шрифтов, pretty и fixed, для всего апплета. Поскольку на разных компьютерах разрешение экрана сильно варьируется, шрифт размером в десять пунктов, выглядящий вполне пристойно на старом мониторе невысокого разрешения, может оказаться совершенно нечитаемым на современном мониторе высокого разрешения. По этой причине вопрос о выборе размера шрифта остается на усмотрение разработчика HTML-страницы. Для установки шрифтов используется следующая последовательность команд:

```
public void init() {
        String fixedName = getParameter("FIXEDFONTNAME");
        if (fixedName==null) {
                fixedName = "Courier";
        }
        int fixedSize = 12;
        String fixedSizestr = getParameter("FIXEDFONTSIZE");
        if (fixedSizestr!=null) {
                try {
                        Integer i = new Integer(fixedSizestr);
                        fixedSize = i.intValue();
                } catch (NumberFormatException e) {}
        }
        fixed = new Font(fixedName, Font.PLAIN, fixedSize);
        String prettyName = getParameter("PRETTYFONTNAME");
        if (prettyName==null) {
                prettyName = "Courier";
        }
        int prettySize = 12;
        String prettySizestr = getParameter("PRETTYFONTSIZE");
        if (prettySizestr!=null) {
                try {
                        Integer i = new Integer(prettySizestr);
                        prettySize = i.intValue();
                } catch (NumberFormatException e) {}
        }
        pretty = new Font(prettyName, Font.PLAIN, prettySize);
```

Теперь, когда мы установили информацию о шрифтах, пора приступать непосредственно к апплету. Мы добавляем метку-заголовок TextArea для отображения данных, поступивших с сервера. Кроме того, мы загружаем изображения фигур в хеш-таблицу:

```
        setLayout(new PackerLayout());
        title = new Label("Internet Chess Server");
        title.setFont(
                new Font(prettyName, Font.PLAIN, prettySize+12));
        title.setAlignment(Label.CENTER);
        add("title;side=top;fill=x", title);
        output = new TextArea(20, 80);
        output.setEditable(false);
        output.setFont(fixed);
        add("output;side=top;fill=x", output);
        input = new TextField(80);
        input.setFont(fixed);
        add("input;side=top;fill=x", input);
        show();
        pieceImages = new Hashtable(10);
        initPieceImages();
}
```

Данная процедура создает хеш-таблицу объетов Image. Символ отображает тип фигуры, которой соответствует данный объект Image. Вначале таблица индексирована в порядке URL (в

формате String). Далее мы проходим по таблице, извлекаем все изображения, на которые ссылаются URL, и строим из них новую хеш-таблицу. Для того чтобы до завершения работы все изображения были гарантированно извлечены, используется объект MediaTracker:

```
protected void initPieceImages() {
        tracker = new MediaTracker(this);
        HashTable h = new HashTable(10);
        h.put("r", "pics/br.gif");
        h.put("n", "pics/bn.gif");
        h.put("b", "pics/bb.gif");
        h.put("q", "pics/bq.gif");
        h.put("k", "pics/bk.gif");
        h.put("p", "pics/bp.gif");
        h.put("P", "pics/wp.gif");
        h.put("R", "pics/wr.gif");
        h.put("N", "pics/wn.gif");
        h.put("B", "pics/wb.gif");
        h.put("Q", "pics/wq.gif");
        h.put("K", "pics/wk.gif");
        Enumeration e = h.keys();
        while(e.hasMoreElements()) {
                try {
                        String key = (String)e.nextElement();
                        String s = (String)g.get(key);
                        URL u = new URL(getCodeBase(), s);
                        Image i = getImage(u);
                        tracker.addImage(u);
                        pieceImages.put(key, i);
                        } catch (Ecxeption ex) {
                                handleException(ex);
                        }
                }
        try {tracker.waitForAll();} catch (InterruptedException ex) {}
}
```

Метод getPieceImage возвращает объект типа Image, на который указывает фигура String. Например, вызов getPieceImage("P") возвращает объект Image, соответствующий белой пешке:

```
public Image getPieceImage(String piece) {
        return (Image)pieceImages.get(piece);
}
```

Стартовав, апплет устанавливает соединение с сервером через сокет. Для объекта InputStream создается соответствующий InputStreamHandler. Как вы помните, данный класс считывает строки из входного потока и передает их с помощью метода receiveInput. Наконец, создается хеш-таблица для хранения объектов ChessFrame. Большинство пользователей предпочитают играть или наблюдать за одной игрой в один момент времени, поэтому хеш-таблица инициализируется для хранения одного элемента:

```
public void start() {
        try {
                String ChessHost = hetParameter("HOST");
                if (ChessHost==null) {
                        ChessHost = getCodeBase().getHost();
                }
                ChessSocket = new Socket(ChessHost, 5000);
                ChessInput = ChessSocket.getInputStream();
                ChessOutput = ChessSocket.getOutputStream();
                ChessInputHandler = new InputStreamHandler(ChessInput, this);
        } catch (Exception e) {
                handleException(e);
        }
games = new Hashtable(1);
```

Заканчивая работу, апплет закрывает InputStreamHandler, сетевые потоки и сокет, а также очищает окна ввода и отображения информации:

```
public void stop() {
        try {
                ChessInputHandler.close();
                ChessOutput.close();
                ChessInput.close();
                ChessSocket.close();
        } catch (IOException e);
                handleException(e);
        }
        output.setText("");
        input.setText("");
}
public void handleException(Exception e) {
        e.printStackTrace();
}
```

Для передачи данных серверу используется метод writeOutput. То, что данный метод - единственный предназначенный для передачи данных, дает нам два преимущества: во-первых, нам не нужно повторять один и тот же набор операторов в каждом месте программы, где нужно выводить данные, во-вторых, поскольку в методе использовано ключевое слово synchronized, у нас есть гарантия, что несколько потоков программы не начнут одновременную передачу. В начале переменной user еще не присвоено значение, однако оно автоматически присваивается, как только сделан первый вызов writeOutput. Так происходит потому, что первая команда, подающаяся на сервер, является именем пользователя:

```
protected synchronized void writeOutput(String s) {
        if (user==null) {
                user = s.trim();
        }
        byte b[] = new byte[s.length()];
        s.getBytes(0, s.length(), b, 0);
        try {
                ChessOutput.write(b);
        } catch (IOException e) {
                handleException(e);
        }
}
```

Метод receiveInput вызывается объектом InputStreamHandler каждый раз, как только получена полная входная строка. Если принятая строка пуста или содержит только приглашение сервера, она полностью игнорируется. Если принятая строка начинается с символов <12>, мы знаем, что она является сигналом к обновлению состояния игры - вызывается метод parseBoard. Если похоже, что строка является сигналом к завершению игры, анализируется ее номер. Если номер игры входит в нашу хеш-таблицу, вызывается метод endGame объекта ChessFrame - и соответствующая игра удаляется из хеш-таблицы. Если строка не соответствует ни одному из вышеописанных событий, она расценивается как стандартная команда сервера и попадает в окно выдачи информации:

```
public void receiveInput(InputStreamHandler ish, Object o) {
        String s = (String)o;
        if (s.trim().equals("") || s.trim().equals("fics%")) {
                return;
        }
        if (s.trim().startsWith("<12>")) {
                parseBoard(s.trim());
        } else if (s.trim().startsWith("{Game")) {
                StringTokenizer st = new StringTokenizer(s);
                String token = st.nextToken(); //{Game
                String number = st.nextToken(); //number
                ChessFrame frame = (ChessFrame)games.get(number);
```

```
                if (frame!=null) {
                        token = st.nextToken(); //(player1
                        token = st.nextToken(); //vs.
                        token = st.nextToken(); //player2)
                        token = st.nextToken(); //loser
                        frame.endGame(token);
                        games.remove(number);
                }
        } else {
                output.appendText(s+"\n");
        }
}
```

Данный метод вызывается, как только метод receiveInput обнаружит сообщение об обновлении игры. Шестнадцатым полем в строке является номер игры. Если данная игра отсутствует в нашей таблице, мы создаем новый объект ChessFrame и вносим его номер в таблицу. Наконец, мы обновляем окно при помощи метода updateGame:

```
protected void parseBoard(String s) {
        StringTokenizer st = new StringTokenizer(s);
        String token = st.nextToken();
        if (! token.equals("<12>")) {
                output.appendText("oops... "+token);
                        return;
        }
        for (int i=0; i<<16; i++) {
                token = st.nextToken();
        }
        ChessFrame frame = (ChessFrame)games.get(token);
        if (frame==null) {
                frame = new ChessFrame(this, user, fixed);
                games.put(token, frame);
        }
        frame.updateGame(s);
}
```

Все события, связанные с ходами игроков, обрабатываются здесь. Если местом назначения события является объект ChessFrame, событием должен быть ход игрока, сгенерированный доской, то есть его необходимо перенаправить серверу. Если местом назначения события является окно ввода, сообщение передается серверу, а окно ввода очищается:

```
public boolean action(Event evt, Object arg) {
        if (evt.target instanceof ChessFrame) {
                writeOutput((String)arg+"\n");
                return true;
        } else if (evt.target==input) {
                writeOutput((String)arg+"\n");
                input.setText("");
                return true;
        } else {
                return false;
        }
}
```

Шахматный клиент написан! Апплет выводит на экран свое название, окно выдачи информации с сервера и поле для ввода команд. Если пользователь начинает другую игру, создается новое окно ChessFrame, в котором можно играть, пользуясь удобным графическим интерфейсом. На рис. 18-2 воспроизведен фрагмент изображения работающего шахматного апплета.

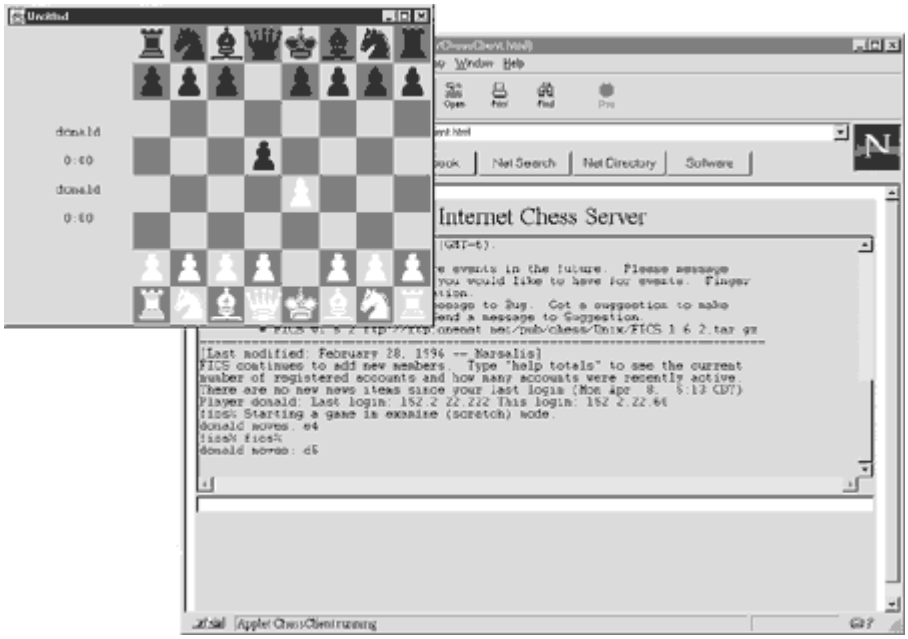

Рис. 18.2.

Возможные усовершенствования

Наш шахматный апплет был распространен среди широкой публики и, похоже, понравился многим. Как всегда, посыпались советы и рекомендации о том, как его можно усовершенствовать. Во-первых, процедура входа на сервер не маскирует вводимый пароль, следовательно, его может подсмотреть любой желающий. Во-вторых, апплет не устанавливает стиль доски 12 вместо 1, оставляя это на усмотрение игрока (см. рис. 18-1). В-третьих, некоторые игроки затруднялись отслеживать партнеров, готовых к сетевой игре, то есть им необходим отдельный список игроков в отдельном окне. Мы решили усовершенствовать апплет, выпустив его следующую версию.

Окно login

Окно login решено было оформить в виде отдельного, внешнего окна. Когда пользователь соединяется сервером, апплет наблюдает за окном login. Как только окно login сформировано, запускается процедура login. Кадр login состоит из двух полей текстового ввода - одно для имени, другое для пароля - и кнопки для запуска процедуры. В качестве окна login можно было бы применить стандартный диалог, поскольку он обладает свойством модальности. Это значит, что программа не реагирует ни на какое событие, не связанное с самим диалогом. К сожалению, на момент написания книги реализация диалога в Netscape Navigator 2.0 для Solaris была немного некорректной, поэтому нам пришлось обходиться без него. Вот исходный код класса UserLogin:

```
import java.awt.*;
import java.util.*;
public class UserLogin extends Frame {
        private Button login;
        private TextField user;
        private TextField password;
        private Component target;
```

В конструкторе объекта UserLogin задается объект Component, которому следует посылать события, и устанавливается шрифт для отображения в окне login. Далее мы добавляем два текстовых поля TextField, одно для имени пользователя, другое - для его пароля. В последнем в качестве эхо-символа используется звездочка (*). Кроме того, мы добавляем кнопку login, генерирующую событие "login" и закрывающую окно:

```
public UserLogin(Component target, Font f) {
        super("Internet Chess Server Login");
```

```
        this.target = target;
        setBackground(Color.gray);
        setLayout(new FlowLayout());
        Panel p = new Panel();
        p.setLayout(new FlowLayout());
        Label l = new Label("Username:");
        p.add(l);
        user = new TextField(16);
        user.setFont(f);
        user.setEditable(true);
        p.add(user);
        l = new Label("Password:");
        p.add(l);
        password = new TextField(16);
        password.setFont(f);
        password.setEditable(true);
        password.setEchoCharacter('*');
        p.add(password);
        login = new Button("Login");
        login.setFont(f);
        p.add(login);
        add(p);
        pack();
        show();
}
```

Как только пользователь нажимает кнопку login, проводится проверка содержимого полей ввода - введено ли хоть что-нибудь. Если нет, мы предполагаем, что кнопка была нажата случайно, и повторяем вывод окна login. Поле пароля не проверяется, поскольку большинство шахматных серверов допускают до игры всех желающих. Если пользователь ввел свое имя в окне username, мы генерируем событие и передаем его объекту, указанному ранее при вызове конструктора окна login. Параметр события - вектор с двумя элементами, именем пользователя и паролем. После того как событие сгенерировано, окно login уничтожается:

```
public boolean action(Event evt, Object arg) {
        if (evt.target==login) {
                if (user.getText().trim().equals("")) {
                        return true;
                }
                Vector v = new Vector(2);
                v.addElement(user.getText());
                v.addElement(password.getText());
                Event e = new Event(this, Event.ACTION_EVENT, v);
                target.deliverEvent(e);
                dispose();
                return true;
        }
        return false;
}
```

Теперь добавим наше окно к основному апплету. Нам необходимо добавить код, обрабатывающий новое событие, генерируемое окном login. Окно UserLogin конструируется при запуске апплета, а исходный текст, добавляемый в конец метода start, выглядит следующим образом:

```
UserLogin ul = new UserLogin(this, pretty);
```

Кроме того, добавляется следующее условие:

```
} else if (evt.target instanceof UserLogin) {
        Vector v = (Vector)evt.arg;
        user = (String)v.elementAt(0);
        String pass = (String)v.elementAt(1);
        writeOutput(user+"\n"+pass+"\n");
        writeOutput("set style 12\n");
```

```
        return true;
}
```

Присваивать значение переменной user в этом методе безопаснее, чем в методе writeOutput. После того как присвоение переменной user из метода writeOutput удалено, можно считать, что усовершенствование состоялось. На рис. 18-3 изображено окно UserLogin.

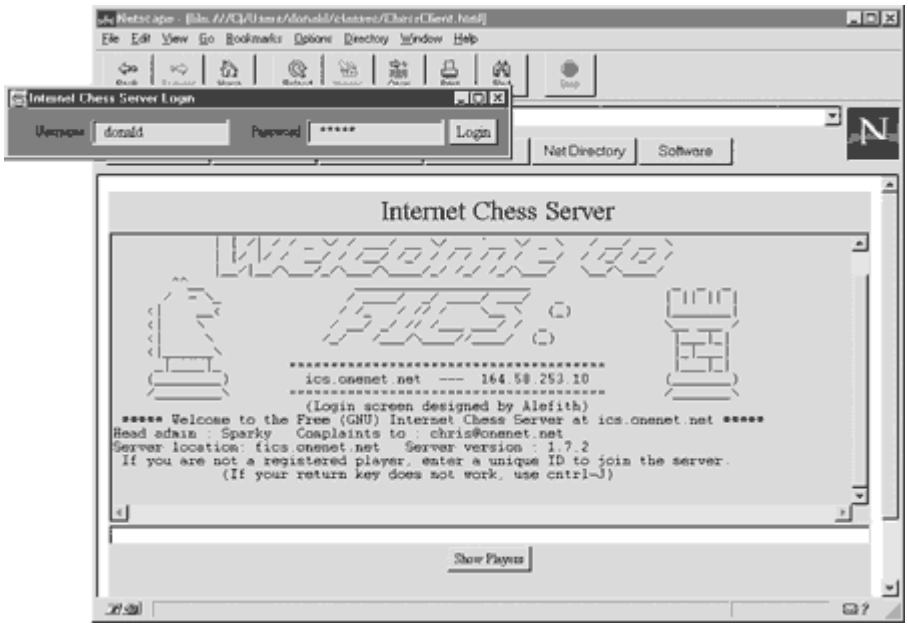

Рис. 18.3.

Список текущих игроков

Добавление списка текущих игроков в отдельном окне - занятие, безусловно, более интересное, чем рассмотренное выше. Разработанное нами отдельное окно состоит из трех колонок. В нем отображаются имена игроков и количество набранных очков. Имя игрока - это кнопка, нажав на которую мы создаем новую игру с выбранным игроком. Одна из проблем отображения списка игроков связана с его динамичностью - список необходимо постоянно обновлять, следя за выбыванием старых и появлением новых игроков. Мы создаем отдельный поток, которой раз в минуту обновляет список. Для этого вводится класс Player, содержащий информацию об имени игрока и его блиц-рейтингом. Вот его исходный текст:

```
public class Player {
        public String name;
        public String blitz;
        public Player(String name, String blitz) {
                this.name = name;
                this.blitz = blitz;
        }
}
```

Класс PlayerFrame создает кадр, в котором отображаются колонки с именами-кнопками и метками-очками. Входные данные передаются методом updatePlayerListing:

```
import java.awt.*;
import java.util.*;
import PackerLayout;
import Player;
import ChessClient;
public class PlayerFrame extends Frame {
        private ChessClient parent;
        private Vector playerList;
        private Panel topplayers;
        private Panel topscores;
```

```
        private Panel midplayers;
        private Panel midscores;
        private Panel lowplayers;
        private Panel lowscores;
        public PlayerFreame(ChessClient parent) {
                this(parent, new Vector(0));
        }
```

Основной конструктор включает два аргумента: ChessClient и список игроков в виде вектора. Мы конструируем три панели для имен игроков и три - для блиц-рейтингов:

```
public PlayerFrame(ChessClient parent, Vector v) {
        super("Chess Players");
        this.parent = parent;
        topplayers = new Panel();
        topscores = new Panel();
        midplayers = new Panel();
        midscores = new Panel();
        lowplayers = new Panel();
        lowscores = new Panel();
        setLayout(new PackerLayout());
        setBackground(Color.gray);
        add("topplayers;side=left", topplayers);
        add("topscores;side=left", topscores);
        add("midplayers;side=left", midplayers);
        add("midscores;side=left", midscores);
        add("lowplayers;side=left", lowplayers);
        add("lowscores;side=left", lowscores);
        updatePlayerList(v);
        pack();
}
```

Метод updatePlayerList обновляет содержимое кадра:

```
public void updatePlayersList(Vector v) {
        playerList = v;
        topplayers.removeAll();
        topscores.removeAll();
        midplayers.removeAll();
        midscores.removeAll();
        lowplayers.removeAll();
        lowscores.removeAll();
        int length = 2+playerList.size()/3;
        topplayers.setLayout(new GridLayout(length, 1));
        topscores.setLayout(new GridLayout(length, 1));
        midplayers.setLayout(new GridLayout(length, 1));
        midscores.setLayout(new GridLayout(length, 1));
        lowplayers.setLayout(new GridLayout(length, 1));
        lowscores.setLayout(new GridLayout(length, 1));
        topplayers.add(new Label("Player Name"));
        topscores.add(new Label("Score");
        midplayers.add(new Label("Player Name"));
        midscores.add(new Label("Score");
        lowplayers.add(new Label("Player Name"));
        lowscores.add(new Label("Score");
        Enumeration e = playerList.elements();
        Player p;
        int count = 0;
        while (e.hasMoreElements()) {
                p = (Player)e.nextElement;
                Button b = new Button(p.name);
                Label l = new Label(p.blitz);
                switch(count%3) {
```

```
                        case 0:
                                topplayers.add(b);
                                topscores.add(1);
                                break;
                        case 1:
                                midplayers.add(b);
                                midscores.add(1);
                                break;
                }
                }
                count++;
        }
        layout();
        pack();
```

При нажатии на кнопку с именем любого игрока генерируется событие для PlayerFrame, которое затем передается объекту ChessClient со строкой-запросом начала новой игры в качестве аргумента. Мы конструируем новое событие, вместо того чтобы использовать стандартное, связанное с нажатием кнопки потому, что апплет ничего не знает про связь кнопки с именем игрока. Все, что необходимо знать апплету, - то, что мы собираемся начать игру с определенным игроком. К главному окну объекта CheeClient добавляется объект MenuBar, поэтому нам необходимо также следить за событиями, связанными с меню. Они передаются без изменений, поскольку объекты MenuItem создаются самим апплетом, а мы должны различать места назначения генерируемых событий:

```
public boolean handleEvent(Event evt) {
        if (evt.target instanceof Button) {
                String s = "match "+(String)evt.arg;
                Event e = new Event(this, Event.ACTION_EVENT, s);
                parent.deliverEvent(e);
                return true;
        } else if (evt.target instanceof MenuItem) {
                parent.deliverEvent(evt);
                return true;
        }
        return false;
}
```

Теперь, когда у нас есть готовый кадр, приступим к заполнению его полей данными. Объект PlayerFrame добавляется к апплету и снабжается меню MenuBar с двумя опциями: обновить список немедленно и приступить к автоматическому обновлению. К апплету необходимо добавить несколько переменных:

```
private MenuBar whobar;
private PlayerFrame players;
private MenuItem whoRefresh;
private MenuItem whoStopStart;
private Button showhidePlayers;
private String whoCommand;
private Vector playerList;
private boolean waitingforwho = false;
```

Пункты меню ассоциированы с объектом PlayerFrame; кнопка находится внизу поля ввода текста апплета и управляет видимостью окна со списком игроков. Управляющая строка содержит запрос к серверу на обновление списка текущих игроков, вектор содержит сам список, а переменная waitingforwho позволяет методу receiveInput распознать состояние ожидания списка игроков от сервера. К методу main апплета добавляется следующий исходный текст:

```
whoCommand = "who an";
whobar = new MenuBar();
whobar.setFont(pretty);
Menu m = new Menu("Update Listing");
whoRefresh = new MenuItem("Refresh Now");
whoStopStart = new MenuItem("Start Updating");
m.add(whoRefresh);
```

```
m.add(whoStopStart);
whobar.add(m);
players = new PlayerFrame(this);
players.setFont(fixed);
players.setMenuBar(whobar);
showhidePlayers = new Button("Show Players");
add("showhideplayers;side=top", showhidePlayers);
```

Как видим, команда-запрос инициализируется строкой "who an". Символ a обозначает, что нам нужен список игроков, доступных в данный момент для игры, а символ n - что мы хотим, чтобы список был представлен в подробной форме, что облегчит задачу по его разбору. Далее создается новое меню, которое добавляется к PlayerFrame. Наконец, создается и добавляется к апплету новая кнопка для управления окном. К методу start необходимо добавить следующие команды - они создают новый список игроков:

```
playerList = new Vector(32);
player.updatePlayerList(playerList);
```

К командам разбора событий добавляются следующие:

```
} else if (evt.target==whoRefresh) {
        writeOutput(whoCommand+"\n");
        playerList = new Vector(32);
        waitingforwho = true;
        return true;
} else if (evt.target==showhidePlayers) {
        if (players.isShowing()) {
                players.hide();
                showhidePlayers.setLabel("Show Players");
        } else {
                players.show();
                showhidePlayers.setLabel("Hide Players");
        }
        return true;
} else if (evt.target==players) {
        writeOutput((String)arg+"\n");
        return true;
}
```

Как вы помните, PlayerFrame передает события MenuItem апплету без изменений. Здесь мы перехватываем запрос пользователя на обновление списка, высылаем команду серверу и устанавливаем переменную waitingforwho в true. Кроме того, мы проверяем состояние кнопки, управляющей видимостью окна списка игроков. В зависимости от результатов проверки состояние PlayerFrame изменяется так же, как и надпись на его кнопке. Если событие генерируется объектом PlayerFrame, запрос на проведение игры передается серверу.

Теперь нам необходимо модифицировать метод receiveInput так, чтобы он оказался способен разбирать принятый от сервера список игроков при установленной в true переменной waitingforwho. Список, генерируемый в ответ на команду "who an", выглядит так:

```
Name          Stand win loss draw Blitz  win  loss draw idle
MchessPre(C) 2337  20  7    8     2474 145  53   22    9
jocelyn       1970  1   0    1     1985 12   11   2
StIdes        1820  5   5    1     1906 28   22   2     13
3 Players Displayed.
```

В данном случае нас интересуют только поля Name и Blitz. Начало списка определяется словом "Name", за которым следует "Stand". Состоящая из дефисов строка игнорируется. Окончание списка определяется словами "Players Displayed". Вот новая версия метода receiveInput:

```
public void receiveInput(InputStreamHandler ish, Object o) {
        String s = (String)o;
```

```
        if (s.trim().equals("") || s.trim().equals("fics%")) {
                return;
        }
        if (s.trim().startsWith("<12>")) {
                parseBoard(s.trim());
        } else if (s.trim().startsWith("{Game")) {
                StringTokenizer st = new StringTokenizer(s);
                String token = st.nextToken(); //{Game
                String number = st.nextToken(); //number
                ChessFrame frame = (ChessFrame)games.get(number);
                if (frame!=null) {
                        token = st.nextToken(); //(player1
                        token = st.nextToken(); //vs.
                        token = st.nextToken(); //player2)
                        token = st.nextToken(); //loser
                        frame.endGame(token);
                        games.remove(number);
                }
```

Отсюда начинается измененная часть. Если мы ожидаем появления списка игроков, для входной строки создается новый StringTokenizer. Если строка начинается с приглашения сервера, она нас не интересует. Но если строка начинается со слова "Name" - это то, что мы ожидаем увидеть. Если следующее слово - "Stand", остаток строки можно проигнорировать. В любом другом случае мы имеем дело с произвольным ответом сервера и добавляем его к окну вывода TextArea.

Далее мы проверяем наличие и количество дефисов в следующей строке и, если все правильно, полностью игнорируем ее. Если нет - проверяем, начинается ли строка с числа, за которым следует "Players". Если да, строка должна быть окончанием списка игроков, поэтому мы обновляем PlayerFrame и устанавливаем переменную waitingforwho в false. Если ни одно из вышеперечисленных условий не выполнено, строка должна быть частью передаваемого списка игроков, и мы разбираем ее, выделяя имя игрока и его блиц-рейтинг. Из полученной информации создается объект Player, который затем добавляется в вектор игроков:

```
} else if (waitingforwho) {
        StringTokenizer st = new StringTokenizer(s);
        String token = st.nextToken();
        if (token.equals("fics%")) {
                return;
        }
        if (token.equals("Name")) {
                if (st.nextToken().equals("Stand")) {
                        return;
                } else {
                        output.appentText(s+"\n");
                        return;
                }
        } else if (token.equals("--------")) {
                return;
        } else {
                try {
                        Integer i = new Integer(token);
                        if (st.nextToken().equals("Players")) {
                                waitingforwho = false;
                                players.updatePlayerList(playerList);
                        }
                        return;
                } catch (NumberFormatExeption e) {}
                String name = token;
                st.nextToken(); //standart rating
                st.nextToken(); // standart win total
                st.nextToken(); // standart loss total
                st.nextToken(); // standart draw total
                String blitz = st.nextToken();
                Player p = new Player(name, blitz);
```

```
                playerList.addElement(p);
                return;
        }
        else {
                output.appendText(s+"\n");
}
```

Теперь апплет обладает работающим объектом PlayerFrame. Видимость PlayerFrame регулируется нажатием на кнопку Show/Hide Players. Пользователь может запросить немедленное обновление списка текущих игроков выбором пункта в меню PlayerFrame. Нам осталось добавить лишь одну функцию - автоматическое обновление списка. Мы создадим поток, активизирующийся каждую минуту и генерирующий событие, требующее обновления списка. К апплету добавляется следующая переменная:

```
private Thread engine;
```

В метод апплета start мы добавляем следующее:

```
engine = new Thread(this);
```

В метод апплета stop добавляем следующее:

```
engine.stop();
```

Вот как выглядит метод апплета run:

```
public void run() {
        while (engine!=null && engine.isAlive()) {
                Event e = new Event(whoRefresh, Event.ACTION_EVENT, "Refresh
now");
                handlrEvent(e);
                try {
                        engine.sleep(60000);
                } catch (InterruptedException ex) {}
        }
}
```

Так как автоматическое обновление может потреблять много временных ресурсов, в особенности если сервер сильно загружен, мы дали возможность пользователю отключать его, выбирая пункт меню whoStopStart, добавленный нами в объект PlayerFrame ранее. К методу action апплета было добавлено следующее условие:

```
} else if (evt.target==whoStopStart) {
        if (!engine.isAlive()) {
                engine.start();
                whoStopStart.setLabel("Stop updating");
                running = true;
        } else if (running) {
                engine.suspend();
                whoStopStart.setLabel("Start updating");
                running = false;
        } else {
                engine.resume();
                whoStopStart.setLabel("Stop updating");
                running = true;
        }
return true;
```

Вот и все изменения, которые потребовалось внести в апплет, чтобы он удовлетворял перечисленным в начале условиям. На рис. 18-4 воспроизведен снимок с экрана конечной версии шахматного апплета.

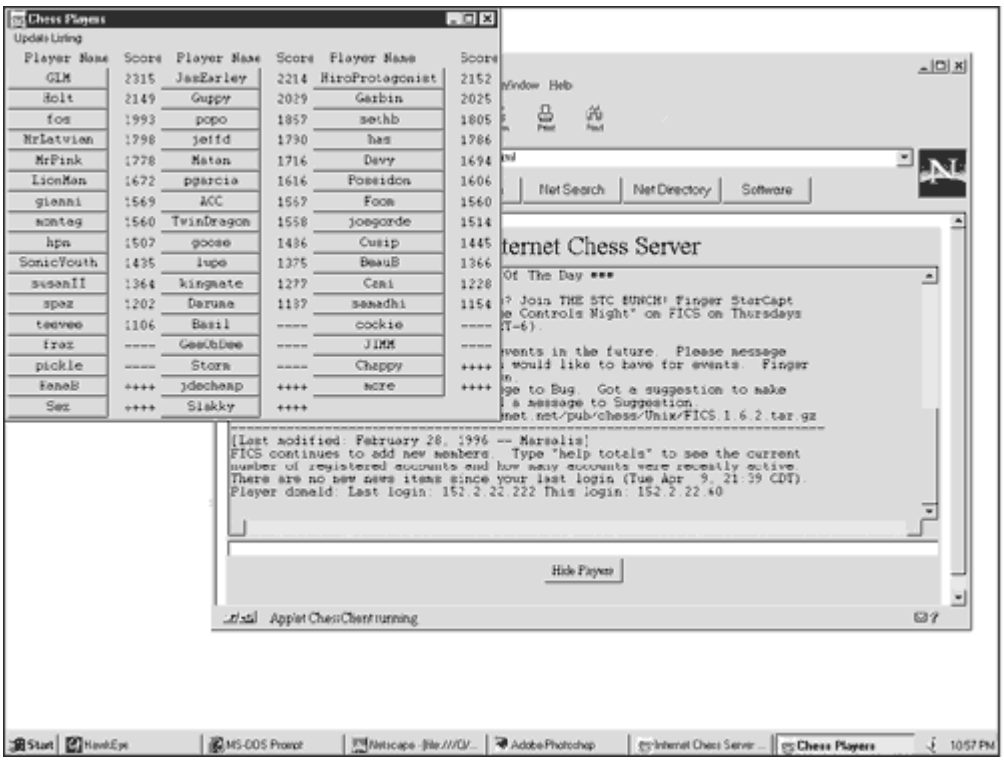

Рис. 18.4.

Дальнейшие усовершенствования

Рассмотренное приложение можно усовершенствовать в различных направлениях. Вот лишь некоторые из идей, которые могут прийти в голову:

- Возможность устанавливать верхнюю и нижнюю границы в списке игроков.
- Возможность изменять окно со списком игроков для отображения стандартного рейтинга вместо блиц-рейтинга (или отображение обоих).
- Возможность открывать окно для передачи текстовых сообщений партнеру.
- Добавление к окну шахматной доски поля, в котором бы отображался последний произведенный ход.
- Добавление к окну шахматной доски меню, дающего игроку допол-нительные возможности, например запрос на переход хода и засчиты-вание поражения сопернику, время игры которого истекло.
- Добавление окна, в котором был бы список текущих партий, позволя-ющий игроку выбрать и наблюдать за ходом интересующей игры так же, как и выбирать соперника из списка игроков.

Глава 19

Как написать свой собственный сервер: планировщик встреч

Что вы узнаете из этой главы

Здесь мы создадим Планировщик встреч (Meeting Scheduler) на Java. Мы рассмотрим подробно следующие темы:

- Проект сервера.
- Обработка особых ситуаций.
- Объектно-ориентированный проект.
- Работа с файлами.

Самый простой способ познакомиться с этим руководством - воспользоваться программой просмотра апплетов и запустить examples/ch19/ui.html с диска CD-ROM, прилагаемого к книге. Для этого необходимо иметь выход в сеть. Если у вас нет доступа к сети, вам нужно установить локальный сервер. О том, как это делается, подробно написано в разделах данной главы "Руководство пользователя" и "Как установить свой собственный сервер".

Контракт

Представьте, что некая маленькая компания попросила нас создать систему, которая будет планировать встречи своих сотрудников. Эти люди в основном работают в других компаниях, но не прочь перейти в данную компанию на полный рабочий день, как только она получит устойчивый доход. Все они живут в одном районе, за исключением одного сотрудника, переехавшего в другую страну. С помощью планировщика можно будет составить собственное расписание для каждого сотрудника и облегчить связь с зарубежным работником. Эту программу можно написать на любом языке, но она должна работать на самой разнообразной технике, потому что члены группы скорее всего будут тайком использовать оборудование тех компаний, где они сейчас работают. Большинство пользователей системы будут работать в несовместимых сетях, так что они не смогут воспользоваться существующими коммерческими планировщиками.

Задачу можно упростить с помощью использования Интернет и языка Java. Тогда нам останется только написать программу, которая сможет работать на разных платформах. Интернет даст клиенту доступ к международным коммуникациям за очень скромную плату. Компания предлагает нам приступить к делу немедленно - через две недели они хотят получить демонстрационную версию.

Свойства планировщика

Основная задача планировщика встреч состоит в том, чтобы помочь людям организовать свое время. Кроме того, он упростит планирование общих собраний, рассчитывая временной промежуток, в который все члены группы находятся на работе. Эта программа является небольшой реализацией популярного программного обеспечения по планированию, аналогичного MeetingMaker или планировщику, поставляемому с LotusNotes. Отличие состоит в том, что наш планировщик встреч будет работать через Интернет. Большинство планировщиков работают только на локальных сетях (LANs), таких как Novell. Использование Интернет позволит группе людей, находящихся по разные стороны государственных границ, пользоваться программой так же легко, как это может делать местная группа пользователей.

Основное свойство планировщика состоит в его способности добавлять и отменять встречи. Следовательно, должны быть созданы приложение для сервера и клиент, написанные на языке Java. В данном руководстве будут в первую очередь освещены вопросы создания сервера на Java.

Руководство пользователя

Прежде чем погрузиться в проектирование сервера, вы, наверное, хотели бы посмотреть, как он работает. Чтобы испробовать программу, вам нужно для начала посетить сервер планировщика. По умолчанию программа обращается к серверу на странице Online Companion. Выбрав в пункте меню File пункт login, вы войдете в систему.

Работать с программой очень просто. После запуска программы и вашего входа на экране появляется ваше расписание на текущий день. Вы можете просмотреть его, перейти к другим дням или заменить это расписание расписанием на любой другой день. Кнопка Add позволит вам добавить в расписание дополнительные пункты.

Основная функция планировщика - планировать встречи с другими людьми. Ниже приводится список сотрудников нашей предполагаемой компании. Первый вход осуществляется под именем Барт, но вы можете выбрать имя другого пользователя командой Select User (Выбор пользователя):

- Барт
- Лиза
- Мэгги
- Хоумер
- Мардж
- Мел
- Красти

Если вы воспользуетесь функцией add, вы тем самым запишете себя на данную встречу. Если вы хотите включить туда других людей, вы должны просто ввести их имена. Список заключается в скобки и должен содержать только имена из общего списка. Желаем вам получить удовольствие, планируя разные безумные встречи со своими виртуальными сотрудниками!

Как установить свой собственный сервер

Установить свой собственный сервер нетрудно, но тут нужно учесть несколько обстоятельств. Программа сервера является отдельным приложением. Вам нужно переписать программу с диска CD-ROM на свой жесткий диск. Чтобы установить сервер, нужно:

1. Создать каталог (в данном примере мы будем называть его "server").
2. Скопировать в этот каталог файлы examples/ch19 с диска CD-ROM.
3. Убедиться в том, что все файлы .dat и .idx доступны для чтения и записи.
4. Запустить сервер, введя команду "java server".

После этого на экране должно появиться сообщение о том, что сервер ждет соединения (waiting for connection). Если этого не произошло, возможно, что-то не в порядке с сокетами сервера. Данный сервер использует сокет 1666. Вы должны настроить свою систему таким образом, чтобы разрешить программе использовать этот сокет. Инструкции по тому, как это сделать, можно найти в разделе TCP/IP services в справочном руководстве к вашей системе.

Проект

Большинство сетевых приложений можно разбить на три основных модуля: пользовательский интерфейс, сетевой интерфейс и модуль, специфический для данного приложения. Такой способ организации позволяет повторно использовать данное приложение. В дальнейшем новые проекты могут использовать только отдельные части - например, сетевой интерфейс - и при этом не обязаны использовать или понимать остальные модули. Со временем вы разработаете группу объектов, с которыми все последующие проекты будут реализовываться легче легкого.

Для того чтобы эти объекты работали как единое целое, они должны иметь хорошо определенные связи друг с другом. Программа пользовательского интерфейса должна особенно хорошо работать именно с пользовательским интерфейсом и передавать все другие действия соответствующему компоненту программы. Допустим, у нас есть кнопка с пометкой login. Пользовательский интерфейс должен определить, что кнопка нажата, а затем передать действие сетевому модулю.

Модуль сетевого интерфейса

Связь через Интернет - это детские игрушки по сравнению с написанием собственных протоколов. При пользовании TCP/IP нам не нужно беспокоиться по поводу таких сложных коммуникационных проблем, как испорченные данные или выход пакета за установленные границы. Конечно, это не так просто, как очистить апельсин, но, по крайней мере, вы разработаете хороший проект, с которым сможете почти забыть об этих внутренних проблемах.

На самом низком уровне связь между двумя компьютерами представляет собой последовательность из нулей и единиц - обстоятельство, мало помогающее работе планировщика. Структурировав этот поток битов в известные типы данных, мы можем значительно облегчить себе работу по созданию планировщика и разбору данных. В Java есть специальный класс для этой работы. Как вы помните, в главе 13, "Работа с сетью на уровне сокетов и потоков", мы обсуждали потоки DataInput и DataOutput. Сейчас эти потоки нам очень пригодятся. Они позволяют пересылать типы данных Java в потоке. Сервер может поместить целое число в поток, и клиент легко сможет прочесть это число - без всякой конвертации и прочей мороки. Нужно только договориться о какой-нибудь структуре данных.

Мы выбрали для наших сообщений очень простую модель. Каждое сообщение будет содержать байт, описывающий тип сообщения, и затем - тело сообщения переменной длины. Каждому типу сообщения будет соответствовать подпрограмма, умеющая проводить разбор данных. Такая схема упростит задачу создания нескольких сообщений и, более того, значительно упростит этот разбор. Если вам когда-нибудь приходилось конвертировать поток битов в реальные данные, вы сможете оценить это удобство.

Этот метод минимизирует сетевой трафик. Самое меньшее, что мы можем послать, это байт. Если вас больше всего заботит скорость передачи данных, то такой протокол вам вряд ли подходит. Зато если вас больше всего волнует скорость кодирования и удобное сопровождение, вы победили. Наш метод не будет эффективно работать при посылке больших файлов данных, но для наших целей он вполне подходит.

Имея сетевую модель, мы можем начать проектировать клиент и сервер. В общем случае они будут содержать дополнительные методы. Когда на одной стороне возникает сообщение, другая сторона должна уметь выполнить его анализ. Мы можем сделать все красиво и создать для каждого сообщения файл-определитель, идентифицирующий каждый элемент данных и вызывающий подпрограмму, которая анализирует каждое сообщение. Это хороший способ, но есть и более простой путь: написать только один метод для создания и один для анализа каждого сообщения. Однако при нарушении синхронности у нас возникнут проблемы. Все, что производитель выкладывает на провод, должно быть получено потребителем.

По вопросу работы с передачей данных по сети у нас есть ряд специальных замечаний, касающихся сервера. В следующем разделе мы подробно обсудим создание Интернет-сервера.

Сервер

Если бы я попросил вас прямо сейчас написать простой сервер, вы, возможно, взяли бы сценарий на языке Perl, использующий CGI (Common Gateway Interface), что долгое время было стандартным ответом на такое предложение. Если бы затем я попросил вас сделать сервер достаточно быстрым, вы, возможно, написали бы программу на C, опять же с интерфейсом CGI.

CGI - это протокол, который применяется для выполнения динамических функций Web. CGI берет некие входные данные, обрабатывает их и создает документ MIME (Multimedia Internet Mail Extensions). Это мощное средство, но у него есть свои ограничения. CGI основан на диалоге с Web-броузером. Web-броузер понимает документы MIME и показывает их, но на этом все кончается. А что если вы хотите забрать данные, еще как-то их обработать и затем отобразить?

Если вы напишете сервер на языке Java, вы получите дополнительные возможности, а запуск клиента на Java расширит их почти до бесконечности. Клиент может обрабатывать входящие

данные и представлять их в удобном виде. При этом не только снимается часть нагрузки с сервера, но и предоставляется больше возможностей клиенту, в результате чего представление данных улучшается. Например, пользователь сможет менять шрифты, и система сможет подгонять отображение данных под размер монитора.

Использование Java для написания сервера - это вопрос личных пристрастий, здесь есть слабые и сильные стороны. Как всегда, для того чтобы выбрать самый подходящий язык, вы должны оценить, что вам нужно получить. Давайте рассмотрим некоторые вопросы, которые нужно обдумать при использовании Java для написания сервера.

Обеспечение безопасности

Написание программы сервера предполагает определенную ответственность, потому что сервер - это, возможно, самое слабое звено в вашей Интернет-броне. Плохо написанная программа сервера может привести к поломке машины или, что еще хуже, к потере данных. Большое количество информации по взлому Интернет, в частности информации о том, как найти лазейку, чтобы проникнуть в программу сервера, является общедоступной. Серверы пишутся для того, чтобы распространять информацию или решать задачи; хочется быть уверенным в том, что информация не является частной и что задача не является разрушительной.

Использование Java в качестве языка для написания сервера практически идеально с точки зрения обеспечения безопасности информации, ведь при создании Java учитывались проблемы секретности информации. Неправильно поставленный указатель не приведет к тому, что ваша система окажется открыта и доступна для атаки посторонних лиц. Важно помнить о том, что Java-приложение имеет меньше ограничений, чем апплет. В частности, оно может обратиться к диску, так что вы должны правильно задать, что разрешено серверу. Существует эмпирическое правило - разрешать серверу доступ только к тем файлам, которые ему нужны.

Обеспечение безопасности информации в Интернет - это настолько важная тема, что можно было бы написать по этому поводу целую книгу. На самом деле, несколько таких книг уже опубликовано. Тем, кто интересуется проблемой безопасности в Интернет, рекомендуется прочитать книгу "Internet Firewall", выпущенную издательством O'Reilly & Associates. Книга "Web-server book" (издательство Ventana) также содержит хороший раздел по этой теме; кроме того, подобные вопросы освещаются в изданиях, рассматривающих общие проблемы, связанные с Интернет-серверами. В программе, написанной на Java, вероятность возникновения лазеек для злоумышленников меньше, чем в программе, написанной на C, и, возможно, меньше, чем при использовании сценария на Perl, но ни один язык не может дать полной гарантии безопасности.

Вопросы скорости и памяти

По скорости выполнения программа на Java обычно сравнима со сценарием на языке Perl. Perl лучше обрабатывает текст, но при выполнении большинства операций Java ему не уступает. Скоро появятся модули Just-In-Time (JIT), конвертирующие байтовую программу на Java в машинные коды; с помощью этих модулей программа на Java будет работать быстрее. Тем не менее, скорость не должна быть решающим аргументом при выборе языка программирования для проекта; она может стать им только в тех случаях, когда время выполнения программы имеет принципиальное значение. Программа, написанная на Java, будет работать не намного медленнее, чем программа, написанная на другом языке.

Java - объектно-ориентированный язык, и в нем есть возможности сборки мусора. Эти два обстоятельства позволяют Java использовать больше памяти, чем другие языки для серверов, такие как Perl или C. Для объектно-ориентированных языков типично использовать больше памяти. Поскольку в языке Java есть сборка мусора, он может делать спорадическую очистку памяти. Система может не чистить память до тех пор, пока вам это не понадобится, не считаясь с тем, что это нужно другим программам. Это означает, что ваш сервер на Java будет держать в резерве больше памяти, чем ему нужно. Чаще всего приходится взвешивать приоритеты. Объектная ориентация языка Java означает, что программа на нем всегда будет использовать больше памяти, чем подобная программа на C. При сравнении Java и Perl приходится учитывать, что Perl версии 5 включает объектно-ориентированные расширения со всеми вытекающими проблемами. И для каждого проекта нужно заново взвешивать все за и против.

Проект сервера

Само по себе проектирование сервера достаточно просто. Когда связь через сокет установлена, создается новый поток для обработки запросов. Этот поток использует для ответов на запросы клиента объект Schedule. Вот и все, что мы имеем. Все это достаточно ясно и легко

программировать. Конкретные вопросы, связанные с реализацией этого алгоритма, мы рассмотрим в разделе "Реализация: модуль сетевого интерфейса" этой главы.

Клиент

Возможность использовать язык Java для написания клиента - это, безусловно, большая удача. Мучения при написании многочисленных программ CGI и необходимость передавать данные от одной программы к другой уходят в прошлое. Разработчики Web, использовавшие CGI, часто сталкивались с разнообразными трудностями. Даже если сервер является программой CGI, полезно использовать Java в качестве клиента. В главе 17, "Взаимодействие с CGI: Java-магазин", показано, как, используя Java, можно связаться с программой CGI.

На самом деле клиент - это просто поток, обрабатывающий приходящие из сети данные. Он действует как диспетчер для входящих пакетов. Клиент раскодирует пакет и вызывает соответствующую программу для работы с этим пакетом. И кроме того, он сам действует как поток, чтобы не возникала необходимость писать другую программу для передачи данных по сети.

Сообщения с сервера обрабатываются, как и любые другие события. Когда мы запрашиваем что-нибудь с сервера, он посылает данные сетевому модулю. Когда пакет раскодирован, с ним можно работать так же, как с событием пользователя. Таким образом, сеть становится простым средством передачи сообщений. Это решение хорошо применять для графических пользовательских интерфейсов, основанных на понятии события.

Прояснив вопросы, связанные с сетью, мы можем теперь обратиться к подпрограммам низкого уровня, которые составляют ядро нашей программы. После этого мы создадим пользовательский интерфейс, и на этом наш проект будет завершен.

Модуль, специфический для данного проекта

Основа планировщика - это структура данных, содержащая все планируемые встречи. И клиент и сервер должны иметь доступ к этой информации. Реализация этой структуры может различаться для разных сторон, но интерфейс должен быть единым. Именно здесь лучше всего воспользоваться механизмом интерфейса Java.

Мы создали интерфейс под названием schedule (расписание). Понятие schedule означает объект, содержащий данные о планируемых встречах данного человека, и подпрограммы, модифицирующие эти данные. Интерфейс позволяет нам определить методы, которые будут содержаться во всех объектах, реализующих schedule. Мы определим, как и что будет передаваться каждой подпрограмме. Непосредственно программированием мы займемся несколько позже, но на данной стадии работы проект планировщика уже готов.

Использование интерфейса позволяет нам изменять способы реализации расписания, а это очень важно для нашего приложения. На стороне сервера мы сохраним расписания на диске и разрешим хранить их постоянно. На стороне клиента мы будем хранить расписания в оперативной памяти по двум причинам: во-первых, Java не позволяет пользователю обращаться к локальным файлам, и во-вторых, хотелось бы, чтобы доступ к расписаниям был быстрым. Хранить расписания в памяти удобно клиенту, но представьте себе, что будет, если машина сервера начнет хранить в памяти расписание каждого сотрудника! Активный сервер может обслуживать одновременно тысячи людей, и такое количество расписаний очень быстро переполнит память машины. На рис. 19-2 изображена иерархия объектов для классов schedule.

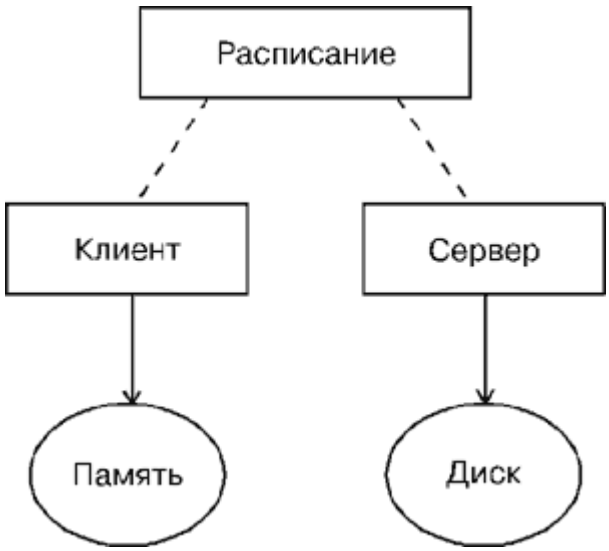

Рис. 19.2.

Эта иерархия очень проста, но она имеет некоторые характерные черты. Заметим, что клиент и сервер реализуют интерфейс schedule (реализация обозначена пунктирными линиями). В свою очередь, они используют различные внутренние структуры данных для сохранения данных. Впоследствии мы сможем создать другой тип расписания - например, распределенное расписание, в котором расписание каждого члена коллектива находится на его машине. С другой стороны, при нашей системе хранения на диске мы будем хранить каждое расписание на машине сервера. Каждый метод имеет свои за и против.

Независимо от того, как реализовано расписание, мы можем просто вставить его непосредственно в нашу программу - и все заработает! Интерфейс - это мощное средство проектирования и изящный путь для развития открытых систем. Когда стандарт разработан, кто угодно может создать собственную реализацию. В табл. 19-1 приведено описание всех методов schedule.

Таблица 19-1. Методы schedule

Метод	Описание
add(scheduleStruct)	Добавляет в расписание новую встречу.
del(ScheduleStruct)	Удаляет встречи из расписания.
find(Date)	Ищет встречи по определенной дате/времени.
getUser()	Возвращает хозяина расписания.

Рассмотрим программу, задающую этот интерфейс:

```
package ventana.scheduler;
import java.io.*;
import java.util.*;
import ventana.util.*;
import ventana.scheduler.*;
public interface schedule {
        // подпрограммы поддержки
        public void add(scheduleStruct newSchedule)
                throws DuplicateException, IOException;
                /*Описание:
                * Добавляет новую встречу в расписание
                *
                * Исключения:
                *Duplicate Exception - временной интервал недействителен
                *ServerException - контакт с сервером невозможен
                *IOException - необрабатываемая ошибка ввода/вывода
                */
        public void del(scheduleStruct del)
        throws NotFoundException, IOException;
                /* Описание:
```

```
                /*Удаляет встречу
                *
                * Исключения:
                *NotFoundException - встреча не существует
                *ServerException - контакт с сервером невозможен
                *IOException - необрабатываемая ошибка ввода/вывода
                */
        // подпрограммы поиска/восстановления
        public scheduleStruct find(Date date)
                throws NotFoundException, IOException;
                /* Описание:
                *Ищет scheduleStrust для определенной даты/времени
                *
                * Исключения:
                *NotFoundException - встреча не существует
                *IOException - необрабатываемая ошибка ввода/вывода
                */
        public scheduleStruct[] findRange(Date start, Date end)
                throws IOException;
                /* Описание:
                * Возвращает все пункты расписания, относящиеся
                *к определенной дате/времени
                *
                * Исключения:
                *IOException - необрабатываемая ошибка ввода/вывода
                */
        public String getUser();
                /* Описание:
                *Возвращает хозяина расписания
                */
}
```

Написав клиент и сервер, мы займемся реализацией этого интерфейса. Тогда мы будем решать, как реализовывать данные, хранящиеся в памяти и на диске. При создании интерфейса мы сосредоточимся на вопросах проектирования и оставим на потом вопросы, связанные с особенностями реализации. Человек, занимающийся реализацией программы, необязательно должен быть ее создателем. Представьте себе, что вы "важная шишка". Вы можете спроектировать систему, получить за нее деньги и переключиться на другие, более важные проекты. После этого придет некто, специализирующийся на написании программ, и займется реализацией системы.

Модуль пользовательского интерфейса

Если есть какое-нибудь средство для графической компоновки интерфейса, то создать пользовательский интерфейс будет нетрудно. В противном случае придется формировать проект вручную. Для компоновки интерфейса на Java нужно с помощью какого-нибудь менеджера размещения расположить отдельные части проекта в окне программы. Как правило, все, что вам нужно, можно выполнить с помощью имеющегося в Java набора менеджеров размещения. Их применение необходимо для того, чтобы программа, работающая в такой разнородной среде, как Интернет, правильно меняла размеры экранных элементов. Для данного руководства мы выбрали удобный менеджер PacketLayout, который свободно распространяется по сети его создателем Дэроном Мейером.

СОВЕТ Разработка пользовательского интерфейса займет у вас в два или в три раза больше времени, чем вы ожидаете. Много часов сэкономит вам использование GUI screen builder.

Работа планировщика начинается с появления основного окна, на котором нет ничего, кроме меню. В этом меню пользователь может выбрать основные функции, такие как вход, открытие расписания и конфигурацию программы. Для выбора остальных функций потребуется выход в подменю.

Все это очень просто. Простота экрана - это совместное достижение правильно построенной

программы и языка Java. Java не разрешает помещать меню на апплет, поэтому нам придется создать отдельный фрейм для размещения в нем меню. Такой метод постепенно приведет к тому, что ваши апплеты будут похожи на целые приложения, хотя обычно это не так уж плохо. Так, наш проект оказывается ближе к реальному приложению, чем к расширению Web-страницы - например, аниматору или доске объявлений (billboard).

В основном меню пользователь обычно открывает расписание (Schedule). В результате должно появиться нечто, похожее на обычный ежедневник. В нем нет симпатичного орнамента или фотографий ваших любимых уголков природы. Это просто функциональный экран, с которого можно переходить на следующие или предыдущие дни и добавлять новые пункты расписания. Хотя, если заказчики захотят, чтобы расписание выглядело наряднее, мы сможем добавить туда красивые картинки. Наш ежедневник приведен на рис. 19-3.

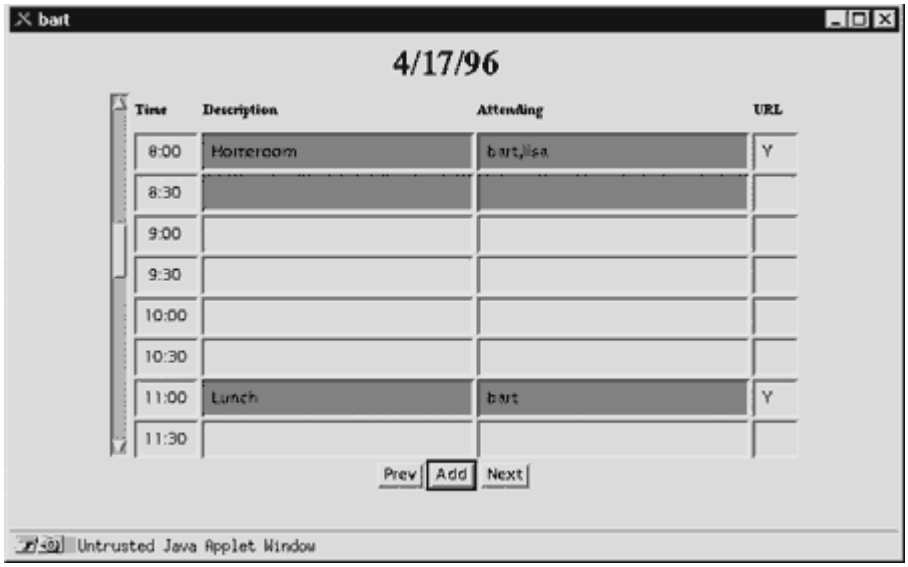

Рис. 19.3.

Остальные окна небольшие и содержат только по одной функции. Мы не приводим их здесь; вы можете сами их посмотреть, полистав приложение. Важно помнить, что все окна являются отдельными фреймами, а это палка о двух концах - дает больше свободы действий, но требует больше работы. Во время реализации проекта мы обратим ваше внимание на некоторые детали, которые важно учитывать при работе с фреймами.

Большая картина

Когда все компоненты будут собраны вместе, мы получим нечто большее, чем набор отдельных частей. Каждый модуль работает в своей области специализации и дает работу другим специалистам. Разработанные модули можно затем использовать в других проектах. Таким образом, мы можем уделить внимание особенностям проекта и меньше заниматься решением общих задач, таких как сетевые коммуникации или создание структур данных, и это одно из самых больших преимуществ использования объектной ориентации. Труд, затраченный на разработку проекта, полностью окупится при создании последующих проектов. На рис. 19-4 выделены связи между модулями.

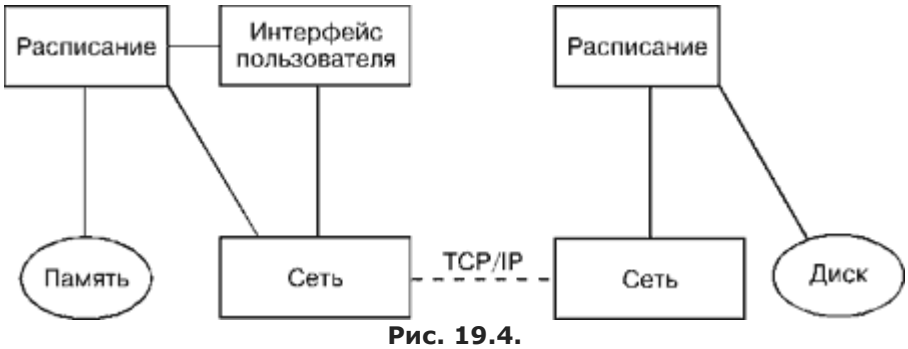

Рис. 19.4.

Спроектировав систему, мы можем заняться написанием программы. На этой стадии работы может возникнуть необходимость изменить проект (скорее всего такой необходимости не возникнет, но иногда такое случается). Поскольку теперь вы лучше научились предвидеть возможные трудности и приобрели опыт работы с Java и объектно-ориентированным проектом, вы сделаете меньше ошибок и ваш проект в целом будет аккуратнее. Необыкновенно приятно, когда кто-нибудь посмотрит на ваш проект и скажет: "Да, вот это вещь!"

Реализация

Создав надежный проект, пора сесть и решить, как реализовать каждый объект, какими структурами данных и алгоритмами пользоваться. Можно еще создать демонстрационную версию, показывающую основные элементы программы, хотя она не будет так эффективна, как конечный продукт. Использование объектно-ориентированной техники позволит нам создать несколько реализаций, а затем пользоваться той из них, которая подходит для данного проекта.

Достаточно обычный путь - написать программу, обращая мало внимания на скорость выполнения и память, а затем вернуться назад и улучшать алгоритм. В смысле возможностей продажи работающая демонстрационная версия почти так же хороша, как и конечный продукт. Мы должны быть уверены, что то, что продается, можно будет реализовать за разумное время. Вы наверняка найдете в программе много мест, нуждающихся в улучшениях, таких как быстрота выполнения и разнообразие функций. Красота объектно-ориентированного проекта состоит в том, что он может быть без большого труда расширен. Если нам не нравится, как что-то работает, мы можем взять новую функцию, которая нам нравится, и заменить ею старую.

Рассмотрим теперь, как реализовать планировщик. Начнем с обзора каталогов и затем обратимся к самой программе.

Обзор программы

В этом руководстве содержится большая программа. Чтобы облегчить работу с проектом, он был разбит на несколько отдельных каталогов. Каждый каталог содержит собственные файлы проекта (makefiles) и, как правило, сам по себе является полезным объектом. Использование пакетов существенно упрощает задачу разработки подручных библиотек. Мы взяли за образец правила образования имен Java и построили наши каталоги аналогичным образом.

Компилируется программа легко. Мы редактируем файл проекта, чтобы изменить местоположение файлов класса. Файл проекта хранит все свои файлы в каталоге ventana. Путь к файлам должен быть записан в CLASSPATH, чтобы имело смысл помещать каталоги туда, где находятся наши классы Java. Модифицировав файл, мы даем команду make. Программа make скомпилирует нашу программу.

Модуль сетевого интерфейса

Основное звено этого руководства - проектирование и реализация сервера. Для того чтобы написать сервер на Java, нужно хорошо разбираться в потоках и исключениях. Поскольку, как правило, нельзя допускать выход сервера из строя, мы должны создать как можно больше исключительных ситуаций. После регистрации исключения программа должна продолжить работу. Самым худшим вариантом будет потеря одной сессии, в то время как все остальные останутся незащищенными.

При написании сервера есть одно неприятное ограничение. Скоро ожидается выпуск базы данных API для Java, но к моменту написания книги эта база недоступна. Это означает, что наш сервер должен выполнять такие функции, как блокировка файла или записи. Мы разработали простой механизм блокировки файлов, хотя можно написать гораздо более изящную систему. В идеале сервер должен иметь доступ к базе данных с блокировкой записи. Мы пока не занимались вопросами возврата транзакции (transaction rollback). Частично транзакция может быть внесена в базу данных. Надеемся, что база данных API отчасти разрешит эти проблемы.

Словарик создателя сервера

Приведем некоторые термины, связанные с работой сервера.
Блокировка файла (file locking). Сделать файл доступным только для одного человека в каждый момент времени, чтобы предотвратить повреждение файла.
Блокировка записи (record locking). Сделать запись доступной только для одного человека в каждый момент времени; это лучше блокировки файла, потому что одновременно может совершаться больше транзакций.
Транзакция (transaction). Операция записи в файл.

Возврат транзакции (transaction rollback). Метод восстановления, позволяющий отменить некоторое количество транзакций. Обычно применяется при возникновении ошибок.

Фактически сервер выполняет очень простой цикл действий. Он слушает порт сокета, принимает связь, а затем запускает поток для обработки запроса. Это приложение, а не апплет, поэтому в нем есть метод main. Приведем программу для основного цикла сервера:

```
public class server {
        public static void main(String args[])
                throws IOException {
                ServerSocket session = new ServerSocket(1666,100);
                handleRequests handler;
                Hashtable users = new Hashtable();
                locks schedLocks = new locks();
                System.out.println("Waiting for connections");
                while(true) {
                        Socket socket = null;
                        try {
                                socket = session.accept();
                        }
                        catch (SocketException e) {
                                // обычно тайм-аут, повторите попытку
                        }
                        if (socket == null) {
                                continue;
                        }
                        System.out.println("Connection made");
                        // создаем отдельный поток для связи
                        handler = new handleRequests(socket,users,
                        schedLocks);
                        // установки для нереентерабельных систем
                        handler.setPriority(handler.getPriority() + 1);
                        handler.start();
                }
        }
}
```

Сервер использует класс под названием serverSocket. Этот класс слушает порт с помощью метода accept и ждет установления соединения. Мы выбрали порт 1666 и задали время ожидания 100 миллисекунд. Обратите внимание на блок try в методе accept. Если в течение заданного промежутка времени соединение не произошло, генерируется исключение socketException. В нашем случае мы будем продолжать совершать шаги цикла, пока соединение не установится.

После установления соединения мы создадим новый поток для обработки запросов. Мы создали класс handleRequest для обслуживания связей. Поскольку он является расширением класса Thread, его основная программа содержится в методе run. Этот метод будет действовать как диспетчер. Когда приходит пакет, первый байт используется для определения типа пакета. Затем вызывается соответствующий метод для выполнения разбора пакета. Вот и все - кроме анализа и обработки запросов, все готово. Класс handleRequest осуществит все операции по запросам клиента:

```
class handleRequests extends Thread {
        DataInputStream in = null;
        DataOutputStream out = null;
        Socket socket = null;
        String user=null;
        Hashtable users;
        locks schedLocks;
        handleRequests(Socket s, Hashtable users,
                locks schedLocks) throws IOException {
                socket = s;
                in = new DataInputStream(
                        new BufferedInputStream(socket.getInputStream()));
                out = new DataOutputStream(new BufferedOutputStream(
```

```
                socket.getOutputStream()));
            this.users = users;
            this.schedLocks = schedLocks;
        }
```

В конструкторе этого класса задается сокет и хеш-таблица и запирается объект класса. Давайте изучим каждый компонент и посмотрим, какую роль он играет в сервере. Когда связь установлена, система создает сокет между пользователем и сервером. Этот сокет представляет собой двухсторонний канал, которым можно пользоваться для общения с сервером. Где при этом находится пользователь, на соседней машине или в другой части света, неважно - связь работает одинаково. Сокет создает два потока: входной и выходной. Для начала создадим для входного и выходного потоков буфер и фильтр данных.

Не забывайте, что для достижения наших целей можно поместить на потоки различные фильтры, в данном случае - буферный фильтр и фильтр данных. Буферный фильтр используется для эффективного обращения к потокам. Символы, поступающие по каналу, временно хранятся в буфере. Это дает возможность более эффективно читать и записывать большие наборы данных. Буферный поток позволяет увеличивать объем передаваемых данных по мере увеличения скорости связи. Мы можем записать пакет данных размером 64 байта, но он будет записываться со скоростью один байт в единицу времени. Команда write сможет записать эти данные и продолжать работу, а не ждать, пока будет записан весь поток. Это сделает передачу данных более компактной и позволит программе работать ровнее.

Фильтр данных используется для того, чтобы посылать через поток простые типы, что даст возможность записывать такие элементы данных, как целые и строки, не проводя их разбор. Использование фильтра данных сделает нашу программу удобнее для чтения и записи.

Второй параметр конструктора - пользовательская хеш-таблица, которая следит за всеми пользователями, вошедшими в наш сервер. Эта информация нужна для того, чтобы известить всех пользователей об изменениях в расписании. Здесь применяется не вполне эффективный алгоритм, который сообщает всем пользователям о любом изменении в расписании. Было бы лучше сообщать каждому пользователю об изменениях только в его личном расписании, но при этом возникнет проблема слежения за индивидуальными расписаниями и потребуется дополнительное сообщение о том, когда каждый пользователь открыл или закрыл расписание. Для целей данного руководства нам достаточно воспользоваться более простой реализацией.

Последним параметром является объект schedLocks. Этот объект нужен для запирания файла. Нельзя допустить, чтобы два пользователя одновременно изменяли файл. Чтобы избежать этого, программа запирает файл перед использованием и отпирает после того, как работа с ним закончена. Хотя это предохраняет файлы, лучше было бы применять механизм записи-блокировки. Вопросы взаимной блокировки и повреждения данных обсуждаются в главе 11, "Многопотоковость".

Создав поток, мы вызываем метод start. После этого система вызывает метод потока run. Метод run нашего класса handleRequest очень прост и действует как диспетчер. Кроме того, он обрабатывает все возникающие исключения. Следующий фрагмент кода демонстрирует анализатор для наших сообщений:

```
public void run() {
        byte b;
        boolean cont=true;
        try {
                loginRequest();
                while(cont) {
                        b = in.readByte();
                        switch(b) {
                                case packetTypes.login :
                                loginParse(); break;
                                case packetTypes.logoff :
                                cont=false; break;
                                case packetTypes.reqSchedule :
                                reqSchedule(); break;
                                case packetTypes.addMeeting :
                                addMeeting(); break;
                                default : System.out.println("Unknown type");
                        }
                }
                out.close();
                in.close();
                socket.close();
```

```
        }
        catch (IOException ignore) {
                System.out.println("IO Exception, thread stopped");
                stop();
        }
        catch (Exception e) {
                System.out.println("Unknown error, thread stopped");
                e.printStackTrace();
                stop();
        }
        finally {
                users.remove(user);
                schedLocks.unlock(user);
        }
}
```

Давайте рассмотрим обработку исключения в этом блоке. Общая посылка состоит в том, что можно позволить разорваться некоему соединению, но мы хотим, чтобы сервер все равно продолжал работать. Проблема в том, что одному потоку нельзя разрешить уничтожать другие потоки. Главная наша задача - убрать все оставшиеся от этого потока запертые объекты и имя пользователя. Обратите внимание на последний блок, который это делает.

Поскольку мы не можем ни предвидеть все ошибки, которые могут возникнуть в программе, ни запрограммировать исправление каждой возможной ошибки, мы выбрали более общий подход. Каждая возникшая ошибка будет регистрироваться, и администратору будет разрешено ее исправить. Такие ошибки, как выход за пределы памяти или ошибка диска, не могут быть исправлены программным путем. В любом случае, мы постараемся поддержать систему в рабочем состоянии.

Основная часть работы сервера - обработка запросов. В то же время это - самая скучная часть программы, по которой мы быстро пройдемся, останавливаясь только на трудных или интересных местах. Приведенная ниже программа показывает, как сервер посылает сообщение клиенту:

```
public void loginRequest() throws IOException {
        System.out.println("Requesting login");
        out.writeByte(packetTypes.login);
        out.flush();
}
```

Метод loginRequest иллюстрирует важное свойство работы буфера с выходными данными: мы не знаем, в какой момент буфер освобождается. Воспользовавшись методом flush, мы можем записать выходные данные сразу же по освобождении буфера. Этот метод полезен для отладки и позволяет осуществлять выполнение программы более гладко. Некоторые буферные системы ждут, пока в очереди не наберется определенное количество байтов, и только после этого посылают сообщение, так что наше однобайтовое сообщение могло бы долго ждать, пока его отправят. После того, как сообщение отправлено, мы укажем системе послать его по каналу:

```
        public void loginParse() throws IOException {
                byte pakType;
                user = in.readLine();
                System.out.println("User " + user + " logged in");
                users.put(user, this);
        }
public void reqSchedule() throws IOException {
        String user;
        Date start, end;
        scheduleStruct result[];
        user = in.readLine();
        start = new Date(in.readLong());
        end = new Date(in.readLong());
        System.out.println("Schedule Request for " + user);
        System.out.println("Start: " + start.toString());
        System.out.println("End: " + end.toString());
        try {
                schedLocks.lock(user);
                serverSchedule schedule = new serverSchedule(user);
```

```
                result = schedule.findRange(start, end);
                out.writeByte(packetTypes.getSchedule);
                out.writeBytes(user + "\n");
                if (result == null) {
                        out.writeInt(0);
                        out.flush();
                        return;
                }
                out.writeInt(result.length);
                System.out.println("Recs to send: " + result.length);
                for(int i=0; i < result.length; i++) {
                        System.out.println("Meeting: " + result[i].desc);
                        out.writeLong(result[i].start.getTime());
                        out.writeLong(result[i].end.getTime());
                        out.writeBytes(result[i].desc + "\n");
                        out.writeBytes(result[i].descURL + "\n");
                        out.writeBytes(result[i].minutesURL + "\n");
                        out.writeBytes(result[i].attending + "\n");
                }
                out.flush();
                System.out.println("Schedule Sent");
        }
        finally {
                schedLocks.unlock(user);
        }
}
```

Метод redSchedule - самый большой поставщик данных на сервере. Зная временной промежуток и имя пользователя, он пошлет данные о всех встречах, назначенных на этот отрезок времени. Эта подпрограмма используется клиентом для того, чтобы заполнить локальный объект расписания. Данные считываются с помощью объекта serverSchedule и затем посылаются в формате сообщения.

Для того чтобы послать это сообщение, нужно знать длину поля, чтобы указать анализатору, сколько записей ему ждать. Если у какого-то пользователя очень плотное расписание, это сообщение может стать слишком длинным. Клиент должен быть спроектирован так, чтобы запрашивать данные на разумные отрезки времени - например, на один день или, может быть, одну неделю. У нас не задан максимальный промежуток - просто не очень хорошо посылать большое количество данных, которое может не понадобиться.

Следующая пара подпрограмм добавляет новые встречи. Это, возможно, самая трудная задача системы. Мы разработали хороший набор необходимых функций, хотя их может быть гораздо больше:

```
public void addNAK(String s) throws IOException {
                out.writeByte(packetTypes.addNAK);
                out.writeBytes("AddMeeting: " + s + "\n");
                out.flush();
        }
        public void addACK() throws IOException {
                out.writeByte(packetTypes.addACK);
                out.flush();
        }
        public void scheduleChange(String s, Date start)
                throws IOException {
                System.out.println("Schedule changed:" + s);
                out.writeByte(packetTypes.scheduleChange);
                out.writeBytes(s + "\n");
                out.writeLong(start.getTime());
                out.flush();
        }
```

AddNAK, addACK и scheduleChange - это подпрограммы оповещения. NAK (negative acknowledge) сообщает, что новая встреча не добавлена. Она принимает строку, которая выдается пользователю. Типичные ошибки - неправильная дата или время и попытки заполнить время человека, который уже занят. Сообщение ACK (acknowledge) говорит системе, что

добавление новой встречи прошло успешно.

Сообщение scheduleChange рассылается всем текущим пользователям при изменении чьего-то расписания. В этом случае клиент смотрит расписание данного человека и решает, надо ли это расписание перезагружать. Наличие текущей информации на стороне клиента поможет защитить пользователя от пересечений в расписании. Теперь рассмотрим программу добавления встречи:

```
public void addMeeting() throws IOException {
                String line;
                String dateLine;
                scheduleStruct ss = new scheduleStruct();
                dateLine = in.readLine();
                line = in.readLine();
                try {
                        ss.start = new Date(dateLine + " " + line);
                } catch ( IllegalArgumentException e) {
                        System.out.println("Illegal argument, aborting");
                        in.readLine();
                        in.readLine();
                        in.readLine();
                        in.readLine();
                        addNAK("Error converting start date");
                        return;
                }
                line = in.readLine();
                try {
                        ss.end = new Date(dateLine + " " + line);
                } catch (IllegalArgumentException e) {
                        System.out.println("Illegal argument, aborting");
                        in.readLine();
                        in.readLine();
                        in.readLine();
                        addNAK("Error converting end date");
                        return;
                }
                ss.desc = in.readLine();
                ss.attending = in.readLine();
                ss.descURL = in.readLine();
                // лексический анализ встречи, это текст в скобках
                int pos;
                String st,st2;
                handleRequests hr;
                Enumeration e;
        st = new String(ss.attending + ",");
        if (st == null) {
                addNAK("Attending field blank");
                return;
        }
        st = st.trim();
        while ((pos = st.indexOf(',')) != -1) {
                st2 = st.substring(0,pos);
                System.out.println("Attending: " + st2);
                st = st.substring(pos + 1);
                schedLocks.lock(st2);
                serverSchedule sched = new serverSchedule(st2);
                try {
                        System.out.println("Adding: " + ss);
                        sched.add(ss);
                } catch(DuplicateException bad) {
                        addNAK(st2 + " already scheduled");
                }
                schedLocks.unlock(st2);
                e = users.elements();
                while(e.hasMoreElements()) {
                        hr = (handleRequests) e.nextElement();
```

```
                        hr.scheduleChange(st2,ss.start);
                }
        }
        addACK();
}
```

Метод addMeeting можно разбить на три части: 1) разбор и верификация, 2) реальное добавление к базе данных и 3) оповещение. При каждом возникновении ошибки посылается пакет addNAK. Клиент должен указать пользователю, в чем ошибка. Затем они могут подогнать входные данные и снова послать пакет.

На этапе разбор и верификации нужно убедиться в правильности дат и времени для встреч. Для разбора применяется конструктор Dates. Конструктор вызывает Date.parse, который придает смысл этой строке. Dates поддерживает хост с форматами дат, но при этом обладает необычным свойством: нумерация месяцев начинается с нуля, а не с единицы. Так что если дата записана как 07/27/1992, это означает 27 августа 1992 года, а не 27 июля. На стороне клиента мы изменили дату так, чтобы посланная дата была правильной, - то есть уменьшили номер месяца на единицу. Если вам интересно, как работает класс Date, посмотрите документацию или почитайте исходный текст программы для класса Java.util.Date.

С помощью объекта schedule легко добавлять данные в базу данных. Если в определенное время пользователь уже занят, мы получим исключение duplicateException. К примеру, если назначается встреча четырех человек, могут возникнуть неприятности. Обратной транзакции здесь не существует, поэтому если мы планируем встречу четырех человек и один из них в это время уже занят, то в расписание этого человека встреча не записывается. Мораль проста: убедитесь в том, что все эти люди свободны и могут прийти на встречу. Поскольку такое бывает редко, возможно, вы захотите создать функцию, которая отыскивает открытые временные интервалы для группы людей и затем выбирает время из полученного списка. Конечно, хотя такой метод работает, подпрограмму addMeeting можно было бы улучшить.

Последний шаг в процедуре добавления встречи состоит в оповещении всех пользователей об изменениях в расписании какого-то сотрудника. Пользователи могут запросить обновленное расписание и посмотреть его. Каждый человек может иметь несколько расписаний, так что мы должны оповестить всех об изменениях. Кроме того, мы можем усовершенствовать наш проект, встроив в него функцию слежения за свободным временем каждого члена группы. Нужно очень тщательно следить за тем, чтобы этот список соответствовал действительности.

Таково краткое описание сервера. Программа получилась очень длинная, но большую ее часть занимают подпрограммы анализатора. Особое внимание уделяется обработке исключительных ситуаций и предохранению сервера от сбоев. Такие вопросы, как запирание файлов и обратная транзакция, рассматриваются, но в общем случае не реализуются до конца. Если вы хотите создать надежный сервер, вам придется их изучить. Существенно улучшит сервер использование системы базы данных для работы serverSchedule.

Модуль, специфический для данного проекта

И клиент и сервер пользуются интерфейсом schedule таким способом, который больше всего подходит для их в какой-то мере пересекающихся интересов. У клиента выполняется определенная защита данных - например, данные не хранятся на диске. Это означает, что информацию по расписаниям мы должны запрашивать по сети. Получив такую информацию, мы можем сохранить ее в памяти для последующего использования.

Вместо того чтобы запрашивать информацию маленькими порциями, более эффективно запрашивать ее несколькими большими партиями. Пока пользователь смотрит на экран, мы можем послать информацию. Нам понадобится место для хранения этих данных, и именно тут нам пригодится clientSchedule.

Клиент Schedule

ClientSchedule реализуется с помощью хеш-таблицы, позволяющей быстро и компактно сохранять данные. Хеш-таблица будет содержать по одному элементу для каждой встречи, которые будут проиндексированы по времени начала встреч. Можно было бы создать массив, проиндексированный по времени начала встреч, но он будет слишком большим. Рассмотрим программу clientSchedule:

```
public class clientSchedule implements schedule {
        Hashtable memoryStore = new Hashtable();
```

Класс clientSchedule реализует интерфейс schedule. Этот класс даст нам программу для описанного предварительно интерфейса. Удобство использования интерфейсов состоит в том, что если вам не нравится данная реализация, вы можете написать свою собственную. Вы даже можете взять для использования некоторые фрагменты программы и заменить те ее части, которые вам не нравятся:

```
public void add(scheduleStruct newSchedule)
        throws DuplicateException, IOException {
        if (memoryStore.put(newSchedule.start, newSchedule)
                != null) {
                throw new DuplicateException();
        }
}
```

Метод add используется для добавления новых встреч. Относительно этой программы у нас есть два важных замечания. Одно касается того, каким образом помещать новые пункты в хеш-таблицу. Мы должны иметь данные и ключ (в данном случае это дата начала встречи), являющиеся объектами. Хеш-таблицы только хранят объект по значению ключа. Таким образом, если мы попытаемся записать две встречи на одно и то же время, мы получим ошибку. Такая система работает, поскольку большинство людей не посещают две встречи одновременно.

Второе важное замечание касается метода выдачи исключений. Как вы помните, мы должны описать все исключения, которые могут возникнуть в результате работы метода. Метод add, по-видимому, может вызвать только одно исключение - DuplicateException. IOException генерируется методом put хеш-таблицы. Поскольку мы здесь не занимаемся ошибками, мы передаем их вызывающему методу. Маловероятно, что мы получим IOException при работе со структурой данных, находящейся в памяти, но тем не менее это возможно. Если такое случится, предоставим программисту решать эту задачу.

DuplicateException генерируется вместо возвращения кода ошибки. В данном случае это не очень важно, но впоследствии мы будем использовать возвращенное значение для получения необходимой информации. Тогда мы захотим не менять возвращенное значение. Возвращенное значение выполняет две функции: возвращает некое число и возвращает -1 для серьезных ошибок. Исключения позволяют легко обойти эту проблему, как это сделано в следующем фрагменте кода:

```
public void del(scheduleStruct del)
        throws NotFoundException {
        if (memoryStore.remove(del.start) == null)
                throw new NotFoundException();
}
public scheduleStruct find(Date date)
        throws NotFoundException {
        scheduleStruct ss;
        ss = (scheduleStruct) memoryStore.get(date);
        if (ss == null) {
                throw new NotFoundException();
        }
```

Методы delete и find просты и практически не нуждаются в пояснениях. Для возвращения информации об ошибках снова используются исключения. Оба метода хеш-таблицы (get и remove) возвращают по запрошенной информации ссылочное значение. Если значение ключа не найдено, возвращается ноль. В этом случае будет сгенерировано NotFoundException.

Последний метод - findRange. Тут мы воспользуемся нумерацией, которая поможет нам вернуть значения данных. Метод findRange получает временной промежуток и возвращает все встречи за указанный период. Поскольку хеш-таблица не содержит данные по временным промежуткам, нам придется производить поиск последовательно. Зато хеш-таблица реализует интерфейс нумерации. Это позволяет последовательно просматривать хеш-таблицу.

Мы будем просматривать таблицу дважды. При первом просмотре мы определим число встреч, которые будут возвращены, а при втором просмотре заполним результирующий массив. Мы вынуждены так делать потому, что размер, по крайней мере, одного измерения массива должен быть определен во время его создания. Заметьте, что мы вызываем элементы метода хеш-таблицы дважды. Каждый раз после просмотра нумерации она уничтожается. При каждом новом обращении нужно заново создавать нумерацию. Ниже приводится программа для метода findRange:

```
public scheduleStruct[] findRange(Date start, Date end)
        throws IOException {
        Enumeration list = memoryStore.elements();
        scheduleStruct ss;
        int cnt=0;
        // считаем число элементов
        while (list.hasMoreElements()) {
                ss = (scheduleStruct) list.nextElement();
                if (start.getTime() <= ss.start.getTime() &&
                        end.getTime() >= ss.end.getTime()) {
                                cnt++;
                        }
                }
                if (cnt < 1) return null;
                scheduleStruct result[] = new scheduleStruct[cnt];
                // нумерация уничтожена, нумеруем заново
                list = memoryStore.elements();
                // создаем результирующий массив
                cnt=0;
                while (list.hasMoreElements()) {
                        ss = (scheduleStruct) list.nextElement();
                        if (start.getTime() <= ss.start.getTime() &&
                                end.getTime() >= ss.end.getTime()) {
                                result[cnt++] = ss;
                        }
                }
                return result;
        }
}
```

Вот и все, что относится к clientSchedule. Воспользовавшись хеш-таблицей, мы передали большую часть нашей работы уже созданному и отлаженному классу. Такое повторное использование проделанной работы позволяет нам писать программу быстрее и с меньшим количеством ошибок.

Сервер Schedule

Сервер schedule несколько сложнее, потому что мы будем иметь дело со структурой данных, хранящейся на диске. Для того чтобы работать с базой данных внушительного размера, реализация сервера должна быть максимально эффективной. Мы будем хранить данные в линейном файле, но для выполнения более быстрого поиска использовать индексный файл. Если вам интересно, как работает индексный файл, посмотрите schedule/IndexFile.java. Программа для него достаточно проста, но слишком длинна для того, чтобы приводить ее здесь. Вот программа для serverSchedule:

```
public class serverSchedule implements schedule {
        IndexFile index;
        RandomAccessFile dataFile;
```

Мы снова реализуем интерфейс schedule. На этот раз мы используем для сохранения данных два файла. Первый - индексный файл, который индексируется по начальной дате и хранит местоположение данных в файле данных. Файл данных хранит записи в формате переменной длины. Оба файла связаны с именем пользователя, причем индексный файл имеет расширение .idx, а файл данных - .dat.

```
public serverSchedule(String user)
        throws IOException {
        this.user = user;
        index = new IndexFile(user + ".idx","rw",Size,Size);
        dataFile = new RandomAccessFile(user + ".dat","rw");
}
```

Конструктор для этого класса содержит строку, идентифицирующую пользователя. Эта строка используется для того, чтобы создать имена индексного файла и файла данных. Заметим, что эта подпрограмма возвращает IOException. Это может случиться, если сервер не имел разрешения записывать в файлы - вы должны найти эту ошибку в вашей программе. Для хранения данных мы используем RandomAccessFile. Файл открывается на чтение и запись (с кодом доступа "rw") под именем, составленным из идентификатора пользователя и расширения .dat. Так, для пользователя с именем maggie файл данных будет называться maggie.dat. Следующий раздел программы добавляет встречу в расписание:

```
public void add(scheduleStruct newSchedule)
        throws DuplicateException, IOException {
        String key;
        String data;
        Long conv;
        long pos;
        dataFile.seek(dataFile.length());
        pos = dataFile.getFilePointer();
        dataFile.writeLong(newSchedule.start.getTime());
        dataFile.writeLong(newSchedule.end.getTime());
        dataFile.writeBytes(newSchedule.desc + "\n");
        dataFile.writeBytes(newSchedule.descURL + "\n");
        dataFile.writeBytes(newSchedule.minutesURL + "\n");
        dataFile.writeBytes(newSchedule.attending + "\n");
        conv = new Long(newSchedule.start.getTime());
        key = conv.toString();
        key = StringUtil.padStringLeft(key,' ',Size);
        conv = new Long(pos);
        data = conv.toString();
        data = StringUtil.padStringLeft(data,' ',Size);
        index.addRecord(key,data);
}
```

Метод add должен добавлять элементы к файлу данных и к индексному файлу. RandomAccessFile находится над фильтром DataOutput и DataInput, так что мы можем читать и писать типы Java непосредственно. Было бы хорошо реально написать объект, но, увы, эта возможность не предусмотрена. Еще одно важное замечание: мы пользовались методом writeBytes. Он записывает данные как 8-битовые байты. Если бы мы пользовались методом writeChars, мы имели бы 16 битов на символ. Метод writeUTF запишет 16-битовые символы Unicode. Если вы собираетесь использовать метод readLine, вам нужно пользоваться методом writeBytes. Соответствующего метода чтения, который бы читал выходные данные writeChars, не существует.

Заметим также, что после каждой строки стоит "\n". В дальнейшем мы будем пользоваться методом readLine - он воспринимает возврат каретки как символ конца строки. Это позволяет задавать строки переменной длины, что экономит немного места и дает возможность не проводить разбор строк.

Индексный файл состоит из двух элементов: значения ключа и значения данных. Оба элемента имеют фиксированную длину. Индексный элемент - это начальная дата. Мы сохранили ее как значение типа long. Оно сохранено как строка, чтобы индексный файл легче было читать. Данные сохранены на смещенной позиции в файле данных и хранятся в формате строки. Если мы хотим уменьшить индексный файл, мы можем сохранить эти значения не как строки, а как длинные слова:

```
public void del(scheduleStruct del)
        throws NotFoundException, IOException {
        Long conv;
        String key;
        String data;
        long pos;
        conv = new Long(del.start.getTime());
        key = conv.toString();
        key = StringUtil.padStringLeft(key,' ',Size);
        data = index.findRecord(key);
        conv = new Long(data.trim());
        pos = conv.longValue();
```

```
        // теперь можно удалить данные на pos,
        // но мы не будем этого делать
        index.delRecord(key);
}
```

Метод delete уничтожает только элемент индексного файла. Если бы мы уничтожили элемент файла данных, у нас образовалась бы дыра в файле. Чтобы ее устранить, нам пришлось бы переместить все данные в файле к началу и заменить все соответствующие элементы индексного файла. Операция такого типа является обычной для программ сжатия и, как правило, выполняется редко. Дисковое пространство достаточно дешево, так что мы можем позволить нашей программе выполняться несколько быстрее и только периодически выполнять операции с базой данных. Приведенный ниже фрагмент кода показывает, как был выполнен метод find для нашего расписания, хранящегося на диске:

```
public scheduleStruct find(Date date)
        throws NotFoundException, IOException {
        scheduleStruct result;
        long pos;
        Long conv;
        String key;
        String data;
        conv = new Long(date.getTime());
        key = conv.toString();
        key = StringUtil.padStringLeft(key,' ',Size);
        data = index.findRecord(key);
        conv = new Long(data.trim());
        pos = conv.longValue();
        return findPos(pos);
}
public scheduleStruct findPos(long pos)
        throws IOException {
        scheduleStruct result = new scheduleStruct();
        dataFile.seek(pos);
        result.start = new Date(dataFile.readLong());
        result.end = new Date(dataFile.readLong());
        result.desc = new String(dataFile.readLine());
        result.descURL = new String(dataFile.readLine());
        result.minutesURL = new String(dataFile.readLine());
        result.attending = new String(dataFile.readLine());
        return result;
}
```

Этот метод состоит из двух подпрограмм. Первая, определенная интерфейсом schedule, просматривает индексный файл и находит местоположение необходимого элемента из файла данных. Вторая подпрограмма - findPos - используется для восстановления данных из файла данных. Здесь применяется метод readLine. Мы ищем соответствующую позицию и читаем каждое поле из файла данных. Затем эти данные возвращаются вызывающему методу.

В результате мы подходим к последней, самой трудной подпрограмме - serversSchedule. Чтобы она быстрее работала, подпрограмма findRange использует отсортированный индексный файл. Мы производим два двоичных поиска в индексном файле, чтобы найти начальный и конечный элементы. Затем мы просматриваем каждый элемент и находим его данные в файле данных. В случае большой базы данных и разумного диапазона данных такая процедура выполняется гораздо быстрее, чем если просматривать всю базу данных. Теперь, когда у нас есть подпрограмма для нахождения определенного расписания, нам нужна подпрограмма, возвращающая диапазон расписаний:

```
public scheduleStruct[] findRange(Date start, Date end)
        throws IOException {
        searchStruct resultStart, resultEnd;
        Long conv;
        String key,data;
        int num;
        conv = new Long(start.getTime());
        key = conv.toString();
```

```
        key = StringUtil.padStringLeft(key,' ',Size);
        resultStart = index.binarySearch(key);
        conv = new Long(end.getTime());
        key = conv.toString();
        key = StringUtil.padStringLeft(key,' ',Size);
        resultEnd = index.binarySearch(key);
        num = (int) (resultEnd.pos - resultStart.pos) /
                              index.getRecordSize();
        if (num < 1) return null;
        scheduleStruct result[] = new scheduleStruct[num];
        int cnt=0;
        for(long i=resultStart.pos; i < resultEnd.pos;) {
                data = index.findPos(i);
                // Data - это индекс в файле .dat
                        conv = new Long(data.trim());
                        result[cnt++] = findPos(conv.longValue());
                        i += index.getRecordSize();
                }
                return result;
        }
}
```

Таким образом осуществляется реализация интерфейса schedule. Каждая реализация решает определенные задачи на обеих сторонах. На машине клиента не хранятся файлы, поэтому ему нужна резервная память. Сервер должен работать быстро и оперировать большими базами данных. При наличии общего интерфейса мы можем разработать классы, выполняющие ту же самую функцию, но совершенно по-разному.

Модуль пользовательского интерфейса

Мы представим вам полную программу пользовательского интерфейса. Нужно сказать, что пока этот процесс более громоздкий, чем он будет впоследствии, когда будут лучше разработаны менеджеры размещения. В данном разделе мы покажем только основные моменты используемого процесса и сконцентрируем внимание на применении фреймов Java.

Апплет Java обычно находится на той Web-странице, на которой он запускается. Это удобно для апплета, рассчитанного на одно окно, но что если мы хотим, чтобы было открыто несколько окон? Кроме того, в окнах апплета не может быть меню. Поэтому, если мы хотим использовать меню, нам придется воспользоваться фреймами.

Первый экран приложения, составляющего расписание, - это экран меню. Он содержит строку меню, которая работает как диспетчер для последующих операций. Пользователь может войти, открыть и закрыть расписания и запустить само приложение. Оно вызовет множество подпрограмм, которые будут выполнять его работу:

```
class MenuFrame extends Frame {
        MenuBar mb = new MenuBar();
public MenuFrame(String s, ui applet) {
        super(s);
        owner = applet;
}
public void createMainMenu() {
        setMenuBar(mb);
}
public void init() {
        createMainMenu();
}
```

Класс MenuFrame содержит дополнительные функции по сравнению с классом Frame. Мы хотели бы рассмотреть фреймы и апплеты под общим углом зрения. Это означает, что мы можем взять несколько конвенций из апплетов и применить их к фрейму.Чтобы это сделать, мы сделали конвенцию, в которой создан метод init. Его функции аналогичны функциям апплета: задать переменную и пространство окна. Наш метод init вызывает createMainMenu, который задает систему меню и выглядит следующим образом:

```
public void createMainMenu() {
                setMenuBar(mb);
                Menu m = new Menu("File");
                m.add(new MenuItem("Login"));
                m.add(new MenuItem("Open Schedule"));
                m.add(new MenuItem("-"));
                m.add(new MenuItem("Quit"));
                mb.add(m);
                m = new Menu("Setup");
                m.add(new MenuItem("User Name"));
                m.add(new MenuItem("Server Name"));
                mb.add(m);
        }
```

После появления меню мы должны знать, когда пользователь выберет пункт меню. Это можно сделать, переопределив метод handleEvent. Каждый раз, когда в данном фрейме возникает событие, вызывается метод handleEvent. Нас касаются только два события.

Первое - WINDOW_DESTROY. Оно возникает, когда пользователь выбирает функцию destroy (уничтожить) в диспетчере окон (window manager). Реальный механизм работы системы зависит от конкретной версии диспетчера окон, но обычно он разрешает пользователю уничтожить окно. Когда это произойдет, нам остается красиво удалиться. Для очистки фрейма вызывается метод освобождения памяти, который освободит все использовавшиеся системные ресурсы и выйдет из системы. Нам еще раз понадобится этот механизм для выхода из приложения.

Другое важное событие - ACTION_EVENT, ассоциирующееся с меню. Для выяснения того, что на MenuItem случилось это событие, воспользуемся оператором instanceof. Поскольку у нас только одно меню, понятно, что событие произошло именно там. Чтобы выяснить название кнопки меню, мы можем посмотреть evt.arg. Потом мы вызовем подпрограмму для работы с этой кнопкой.

При обработке события во фреймах важно помнить, что они не порождают цепочку событий. Каждый фрейм является независимым объектом и не имеет реального родителя. Если вы думаете, что событие Unhandled будет передано новому окну (например, апплету), то вы ошибаетесь. Вам придется обрабатывать события на месте или в явном виде выдавать событие апплету:

```
public boolean handleEvent(Event evt) {
        if (evt.id == Event.WINDOW_MOVED) {
        }
        else if (evt.id == Event.WINDOW_DESTROY) {
                dispose();
        }
        else if (evt.id == Event.ACTION_EVENT) {
                if (evt.target instanceof MenuItem) {
                        String st = (String) evt.arg;
                        if (st.equals("Quit")) {
                                quit();
                                return true;
                        }
                        else if (st.equals("Login")) {
                                login();
                                return true;
                        }
                        else if (st.equals("Open Schedule")) {
                                open();
                                return true;
                        }
                        else if (st.equals("Server Name")) {
                                snFrame sname = new snFrame("Change
Server",owner);
                                sname.init();
                                sname.show();
                                return true;
                        }
                        else if (st.equals("User Name")) {
                                unFrame uname = new unFrame("User Name",owner);
```

```
                                uname.init();
                                uname.show();
                                return true;
                        }
                        else System.out.println("Menu: " + (String) evt.arg);
                        return true;
                }
        }
        return false;
        }
}
```

Класс acFrame используется для того, чтобы добавить новые встречи в систему. Приведенная программа демонстрирует использование нестандартного менеджера размещения. Мы пользуемся пакетом менеджера TK. Если вам не нравится, как работает менеджер размещения в Java, вы можете написать свой собственный. Рассмотрим для примера ventana/awt/PackerLayout.java:

```
class acFrame extends Frame {
        ui owner;
        Panel datePanel = new Panel();
        Panel timePanel = new Panel();
        Label mdLabel = new Label("Meeting Date");
        TextField mdField = new TextField(7);
        Label tsLabel = new Label("Time Start");
        TextField tsField = new TextField(4);
        Label endLabel = new Label("End");
        TextField endField = new TextField(4);
        Panel dataPanel = new Panel();
        Panel labPanel = new Panel();
        Panel fieldPanel = new Panel();
        Label descLabel = new Label("Description");
        TextField descField = new TextField(40);
        Label attendLabel = new Label("Attending");
        TextField attendField = new TextField(40);
        Label urlLabel = new Label("URL");
        TextField urlField = new TextField(40);
        Panel buttonPanel = new Panel();
        Button schedButton = new Button("Schedule");
        Button cancelButton = new Button("Cancel");
        public acFrame(String s, String user, ui owner) {
                super(s);
                this.owner = owner;
        }
        public void init() {
                setLayout(new PackerLayout());
                datePanel.setLayout(new PackerLayout());
                timePanel.setLayout(new PackerLayout());
                datePanel.add("mdLabel;side=left",mdLabel);
                datePanel.add("mdField;side=left",mdField);
                timePanel.add("tsLabel;side=left",tsLabel);
                timePanel.add("tsField;side=left",tsField);
                timePanel.add("endLabel;side=left",endLabel);
                timePanel.add("endField;side=left",endField);
                dataPanel.setLayout(new PackerLayout());
                labPanel.setLayout(new PackerLayout());
                fieldPanel.setLayout(new PackerLayout());
                dataPanel.add("labPanel;side=left",labPanel);
                dataPanel.add("fieldPanel;side=left",fieldPanel);
                labPanel.add("descLabel;anchor=w;ipady=3",descLabel);
                labPanel.add("attendLabel;anchor=w;ipady=3",attendLabel);
                labPanel.add("urlLabel;anchor=w;ipady=3",urlLabel);
                fieldPanel.add("descField",descField);
                fieldPanel.add("attendField",attendField);
```

```
                fieldPanel.add("urlField",urlField);
                buttonPanel.setLayout(new PackerLayout());
                buttonPanel.add("schedButton;side=left;padx=5",schedButton);
                buttonPanel.add("cancelButton;side=left",cancelButton);
                add("datePanel;anchor=w;pady=5",datePanel);
                add("timePanel;anchor=w;pady=5",timePanel);
                add("dataPanel;anchor=w;pady=5",dataPanel);
                add("buttonPanel",buttonPanel);
        }
}
```

Создание даже такой простой страницы - занятие достаточно нудное. С использованием PackerLayout это сделать проще, хотя программирование расположения элементов экрана не может быть увлекательным занятием. Новая интегрированная среда разработки (New Integrated Development System, IDEs) вскоре позволит перемещать компоненты в пределах экрана. Тогда будет создана программа AWT. Вам нужно будет только обрабатывать события и склеивать вместе программы.

Возможные улучшения

Мы создали только основу апплета планировщика встреч. Чтобы добавить в него все свойства, которые мы захотим, может потребоваться несколько недель. Окончательный планировщик может много чего уметь, и программа его будет очень длинной. Если вы хотели бы расширить возможности программы, вы можете обдумать следующие возможности:

- Добавление общего вида еженедельника/ежемесячника.
- Разрешение пользователю или компании задавать дни, когда пользователь занят, или выходные.
- Указание группы людей, для которой компьютер будет подбирать подходящий временной интервал определенной длины.
- Учет проблемы безопасности информации, например определение людей, которым разрешено составлять расписание для данного пользователя.
- Посылка некоторых сообщений по электронной почте:
- Извещение пользователя, когда его расписание изменилось.
- Предупреждение о важных встречах или днях (скажем, годовщинах свадеб).
- Разрешение планирования использования ресурсов - таких, как залы заседаний и аудио-, видеоаппаратура.

Приложение A

О странице Online Companion

Знание - сила! Web-страница Online Companion соединит вас с лучшими информационными источниками в Интернет, посвященными программированию на Java.

Online Companion доступна по адресу http://www.vmedia.com/java.html. Некоторые из расположенных там ссылок указывают на другие полезные Web-страницы, архивы постоянно обновляющегося программного обеспечения и информацию о публикуемых издательством Ventana книгах.

Online Companion соединит вас с книжным магазином, где вы найдете массу интересных предложений и новой продукции. Библиотека Ventana's Online Library позволит заказать интересующие вас книги прямо по сети.

Страница Online Companion постоянно развивается, на ней появляется все больше информации, указателей, справочников и разделов книг, опубликованных издательством.

Приложение B

Диск CD-ROM

В состав CD-ROM диска, прилагаемого к книге, входят коллекция апплетов вместе с примерами исходных текстов, рассматриваемых в книге, и несколько полезных программ-утилит.

Для установки программ с диска вы должны выполнить следующие действия:

- *Для Windows 95/NT*. Дважды щелкните по пиктограмме CD-ROM и затем дважды щелкните по появившейся пиктограмме viewer.exe из окна Проводника или File Manager.
- *Для Macintosh.* Дважды щелкните по пиктограмме CD-ROM и затем по пиктограмме Viewer.

Перед вами появится экран, предлагающий выбор. Пункты меню следующие: выйти из программы просмотра, получить подсказку, просмотреть исходные коды примеров, получить информацию об издательстве или о выпускающихся книгах.

Чтобы просмотреть примеры апплетов, необходимо указать на броузер, способный выполнять апплеты. Затем нужно нажать кнопку Launch.

СОВЕТ Для того чтобы вернуться в программу просмотра после того, как был вызван броузер, необходимо еще раз дважды щелкнуть мышью по viewer.exe. Если вы просматриваете текст апплета в текстовом редакторе, для возврата в главное меню CD-ROM необходимо нажать кнопку Main после закрытия текстового редактора.

Для того чтобы использовать программы, расположенные на диске, их необходимо скопировать на жесткий диск.

Примеры исходных текстов апплетов и сами апплеты находятся на диске в каталоге Applets. Там же находится файл index.html, просматривая который при помощи броузера можно на практике познакомиться с работой того или иного апплета.