OSBORNE

// Изучите секреты, которые раскрывают вам гуру языка программирования Java

// Освойте грандиозные возможности языка Java и сопровождающих его библиотек

// Создайте необходимые во многих случаях приложения, такие как интерпретатор языка, Web-червь, диспетчер загрузки, синтаксический анализатор выражений, финансовые аплеты и др.

Искусство программирования на JAVA

Герберт Шилдт / Джеймс Холмс

THE ART OF JAVA

HERBERT SCHILDT,
JAMES HOLMES

McGraw-Hill/Osborne

New York Chicago San Francisco
Lisbon London Madrid Mexico City Milan
New Delhi San Juan Seoul Singapore Sydney Toronto

ИСКУССТВО программирования на JAVA

ГЕРБЕРТ ШИЛДТ,
ДЖЕЙМС ХОЛМС

Москва • Санкт-Петербург • Киев
2005

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *Г.В. Галисеева*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Шилдт, Герберт, **Холмс**, Джеймс.

Ш57 Искусство программирования на Java. : Пер. с англ. — М. : Издательский дом "Вильямс", 2005. — 336 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0786-1 (рус.)

Эта книга отличается от множества других книг по языку Java. В то время как другие книги обучают основам языка, эта книга показывает, как использовать язык наиболее эффективно, с большей пользой и отдачей для решения запутанных задач программирования. На страницах книги постепенно раскрывается мощь, универсальность и элегантность языка Java.

Как и можно ожидать, несколько описанных приложений связаны непосредственно с Internet. Многие главы посвящены анализу кода, который иллюстрирует выразительные возможности Java независимо от Internet. Легкость, с которой эти программы могут быть написаны на языке Java, подтверждает гибкость и элегантность языка.

В каждой главе рассматриваются фрагменты кода, который можно использовать "как есть". Например, синтаксический анализатор может послужить отличным дополнением для многих разработок. Однако наибольшую пользу от этих программ можно получить, если их использовать как базовые для разработки собственных приложений. Например, Web-червь, подробное описание которого приводится в книге, может послужить основой для разработки архиватора Web-сайта или детектора разрыва связи.

Исходные тексты всех примеров, рассмотренных в книге, доступны на Web-сайте издательства. Книга рассчитана на студентов, преподавателей и специалистов в области компьютерных технологий.

ББК 32.973.26-018.2.75

ISBN 5-8459-0786-1 (рус.)
ISBN 0-07-222971-3 (англ.)

Оглавление

Содержание

Об авторах

Герберт Шилдт является известным автором книг по языкам программирования Java, C, C++ и C#, а также экспертом по программированию для Windows. Его книги по программированию разошлись тиражом более чем три миллиона экземпляров и переведены на все основные языки мира. Он автор многочисленных бестселлеров серий *Полный справочник и Руководство для начинающих*. Его перу принадлежат книги: *Полный справочник по Java 2*, по C, C++, C#. Герберт Шилд имеет степень магистра компьютерных наук в университете штата Иллинойс. С Шилдтом можно связаться по телефону (217) 586-4683.

Джеймс Холмс является признанным лидером по программированию на языке Java. Он был признан лучшим специалистом 2002 года журналом *Oracle Magazine Java Developer*. Джеймс входит в группу разработчиков открытого программного кода *Jakarta Struts*. В настоящее время он является независимым консультантом по языку Java и имеет сертификации Sun Certified Java Programmer и Sun Certified Web Component Developer. К Джеймсу можно обратиться по электронной почте `james@jamesholmes.com` или посетить его Web-сайт `http://www.JamesHolmes.com`.

Предисловие

Собравшись в 1991 году в фирме Sun Microsystems, Джеймс Гослинг, Патрик Нагхон, Крис Уорш, Ед Франк и Майк Шеридан начали работу над новым языком программирования, который вскоре потряс основы программирования. Изначально названный Oak (Дуб), новый язык программирования в 1995 был переименован в Java и именно с тех пор начал изменять основы программирования. Изменения происходили по двум важным направлениям: во-первых, в Java были включены средства, облегчающие использование и разработку приложений Internet. Java стал первым языком, по-настоящему приспособленным для работы с Internet.

Во-вторых, Java значительно изменил среду разработки программ и повысил скорость и качество разработки проектов. Например, была заново пересмотрена парадигма объектно-ориентированного программирования, получила завершенный вид обработка исключительных ситуаций, многопоточная обработка была встроена в язык, а также был использован промежуточный язык, названный байт-кодом, который позволил использовать язык Java на различных платформах.

Успеху Java способствовала опора на две важные концепции: встроенная поддержка Internet и удобная среда разработки. Реализация каждой из этих концепций по отдельности уже сделали бы Java отличным языком программирования, но комбинация этих концепций сделали Java поистине великим языком.

В этой книге объяснены многие причины, по которым Java стал действительно экстраординарным языком программирования.

О чем эта книга

Эта книга отличается от множества других книг по языку Java. В то время как другие книги обучают основам языка, эта книга показывает, как использовать язык наиболее эффективно, с большей пользой и отдачей для решения запутанных задач

программирования. На страницах книги постепенно раскрывается мощь, универсальность и элегантность языка Java.

Как и можно ожидать, несколько приложений, таких как менеджер загрузки в главе 4 или фрагменты электронной почты в главе 5, связаны непосредственно с Internet. Многие главы посвящены анализу кода, который иллюстрирует выразительные возможности Java независимо от Internet. Например, интерпретатор языка в главе 3 или подпрограммы поиска в главе 10, которые мы называем примерами "чистого кода". Ни одно из этих приложений не связано с Internet и не использует графический интерфейс. Они являются примерами фрагментов кода, которые можно найти ранее написанными на языке C++. Легкость, с которой эти программы могут быть написаны на языке Java, подтверждает гибкость и элегантность языка.

В каждой главе рассматриваются фрагменты кода, который можно использовать "как есть". Например, синтаксический анализатор из главы 2 может послужить отличным дополнением для многих разработок. Однако наибольшую пользу от этих программ можно получить, если их использовать как базовые для разработки собственных приложений. Например, Web-червь, описание которого находится в главе 7, может послужить основой для разработки архиватора Web-сайта или детектора разрыва связи. В общем, рассматривайте приведенные программы как основу для своих дальнейших разработок.

Необходимые знания для чтения книги

Читатели этой книги должны иметь хорошие знания основ языка Java. Необходимо уметь создавать, компилировать и запускать Java-программы, пользоваться основными возможностями интерфейса прикладного программирования Java, уметь обрабатывать исключительные ситуации и создавать многопоточные программы. Таким образом, предполагается, что, принимаясь за эту книгу, читатели уже знакомы с языком Java и разработали на нем несколько программ.

Если читатель считает себя недостаточно подготовленным для работы с языком Java, то такие книги, как *Java 2: A Beginner's Guide* и *Java 2: The Complete Reference*[1] издательства McGraw-Hill/Osborne, позволят лучше понять основы Java или освежить свои знания.

Эффект командной работы

Я пишу о программировании уже много лет и сейчас я редко работаю с соавторами. Однако эта книга является исключением. Я встретился с одним из ярчайших талантов в программировании, Джеймсом Холмсом. Джеймс является выдающимся программистом с впечатляющими достижениями. Он признан лучшим специалистом 2002 года по результатам конкурса, проведенного журналом *Oracle Magazine Java Developer*. Также отмечена его деятельность в группе разработчиков открытого программного кода *Jakarta Struts*. Поскольку Джеймс обладает уникальными знаниями в программировании для Internet, я рад, что смог привлечь его сотрудничать со мной. В результате Джеймс написал главы 4, 5, 6 и 7, которые включают большинство рассмотренных в этой книге приложений для Internet.

---

[1] *Перевод этой книги на русском языке выйдет в 2005 году в ИД "Вильямс"*

Не забудьте

Не забывайте, что исходный код примеров для всех глав и проекты из книги доступны на сайте `www.osborne.com`.

Что еще написано Гербертом Шилдтом

Данная книга входит в серию книг по программированию. Ниже приведены еще несколько различных серий, которые могут представлять интерес.

При изучении Java:

Java 2: The Complete Reference

Java 2: A Beginner's Guide

Java 2: Programmer's Reference

При изучении C++:

C++: The Complete Reference (Полный справочник по C++)

C++: A Beginner's Guide (C++: Руководство для начинающих)

Teach Yourself C+ + (Освой самостоятельно C++ за 21 день)

C++ From the Ground Up (C++: базовый курс)

STL Programming From the Ground Up

При изучении C#:

C#: A Beginner's Guide

C#: The Complete Reference (Полный справочник по C#)

Если есть необходимость более подробно познакомиться с языком C и основами современного программирования, то используйте следующие книги:

C: The Complete Reference (Полный справочник по C)

Teach Yourself C (Освой самостоятельно C за 21 день)

Что написано Джеймсом Холмсом

Для знакомства с распорками — открытыми фрагментами программного кода каркасов приложений для Web-разработчиков — рекомендуется следующая книга:

Struts: The Complete Reference

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

This page is intentionally left blank

ГЛАВА

1

Таланты Java

Всемирная история состоит и из таких небольших фрагментов, как история программирования, изучая которую, можно провести определенные аналогии. Как и в человеческой истории, прошедшей все этапы от простого к сложному, в истории программирования время от времени тоже происходил качественный скачок. Подобно цивилизациям, языки программирования возникают, развиваются и исчезают. Именно расцвет и упадок способствуют прогрессу. И новое время рождало новый язык, который свергал своего предшественника и вводил более совершенные технологии программирования. Оглядываясь назад, можно отметить несколько ключевых событий, таких как распад Римской империи, вторжение в Британию в 1066 году или первый ядерный взрыв, которые изменили мир. То же самое происходило и с языками программирования, хотя и в несколько меньших масштабах. Например, изобретение языка FORTRAN навсегда изменило методы программирования. Аналогичным ключевым событием можно считать и создание Java. Этот язык можно считать вехой, отметившей начало эры программирования для Internet. Разработанный специально для создания приложений для Internet и использующий принцип "Написан один раз, но может запускаться везде", Java положил начало новой парадигме программирования. Еще в начале работы Гослинг и его коллеги увидели, что решение небольшого класса проблем является катализатором для появления новейших технологий программирования. Java внес настолько фундаментальные понятия в принятые нормы программирования, что историю компьютерных языков программирования можно разделить на два этапа: до и после Java. Программисты эпохи, предшествующей Java, создавали программы и запускали их на отдельном компьютере. Программисты, использующие Java, создают программы для компьютеров, распределенных по сети и работающих в сетевом окружении. Программист больше не оперирует терминами отдельно стоящего компьютера, вместо этого он использует понятие сети и термины серверов, клиентов и хостов.

Хотя разработчики Java ориентировались на создание языка для Internet, Java является языком не только для Internet. Это по-настоящему универсальный язык с широкими возможностями, разработанный для современного сетевого мира. Java можно использовать для решения любых задач программирования, хотя в нем и сделаны акценты на решении задач программирования для сетей. Помимо этого, Java вобрал в себя много новых возможностей и поднял искусство программирования на новую высоту. Эти инновации все еще будоражат компьютерное сообщество. Например, некоторые аспекты языка C# основаны на элементах, впервые введенных в Java.

На протяжении всей книги мы будет демонстрировать широкие возможности Java, используя его при рассмотрении различных элементов приложений. Некоторые из приложений демонстрируют мощь языка, независимо от его возможностей работы в сети. Мы называем это примерами "чистого кода", потому что они показывают выразительные средства Java и философию проектирования. Другие примеры иллюстрируют легкость, с которой могут быть разработаны запутанные программы для работы в сети, используя язык Java и классы прикладного программирования.

Перед углубленным изучением Java, мы потратим некоторое время на объяснение тех возможностей, которые делают Java столь привлекательным. Все эти возможности мы объединили словосочетанием "таланты Java".

Простые типы и объекты: поиск равновесия

Одной из значительных задач, стоящих перед разработчиками объектно-ориентированных языков программирования, является дилемма выбора в пользу объектов или простых типов. Это настоящая проблема. С концептуальной точки зрения каждый тип данных должен быть объектом и каждый тип должен быть унаследован от универсального родительского типа. Это позволяет одинаково обращаться со всеми типами, т.к. каждый тип имеет набор общих свойств и общие правила поведения. Сложность заключается в том, что для повышения производительности необходимо использовать простые типы, такие как целые числа или удвоенные слова, которые не трактуются как объекты и поэтому не используют всю ту надстройку, которая необходима для создания объекта. Поэтому при работе с простыми типами выполняющаяся программа не подключает дополнительные фрагменты кода и работа с ними происходит очень быстро.

Поскольку простые типы обычно используются в таких часто применяемых конструкциях, как циклы или вычисления условия, то замена их объектами приведет к резкому снижению производительности. И нахождение оптимального соотношения между принципами "все является объектом" и "производительность любой ценой" является довольно сложным делом.

В Java эта проблема решена очень элегантным образом. Во-первых, существует восемь простых типов: `byte`, `short`, `int`, `long`, `char`, `float`, `double` и `boolean`. Эти типы транслируются непосредственно в бинарные значения. Таким образом, переменные такого типа могут обрабатываться процессором непосредственно, без дополнительного расхода ресурсов. Простые типы Java работают быстро и эффективно, как, впрочем, и в любом другом языке программирования. Поэтому циклы с использование простых типов выполняются с максимально возможной скоростью.

В отличие от простых типов, все другие типы Java являются объектами, унаследованными от универсального суперкласса `Object`, и должны содержать в себе дополнительный код, обеспечивающий необходимую функциональность. Например, все объекты имеют метод `toString()`, т.к. этот метод объявлен в базовом классе `Object`.

Поскольку простые типы не являются объектами, Java по различному трактует объекты и простые типы. В этом проявляется талант разработчиков Java. В Java обращение ко всем объектам производится при помощи ссылок, а не напрямую, как в случае с простыми типами. Поэтому в программе нет непосредственного управления объектами. Использование этого отдельного правила позволяет получить значительные преимущества, которые на первый взгляд не так заметны. Например, можно организовать автоматический сборщик мусора. Так как обращение к объектам происходит через ссылки, то сборщик мусора может эффективно отслеживать наличие ссылок и удалять объекты, на которые ссылок уже нет. Второе преимущество заключается в том, что, используя механизм ссылок, можно ссылаться на любой объект системы.

Конечно, обращение к объектам с помощью ссылок требует дополнительных затрат. При этом в качестве ссылок используются адреса (указатели), т.е. доступ к каждому объекту происходит косвенно, через считывание его адреса. Хотя современные процессоры обрабатывают косвенные ссылки довольно эффективно, все равно

такое обращение происходит медленнее, чем в случае прямого обращения, как с простыми типами. Но хотя обращение к простым типам происходит быстро, существует множество случаев, когда необходимо, чтобы с простыми объектами можно было работать, как с объектами. Например, необходимо во время выполнения создать список целых чисел, причем этот список должен быть виден сборщику мусора, который автоматически должен его уничтожить, когда список станет уже не нужен. Для учета таких ситуаций Java заключает простые типы в оболочку и создает объекты, такие как `Integer` или `Double`. Поэтому при необходимости с простыми типами можно работать как с объектами.

Таким образом Java соблюдает необходимый баланс между объектами и простыми типами, необходимый в программе. Это позволяет писать эффективные программы, используя объектную модель и позволяя избежать непроизводительных расходов, всегда используя простые типы там, где только это возможно.

Управление памятью с помощью сборщика мусора

Управление памятью с помощью сборщика мусора не является абсолютно новой идеей, но в Java она выражена особенно четко и логично. В таких языках программирования, как C++, управление памятью осуществлялось вручную, когда программист явно уничтожал уже ненужные объекты. Но это порождало многочисленные проблемы, т.к. обычно разработчик частично забывал освободить ненужные объекты, которые занимали память и в лучшем случае снижали производительность, а иногда приводили к зависанию программы. Java избавил разработчиков от этой проблемы, полностью взяв управление памятью на себя. Такой режим может быть реализован очень эффективно благодаря тому, что обращение к объектам производится в помощью ссылок. И когда сборщик мусора находит объект, на который нет ссылки, он ставит этот объект в очередь на уничтожение. В Java разрешено и прямое управление объектами наподобие простых типов, но в этом случае программист должен сам следить за уничтожением объектов, т.к. сборщик мусора не может отслеживать такие объекты. Использование сборщика мусора отражает общую философию Java. Разработчики Java заложили в этот язык достаточно много средств, облегчающих работу программистов и позволяющих автоматически решать проблемы, осложняющие работу программиста в большинстве языков программирования. И решение такой проблемы, как уборка мусора, позволяет больше не задумываться над этим и посвятить время не техническим, а логическим проблемам. Поэтому сборщик мусора снимает целый набор проблем.

Замечательно простая модель многозадачности

Разработчики Java смогли предусмотреть сложные проблемы, которые могут возникнуть при разработке проектов многозадачного программирования. Напомним, что существует два основных вида многозадачности: на основе процесса и на основе потока. Многозадачность на основе процесса позволяет компьютеру выполнять две или более программ одновременно. При многозадачности на основе процесса, поток является наименьшим планируемым модулем. Поток является частью выполняемой про-

граммы. Таким образом, процесс может состоять из двух или более одновременно выполняемых самостоятельных потоков.

Хотя поддержка многозадачности на основе процесса является функцией операционной системы, язык программирования должен использовать эти преимущества и взаимодействовать с операционной системой для поддержки этих возможностей. Например, язык C++, который не имеет встроенной поддержки многозадачности, должен непосредственно использовать функции операционной системы и полагаться только на эти функции. Это означает, что создание, начало, синхронизация и окончание режима многозадачности требует многочисленных обращений к функциям операционной системы. В результате код, написанный на C++ для многозадачного режима, не является переносимым. При этом программы становятся громоздкими и плохо понимаемыми.

Поскольку в Java встроена поддержка многозадачности, то это освобождает программиста от многих проблем, возникающих при написании программ на других языках. Одним из наиболее элегантных решений при создании много задачной модели является обеспечение синхронизации. Синхронизация основана на двух передовых технологиях. Во-первых, в Java все объекты имеют встроенный монитор, который действует как взаимоисключающий блокирующий узел. Только один поток может использовать монитор в определенное время. Включение блокировки производится определенным методом совместно с ключом синхронизации. Когда вызывается метод синхронизации, объект блокируется и другой поток ожидает разрешения на доступ к объекту.

Во-вторых, поддержка синхронизации встроена в базовый класс `Object`, который является суперклассом для всех других классов. В этом классе объявлены следующие методы: `wait()`, `notify()` и `notifyAll()`. Эти методы обеспечивают связь между потоками. Таким образом, все объекты обладают возможность взаимодействия между потоками. При взаимодействии с методами синхронизации эти методы обеспечивают высокий уровень контроля за взаимодействием между потоками.

Имея встроенные средства взаимодействия между потоками (очень легкие в использовании), Java изменил подходы к фундаментальному построению архитектуры программ. До Java большинство программистов рассматривали программу как монолитную структуру, которая является единым выполняемым модулем. После появления Java программисты начали рассматривать программу как множество параллельных задач, которые взаимодействуют одна с другой. Это оказало чрезвычайно сильное влияние на дальнейшее развитие технологий программирования, но, возможно, наибольшее влияние оказала технология использования программных компонентов.

Встроенная обработка исключительных ситуаций

Концептуально средства обработки исключительных ситуаций (исключения) были разработаны до Java. Таким образом, встроенные программные фрагменты для обработки исключительных ситуаций использовались и в других языках программирования. Например, исключения были добавлены в язык C++ за несколько лет до того, как появился Java. Но подход, принятый в Java, отличался от того, что было в других языках. Здесь исключениям отводилась довольно значительная роль, и они

стали частью среды программирования Java. Обработка исключений не добавлялись при необходимости, а была полностью встроена в язык Java как одно из базовых свойств. Ключевым аспектом механизма обработки исключений Java является то, что его использование не является дополнительным. Обработки исключения является правилом и необходимые программные конструкции встроены в язык, в отличие, например, от C++, где поддерживается обработка исключений, но этот механизм не встроен в язык. Рассмотрим наиболее общую ситуацию открытия и чтения файлов. В Java, когда ошибка происходит именно в этот момент, возникает исключительная ситуация. В отличие от Java, например, в C++ при возникновении ошибки не происходит ничего подобного и метод, который использовался при этих операциях, просто возвращает код ошибки. Поскольку в C++ нет встроенной поддержки исключений, все библиотечные программы возвращают коды ошибок и программист должен постоянно об этом помнить и постоянно производить проверку кодов возврата вручную. В Java код, который может привести к ошибке, просто заключается в блок `try...catch`. Все ошибки будут перехвачены автоматически.

Естественная поддержка полиморфизма

Полиморфизм является необходимым атрибутом объектно-ориентированного программирования, который позволяет использовать один интерфейс множеством методов. Java поддерживает различные формы полиморфизма, и относительно этого выделить два аспекта. Во-первых, все методы могут быть переопределены в производных классах. Во-вторых, используется ключевое слово `interface`. Рассмотрим это подробнее.

Поскольку методы суперкласса могут быть переопределены в производных классах, тривиально легко создать иерархию, в которой производные классы конкретизируют суперкласс, т.е. приспосабливают его возможности к возникшей ситуации. Напомним, что ссылка на суперкласс может использоваться для ссылки на любой производный от него класс. Кроме того, вызов метода объекта производного класса с помощью ссылки на суперкласс автоматически приведет к вызову переопределенной версии этого метода. Таким образом, суперкласс определяет основные свойства объекта, используемые по умолчанию. Эти свойства могут быть подстроены в производных классах в зависимости от возникшей ситуации. Поэтому один базовый интерфейс может служить основой для различных реализаций.

Конечно, Java использует концепцию "один интерфейс — множество методов" и развивает ее. В языке используется ключевое слово `interface`, которое позволяет полностью отделить методы класса от их реализации. Хотя интерфейс является абстрактным понятием, можно объявлять ссылки на интерфейсные типы. Эти ссылки можно использовать для обращения к любому объекту, в котором реализован данный интерфейс. Это является очень мощным средством для реализации полиморфизма. Если в классе реализован определенный интерфейс, то объекты на основе этого класса можно использовать в любом месте, благодаря чему программируется функциональность, определяемая интерфейсом. Например, предположим, что есть интерфейс с именем `MyIF`, и рассмотрим следующий метод.

```
void myMeth(MyIF ob) {
// . . . }
```

Любой объект, который реализует интерфейс `MyIF`, может быть передан в метод `myMeth()`. И не имеет значения, что делает данный объект. Если в нем реализован интерфейс `MyIF`, метод `myMeth()` выполнит его.

Переносимость и защищенность за счет использования байт-кода

Несмотря на все описанные выше нововведения, Java остался бы в истории программирования не более, чем удобным языком программирования для решения определенных задач, если бы не такая часть языка, как байт-код. Как известно, после компилятора Java создается не машинный код, рассчитанный на использование непосредственно процессором, а высоко оптимизированный набор команд, называемый байт-кодом, который выполняется виртуальной машиной Java (Java Virtual Machine — JVM). Оригинальная JVM является просто интерпретатором команд байт-кода. В настоящее время JVM можно рассматривать как компилятор времени выполнения байт-кода в машинные команды. Однако использование байт-кода расставило чрезвычайно важные акценты и способствовало огромному успеху Java.

Первым преимуществом является переносимость. Компиляция программ Java в байт-код позволяет затем выполнять их на любом компьютере, с любым типом процессора и любой операционной системой, т.к. JVM можно разработать для любой платформы. Другими словами, если для некоторого окружения разработана JVM, то с ее помощью можно запускать все программы на языке Java. При этом нет необходимости специально подстраивать программу для этого окружения. Один и тот же байт-код используется во всех JVM. Поэтому и можно с полной уверенностью сказать о программах на языке Java: "написанная однажды, запускается всегда и везде".

Вторым преимуществом байт-кода является защищенность. Поскольку выполнение байт-кода происходит под управлением JVM, то JVM может легко защитить программы на языке Java от выполнения неразрешенных действий и тем самым защитить компьютер. Именно эта способность защитить компьютер от злонамеренных вторжений в значительной степени способствовала успеху Java и привела к широкому использованию аплетов. Так как аплет является небольшой, динамически загружаемой программой, которая распространяется через Internet, то средства защиты позволяют широко использовать эти удобные средства. Комбинация байт-кода и JVM позволяет создать механизм безопасного использования аплетов. Без байт-кода всемирная паутина занимала бы несколько другое место в мире.

Разнообразие прикладного интерфейса Java

Концептуально компьютерный язык состоит из двух частей. Первая часть — это собственно сам язык, его ключевые слова и синтаксис. Вторую часть составляют стандартные библиотеки, которые содержат наборы классов, интерфейсы и методы, которые доступны для программиста. Почти все основные современные языки программирования включают большие библиотеки, среди которых своим разнообразием и богатством предложений для программиста выделяются именно библиотеки Java. Когда Java был только создан, его библиотеки содержали только набор базовых

пакетов, таких как `java.lang`, `java.io` и `java.net`. С каждой новой реализацией в Java добавлялись новые классы и пакеты. В настоящее время Java предлагает программистам впечатляющий набор средств и поразительную функциональность.

С самого начала одним из ключевых элементов, который отличал библиотеки Java от библиотек других языков, была поддержка сети. До создания Java другие языки, такие как C++, не поставляли стандартных библиотечных элементов с возможностью работы в сети. Java поставлялся с классами, позволяющими легко устанавливать соединения через сеть и использовать Internet. Благодаря Java Internet стал открытым для всех программистов, а не только для тех, которые специализировались по сетевым технологиям. Такая широкая поддержка сетевых соединений в Java изменила привычные представления об обработке данных.

Другим ключевым пакетом ядра библиотеки Java является файл `java.awt`, который поддерживает инструментальные средства для создания примитивов Window (Abstract Window Toolkit — AWT). Пакет AWT позволяет программисту создавать переносимые графические интерфейсы. Используя классы AWT, можно создавать оконные приложения, которые используют стандартные графические примитивы, такие как полосы прокрутки, элементы проверки и радио кнопки. С помощью AWT можно создавать графические приложения, которые могут выполняться в любой среде, которую поддерживает виртуальная машина Java. Такой уровень переносимости до Java был неизвестен.

Включение в Java средств AWT революционизировало путь, который использовался программистами для создания интерфейса приложения. До Java, программы с использование графического интерфейса пользователя должны были создаваться для каждой среды, в которой программы будут запускаться. После появления Java программы, написанные, например, для Windows, могут без изменений запускаться на компьютерах Apple. Программисты могут написать всего одну программу, которая будет выполняться в обоих средах. Благодаря использование переносимого графического интерфейса, Java унифицировал среду программирования.

Несколько лет спустя в Java был добавлен облегченный альтернативный пакет AWT — Swing. Компоненты Swing содержатся в пакете `javax.swing` и дополнительных к нему пакетах. Swing предлагает программистам богатый набор графических примитивов, которые улучшили переносимость Java. Многие примеры из этой книги используют как компоненты AWT, так и компоненты Swing, давая программисту возможность разрабатывать высокоэффективные и переносимые приложения с использованием графического интерфейса пользователя.

В настоящее время библиотека Java выросла значительно по сравнению с начальным кодом. Каждая новая реализация Java сопровождалась увеличением существующей библиотеки. Добавлялись новые пакеты и расширялась функциональность существующих пакетов. Библиотека находилась в постоянном изменении, пока не был достигнут достаточный уровень для эффективного решения задач в любой вычислительной среде. Эта способность изменяться и адаптироваться считается одним из самых привлекательных достоинств Java.

Аплеты

Сегодня можно сказать, что аплеты являются наиболее революционным достижением Java, поскольку с их помощью создаются переносимые, динамически загружаемые программы, которые могут безопасно выполняться и просматриваться браузером. До появления Java исполняемая часть всегда "подозревалась" в наличии злоумышленного кода, который мог принести вред компьютеру пользователя. Кроме того, код компилировался для одного типа процессора и операционной системы и не мог выполняться на другой системе. Поскольку к Internet подключались компьютеры в различной архитектурой, различными процессорами и операционными системами, то практически было невозможно создать программу, совместимую со всеми средами. Аплеты Java легко решили все эти проблемы. Использование аплетов позволило легко добавлять динамичные фрагменты в ранее статичный мир HTML. Аплеты сделали Web динамичным, и возврат назад уже невозможен.

В дополнение к тому, что изменился вид содержимого Web, аплеты произвели и другой эффект, точнее, побочный эффект — они ввели движение и в компоненты. Так как аплеты являются небольшими программами, они обычно представляют небольшой функциональный фрагмент, который можно рассматривать как компонент. Размышляя о терминах аплетов, мы вполне естественно перейдем к технологии Beans и далее. Сегодня компонентно-ориентированная архитектура, согласно которой приложение состоит из набора интерактивных компонентов, заменяет монолитную модель, используемую ранее.

Продолжение революции

Можно отметить много аспектов Java, которые отражают выдающиеся способности этого языка, хотя это не всегда только языковые конструкции. Java порождает новую культуру новаторства, в которой приветствуются новые идеи и для этих идей создается благоприятная среда, позволяющая их быстро реализовывать. В отличие от других, медленно развивающихся языков программирования, Java постоянно эволюционирует и приспосабливается. Более того, этот процесс открыт для всех через сообщество Java — Java Community Process (JCP). Сообщество JCP предлагает механизм, с помощью которого все пользователи Java могут влиять на будущее развитие языка, прикладных программ и связанных с этим технологий. Таким образом, пользователи, активно использующие язык Java, могут чувствовать себя разработчиками языка. С самого начала Java революционизировал среду программирования, и эта революция продолжается до сих пор. Java и сейчас находится на передовой линии разработки языков программирования. Этот язык занял выдающееся место в истории компьютерных вычислений.

ГЛАВА

2

Рекурсивно-последовательный синтаксический анализатор выражений

Как написать программу, которая анализирует строки, содержащие выражения, подобные (10-5)*3, и вычисляет правильный ответ? Является ли это привилегией только тех, кто глубоко понимает суть того, как это делается? Для многих этот процесс связан с некими "мистическими" манипуляциями, с помощью которых языки высокого уровня преобразовывают алгебраические выражения в команды, которые может понять и выполнить компьютер. Такая процедура называется синтаксическим анализом, и она является основной для всех языков при компиляции и интерпретации программ. Используется она и в электронных таблицах, и во многих других программах, где необходимо преобразовывать числовые выражения в форму, которую может использовать компьютер.

Но мистически это выглядит только для непосвященных, синтаксический анализатор выражений выполняет ряд четко определенных задач, для которых существуют по настоящему элегантные решения. Все это происходит потому, что проблема уже хорошо изучена и синтаксический анализатор работает в соответствии с правилами алгебры. В этой главе будет представлен анализатор, на который ссылаются как на рекурсивно-последовательный синтаксический анализатор, а также все дополнительные подпрограммы, которые позволят преобразовывать числовые выражения. Когда вы поймете логику работы анализатора, вы сможете легко расширить его возможности и приспособить его для решения ваших собственных задач.

Независимо от полезности отдельных фрагментов кода, анализатор был выбран как первый пример для этой книги потому, что он иллюстрирует мощь и диапазон применимости языка Java. Анализатор является фрагментом "чистого кода". Я имею ввиду, что он не ориентирован на работу в сети, не связан с графическим интерфейсом, не является аплетом или сервлетом и т.д. Такой тип кода можно найти написанным на языках C или C++, но не на Java. Поскольку Java внес революционные изменения на пути программирования для Internet, мы иногда забываем, что он не ограничен только этой средой. Наоборот, Java является универсальным языком программирования, который можно использовать для решения различных задач программирования. Разработка синтаксического анализатора отлично подтверждает эту точку зрения.

Выражения

Поскольку анализатор обрабатывает выражения, необходимо уточнить, что такое выражение и из каких частей оно состоит. Хотя существует множество различных выражений, в этой главе рассматривается только один тип выражений — арифметические выражения. Выражения, которые будут обрабатываться нашим синтаксическим анализатором, состоят из следующих элементов:

числа;

операторы +, –, /, *, ^, %, =;

круглые скобки;

переменные.

Здесь оператор "^" определяет возведение в степень (не XOR, как это сделано в Java), а оператор "=" является оператором присваивания. Эти элементы могут

комбинироваться друг с другом в соответствии с алгебраическими правилами. Ниже показано несколько примеров:

```
10 - 8
(100 - 5) * 14 / 6
a + b - c
10^5
a = 10 - b
```

Зададим следующее старшинство операторов:

высший	+, – (унарный)
	^
	*, /, %
	+, –
низший	=

Правила старшинства прилагаются к операторам при просмотре слева направо.

Разрабатываемый анализатор будет иметь некоторые ограничения. Во-первых, переменные могут быть выражены только отдельными буквами (другими словами, доступны только 26 переменных, от A до Z). Переменные не чувствительны к регистру (a и A определяют одну и ту же переменную). Во-вторых, все числовые значения должны быть числами с двойной точностью, хотя не представляет особого труда модифицировать анализатор для обработки других типов переменных. Наконец, для того чтобы не усложнять логическую составляющую и легкость понимания, производится проверка только на наличие элементарных ошибок.

Анализ выражения: трудности

Если глубоко не задумываться о проблемах, которые могут возникать при анализе выражений, можно решить, что все делается очень просто, но это совсем не так. Для того чтобы лучше понять проблему, проанализируем приведенное ниже выражение:

```
10 - 2 * 3
```

Понятно, что значение этого выражения равно 4. Хотя написать программу для решения именно этого выражения не составляет труда, проблема состоит в том, как написать программу, которая правильно решит любое выражение. Сначала необходимо придумать алгоритм подобно приведенному ниже:

```
a = получить первый операнд
while(текущий операнд) {
  op = получить оператор
  b = получить второй операнд
  a = a op b
}
```

Исходя из этого алгоритма сначала необходимо получить первый операнд, оператор и второй операнд для выполнения первой операции. Затем получить следующий оператор и операнд и выполнить следующую операцию и т.д. Однако если на основе этого алгоритма рассчитать выражение `10-2*3`, то получим значение 24 (8*3) вместо 4, поскольку в данном алгоритме не учитывается старшинство операторов. Нельзя только последовательно брать операнды и операторы слева направо, поскольку,

согласно правилам алгебры, умножение должно выполняться перед вычитанием. Сначала возникает уверенность, что эту проблему легко преодолеть, и в некоторых отдельных случаях ее можно решить. Но проблема возникнет вновь, как только будут добавлены круглые скобки, возведение в степень, переменные, унарные операторы и тому подобное. Хотя существует множество путей для решения всех возникающих проблем, обсуждаемый ниже анализатор является наиболее простым из всех. Он называется рекурсивно-последовательным синтаксическим анализатором, и далее вы поймете, почему его так назвали. Некоторые другие методы, используемые при разработке анализаторов, используют комплексные таблицы, обычно генерируемые другой программой. Такие анализаторы часто называют табличными (или управляемыми таблицами) синтаксическими анализаторами.

Синтаксический анализ выражений

Существует несколько путей проанализировать и определить значения выражения. Для понимания работы рекурсивно-последовательного синтаксического анализатора необходимо рассматривать выражение как рекурсивную структуру данных, т.е. выражение должно быть определено в терминах самого выражения. Если на мгновение представить, что в выражении могут использоваться только операторы +, -, *, / и круглые скобки, то все выражения могут быть определены в соответствии со следующими правилами:

```
выражение -> терм [+ терм] [- терм]
терм -> фактор [* фактор] [/ фактор]
фактор -> переменная, число, (выражение)
```

В квадратные скобки заключены дополнительные элементы, а символ "->" означает соответствие. Обычно такие правила называют порождающими правилами выражения. Поэтому для определения терма можно сказать: "Терм представляет фактор, который может быть умноженным или разделенным на фактор". Обратите внимание, что старшинство операторов при просмотре выражения определено неявно.

Рассмотрим пример. Выражение

```
10 + 5 * B
```

имеет два терма: `10` и `5*B`. Второй терм состоит из двух факторов: `5` и `B`. Эти факторы представляют одно число и одну переменную. С другой стороны, выражение

```
14 * (7 - C)
```

имеет два фактора: `14` и `(7-C)`. Эти факторы представляют число и выражение, заключенное в круглые скобки. Выражение в круглых скобках содержит два терма: одно число и одну переменную.

Такой способ формирует основы для создания рекурсивно-последовательного синтаксического анализатора, который представляет набор совместно рекурсивных методов, последовательно выполняющихся и реализующих порождающие правила. На каждом шаге анализатор выполняет специфические операции в соответствии с алгебраическими правилами. Для лучшего понимания того, как порождающие правила используются при анализе выражения, рассмотрим пример анализа следующего выражения:

```
9 / 3 - (100 + 56).
```

При этом должна выполняться следующая последовательность действий.

- Получить первый терм `9/3`.
- Получить каждый фактор и разделить целые числа. Результирующее значение будет равно `3`.
- Получить второй терм `(100+56)`. С этого момента начинается рекурсивный анализ этого подвыражения.
- Получить каждый терм и сложить их. Результирующее значение будет равно `156`.
- Вернуться из рекурсивного вычисления второго терма.
- Вычесть `156` из `3`. Результат равен `-153`.

Если вам пока не все понятно, не расстраивайтесь. Это достаточно общая концепция, которую будем уточнять в дальнейшем. Необходимо запомнить два основных принципа рекурсивного обзора выражения. Во-первых, старшинство операторов неявно учитывается в процессе реализации порождающих правил. Во-вторых, этот метод анализа и вычисления выражений подобен тому, как человек решает математическое уравнение.

В конце этой главы будут представлены два анализатора. Первый будет анализировать и вычислять выражения с плавающей запятой удвоенной точности, которые состоят только из константных значений. Второй добавит возможность использовать переменные.

Анализ выражения

Для того, чтобы вычислить выражение, анализатору необходимо обрабатывать отдельные составляющие части всего выражения. Например, выражение

```
A * B - (W + 10)
```

содержит отдельные части: `A`, `*`, `B`, `-`, `(`, `W`, `+`, `10` и `)`. На языке анализатора каждая часть выражения называется лексемой (token), и каждая лексема является неразложимой частью выражения. Разложение выражения на лексемы лежит в основе работы анализатора, поэтому рассмотрим это подробнее, прежде чем изучать сам анализатор.

Для разложения выражения на лексемы, необходим алгоритм, который будет последовательно получать каждую лексему при проходе выражения от начала до конца. В этот алгоритме также должен учитываться тип лексемы и определяться конец выражения. В анализаторе, приведенном ниже, метод, который реализует этот алгоритм, имеет имя `getToken()`.

Оба анализатора из этой главы объединены в классе `Parser`. Хотя этот класс будет подробно описан позже, первую часть необходимо представить сейчас для того, чтобы понять, как работает метод `getToken()`. Начало анализатора с входящими переменными и полями показано ниже.

```
class Parser {
  // Объявление лексем.
  final int NONE = 0;
  final int DELIMITER = 1;
  final int VARIABLE = 2;
```

```
final int NUMBER = 3;

// Объявление констант синтаксических ошибок.
final int SYNTAX = 0;
final int UNBALPARENS = 1;
final int NOEXP = 2;
final int DIVBYZERO = 3;

// Эта лексема определяет конец выражения.
final String EOE = "\0";

private String exp;     // Ссылка на строку с выражением.
private int expIdx;     // Текущий индекс в выражении.
private String token;   // Сохранение текущей лексемы.
private int tokType;    // Сохранение типа лексемы.
```

Анализатор сначала определяет значение, которое указывает тип лексемы. При анализе выражения каждая лексема должна быть связана с определенным типом. Для разработанного в этой главе анализатора используются только три типа: переменная, число и разделитель. Они представлены соответствующими значениями: VARIABLE, NUMBER и DELIMITER. Разделителями также являются как операторы, так и круглые скобки. Тип NONE используется только для замены еще не определенной лексемы.

Затем в анализаторе указаны значения, представляющие различные ошибки, которые могут произойти в процессе анализа выражения. Значение SYNTAX представляет широкую категорию ошибок, которые происходят при неправильно записанном выражении. Значение UNBALPARENS возвращается при несовпадении количества открывающихся и закрывающихся скобок. Значение NOEXP говорит об отсутствии выражения при запуске анализатора. Значение DIVBYZERO указывает на деление на нуль.

Наконец, переменная окончания EOE является лексемой, которая указывает на достижение конца выражения.

Ссылка на строку, которая содержит выражение для анализа, сохраняется в переменной exp. Таким образом, переменная exp будет ссылаться на строку, подобную "10+4". Индекс очередной лексемы в строке сохраняется в переменной expIdx, которая при начальном запуске устанавливается в нуль. Текущая лексема сохраняется в переменной token, а ее тип сохраняется в переменной tokType. Эти поля являются закрытыми, поскольку они используются только в самом анализаторе и не должны модифицироваться внешним кодом.

Листинг метода getToken() приведен ниже. При каждом обращении к этому методу будет получена очередная лексема из строки, представляющей выражение, ссылка на которую находится в переменной exp. Подсчет лексем происходит в переменной expIdx. Другими словами, каждый раз при обращении к методу getToken(), возвращается лексема exp[expIdx]. Затем лексема помещается в поле token. Тип лексемы помещается в поле tokType. В методе getToken() используется метод isDelim(), который также приведен ниже.

```
// Получить следующую лексему.
private void getToken() {
  tokType = NONE;
  token = "";
  // Проверить окончание выражения.
  if(expIdx == exp.length()) {
    token = EOE;
```

```
    return;
  }
  // Пропустить пробелы.
  while(expIdx < exp.length() &&
    Character.isWhitespace(exp.charAt(expIdx))) ++expIdx;
  // Проверить окончание выражения.
  if(expIdx == exp.length()) {
    token = EOE;
    return;
  }
  if(isDelim(exp.charAt(expIdx))) {                    // Оператор.
    token += exp.charAt(expIdx);
    expIdx++;
    tokType = DELIMITER;
  }
  else if(Character.isLetter(exp.charAt(expIdx))) {   // Переменная
    while(!isDelimfexp.charAt(expIdx))) {
      token + = exp.charAt(expIdx);
      expIdx++;
      if(expIdx >= exp.length()) break;
    }
    tokType = VARIABLE;
  }
  else if(Character.isDigit(exp.charAt(expIdx))) {    // Число.
    while(!isDelimfexp.charAt(expIdx))) {
      token += exp.charAt(expIdx);
      expIdx++;
      if(expIdx >= exp.length()) break;
    }
    tokType = NUMBER;
  }
  else {                                          // Неизвестный символ.
    token = EOE;
    return;
  }
}

// Возвращает true, если с является разделителем.
private boolean isDelim(char c)
{
  if((" +-/*%^=()".indexOf(c) != -1))
    return true;
  return false;
}
```

Давайте подробнее рассмотрим метод `getToken()`. После инициализации переменных, в методе проверяется достижение конца выражения, которое определяется сравнением значения переменной `expIdx` с длиной строки `exp.length()`. Так как значением переменной `expIdx` является номер (индекс) очередной лексемы, то как только этот номер станет равным последнему номеру строки, то считается, что выражение полностью пройдено.

При нахождении лексем `getToken()` пропускает все предшествующие пробелы. Если в конце выражения также находятся пробелы, то возвращается лексема EOE. Во всех остальных случаях возвращается или цифра, или переменная, или оператор как значение `exp[expIdx]`. Если следующим символом является оператор, то он передается как строковое значение в поле `token`, а значение DELIMITER сохраняется в поле `tokType`. Если следующим символом является буква, то считается, что это

переменная. Она передается как строковое значение в поле token, а полю tokType присваивается значение VARIABLE. Если следующим символом является цифра, то полностью считывается число и оно передается как строковое значение в поле token, а полю tokType присваивается значение NUMBER. Наконец, если следующий символ отсутствует, то полю token присваивается значение EOE.

Чтобы не усложнять метод getToken(), пропущены проверки некоторых ошибок и сделаны отдельные допущения. Например, любой нераспознанный символ с предшествующими пробелами приводит к выходу из метода. Также в приведенной версии переменные могут быть любой длины, но только первый символ является значимым. В дальнейшем вы можете добавить больше проверок и уточнить правила в соответствии с вашими требованиями.

Для лучшего понимания процесса анализа подробно рассмотрим все этапы анализа для следующего выражения:

```
A + 100 - (B * C) / 2
```

Лексема	*Тип лексемы*
A	VARIABLE (Число)
+	DELIMITER (Разделитель)
100	NUMBER (Число)
-	DELIMITER (Разделитель)
(	DELIMITER (Разделитель)
B	VARIABLE (Переменная)
*	DELIMITER (Разделитель)
C	VARIABLE (Переменная)
)	DELIMITER (Разделитель)
/	DELIMITER (Разделитель)
2	NUMBER (Число)

Помните, что поле token всегда сохраняет строку, даже если оно содержит один символ.

Замечание. Хотя Java включает некоторые собственные средства синтаксического анализа, которые можно найти в классе StringTokenizer, лучше пока их не использовать и изучить данный анализатор полностью, используя метод getToken().

Простой синтаксический анализатор выражений

Ниже представлена первая версия анализатора. Он может анализировать выражения, которые состоят только из литералов, операторов и круглых скобок. Хотя метод getToken() может обрабатывать переменные, в анализаторе это не используется. После того как вы поймете работу этого простейшего анализатора, будут расширены его возможности и будет добавлена обработка переменных.

```
/*
Этот модуль содержит рекурсивно-последовательный
анализатор, который не использует переменных
*/
```

```
// Класс исключений для ошибок анализатора.
class ParserException extends Exception {
  String errStr; // describes the error
  public ParserException(String str) {
    errStr = str;
  }
  public String toStringO {
    return errStr;
  }
}

class Parser {
  // Типы лексем.
  final int NONE = 0;
  final int DELIMITER = 1;
  final int VARIABLE = 2;
  final int NUMBER = 3;

  // These are the types of syntax errors,
  final int SYNTAX = 0;
  final int UNBALPARENS = 1;
  final int NOEXP = 2;
  final int DIVBYZERO = 3;

  // Лексема, отмечающая конец выражения.
  final String EOE = "\0";
  private String exp;      // Ссылка на строку с выражением.
  private int expIdx;      // Текущий индекс.
  private String token;    // Содержит текущую лексему.
  private int tokType;     // Содержит тип текущей лексемы.
  // Точка входа анализатора.
  public double evaluate(String expstr) throws ParserException
  {
    double result;
    exp = expstr;
    expIdx = 0;
    getToken();
    if(token.equals(EOE))
      handleErr(NOEXP);        // Нет выражения.
    // Анализ и вычисление выражения.
    result = evalExp2();
    if(!token.equals(EOE)) // Последняя лексема должна быть EOE
      handleErr(SYNTAX);
      return result;
  }
  // Сложить или вычесть два терма.
  private double evalExp2() throws ParserException
  {
    char op;
    double result;
    double partialResult;
    result = evalExpS();
    while((op = token.charAt(0)) == '+' || op == '-') {
      getToken ();
      partialResult = evalExpB();
      switch(op) {
        case '-':
          result = result - partialResult;
          break;
        case '+':
          result = result + partialResult;
```

```
        break;
      }
    }
    return result;
  }
  // Умножить или разделить два фактора.
  private double evalExp3() throws ParserException
  {
    char op;
    double result;
    double partialResult;
    result = evalExp4();
    while((op = token.charAt(0)) == '*' ||
          op == '/' | op =='%'){
      getToken();
      partialResult = evalExp4();
      switch(op) {
        case '*':
          result = result * partialResult;
          break;
        case '/':
          if(partialResult == 0.0)
            handleErr(DIVBYZERO);
          result = result / partialResult;
          break;
        case '%':
          if(partialResult == 0.0)
            handleErr(DIVBYZERO);
          result = result % partialResult;
          break;
      }
    }
    return result;
  }
  // Выполнить возведение в степень.
  private double evalExp4() throws ParserException
  {
    double result;
    double partialResult;
    double ex;
    int t;
    result = evalExp5();
    if(token.equals ("*") ) {
      getToken();
      partialResult = evalExp4();
      ex = result;
      if(partialResult ==0.0) {
        result = 1.0;
      } else
        for(t=(int)partialResult-1; t > 0; t--)
          result = result * ex;
    }
    return result;
  }
  // Определить унарные + или -.
  private double evalExpB() throws ParserException
  {
    double result;
    String  op;
    op = " " ;
    if((tokType == DELIMITER) &&
```

```
      token.equals("+") || token.equals("-")) {
    op = token;
    getToken() ;
  }
  result = evalExp6();
  if(op.equals("-")) result = -result;
  return result;
}
// Обработать выражение в скобках.
private double evalExp6() throws ParserException
{
double result;
if(token.equals("(")) {
  getToken();
  result = evalExp2();
  if(!token.equals(")"))
     handleErr(UNBALPARENS);
  getToken();
  }
  else result = atom();
  return result;
}
// Получить значение числа.
private double atom() throws ParserException
{
double result =0.0;
switch(tokType) {
  case NUMBER:
    try {
      result = Double.parseDouble(token);
    } catch (NumberFormatException exc) {
      handleErr(SYNTAX);
    }
    getToken(); break;
  default:
    handleErr(SYNTAX);
    break;
  }
  return result;
}
// Handle an error.
private void handleErr(int error) throws ParserException
{
  String[] err = {
    "Syntax Error",
    "Unbalanced Parentheses",
    "No Expression Present",
    "Division by Zero"
  };
  throw new ParserException(err[error]);
}
// Получить следующую лексему. .
private void getToken()
{
  tokType = NONE;
  token = "";
  // Проверить конец выражения.
  if(expIdx == exp.length()) {
    token = EOE;
    return;
  }
```

```
    // Пропустить пробелы.
    while(expIdx < exp.length() &&
      Character.isWhitespace(exp.charAt(expIdx))) ++expIdx;
    // Пробелы в конце выражения.
    if(expIdx == exp.length()) {
      token = EOE;
      return;
    }
    if(isDelim(exp.charAt(expIdx))) { // is operator
      token += exp.charAt(expIdx); expIdx++;
      tokType = DELIMITER;
    }
    else if(Character.isLetter(exp.charAt(expIdx))) { // is variable
      while(!isDelimfexp.charAt(expIdx))) {
        token + = exp.charAt(expIdx);
        expIdx++;
        if(expIdx >= exp.length()) break;
      }
      tokType = VARIABLE;
    }
    else if(Character.isDigit(exp.charAt(expIdx))) {   // Число.
      while(!isDelim(exp.charAt(expIdx))) {
        token += exp.charAt(expIdx);
        expIdx++;
       if(expIdx >= exp.length()) break;
      }
      tokType = NUMBER;
    }
    else {               // Неизвестный символ. Выход.
      token = EOE;
      return;
    }
  }
  // Возвратать true если с является ограничителем.
  private boolean isDelim(char c)
  {
    if((" +-/*%*=(}".indexOf(c) != -1))
      return true;
    return false;
  }
}
```

Обратите внимание, что класс `ParserException` объявлен в начале кода. Он определяет исключения, которые могут произойти в процессе анализа выражения. Эти исключения должны быть обработаны в коде, который использует анализатор.

Как видно из листинга, анализатор обрабатывает следующие операторы: `+`, `–`, `*`, `/`, `%`. В дополнение к этому анализатор может произвести целочисленное возведение в степень (`^`) и унарный минус. Так же анализатор может корректно обращаться с круглыми скобками.

Для использования анализатора необходимо сначала создать объект типа `Parser`. Затем вызывается метод `evaluate()`, в который передается строка с выражением, которое необходимо анализировать. После окончания анализа и вычисления методом возвращается результат. При возникновении ошибки генерируется исключительная ситуация и выполняется обработка ошибка, при этом происходит обращение к методу `ParserException`. В листинге, приведенном ниже, показано использование анализатора.

```
// Демонстрационная версия. Импортируется java.io.*;
class PDemo {
  public static void main(String args[])
    throws IOException
  {
    String expr;
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    Parser p = new Parser();
    System.out.println("Для выхода из программы введите пустое
выражение.");
    for(;;) {
      System.out.print("Введите выражение: ");
      expr = br.readLine();
      if(expr.equals("")) break;
      try {
        System.out.println("Результат: " + p.evaluate(expr));
        System.out.println();
      }
      catch (ParserException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

Ниже приведен пример запуска программы:

```
Для выхода из программы введите пустое выражение.
Введите выражение: 10-2*3
Результат: 4.0

Введите выражение: (10-2)*3
Результат: 24.0

Введите выражение: 10/3.5
Результат: 2.857142857142857
```

Разбираемся в анализаторе

Давайте более внимательно рассмотрим анализатор. На строку, содержащую выражение для анализа, содержится ссылка в поле `exp`. Это поле устанавливается каждый раз при вызове метода `evaluate()`. Важно понимать, что анализатор может обрабатывать выражения, содержащиеся в стандартных строках Java. Например, следующие строки содержат выражения, которые анализатор может обработать:

```
"10 - 5"
"2 * 3.3 / (3.1416 * 3.3)"
```

Текущий индекс для поля `exp` сохраняется в поле `expIdx`. Когда анализатор начинает обработку выражения, поле `expIdx` устанавливается в нуль. При последовательном прохождении выражения, поле `expIdx` инкрементируется. Поле `token` содержит текущую лексему, а поле `tokType` содержит тип лексемы.

Анализ начинается с вызова метода `evaluate()`, в качестве аргумента в который передается строка, содержащая выражение для анализа. Методы от `evalExp2()` до `evalExp6()` вместе с методом `atom()` формируют рекурсивно-последовательный синтаксический анализатор. Они реализуют улучшенный набор порождающих правил

для выражения, которые уже обсуждались выше. Комментарии в начале каждого метода описывают функциональные возможности метода. В следующей версии анализатора будет добавлен метод `evalExp1()`.

Метод `handleErr()` используется для нахождения синтаксических ошибок в выражении. Методы `getToken()` и `isDelim()` разбивают выражение на составные части, как описано выше. Анализатор использует метод `getToken()` для получения очередной лексемы из выражения, начиная с первой лексемы и заканчивая последней. В зависимости от полученных лексем, выполняются различные действия.

Для более четкого представления о том, как анализатор производит обработку выражения, рассмотрим следующий пример:

```
10 - 3 * 2
```

Когда производится переход на входную точку анализатора, т.е. вызывается метод `evaluate()`, он выделяет первую лексему. Если это будет значение EOE, то метод был вызван с пустой строкой и будет сгенерированы ошибка NOEXP. Однако в нашем примере это будет лексема, представляющая число 10. После чего вызывается метод `evalExp2()`. Этот метод вызывает метод `evalExp3()`, а `evalExp3()` в свою очередь вызывает метод `evalExp4()`, который затем вызывает метод `evalExp5()`. Метод `evalExp5()` проверяет, не является ли лексема унарным минусом или унарным плюсом. В нашем случае это не так, поэтому вызывается метод `evalExp6()`. Начиная в этого места может происходить рекурсивный вызов `evalExp2()`, если выражение заключено в круглые скобка, или может быть вызван метод `atom()` для получения значения числа. Так как в нашем случае скобка не обнаружена, то вызывается метод `atom()` и возвращается значение `10`. Затем, извлекается следующая лексема и начинается повторение описанного цикла. В нашем случае будет извлечет минус (`-`), и произойдет вызов метода `evalExp2()`.

Дальнейшие действия очень важны. Так как лексема является знаком "`-`", она сохраняется в поле `op`. Затем анализатор получает следующую лексему, которая представляет число `3`, и цикл начинается вновь. Будет вызван метод `atom()`. Значение `3` будет возвращено как результирующее значение и затем будет считана лексема "`*`". Это вызовет возврат к методу `evalExp3()`, в котором получена конечная лексема `2`. После этого будет выполнен арифметический оператор и будет произведено умножение `2` на `3`. Результат будет передан в метод `evalExp2()` и будет выполнено вычитание. В результате будет получено значение `4`. Хотя на первый взгляд все это выглядит довольно сложно, но такая последовательность приводит к правильным результатам при вычислении различных выражений, в чем можете убедиться самостоятельно.

Если в процессе анализа произойдет ошибка, то вызывается метод `handleErr()`. Этот метод генерирует исключения на основе класса `ParserException`, где описаны соответствующие ошибки. Эти ошибки должны быть обработаны в том коде, в котором используется анализатор.

Такой анализатор хорошо подходит для использования в простом калькуляторе, что и продемонстрировано в предшествующей программе. Но до того, как его использовать с языками программирования, базами данных или в сложных вычислительных системах, необходимо научить его обрабатывать переменные. Как это сделать, будет показано в следующем разделе.

Добавление переменных в анализатор

Все языки программирования, многие калькуляторы и электронные таблицы используют переменные для сохранения переменных и их дальнейшего использования. Для того чтобы рассмотренный анализатор можно было использовать в указанных приложениях, необходимо его дополнить возможностями анализа переменных. Для достижения необходимых функциональных возможностей необходимо дополнить анализатор новыми фрагментами кода. Во-первых, необходимо определить сами переменные. Как уже было сказано ранее, в качестве переменных будут использоваться буквы от A до Z. Переменные будут сохраниться в массиве внутри класса `Parser`. Каждая переменная будет использовать одно число двойной точности из 26-элементного массива. Для этого добавим в класс `Parser` следующее поле.

```
// Массив для переменных.
private double vars[] = new double[26];
```

Каждый элемент массива автоматически инициализируется нулем при создании объекта типа `Parser`.

Также будет необходим метод для просмотра значений отдельных переменных. Поскольку переменные имеют имена от A до Z, то их можно легко находить по индексу массива `vars`, преобразуя значение ASCII для буквы в числовое значение, что легко делается вычитанием из значения заданного символа значения символа A. В методе `findVar()` показано, как это сделать.

```
// Возврат значения переменной.
private double findVar(String vname) throws ParserException {
  if ( !Character.isLetter(vname.charAt(0))){
    handleErr(SYNTAX);
    return 0.0;
  }
  return vars[Character.toUpperCase(vname.charAt(0))-'A']; }
```

В таком виде этот метод может принимать длинные имена переменных, таких как `A12` или `test`, но только первый символ является значимым. Легко можно изменить это положение и использовать полные имена.

Также необходимо модифицировать метод `atom()` для того, чтобы обрабатывать как числа, так и переменные. Новая версия метода приведена ниже.

```
// Получить значение числа или переменной.
private double atom() throws ParserException {
  double result = 0.0;
  switch(tokType) {
  case NUMBER: try {
    result = Double.parseDouble(token);
  }
  catch (NumberFormatException exc) {
    handleErr(SYNTAX);
  }
  getToken();
  break;
  case VARIABLE:
    result = findVar(token);
    getToken();
    break;
  default:
    handleErr(SYNTAX);
    break;
```

```
  }
  return result;
}
```

В данном методе сделано все, чтобы анализатор использовал переменные правильно, однако, здесь нет возможности присваивать переменным значения. Для того чтобы переменным можно было присваивать значения, анализатор должен обрабатывать оператор присваивания, такой как "=". Для реализации оператора присваивания необходимо в класс `Parser` добавить еще один метод с именем `evalExpl()`. Этот метод будет начинать рекурсивно-последовательный анализ. Это означает, что именно он, а не метод `evalExp2()`, должен вызываться методом `evaluate()` перед началом анализа выражения. Метод `evalExpl()` приведен ниже.

```
// Присваивание значений.
private double evalExpl() throws ParserException
{
  double result;
  int varIdx;
  int ttokType;
  String temptoken;
  if(tokType == VARIABLE) {
    // Сохранение предыдущей лексемы.
    temptoken = new String(token);
    ttokType = tokType;
    // Расчет индекса переменной.
    varIdx = Character.toUpperCase(token.charAt(0)) - 'A';
    getToken();
    if(!token.equals("=")) {
      putBack();         // Возвратить текущую лексему.
      // Восстановить предыдущую лексему - не присваивать.
      token = new String(temptoken);
      tokType = ttokType;
    }
    else {
      getToken();        // Получить следующую часть выражения.
      result = evalExp2();
      vars[varIdx] = result;
      return result;
    }
  }
  return evalExp2();
}
```

Метод `evalExpl()` необходим для просмотра вперед и определения того, должно ли быть сделано присваивание. Это происходит потому, имена переменных всегда предшествуют присваиванию, но одно имя переменной не гарантирует, что за ним следует символ присваивания. Таким образом, анализатор становится достаточно разумным, чтобы понять, что выражение `A=100` является присваиванием, а выражение `A/10` не является присваиванием. Для обеспечения этого метод `evalExpl()` считывает следующую лексему из входного потока. Если следующая лексема не является символом присваивания, то лексема возвращается во входной поток для дальнейшего использования. Для возврата вызывается метод `putBack()`, который показан ниже.

```
// Возвращение лексемы во входной поток.
private void putBack()
{
```

```
  if(token == EOE) return;
  for(int i=0; i < token.length(); i++) expIdx--;
}
```

После внесения необходимых изменений анализатор будет выглядеть так, как показано в листинге ниже.

```
/*
  Этот модуль содержит рекурсивно-последовательный
  анализатор, который использует переменные
*/

// Клас исключений для ошибок анализатора.
class ParserException extends Exception {
  String errStr;          // Описание ошибки.

  public ParserException(String str) {
    errStr = str;
  }

  public String toString() {
    return errStr;
  }
}

class Parser {
  // Здесь перечисляются типы лексем.
  final int NONE = 0;
  final int DELIMITER = 1;
  final int VARIABLE = 2;
  final int NUMBER = 3;

  // Типы синтаксических ошибок.
  final int SYNTAX = 0;
  final int UNBALPARENS = 1;
  final int NOEXP = 2;
  final int DIVBYZERO = 3;

   // Лексема, отмечающая конец выражения.
  final String EOE = "\0";

  private String exp;   // Ссылка на строку с выражением.
  private int expIdx;   // Текущий индекс.
  private String token; // Хранение текущей лексемы.
  private int tokType;  // Хранение типа текущей лексемы.

  // Массив для переменных.
  private double vars[] = new double[26];

  // Входная точка анализатора.
  public double evaluate(String expstr) throws ParserException
  {
    double result;
    exp = expstr;
    expIdx = 0;

    getToken();
    if(token.equals(EOE))
      handleErr(NOEXP);       // Выражение отсутствует.

    // Анализ и вычисление выражения.
    result = evalExp1();
```

```
    if(!token.equals(EOE))     // Последняя лексема должна быть EOE
      handleErr(SYNTAX);

    return result;
  }

  // Присваивание.
  private double evalExp1() throws ParserException
  {
    double result;
    int varIdx;
    int ttokType;
    String temptoken;

    if(tokType == VARIABLE) {
      // Сохранение предыдущей лексемы.
      temptoken = new String(token);
      ttokType = tokType;

      // Расчет индекса.
      varIdx = Character.toUpperCase(token.charAt(0)) - 'A';

      getToken();
      if(!token.equals("=")) {
        putBack();     // Возвратить текущую лексему
        // Восстановить предшествующую лексему. Нет присваивания.
        token = new String(temptoken);
        tokType = ttokType;
      }
      else {
        getToken(); // Получить следующую часть выражения.
        result = evalExp2();
        vars[varIdx] = result;
        return result;
      }
    }

    return evalExp2();
  }

  // Сложение или вычитание двух термов.
  private double evalExp2() throws ParserException
  {
    char op;
    double result;
    double partialResult;

    result = evalExp3();

    while((op = token.charAt(0)) == '+' || op == '-') {
      getToken();
      partialResult = evalExp3();
      switch(op) {
        case '-':
          result = result - partialResult;
          break;
        case '+':
          result = result + partialResult;
          break;
      }
```

```
    }
    return result;
  }

  // Перемножение или деление двух факторов.
  private double evalExp3() throws ParserException
  {
    char op;
    double result;
    double partialResult;

    result = evalExp4();

    while((op = token.charAt(0)) == '*' ||
          op == '/' || op == '%') {
      getToken();
      partialResult = evalExp4();
      switch(op) {
        case '*':
          result = result * partialResult;
          break;
        case '/':
          if(partialResult == 0.0)
            handleErr(DIVBYZERO);
          result = result / partialResult;
          break;
        case '%':
          if(partialResult == 0.0)
            handleErr(DIVBYZERO);
          result = result % partialResult;
          break;
      }
    }
    return result;
  }

  // Выполнить возведение в сстепень.
  private double evalExp4() throws ParserException
  {
    double result;
    double partialResult;
    double ex;
    int t;

    result = evalExp5();

    if(token.equals("^")) {
      getToken();
      partialResult = evalExp4();
      ex = result;
      if(partialResult == 0.0) {
        result = 1.0;
      } else
        for(t=(int)partialResult-1; t > 0; t--)
          result = result * ex;
    }
    return result;
  }

  // Определить унарные + или -.
  private double evalExp5() throws ParserException
```

```
{
  double result;
  String  op;

  op = "";
  if((tokType == DELIMITER) &&
      token.equals("+") || token.equals("-")) {
    op = token;
    getToken();
  }
  result = evalExp6();

  if(op.equals("-")) result = -result;

  return result;
}

// Обработать выражение в скобках.
private double evalExp6() throws ParserException
{
  double result;

  if(token.equals("(")) {
    getToken();
    result = evalExp2();
    if(!token.equals(")"))
      handleErr(UNBALPARENS);
    getToken();
  }
  else result = atom();

  return result;
}

// Получить значение числа или переменной.
private double atom() throws ParserException
{
  double result = 0.0;

  switch(tokType) {
    case NUMBER:
      try {
        result = Double.parseDouble(token);
      } catch (NumberFormatException exc) {
        handleErr(SYNTAX);
      }
      getToken();
      break;
    case VARIABLE:
      result = findVar(token);
      getToken();
      break;
    default:
      handleErr(SYNTAX);
      break;
  }
  return result;
}

 // Возвратить значение переменной.
private double findVar(String vname) throws ParserException
```

```
{
  if(!Character.isLetter(vname.charAt(0))){
    handleErr(SYNTAX);
    return 0.0;
  }
  return vars[Character.toUpperCase(vname.charAt(0))-'A'];
}

// Возвратить лексему во входной поток.
private void putBack()
{
  if(token == EOE) return;
  for(int i=0; i < token.length(); i++) expIdx--;
}

// Обработать ошибку.
private void handleErr(int error) throws ParserException
{
  String[] err = {
    "Syntax Error",
    "Unbalanced Parentheses",
    "No Expression Present",
    "Division by Zero"
  };

  throw new ParserException(err[error]);
}

// Obtain the next token.
private void getToken()
{
  tokType = NONE;
  token = "";

  // Проверка на окончание выражения.
  if(expIdx == exp.length()) {
    token = EOE;
    return;
  }

  // Пропустить пробелы.
  while(expIdx < exp.length() &&
    Character.isWhitespace(exp.charAt(expIdx))) ++expIdx;

  // Trailing whitespace ends expression.
  if(expIdx == exp.length()) {
    token = EOE;
    return;
  }

  if(isDelim(exp.charAt(expIdx))) { // is operator
    token += exp.charAt(expIdx);
    expIdx++;
    tokType = DELIMITER;
  }
  else if(Character.isLetter(exp.charAt(expIdx))) { // is variable
    while(!isDelim(exp.charAt(expIdx))) {
      token += exp.charAt(expIdx);
      expIdx++;
      if(expIdx >= exp.length()) break;
    }
```

```
      tokType = VARIABLE;
    }
    else if(Character.isDigit(exp.charAt(expIdx))) { // is number
      while(!isDelim(exp.charAt(expIdx))) {
        token += exp.charAt(expIdx);
        expIdx++;
        if(expIdx >= exp.length()) break;
      }
      tokType = NUMBER;
    }
    else { // Неизвестный символ. Выход.
      token = EOE;
      return;
    }
  }

  // Возвратить true если с является ограничителем.
  private boolean isDelim(char c)
  {
    if((" +-/*%^=()".indexOf(c) != -1))
      return true;
    return false;
  }
}
```

Для получения улучшенного анализатора можно модифицировать ранее написанный простой анализатор. С помощью улучшенного анализатора можно обработать выражения, подобные следующим:

```
A = 10/4
A - B
C = A * (F -21)
```

Синтаксический контроль в рекурсивно-последовательном анализаторе

При анализе выражения синтаксические ошибки представляют ситуации, при которых входное выражение не соответствует точным правилам, заложенным в анализатор. В большинстве случаев это вызывается ошибками при написании выражения, т.е. человеческим фактором. Например, следующие выражения не являются правильным и не будут обработаны созданным нами анализатором:

```
10**
8 ((10 - 5) * 9
/8
```

В первом выражении на одной строке содержится два оператора подряд, во втором — не сбалансированы скобки, и в третьем в начале выражения находится оператор деления. Поскольку такие выражения могут привести к непредсказуемым результатам, необходимо обеспечить защиту и не принимать для обработки такие выражения.

Для этого в анализаторе при возникновении любой ошибки вызывается метод `handleErr()`. В отличие от некоторых других типов анализаторов, рекурсивно-последовательный метод анализа значительно облегчает нахождение ошибок, поскольку

в основном они происходят в методах `atom()`, `findVar()` или `evalExp6()`, где производится проверка круглых скобок.

Когда вызывается метод `handleErr()`, он использует класс `ParserException`, в котором содержится описание всех ошибок. Эти исключения не перехватываются в классе `Parser`, и происходит переход в вызывающий код. Таким образом, работа анализатора немедленно прекращается при возникновении ошибки. Разумеется, такое поведение анализатора можно изменить, если возникнет необходимость.

С другой стороны, может понадобиться расширить информацию, содержащуюся в объекте типа `ParserException`. В обычном режиме этот класс сохраняет только строку, описывающую ошибку. Можно добавить более подробное описание ошибки, значение индекса, когда возникла ошибка, или другую информацию.

Аплет "Калькулятор"

Синтаксический анализатор является крайне простым в использовании и может быть добавлен почти в любое приложение. Для того чтобы лучше понять, как легко пользоваться анализатором, рассмотрим следующий пример. Написав всего несколько строк кода, можно создать полностью функциональный аплет с функциями калькулятора. Калькулятор использует два текстовых поля. Первое содержит выражение, которое необходимо вычислить, а второе отображает результат. Поле с результатом рассчитано только на чтение. Сообщения об ошибках отображаются в панели состояния. Пример выполнения (с использованием Applet Viewer) показан на рис. 2.1.

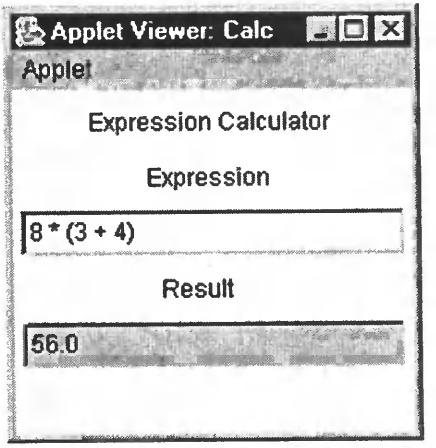


Рис. 2.1. Простой аплет с функциями калькулятора

```
// Простой Аплет с функциями калькулятора.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="Calc" width=200 height=150>
  </applet>
*/
```

```
public class Calc extends Applet
  implements ActionListener {

  TextField expText, resText;
  Parser p;

  public void init() {
    Label heading = new
          Label("Expression Calculator ", Label.CENTER);

    Label explab = new Label("Expression ", Label.CENTER);
    Label reslab = new Label("Result     ", Label.CENTER);
    expText = new TextField(24);
    resText = new TextField(24);

    resText.setEditable(false); // Поле результата только для отображения

    add(heading);
    add(explab);
    add(expText);
    add(reslab);
    add(resText);

    /* Регистрация текстового поля выражения
       для приема события.
    */
    expText.addActionListener(this);

    // Создать экземпляр анализатора.
    p = new Parser();
  }

  // Пользователь нажал клавишу <Enter>.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
    double result = 0.0;
    String expstr = expText.getText();

    try {
      if(expstr.length() != 0)
        result = p.evaluate(expstr);

// Для очистки выражения после нажатия клаваши <ENTER>
// Использовать следующее утверждение:
//    expText.setText("");

      resText.setText(Double.toString(result));

      showStatus(""); // Удалить все предыдущие сообщения об ошибках.
    } catch (ParserException exc) {
      showStatus(exc.toString());
      resText.setText("");
    }
  }
}
```

Класс `Calc` начинается с объявления трех переменных. Две первые переменные `expText` и `resText` используются для хранения ссылок на текстовые поля для выражения и отображения результата. Ссылка на анализатор хранится в переменной `p`.

В методе `init()` создаются текстовые поля и добавляются в аплет. Слушатель действий зарегистрирован для переменной `expText`, которая принадлежит аплету. Событие создается при нажатии клавиши <ENTER> после того, как введено выражение. Поскольку текстовое поле для результата `resText` должно только отображать результат, оно вызывается только для чтения с помощью метода `setEditable(false)`. При этом текстовое поле отображается с серым фоном и не реагирует на команды пользователя. Наконец, созданный экземпляр типа `Parser` присваивается переменной `p`.

При использовании калькулятора просто вводите выражение и нажимайте клавишу <ENTER>. При этом будет генерироваться событие `ActionEvent`, для обработки которого используется метод `actionPerformed()`. Это приводит к вызову метода `repaint()`, который вызывает метод `paint()`. При выполнении метода `paint()` анализатор рассчитывает значение выражения и отображает результат. Ошибка отображается в строке состояния.

Несколько экспериментов

Анализатор выражений, показанный в этой главе, будет полезен во многих приложениях, поскольку он обеспечивает дополнительные удобства без затраты значительных усилий с вашей стороны. Рассмотрим ситуацию, при которой требуется, чтобы в вашей программе пользователь мог вводить числовые значения. Например, приложение может запросить пользователя ввести число копий документа для печати. Обычно для этого используется текстовое поле, в котором происходит ожидание ввода, и затем введенный текст преобразовывается во внутренний числовой формат. В таком приближенном подходе пользователь просто вводит значение, например, 100. Но как сделать, чтобы автоматически получить по 72 копии для каждого из 9 департаментов? Придется произвести подсчет общего количества копий и ввести в текстовое поле число 648. Но если при вводе использовать анализатор, то подсчет числа копий будет производится автоматически и пользователю достаточно будет ввести выражение 9*72 и не подсчитывать самому общее число копий, что значительно удобнее. Способность анализатора рассчитывать числовое значение выражения позволит придать вашим программам завершенный и профессиональный вид и этим выгодно будет выделять их среди любительских программ. Старайтесь использовать способность анализатора рассчитывать числовые значения во всех ваших приложениях.

Как уже упоминалось ранее в этой главе, производится проверка ошибок только с минимальным уровнем анализа. Может потребоваться более детальное раскрытие составляющих ошибки. Например, можно подсветить то место в выражении, где произошла ошибка. Это позволит пользователю более оперативно найти синтаксическую ошибку и исправить ее.

В том виде, в каком разработан анализатор, он может анализировать только числовые выражения. Однако, если ввести насколько несложных дополнений, то можно заставить анализатор обрабатывать другие типы выражений, такие как строковые

выражения, выражения с пространственными координатами или с комплексными числами. Например, для того чтобы анализатор мог обрабатывать строковые выражения, необходимо сделать следующие изменения.

1. Определить новую лексему с именем `STRING`;
2. Дополнить метод `getToken()` таким образом, чтобы он мог распознавать строки;
3. Добавить новый пункт выбора в метод `atom()`, с помощью которого можно обрабатывать лексемы `STRING`.

После реализации этих шагов анализатор сможет обрабатывать строковые выражения, подобные следующим:

```
a = "one"
b = "two"
c = a + b
```

В результате переменная `c` будет представлять конкатенацию строк `a` и `b`, или строку `"onetwo"`.

ГЛАВА

3

Реализация интерпретатора языка на Java

Появлялось ли у вас когда-либо желание написать свой собственный язык программирования? Наверно, да, как и у большинства людей, занимающихся программированием. Действительно, идея создания, управления и улучшения своего собственного языка программирования очень привлекательна. Однако не все осознают, как легко и даже с наслаждением это можно делать. При этом подразумевается, что это будет не полнофункциональный компилятор высокого уровня, подобный Java, а достаточно простой интерпретатор языка, что является несравненно более легкой задачей.

Как интерпретатор, так и компилятор обрабатывают исходный код программы, но делают они это совершенно по-разному. Компилятор конвертирует исходный код программы в форму исполняемого файла. Очень часто в таких случаях исполняемый файл состоит из команд процессора и может непосредственно выполняться компьютером. В других случаях, компилятор создает код на переносимом промежуточном языке, для выполнения которого используется специальная система реального времени. Так построена среда программирования и исполнения Java. Промежуточный язык Java называется байт-кодом.

Интерпретатор работает совершенно по другому принципу. Он не транслирует исходный код в объектный файл, а во время выполнения непосредственно выполняет команды исходного кода. Хотя при этом программа выполняется значительно медленнее, т.к. одновременно происходят два процесса, компиляция и обработка машинного кода, интерпретаторы все еще широко используются по следующим причинам.

Во-первых, с интерпретаторами можно получить по настоящему интерактивный режим выполнения, т.е. можно приостановить программу, внести необходимые изменения и продолжить работу без перекомпиляции всего исходного модуля. Например, такое интерактивное построение используется в робототехнике. Во-вторых, интерпретатор языка программирования позволяет создать удобный режим отладки. В-третьих, интерпретаторы превосходно справляются с "языками сценариев", такими как язык запросов для баз данных. И в-четвертых, использование интерпретаторов позволяет выполнять программы на различных платформах. Интерпретатор можно приспособить для работы в любом новом окружении.

Иногда термин "интерпретатор" используется в несколько других ситуациях, не таких, как описано выше. Например, оригинальная система реального времени Java называется интерпретатором байт-кода. Но это не тот тип интерпретатора, который будет рассмотрен в данной главе. Система реального времени Java не использует исходный код, а обрабатывает высоко оптимизированный набор переносимых машинных команд, который создается компилятором Java. Именно поэтому система реального времени Java называется *Виртуальной Машиной Java* (Java Virtual Machine).

Для разработки интересного и полезного фрагмента кода, при разработке интерпретатора были поставлены следующие задачи: продемонстрировать рациональную элегантность языка Java. Подобно синтаксическому анализатору из главы 2, интерпретатор языка должен быть примером "чистого кода". При этом все логически сложные моменты интерпретатора должны быть наглядно отображены. Легкость, с которой интерпретатор может быть реализован на языке Java, должна еще раз показать универсальность и гибкость языка. При этом хорошо понимаемый код показывает выразительность и мощность языка Java, его удачный синтаксис и хорошо подобранный набор библиотек.

Какой язык интерпретировать?

Перед тем как начать разрабатывать интерпретатор, необходимо выбрать язык, для которого будем создавать интерпретатор. Хотя Java кажется наиболее подходящим выбором, он является достаточно большим и сложным, т.к. должен решать большой круг задач. Исходный код интерпретатора языка Java, даже если он будет решать очень ограниченный круг задач, будет слишком велик для описания в одной главе. При этом, во избежание затруднений с пониманием логики работы интерпретатора, не следует писать интерпретатор для такого сложного языка, как Java. Таким образом, лучшим выбором для написания интерпретатора с учебной целью будет компактный язык, который изначально приспособлен для работы с интерпретатором. Всем необходимым критериям отвечает одна из первых версий языка BASIC, и разработанный в данной главе интерпретатор будет полностью реализовывать возможности этого языка. В дальнейшем будем ссылаться на эту версию языка, как на Small BASIC.

Язык, подобный BASIC, был выбран по трем причинам. Во-первых, язык BASIC изначально разрабатывался для работы с интерпретатором. Поэтому относительно легко реализовать интерпретатор для такого языка. Например, оригинальная версия BASIC не поддерживает локальные переменные, рекурсивные методы, блоки, классы, перегрузку и т.д., т.е. все те возможности, которые значительно увеличивают сложность языка. При этом можно будет использовать идеи, заложенные в реализацию интерпретатора BASIC, для разработки интерпретаторов других языков, а разработанный код использовать как стартовый. Второй причиной, способствующей выбору BASIC, является то, что в результате получается сравнительно небольшой код. Наконец, синтаксис языка BASIC легко воспринимается и не составляет большого труда быстро его освоить. Даже если вы незнакомы с BASIC, у вас не будет никаких трудностей с изучением Small BASIC.

В следующем примере приведен фрагмент программы на языке Small BASIC, из которого можно понять, как доступен и нагляден язык. Даже если вы ранее не видели программ на языке BASIC, вы можете понять большинство операторов.

```
PRINT "Простая программа на языке Small BASIC"
FOR X = 1 TO 10
  GOSUB 100 NEXT
END
100 PRINT X
RETURN
```

При выполнении программа напечатает следующее.

```
Простая программа на языке Small BASIC
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
```

Хотя все ключевые слова Small BASIC интуитивно понятны, их полное объяснение будет приведено позже в этой главе.

Замечание: Small BASIC является наследником оригинальной версии BASIC, а не Visual BASIC, и был разработан всего несколько лет назад. Visual BASIC имеет значительные отличия от оригинальной версии языка BASIC. Разумеется, если вы хорошо разберетесь в работе интерпретатора, то не будет представлять большого труда внести необходимые изменения в исходный текст и заставить интерпретатор работать с различными вариантами языка BASIC.

Обзор интерпретатора

Сначала необходимо упомянуть, что интерпретатор, разработанный в этой главе, является интерпретатором исходного кода. Это означает, что исполнение программы происходит путем последовательного считывания утверждений, при этом выполняется текущий оператор. Это отличается от интерпретатора псевдокода, такого как оригинальная система реального времени Java, который интерпретирует байт-код. В нашем случае интерпретируются непосредственно коды исходной программы, при этом получается более простой цикл обработки исходного кода, так как не требуется этапа компиляции для создания промежуточного псевдокода.

Интерпретатор языка Small BASIC содержит две основные подсистемы: синтаксический анализатор выражений, который обрабатывает числовые выражения, и интерпретатор, которые непосредственно выполняет программу. За основу синтаксического анализатора выражений взят анализатор, описанный в главе 2. В нашем случае он адаптирован для обработки числовых выражений, которые встречаются в программе, в отличие от независимого выражения, как это сделано в главе 2.

Как сам интерпретатор, так и синтаксический анализатор выражений включены в один класс, названный `SBasic`. Хотя можно использовать и два отдельных класса, один для интерпретатора и один для анализатора выражений, но более эффективно будет включить.их в один класс. Причина заключается в том, что коды синтаксического анализатора выражений и интерпретатора довольно плотно переплетены. Например, оба обрабатывают один и тот же символьный массив, который содержит программу. Разделение их на два класса повлечет значительные дополнительные издержки, потерю производительности и дублирование функциональных возможностей.

Более того, поскольку анализатор и интерпретатор являются частью одной задачи по интерпретации программы, то имеет смысл создать общий механизм управления, содержащийся в одном классе.

Работа интерпретатора заключается в последовательном считывании исходного кода программы, по одной лексеме за раз. Если встречается ключевое слово, то выполняется работы, связанная с этим словом. Например, когда интерпретатор считает ключевое слово PRINT, он напечатает значение выражения, которое следует за ключевым словом. Если встречается слово GOSUB, то выполняется заданная подпрограмма. Этот процесс продолжается до конца программы. Таким образом, интерпретатор точно выполняет ту последовательность действий, которая заложена в исходной программе.

Интерпретатор для Small BASIC

Код программы для интерпретатора Small BASIC достаточно большой — больше чем обычно размещают фрагменты программ в главах книг. Однако, не обращайте внимания на его размер. Несмотря на свою длину, интерпретатор концептуально очень прост и если вы хорошо поймете его основной режим работы, то изучение остальных частей программы не составит особых сложностей.

Полностью программный код интерпретатора языка Small BASIC представлен ниже. Остальная часть главы посвящена объяснению того, как интерпретатор работает и как им пользоваться.

```
// Интерпретатор языка Small Basic.
import java.io.*;
import java.util.*;

// Класс исключений для ошибок интерпретатора.
class InterpreterException extends Exception {
  String errStr;                    // Описание ошибки
  public InterpreterException(String str) {
    errStr = str;
  }

  public String toString() {
    return errStr;
  }
}

// Класс интерпретатора языка Small Basic.
class SBasic {
  final int PROG_SIZE = 10000;     // Максимальный размер программы.

  // Типы лексем.
  final int NONE = 0;
  final int DELIMITER = 1;
  final int VARIABLE = 2;
  final int NUMBER = 3;
  final int COMMAND = 4;
  final int QUOTEDSTR = 5;

  // Типы ошибок.
  final int SYNTAX = 0;
  final int UNBALPARENS = 1;
  final int NOEXP = 2;
  final int DIVBYZERO = 3;
  final int EQUALEXPECTED = 4;
  final int NOTVAR = 5;
  final int LABELTABLEFULL = 6;
  final int DUPLABEL = 7;
  final int UNDEFLABEL = 8;
  final int THENEXPECTED = 9;
  final int TOEXPECTED = 10;
  final int NEXTWITHOUTFOR = 11;
  final int RETURNWITHOUTGOSUB = 12;
  final int MISSINGQUOTE = 13;
  final int FILENOTFOUND = 14;
  final int FILEIOERROR = 15;
  final int INPUTIOERROR = 16;
```

```
// Внутреннее представление ключевых слов Small Basic.
final int UNKNCOM = 0;
final int PRINT = 1;
final int INPUT = 2;
final int IF = 3;
final int THEN = 4;
final int FOR = 5;
final int NEXT = 6;
final int TO = 7;
final int GOTO = 8;
final int GOSUB = 9;
final int RETURN = 10;
final int END = 11;
final int EOL = 12;

// Лексема конца программы.
final String EOP = "\0";

// Коды для двойных операторов, таких как <=.
final char LE = 1;
final char GE = 2;
final char NE = 3;

// Массив для переменных.
private double vars[];

// В этом классе связываются ключевые слова с их лексемами.
class Keyword {
  String keyword; // Строка.
  int keywordTok; // Внутреннее представление.

  Keyword(String str, int t) {
    keyword = str;
    keywordTok = t;
  }
}

/* Таблица ключевых слов с их внутренним представлением.
   Все слова должны вводиться в нижнем регистре. */
Keyword kwTable[] = {
  new Keyword("print", PRINT),
  new Keyword("input", INPUT),
  new Keyword("if", IF),
  new Keyword("then", THEN),
  new Keyword("goto", GOTO),
  new Keyword("for", FOR),
  new Keyword("next", NEXT),
  new Keyword("to", TO),
  new Keyword("gosub", GOSUB),
  new Keyword("return", RETURN),
  new Keyword("end", END)
};
private char[] prog;   // Ссылка на массив с программой.
private int progIdx;   // Текущий индекс в программе.
private String token;  // Сохраняет терущую лексему.
private int tokType;   // Сохраняет тип лексемы.
private int kwToken;   // Внутреннее представление ключевого слова.

// Подержка цикла FOR.
class ForInfo {
  int var;             // Счетчик.
```

```
    double target;     // Значение.
    int loc;           // Индекс в исходном коде для цикла.
  }

  // Стек для циклов FOR.
  private Stack fStack;

  // Определяем таблицу вхождения меток.
  class Label {
    String name;      // Метка.
    int loc;          // Индекс положения метки в исходном файле.
    public Label(String n, int i) {
      name = n;
      loc = i;
    }
  }

  // Распределение меток.
  private TreeMap labelTable;

  // Стек для gosubs.
  private Stack gStack;

  // Операторы отношения.
  char rops[] = {
    GE, NE, LE, '<', '>', '=', 0
  };

  /* Создать строку, содержащую операторы отношения,
     для того, чтобы сделать их проверку более удобной. */
  String relops = new String(rops);

  // Конструктор для Small Basic.
  public SBasic(String progName)
      throws InterpreterException {
    char tempbuf[] = new char[PROG_SIZE];
    int size;

    // Загрузить программу для выполнения.
    size = loadProgram(tempbuf, progName);
    if(size != -1) {
      // Создать соответсвующий массив для хранения программы.
      prog = new char[size];

      // Копировать программу в массив.
      System.arraycopy(tempbuf, 0, prog, 0, size);
    }
  }

  // Загрузить программу.
  private int loadProgram(char[] p, String fname)
    throws InterpreterException
  {
    int size = 0;
    try {
      FileReader fr = new FileReader(fname);
      BufferedReader br = new BufferedReader(fr);
      size = br.read(p, 0, PROG_SIZE);
      fr.close();
    } catch(FileNotFoundException exc) {
      handleErr(FILENOTFOUND);
```

```
    } catch(IOException exc) {
      handleErr(FILEIOERROR);
    }

    // Если файл оканчивается маркером EOF, сделать возврат.
    if(p[size-1] == (char) 26) size--;
    return size; // Возвратить размер программы.
  }

  // Выполнить программу.
  public void run() throws InterpreterException {

    // Инициализировать для запуска новой программы.
    vars = new double[26];
    fStack = new Stack();
    labelTable = new TreeMap();
    gStack = new Stack();
    progIdx = 0;
    scanLabels();   // Найти метки в программе.
    sbInterp();     // Выполнить.
  }

  // Точка входа для интерпретатора Small Basic.
  private void sbInterp() throws InterpreterException
  {

    // Основной цикл интерпретатора.
    do {
      getToken();
      // Проверка на наличие оператора присваивания.
      if(tokType==VARIABLE) {
        putBack();    // Возвратить переменную во входной поток.
        assignment(); // Обработать оператор присваивания.
      }
      else // is keyword
        switch(kwToken) {
          case PRINT:
            print();
            break;
          case GOTO:
            execGoto();
            break;
          case IF:
            execIf();
            break;
          case FOR:
            execFor();
            break;
          case NEXT:
            next();
            break;
          case INPUT:
            input();
            break;
          case GOSUB:
            gosub();
            break;
          case RETURN:
            greturn();
            break;
          case END:
```

```
            return;
          }
      } while (!token.equals(EOP));
    }

    // Найти все метки.
    private void scanLabels() throws InterpreterException
    {
      int i;
      Object result;

      // Посмотреть, является ли первая лексема в файле меткой.
      getToken();
      if(tokType==NUMBER)
        labelTable.put(token, new Integer(progIdx));
      findEOL();
      do {
        getToken();
        if(tokType==NUMBER) {  // Должен быть номер строки.
          result = labelTable.put(token,
                                  new Integer(progIdx));
          if(result != null)
            handleErr(DUPLABEL);
        }

        // Найти следующую строку.
        if(kwToken != EOL) findEOL();
      } while(!token.equals(EOP));
      progIdx = 0;     // Переустановить индекс в начало программы.
    }

    // Найти начало следующей строки.
    private void findEOL()
    {
      while(progIdx < prog.length &&
            prog[progIdx] != '\n') ++progIdx;
      if(progIdx < prog.length) progIdx++;
    }

    // Присвоить переменной значение.
    private void assignment() throws InterpreterException
    {
      int var;
      double value;
      char vname;

      // Получить имя переменной.
      getToken();
      vname = token.charAt(0);

      if(!Character.isLetter(vname)) {
        handleErr(NOTVAR);
        return;
      }

      // Преобразовать индекс по таблице переменных.
      var = (int) Character.toUpperCase(vname) - 'A';

      // Получить знак равенства.
      getToken();
      if(!token.equals("=")) {
```

```
      handleErr(EQUALEXPECTED);
      return;
    }

    // Получить значение для присваивания.
    value = evaluate();

    // Присвоить значение.
    vars[var] = value;
  }

  // Выполнить простую версию утверждения PRINT.
  private void print() throws InterpreterException
  {
    double result;
    int len=0, spaces;
    String lastDelim = "";
    do {
      getToken();           // Получитьследующий элемент списка.
      if(kwToken==EOL || token.equals(EOP)) break;
      if(tokType==QUOTEDSTR) {   // Строка.
        System.out.print(token);
        len += token.length();
        getToken();
      }
      else {                // Выражение.
        putBack();
        result = evaluate();
        getToken();
        System.out.print(result);

        // Добавить длину выхода для полного выполнения.
        Double t = new Double(result);
        len += t.toString().length(); // Сохранить длину.
      }
      lastDelim = token;

      // Если запятая, переместиться к следующей точке.
      if(lastDelim.equals(",")) {
        // Подсчитать число пробелов для табуляции.
        spaces = 8 - (len % 8);
        len += spaces; // Добавить позицию табуляции.
        while(spaces != 0) {
          System.out.print(" ");
          spaces--;
        }
      }
      else if(token.equals(";")) {
        System.out.print(" ");
        len++;
      }
      else if(kwToken != EOL && !token.equals(EOP))
        handleErr(SYNTAX);
    } while (lastDelim.equals(";") || lastDelim.equals(","));
    if(kwToken==EOL || token.equals(EOP)) {
      if(!lastDelim.equals(";") && !lastDelim.equals(","))
        System.out.println();
    }
    else handleErr(SYNTAX);
  }
```

```
// Выполнить утверждение GOTO.
private void execGoto() throws InterpreterException
{
  Integer loc;
  getToken(); // Получить метку.

  // Найти положение метки.
  loc = (Integer) labelTable.get(token);
  if(loc == null)
    handleErr(UNDEFLABEL); // Метка не определена.
  else // start program running at that loc
    progIdx = loc.intValue();
}

// Выполнить утверждение IF.
private void execIf() throws InterpreterException
{
  double result;
  result = evaluate(); // Получить значение выражения.

  /* Если результат true (не-нуль),
     обработать IF. В противном случае перейти
     к следующей строке программы. */
  if(result != 0.0) {
    getToken();
    if(kwToken != THEN) {
      handleErr(THENEXPECTED);
      return;
    }             // Иначе утверждение будет выполнено.
  }
  else findEOL(); // Найти начало следующей строки.
}

// Выполнить утверждение FOR.
private void execFor() throws InterpreterException
{
  ForInfo stckvar = new ForInfo();
  double value;
  char vname;
  getToken(); // Считать контрольную переменную.
  vname = token.charAt(0);
  if(!Character.isLetter(vname)) {
    handleErr(NOTVAR);
    return;
  }

  // Сохранить индекс контрольной переменной.
  stckvar.var = Character.toUpperCase(vname) - 'A';

  getToken();               // Считать символ равенства.
  if(token.charAt(0) != '=') {
    handleErr(EQUALEXPECTED);
    return;
  }
  value = evaluate();       // Инициализировать.
  vars[stckvar.var] = value;
  getToken(); // Считать и отбросить TO.
  if(kwToken != TO) handleErr(TOEXPECTED);
  stckvar.target = evaluate(); // Получить значение.

  /* Если цикл может выполниться по крайней мере один раз,
```

```
      то поместить в стек. */
    if(value >= vars[stckvar.var]) {
      stckvar.loc = progIdx;
      fStack.push(stckvar);
    }
    else // В противном случае пропустить цикл полностью.
      while(kwToken != NEXT) getToken();
  }

  // Выполнить утверждение NEXT.
  private void next() throws InterpreterException
  {
    ForInfo stckvar;
    try {
      // Извлечь информацию для цикла For.
      stckvar = (ForInfo) fStack.pop();
      vars[stckvar.var]++; // Инкрементировать управляющую переменную.

      // Если сделано, вернуться.
      if(vars[stckvar.var] > stckvar.target) return;
      // Otherwise, restore the info.
      fStack.push(stckvar);
      progIdx = stckvar.loc;        // Цикл.
    } catch(EmptyStackException exc) {
      handleErr(NEXTWITHOUTFOR);
    }
  }

  // Выполнить простую форму INPUT.
  private void input() throws InterpreterException
  {
    int var;
    double val = 0.0;
    String str;
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    getToken(); // Посмотреть на присутствие приглашения.
    if(tokType == QUOTEDSTR) {
      // Если так, напечатать и проверить запятую.
      System.out.print(token);
      getToken();
      if(!token.equals(",")) handleErr(SYNTAX);
      getToken();
    }
    else System.out.print("? "); // Иначе сделать приглашение с "?".

    // Получить входную переменную.
    var =  Character.toUpperCase(token.charAt(0)) - 'A';
    try {
      str = br.readLine();
      val = Double.parseDouble(str); // Считать значение.
    } catch (IOException exc) {
      handleErr(INPUTIOERROR);
    } catch (NumberFormatException exc) {
      /* Возможно, вы захотите обработать эту ошибку
         по другому, чем другие ошибки интерпретатора. */
      System.out.println("Invalid input.");
    }
    vars[var] = val; // Сохранить.
  }
```

```
// Выполнить утверждение GOSUB.
private void gosub() throws InterpreterException
{
  Integer loc;
  getToken();

  // Найти метку для вызова.
  loc = (Integer) labelTable.get(token);
  if(loc == null)
    handleErr(UNDEFLABEL); // Метка не определена.
  else {
    // Сохранить место для возврата.
    gStack.push(new Integer(progIdx));

    // Начать выполнение с этого места.
    progIdx = loc.intValue();
  }
}

// Возврат из GOSUB.
private void greturn() throws InterpreterException
{
  Integer t;
  try {
    // Востановить индекс программы.
    t = (Integer) gStack.pop();
    progIdx = t.intValue();
  } catch(EmptyStackException exc) {
    handleErr(RETURNWITHOUTGOSUB);
  }
}

// ***************** Синтаксический анализатор ****************
// Входная точка анализатора.
private double evaluate() throws InterpreterException
{
  double result = 0.0;
  getToken();
  if(token.equals(EOP))
    handleErr(NOEXP); // Выражения нет.

  // Проанализировать и подсчитать выражение.
  result = evalExp1();
  putBack();
  return result;
}

// Обработать операторы отношения.
private double evalExp1() throws InterpreterException
{
  double l_temp, r_temp, result;
  char op;
  result = evalExp2();
  // Если конец программы, то возврат.
  if(token.equals(EOP)) return result;
  op = token.charAt(0);
  if(isRelop(op)) {
    l_temp = result;
    getToken();
    r_temp = evalExp1();
    switch(op) { // Выполнить оператор отношения.
```

```
      case '<':
        if(l_temp < r_temp) result = 1.0;
        else result = 0.0;
        break;
      case LE:
        if(l_temp <= r_temp) result = 1.0;
        else result = 0.0;
        break;
      case '>':
        if(l_temp > r_temp) result = 1.0;
        else result = 0.0;
        break;
      case GE:
        if(l_temp >= r_temp) result = 1.0;
        else result = 0.0;
        break;
      case '=':
        if(l_temp == r_temp) result = 1.0;
        else result = 0.0;
        break;
      case NE:
        if(l_temp != r_temp) result = 1.0;
        else result = 0.0;
        break;
    }
  }
  return result;
}

// Сложить или вычесть два терма.
private double evalExp2() throws InterpreterException
{
  char op;
  double result;
  double partialResult;
  result = evalExp3();
  while((op = token.charAt(0)) == '+' || op == '-') {
    getToken();
    partialResult = evalExp3();
    switch(op) {
      case '-':
        result = result - partialResult;
        break;
      case '+':
        result = result + partialResult;
        break;
    }
  }
  return result;
}

// Перемножить или разделить два фактора.
private double evalExp3() throws InterpreterException
{
  char op;
  double result;
  double partialResult;
  result = evalExp4();
  while((op = token.charAt(0)) == '*' ||
        op == '/' || op == '%') {
    getToken();
```

```
      partialResult = evalExp4();
      switch(op) {
        case '*':
          result = result * partialResult;
          break;
        case '/':
          if(partialResult == 0.0)
            handleErr(DIVBYZERO);
          result = result / partialResult;
          break;
        case '%':
          if(partialResult == 0.0)
            handleErr(DIVBYZERO);
          result = result % partialResult;
          break;
      }
    }
    return result;
  }

  // Выполнить возведение в степень.
  private double evalExp4() throws InterpreterException
  {
    double result;
    double partialResult;
    double ex;
    int t;
    result = evalExp5();
    if(token.equals("^")) {
      getToken();
      partialResult = evalExp4();
      ex = result;
      if(partialResult == 0.0) {
        result = 1.0;
      } else
        for(t=(int)partialResult-1; t > 0; t--)
          result = result * ex;
    }
    return result;
  }

  // Оценить унарные + или -.
  private double evalExp5() throws InterpreterException
  {
    double result;
    String  op;
    op = "";
    if((tokType == DELIMITER) &&
        token.equals("+") || token.equals("-")) {
      op = token;
      getToken();
    }
    result = evalExp6();
    if(op.equals("-")) result = -result;
    return result;
  }

  // Обработать выражение в круглых скобках.
  private double evalExp6() throws InterpreterException
  {
    double result;
```

```
    if(token.equals("(")) {
      getToken();
      result = evalExp2();
      if(!token.equals(")"))
        handleErr(UNBALPARENS);
      getToken();
    }
    else result = atom();
    return result;
  }

  // Получить значение числа или переменной.
  private double atom() throws InterpreterException
  {
    double result = 0.0;
    switch(tokType) {
      case NUMBER:
        try {
          result = Double.parseDouble(token);
        } catch (NumberFormatException exc) {
          handleErr(SYNTAX);
        }
        getToken();
        break;
      case VARIABLE:
        result = findVar(token);
        getToken();
        break;
      default:
        handleErr(SYNTAX);
        break;
    }
    return result;
  }

   // Возвратить значение переменной.
  private double findVar(String vname)
    throws InterpreterException
  {
    if(!Character.isLetter(vname.charAt(0))){
      handleErr(SYNTAX);
      return 0.0;
    }
    return vars[Character.toUpperCase(vname.charAt(0))-'A'];
  }

  // Возвратить лексему во входной поток.
  private void putBack()
  {
    if(token == EOP) return;
    for(int i=0; i < token.length(); i++) progIdx--;
  }

  // Обработать ошибку.
  private void handleErr(int error)
    throws InterpreterException
  {
    String[] err = {
      "Syntax Error",
      "Unbalanced Parentheses",
      "No Expression Present",
```

```
      "Division by Zero",
      "Equal sign expected",
      "Not a variable",
      "Label table full",
      "Duplicate label",
      "Undefined label",
      "THEN expected",
      "TO expected",
      "NEXT without FOR",
      "RETURN without GOSUB",
      "Closing quotes needed",
      "File not found",
      "I/O error while loading file",
      "I/O error on INPUT statement"
    };
    throw new InterpreterException(err[error]);
  }

  // Получить следующую лексему.
  private void getToken() throws InterpreterException
  {
    char ch;
    tokType = NONE;
    token = "";
    kwToken = UNKNCOM;

    // Проверить конец программы.
    if(progIdx == prog.length) {
      token = EOP;
      return;
    }

    // Пропустить пробелы.
    while(progIdx < prog.length &&
          isSpaceOrTab(prog[progIdx])) progIdx++;

    // Заключительные пробелы программы.
    if(progIdx == prog.length) {
      token = EOP;
      tokType = DELIMITER;
      return;
    }
    if(prog[progIdx] == '\r') { // handle crlf
      progIdx += 2;
      kwToken = EOL;
      token = "\r\n";
      return;
    }

    // Обработка операторов отношения.
    ch = prog[progIdx];
    if(ch == '<' || ch == '>') {
      if(progIdx+1 == prog.length) handleErr(SYNTAX);
      switch(ch) {
        case '<':
          if(prog[progIdx+1] == '>') {
            progIdx += 2;;
            token = String.valueOf(NE);
          }
          else if(prog[progIdx+1] == '=') {
            progIdx += 2;
```

```
        token = String.valueOf(LE);
      }
      else {
        progIdx++;
        token = "<";
      }
      break;
    case '>':
      if(prog[progIdx+1] == '=') {
        progIdx += 2;;
        token = String.valueOf(GE);
      }
      else {
        progIdx++;
        token = ">";
      }
      break;
  }
  tokType = DELIMITER;
  return;
}
if(isDelim(prog[progIdx])) {
  // Операгоор.
  token += prog[progIdx];
  progIdx++;
  tokType = DELIMITER;
}
else if(Character.isLetter(prog[progIdx])) {
  // Переменная или ключевое слово.
  while(!isDelim(prog[progIdx])) {
    token += prog[progIdx];
    progIdx++;
    if(progIdx >= prog.length) break;
  }
  kwToken = lookUp(token);
  if(kwToken==UNKNCOM) tokType = VARIABLE;
  else tokType = COMMAND;
}
else if(Character.isDigit(prog[progIdx])) {
  // Число.
  while(!isDelim(prog[progIdx])) {
    token += prog[progIdx];
    progIdx++;
    if(progIdx >= prog.length) break;
  }
  tokType = NUMBER;
}
else if(prog[progIdx] == '"') {
  // Строка в кавычках.
  progIdx++;
  ch = prog[progIdx];
  while(ch !='"' && ch != '\r') {
    token += ch;
    progIdx++;
    ch = prog[progIdx];
  }
  if(ch == '\r') handleErr(MISSINGQUOTE);
  progIdx++;
  tokType = QUOTEDSTR;
}
else { // Неизвестный символ, прервать выполнение.
```

```
      token = EOP;
      return;
    }
  }

  // Возвратить true если c является разделителем.
  private boolean isDelim(char c)
  {
    if((" \r,;<>+-/*%^=()".indexOf(c) != -1))
      return true;
    return false;
  }

  // Возвратить true если c является пробелом или символом табуляции.
  boolean isSpaceOrTab(char c)
  {
    if(c == ' ' || c =='\t') return true;
    return false;
  }

  // Возвратить true если c является оператором отношения.
  boolean isRelop(char c) {
    if(relops.indexOf(c) != -1) return true;
    return false;
  }

  /* Найти внутренее представление лексемы
     в таблице лексем. */
  private int lookUp(String s)
  {
    int i;

    // Преобразовать в нижний регистр.
    s = s.toLowerCase();

    // Проверить лексему в таблице.
    for(i=0; i < kwTable.length; i++)
      if(kwTable[i].keyword.equals(s))
        return kwTable[i].keywordTok;
    return UNKNCOM; // Неизвестное ключевое слово.
  }
}
```

Синтаксический анализатор для языка Small BASIC

Ядром интерпретатора является синтаксический анализатор выражений. Как уже упоминалось, анализатор, используемый для Small BASIC является модернизированной версией анализатора, описанного в главе 2. Если вы еще не познакомились с ним, то прочитайте главу 2 и изучите принцип его работы. Все основные операции, описанные в главе 2, используются и в анализаторе для Small BASIC, хотя есть и изменения.

Многие из изменений введены для того, чтобы обрабатывать синтаксис языка Small BASIC. Например, анализатор должен уметь распознавать ключевые слова языка, он не должен трактовать знак равенства (=) как оператор, но при этом понимать операторы отношений. Поэтому метод `getToken()` в значительной степени изменен для учета всех новых требований.

Еще одно отличие между анализатором из главы 2 и вновь разработанным для Small BASIC заключается в том, что новый анализатор должен вызываться в программе по мере необходимости. Например, анализатор из главы 2 ссылается на выражение, которое передается в анализатор. В версии для Small BASIC, ссылка на часть программы, подлежащей интерпретации, содержится в переменной, которая должна использоваться как интерпретатором, так и анализатором. Таким образом, исключаются дополнительные расходы, связанные с передачей ссылок. Поскольку процесс интерпретации довольно медленный, то многочисленные даже незначительные улучшения производительности очень важны.

Еще одно изменение в анализаторе вызвано тем, что версия для Small BASIC обрабатывает массив символов, а не отдельную строку. Вспомните, что в анализатор из главы 2 передается строка, содержащая необходимое выражение. Причина, по которой сделаны эти изменения, заключается в необходимости повысить производительность. Так как программа сохраняется в обычном текстовом файле, который представляет последовательность символов, а не отдельных строк, то это необходимо использовать. Поэтому когда анализатор загружает файл для исполнения, то он считывает файл в символьный массив. Хотя и можно преобразовать этот массив в набор строк, но зачем выполнять дополнительные операции и значительно терять в производительности?

Так как синтаксический анализатор Small BASIC использует те же идеи, что заложены в анализаторе из главы 2, то нет необходимости подробно рассматривать все детали построения анализатора. Однако по мере рассмотрения работы интерпретатора, несколько замечаний по работе анализатора будет сделано. Начнем со знакомства с выражениями языка Small BASIC.

Выражения языка Small BASIC

Так как интерпретатор, представленный в этой главе, должен обрабатывать выражения для Small BASIC, то должны использоваться следующие элементы.

Целые числа;
Операторы +, –, /, *, ^. =. (,) , <, >, >=, <=, <>;
Переменные.

В Small BASIC символ "^" используется для возведения в степень. Символ "=" используется как для присваивания, так и для условия равенства. Хотя в оригинальном BASIC этот оператор используется только в выражениях отношения. Неравенство определяется оператором "<>". Все эти операторы могут комбинироваться в выражениях в соответствии с правилами алгебры. Ниже приведено несколько примеров:

```
7 - 8
(100 - 5) * 14 / 6
a + b - c
10 ^ 5
A < B
```

Старшинство операторов показано ниже:

```
высший     (  )
           унарные +  -
           ^
           *  /
           +  -
низший     <  >  <=  >=  <>  =
```

Старшинство операторов учитывается при проходе слева направо. В Small BASIC сделаны следующие допущения.

- Переменные могут быть представлены только отдельными буквами. Это означает, что может быть только 26 переменных, это буквы от A до Z.
- Переменные на являются чувствительными к регистру клавиатуры. Поэтому буквы "a" и "A" будут представлять одну и ту же переменную.
- Все числовые значения представлены числами двойной точности.
- Не поддерживаются строковые переменные, хотя строковая константа, заключенная в кавычки, может использоваться для вывода сообщений на экран.

Эти допущения встроены в синтаксический анализатор для Small BASIC.

Лексемы языка Small BASIC

Ядром синтаксического анализатора для языка Small BASIC является метод `getToken()`. Этот метод является измененной версией аналогичного метода, описанного в главе 2. Вновь введенные возможности позволяют обрабатывать не только числовые выражения, но и другие элементы языка Small BASIC, такие как ключевые слова и строки.

В языке Small BASIC каждая лексема для ключевого слова имеет два формата: внешний и внутренний. Внешний формат представлен в виде текста, который используется при написании программ. Например, строка "PRINT" является внешней формой ключевого слова `PRINT`. Хотя можно сделать интерпретатор таким образом, что каждая лексема будет использовать только внешний формат (это обычно не делается из-за потери эффективности). Вместо этого, Small BASIC работает с внутренними формами лексем, которые являются просто целочисленными значениями. Например, команда `PRINT` представлена числом 1, команда `INPUT` — числом 2 и т.д. Преимуществом такого внутреннего представления является высокая скорость работы, т.к. обрабатываются только числа, а не целые строки. При этом в методе `getToken()` происходит преобразование лексем из внешнего формата во внутренний.

Метод `getToken()` для анализатора Small BASIC представлен ниже. Он последовательно просматривает программу, символ за символом.

```
// Получить следующую лексему.
private void getToken() throws InterpreterException
{
  char ch;
  tokType = NONE;
  token = "";
  kwToken = UNKNCOM;

  // Проверка конца программы.
  if(progIdx == prog.length) {
    token = EOF;
    return;
  }

  // Пропустить пробелы.
  while(progIdx < prog.length &&
    isSpaceOrTab(prog[progIdx])) progIdx++;
  // Пропустить пробелы в конце программы.
```

```
  if(progIdx == prog.length) {
    token = EOF;
    tokType = DELIMITER;
    return;
  }

  if(prog[progIdx] == '\r') {  // Обработать crlf.
    progIdx += 2;
    kwToken = EOL;
    token = "\r\n";
    return;
  }
// Проверить на наличие операторов отношения.
  ch = prog[progIdx];
  if(ch == '<' || ch == '>') {
    if(progIdx+1 == prog.length) handleErr(SYNTAX);
    switch(ch) {
      case '<':
        if(prog[progIdx+1] == '>') {
          progIdx += 2;
          token = String.valueOf(NE);
        }
        else if(prog[progIdx+1] == '=') {
          progIdx += 2;
          token = String.valueOf(LE);
        }
        else {
          progIdx++;
          token = "<";
        }
        break;
      case '>':
        if(prog[progIdx+1] == '=') {
          progIdx += 2;
          token = String.valueOf(GE);
        }
        else {
          progIdx++;
          token = ">";
        }
        break;
    }
    tokType = DELIMITER;
  return; }
  if(isDelim(prog[progIdx])) {
    // Это оператор.
    token += prog[progIdx];
    progIdx++;
    tokType = DELIMITER;
  }
  else if(Character.isLetter(prog[progIdx])) {
  // Переменная или ключевое слово.
    while(!isDelim(prog[progIdx])) {
      token += prog[progIdx];
      progIdx++;
      if(progIdx >= prog.length) break;
    }
    kwToken = lookup(token);
    if(kwToken==UNKNCOM) tokType = VARIABLE;
    else tokType = COMMAND;
  }
```

```
  else if(Character.isDigit(prog[progIdx])) {
    // Номер.
    while(!isDelim(prog[progIdx])) {
      token += prog[progIdx];
      progIdx++;
      if(progIdx >= prog.length) break;
    }
    tokType = NUMBER;
  }
  else if(prog[progIdx] == '"') { // Строка в кавычках.
    progIdx++;
    ch = prog[progIdx];
    while(ch !='"' && ch != '\r') {
      token + = ch;
      progIdx++;
      ch = prog[progIdx];
    }
    if(ch == '\r') handleErr(MISSINGQUOTE);
    progIdx++;
    tokType = QUOTEDSTR;
  }
  else { // Неизвестный символ. Выход из программы.
    token = EOF;
    return;
  }
}
```

В языке Small Basic определены следующие экземпляры переменных, которые интенсивно используются как методом `getToken()`, так и в кодах интерпретатора.

```
private char[] prog;    // Ссылка на массив программы.
private int progIdx;    // Текущий индекс в программе.
private String token;   // Хранение текущей лексемы.
private int tokType;    // Хранение типа лексемы.
private int kwToken;    // Внутреннее представление ключевого слова.
```

Программа сохраняется в символьном массиве, на который сделана ссылка с переменной `prog`. Число, определяющее положение рабочего символа программы для интерпретатора, сохраняется в переменой `progIdx`. Строковая версия лексемы содержится в переменной `token`. Тип лексемы сохраняется в `tokType`. Внутреннее представление лексемы, определяющее ключевые слова, сохраняется в переменной `kwToken`.

Синтаксический анализатор языка Small BASIC распознает пять типов лексем: `DELIMITER`, `VARIABLE`, `NUMBER`, `COMMAND` и `QUOTESTR`. Тип `DELIMITER` используется и с операторами, и со скобками. Тип `VARIABLE` используется при определении переменной в программе. Тип `NUMBER` используется для чисел. Тип `COMMAND` присваивается ключевым словам Small BASIC. Тип `QUOTESTR` представляет строку в кавычках.

Рассмотрим более подробно метод `getToken()`. При достижении конца программы, лексема принимает значение `EOF` и происходит выход из метода. При этом предшествующие пробелы пропускаются с помощью метода `isSpaceOrTab()`, который возвращает значение `true`, если его аргументом является символ пробела или символ табуляции. В данном случае нельзя использовать стандартный метод Java `Character.isWhitespace()` (который возвращает значение `true` не только для символов пробела или табуляции), поскольку Small BASIC распознает символ новой

строки как окончание. Для языка Small BASIC пробелы включают только символы пробела и табуляции. Таким образом, пробелы не могут находиться в конце программы, так как будут пропущены, поэтому метод `prog[progIdx]` будет ссылаться на число, переменную, ключевое слово, последовательность "возврат каретки/пропуск строки", на оператор или на строку в кавычках.

Если следующим символом будет возврат каретки, то переменная `kwToken` принимает значение `EOL`, а последовательность "возврат каретки/пропуск строки" сохраняется в переменной `token`. При этом переменная `tokType` принимает значение `DELIMITER`.

Также метод `getToken()` проверяет наличие операторов отношения, которые могут состоять из двух символов, таких как "<=". Такая двухсимвольная последовательность конвертируется в методе `getToken()` в их внутреннее односимвольное представление. Значения `NE`, `GE` и `LE` в языке Small Basic считаются константами. Метод `getToken()` проверяет и наличие других операторов. Если обнаружен оператор любого типа, он возвращается как строковая лексема и тип `DELIMITER` помещается в переменную `tokType`.

Если следующий символ не является оператором, то `getToken()` производит проверку на наличие буквы. Если это буква, то это может быть либо переменная, такая как A и X, либо ключевое слово, например `PRINT`. При этом используется метод `lookUp()` для проверки наличия ключевого слова. Если обнаружено ключевое слово, то метод `lookUp()` возвращает соответствующее внутреннее представление ключевого слова. Если это не ключевое слово, тогда предполагается, что лексема является переменной.

Если следующий символ является цифрой, тогда метод `getToken()` считывает число. Если следующий символ представляет кавычки, то считывается строка в кавычках. Наконец, если следующий символ не является ни одним из перечисленных, то предполагается, что достигнут конец выражения.

Остальную часть работы анализатор выполняет точно так, как и анализатор из главы 2, за исключением метода `evalExp1()`. В анализаторе из главы 2 метод `evalExp1()` используется для обработки оператора присваивания. Однако в языке BASIC присваивание производится в утверждении и не с помощью оператора. Поэтому метод `evalExp1()` не используется при анализе выражений, найденных в программе на языке Small BASIC. Вместо этого он используется для обработки операторов отношения. Если вы будете использовать интерпретатор для экспериментов с другими языками программирования, тогда может возникнуть необходимость добавить метод `evalExp0()`, который будет использоваться для обработки оператора присваивания.

Другое важное отличие между двумя анализаторами заключается в том, что анализаторе из главы 2 окончание выражения определяется по концу строки. В версии для языка Small BASIC окончание выражения определяется не только по концу строки, но и по лексеме, которая не может входить в выражение, например, ключевое слово.

Синтаксический анализатор Small BASIC распознает переменные только как буквы от A до Z. Хотя имена переменных могут быть достаточно длинными — распознается только первая буква. Можно легко изменить анализатор и заставить его распознавать только однобуквенные имена переменных.

Интерпретатор

Интерпретатор Small BASIC представляет часть кода, которая непосредственно обрабатывает программу. Необходимо отметить, что интерпретировать программы для Small BASIC довольно легко, поскольку каждое утверждение (за исключением присваивания) начинается с ключевого слова. Таким образом, работа интерпретатора заключается в получении ключевого слова в начале каждой строки и выполнении связанных с ним специфических действий. Этот процесс повторяется до тех пор, пока вся программа не будет интерпретирована. В оставшейся части главы будут подробно рассмотрены все блоки интерпретатора.

Класс InterpreterException

В начале листинга для интерпретатора находится класс `InterpreterException`. Этот класс определяет тип исключения, которое будет генерировать интерпретатор при возникновении ошибки. Код, в котором используется класс `Sbasic`, будет обрабатывать это исключение. Исключительные ситуации могут быть вызваны синтаксическими ошибками, ошибками ввода-вывода и ошибками в числовых выражениях.

Конструктор для SBasic

Листинг конструктора для класса `SBasic` приведен ниже.

```
// Конструктор для SBasic.
public SBasic(String progName)
    throws InterpreterException {
  char tempbuf[] = new char[PROG_SIZE];
  int size;
  // Загрузить программу для выполнения.
  size = loadProgram(tempbuf, progName);
  if(size != -1) {
  // Создать массив соответвующего размера для хранения программы.
    prog = new char[size];
  // Копировать программу в массив.
    System.arraycopy(tempbuf, 0, prog, 0, size);
  }
}
```

В конструктор передается имя файла с программой для Small BASIC, которую необходимо интерпретировать. При этом создается временный буфер, в который будет считан файл. Размер файла ограничивается константой `PROG_SIZE`, которая в нашем случае имеет значение 10000. То есть максимально возможное значение для файла с исходным текстом не должно превышать 10000 байт. Это определяет и максимально возможный размер программы, которая может интерпретироваться с помощью класса `SBasic`. При желании этот размер можно изменить.

Затем конструктор вызывает метод `loadProgram()`, который считывает программу из файла и возвращает ее размер в символах, или –1 в случае ошибки. Затем создается массив с размером, равным длине файла и ссылка на него помещается в переменную `prog`. Наконец, программа копируется в массив. Таким образом, размер массива, на который производится ссылка с помощью значения в переменной `prog`, будет в точности равен размеру программы.

Метод `loadProgram()` приведен ниже.

```
// Загрузка программы.
private int loadProgram(char[] p, String fname)
    throws InterpreterException {
  int size = 0;
  try {
    FileReader fr = new FileReader(fname);
    BufferedReader br = new BufferedReader(fr);
    size = br.read(p, 0, PROG_SIZE);
    fr.close();
  }
  catch(FileNotFoundException exc) {
    handleErr(FILENOTFOUND);
  }
  catch(IOException exc) {
    handleErr(FILEIOERROR);
  }
  // Если файл оканчивается маркером EOF, вернуться назад.
  if(p[size-1] == (char) 26) size--;
  return size; // return size of program }
```

Большинство из этих методов легко понимаются, но особое внимание обратим на следующие строки.

```
// Если файл оканчивается маркером EOF, вернуться назад.
  if(p[size-1] == (char) 26) size--;
```

Как отмечено в комментарии, в этой строке отбрасывается маркер EOF, который может находиться в конце файла. Как известно, некоторые текстовые редакторы ставят в конце файла маркер конца файла (End-Of-File — EOF), который обычно имеет значение 26. Другие редакторы этого не делают. Метод `loadProgram()` учитывает оба случая и удаляет маркер, если он есть.

Ключевые слова

В подмножестве языка BASIC, каким является Small BASIC, включены следующие ключевые слова.

PRINT	INPUT	IF
THEN	FOR	NEXT
TO	GOTO	GOSUB
RETURN	END	

Внутренний формат этих ключевых слов, а также маркер EOL для конца строки, объявлены как переменные с директивой `final` в классе `Sbasic`, как показано ниже.

```
// Внутреннее представление ключевых слов Small BASIC.
final int UNKNCOM = 0;
final int PRINT = 1;
final int INPUT = 2;
final int IF = 3;
final int THEN = 4;
final int FOR = 5;
final int NEXT = 6;
final int TO = 7;
final int GOTO = 8;
final int GOSUB = 9;
```

```
final int RETURN = 10;
final int END = 11;
final int EOL = 12;
```

Обратите внимание на переменную UNKNCOM. Это значение используется в методе lookUp() для ссылок на неизвестные ключевые слова.

Для более удобного преобразования ключевых слов из внешнего формата во внутренний, оба формата представлены в таблице с именем kwTable, содержащейся в классе Keyword, как показано ниже.

```
// Этот класс сопоставляет ключевые слова
// с соответствующими лексемами ключевых слов.
class Keyword {
  String keyword;     // Строковый формат.
  int keywordTok;     // Внутренее представление.
  Keyword(String str, int t) {
    keyword = str;
    keywordTok = t;
  }
}
// Таблица ключевых слов с их внутренним форматом.
// Все ключевые слова должны вводится в нижнем регистре клавиатуры.
Keyword kwTable[] = {
  new Keyword("print", PRINT),
  new Keyword("input", INPUT),
  new Keyword("if", IF),
  new Keyword("then", THEN),
  new Keyword("goto", GOTO),
  new Keyword("for", FOR),
  new Keyword("next", NEXT),
  new Keyword("to", TO),
  new Keyword("gosub", GOSUB),
  new Keyword("return", RETURN),
  new Keyword("end", END) };
```

Метод lookUp(), листинг которого приведен ниже, использует таблицу kwTable для преобразования лексем в их внутренний формат. Если соответствие не найдено, то возвращается значение UNKNCOM.

```
// Нахождение внутреннего представления ключевых слов
// в таблице лексем.
private int lookup(String s) {
  int i ;
  // Преобразовать в строчные.
  s = s.toLowerCase();
  // Найти лексему в таблице.
  for(i=0; i < kwTable.length; i++)
    if(kwTable[i].keyword.equals(s))
      return kwTable[i].keywordTok;
  return UNKNCOM; // Неизвестное ключевое слово.
}
```

Метод run()

После того как создан объект Sbasic, исходная программа начинает обрабатываться с помощью метода run(), листинг которого приведет ниже.

```
// Выполнить программу.
  public void run() throws InterpreterException {
  // Инициализировать переменные.
```

```
  vars = new double[26];
  fStack = new Stack();
  labelTable = new TreeMap();
  gStack = new Stack();
  progIdx = 0;
  scanLabels();   // Найти метки в программе.
  sbInterp();     // Выполнить. }
```

В методе `run()` распределяется память для массива, который будет содержать значения переменных, выделяется стек для цикла FOR, распределяются метки и выделяется стек для ключевого слова GOSUB. Затем переменная `progIdx`, в которой содержится положение очередного символа программы, устанавливается в 0. Эти поля устанавливаются каждый раз при вызове метода `run()`, поэтому можно повторять обработку одной и той же программы.

Затем вызывается метод `scanLabels()`, который сканирует программу в целях поиска меток. Когда метка обнаружена, ее положение сохраняется в таблице размещения меток `labelTable`. Знать расположение всех меток необходимо для повышения скорости работы программы.

Наконец, вызывается метод `sbInterp()`, который начинает обработку программы.

Метод sbInterp()

Метод `sbInterp()` начинает и контролирует выполнение программы на языке Small BASIC. Листинг метода приведен ниже.

```
// Точка входа для интерпретатора языка Small BASIC.
private void sbInterp() throws InterpreterException
{
  // Основной цикл интерпретатора.
  do {
  getToken();
  // Проверка на присваивание.
  if(tokType==VARIABLE) {
    putBack();    // Возвратить переменную во входной поток.
    assignment(); // Обработать утверждение присваивания.
  }
  else // Ключевое слово.
    switch(kwToken) {
      case PRINT:
        print();
        break;
      case GOTO:
        execGoto();
        break;
      case IF:
        execIf();
        break;
      case FOR:
        execFor();
        break;
      case NEXT:
        next();
        break;
      case INPUT:
        input();
        break;
      case GOSUB:
```

```
        gosub();
        break;
      case RETURN:
        greturn() ;
        break;
      case END:
        return;
    }
  } while (!token.equals(EOF));
}
```

Все интерпретаторы используют основной цикл, в котором считывается очередная лексема из программы и выбирается соответствующая последовательность действий для ее обработки. Интерпретатор для языка Small BASIC не содержит исключений. Основной цикл включен в метод `sbInterp()` и работа происходит так. Сначала считывается очередная лексема из программы. Предположим, что ошибок нет и лексема является переменной, тогда выполняется присваивание. В противном случае лексема может представлять или номер строки (который игнорируется), или ключевое слово. Если это ключевое слово, то происходит преобразование во внутренний формат и выбирается соответствующий пункт утверждения `case`. Каждое ключевое слово обрабатывается своим собственным методом, которые описаны ниже.

Присваивание

В традиционном языке BASIC присваивание является утверждением, а не оператором, и, соответственно, в Small BASIC присваивание также является утверждением. Общая форма присваивания в BASIC выглядит таким образом.

```
Имя_переменной = выражение
```

Утверждение присваивания интерпретируется с помощью метода присваивания `assignment()`, листинг которого приведен ниже.

```
// Присваивание значения переменной.
private void assignment() throws InterpreterException
{
  int var;
  double value;
  char vname;
  // Получить имя переменной.
  getToken();
  vname = token.charAt(0);
  if((Character.isLetter(vname)) {
    handleErr(NOTVAR);
    return;
  }
  // Преобразовать индекс с соответствии с таблицей переменных.
  var = (int) Character.toUpperCase(vname) - 'A';
  // Получить знак равенства.
  getToken();
  if(!token.equals("=")) {
    handleErr(EQUALEXPECTED);
    return;
  }
  // Получить значение для присваивания.
  value = evaluate();
  // Присвоить значение.
  vars[var] = value;
}
```

Первое, что делает метод `assignment()`, — считывает лексему из программы. Это будет переменная, которой должно быть присвоено значение. Если имя переменной является недопустимым, то будет сгенерирована ошибка. следующим считывается знак равенства. Затем вызывается метод `evaluate()` для получения значения, которое должно быть присвоено переменной. Наконец, это значение присваивается переменной. Этот метод на удивление прост и понятен, поскольку в нем производится обращение к синтаксическому анализатору выражений и методу `getToken()`, которые и выполняют большую часть рутинной работы.

Утверждение PRINT

В языке BASIC утверждение PRINT является довольно мощным и гибким. Но в этой главе не будем создавать полнофункциональный метод со всеми возможностями традиционного языка BASIC. Разработанный здесь метод поддерживает наиболее важные возможности. Общая форма записи утверждения PRINT представлена ниже.

```
PRINT список_аргументов
```

Где список аргументов представляет список выражения или строк в кавычках, разделенных запятыми или точками с запятыми. Метод `print()`, листинг которого приведен ниже, интерпретирует утверждение PRINT.

```
// Простая версия утверждения PRINT.
private void print() throws InterpreterException
{
  double result;
  int len=0, spaces;
  String lastDelim = " " ;
  do {
    getToken(); // get next list item
    if(kwToken==EOL || token.equals(EOF))
      break;
    if(tokType==QUOTEDSTR) {          // Строка.
      System.out.print(token);
      len += token.length();
      getToken();
    }
    else {                            // Выражение.
      putBack();
      result = evaluate();
      getToken();
      System.out.print(result);
      // Добавить длину выхода для полного выполнения.
      Double t = new Double(result);
      len + = t.toString().length(); // Сохранить длину.
    }
    lastDelim = token;
    // Если запятая, перейти к следующей точке.
    if(lastDelim.equals(",")) {
      // Подсчитать число пробелов для перехода.
      spaces = 8 - (len % 8);
      len += spaces; // Добавить в распределение табуляций.
      while(spaces != 0) {
        System.out.print(" ");
        spaces--;
      }
    }
    else if(token.equals(";")) {
```

```
      System.out.print(" ");
      len++;
    }
    else if(kwToken != EOL && !token.equals(EOF))
      handleErr(SYNTAX);
    }
    while (lastDelim.equals(";") || lastDelim.equals(","));
    if(kwToken==EOL || token.equals(EOF)) {
      if(!lastDelim.equals(";") && !lastDelim.equals(","))
        System.out.println();
    }
    else handleErr(SYNTAX);
}
```

Утверждение PRINT можно использовать для вывода списка переменных и строк в кавычках на экран. Если один элемент отделен от следующего с помощью точки с запятой, тогда между ними печатается один пробел. Если два элемента разделяются с помощью запятой, тогда следующий элемент будет отображаться с начала следующей позиции табуляции. Если список оканчивается запятой или точкой с запятой, то производится переход на новую строку. Ниже приведено несколько примеров допустимых утверждений PRINT.

```
PRINT X; Y; "THIS IS A STRING"
PRINT 10 / 4
PRINT
```

В последнем случае будет выведена пустая строка.

Операция вывода с помощью метода `print()` непосредственно производит отображение на экран. Однако, обратите внимание, что совместно с методом `print()` используется метод `putBack()` для возврата символа обратно в поток. Это делается потому, что метод `print()` должен просматривать символы вперед для определения строки в кавычках или числового выражения. Если это выражение, то первый терм из этого выражения должен быть возвращен во входной поток для того, чтобы синтаксический анализатор выражений мог корректно произвести анализ этого выражения.

Утверждение INPUT

В языке BASIC утверждение INPUT используется для считывания значений, вводимых с клавиатуры и присваивания этих значений переменным. Для этого утверждения применяются две формы записи. Первой будет следующая запись.

```
INPUT var
```

При этом будет отображен знак вопроса и система будет ожидать ввода. Вторая форма записи выглядит следующим образом.

```
INPUT "строка_приглашение", var
```

В этом случае будет отображаться приглашение для ввода с последующим ожиданием ввода. В обоих случаях значения, введенные пользователем, сохраняются в переменной `var`. Например, утверждение

```
INPUT "Введите ширину: ", w
```

отобразит строку "Введите ширину: " и сохранит значение, введенное пользователем, в переменной `w`. Метод `input()`, используемый в листинге, реализует утверждение INPUT.

```
// Выполнить увержденне INPUT.
private void input() throws InterpreterException
{
  int var;
  double val =0.0;
  String str;
  BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in));
  getToken(); // Есть ли строка приглашения?
  if(tokType == QUOTEDSTR) {
    // Если да, напечатать и проверить запятую.
    System.out.print(token);
    getToken();
    if(!token.equals(",")) handleErr(SYNTAX);
    getToken();
  }
  else System.out.print("? "); // В противном случае вывести "?"
  // Получить переменную.
  var = Character.toUpperCase(token.charAt(0)) - 'A';
  try {
    str = br.readLine(); // Считать значение.
    val = Double.parseDouble(str);
  }
  catch (IOException exc) {
    handleErr(INPUTIOERROR);
  }
  catch (NumberFormatException exc) {
    // Можно обработать эту ошибку отдельно от других ошибок.
    System.out.println("Invalid input.");
  }
  vars[var] = val; // Сохранить.
}
```

Все операции, выполняющиеся в этом методе довольно прозрачны, а для лучшего понимания они снабжены комментариями. Только несколько замечаний. Объект `BufferedReader` создается для считывания нажатия клавиш. Следующая полученная лексема должна быть или строкой приглашения, или именем переменной, сохраняющей введенное значение. Если это строка приглашения, то она отображается и вызывается метод `getToken()` для получения имени переменной. Затем, вводимые числа преобразовываются в значение двойной точности. Это значение присваивается переменной.

Утверждение GOTO

В традиционном языке BASIC наиболее важным для управления программой является утверждение GOTO. Утверждение GOTO сопровождается номером строки. В языке Small BASIC этот традиционный подход сохранен.

Наверно, вы знаете, что в ранних версиях языка BASIC каждая строка должна была начинаться с номера строки. Однако в Small BASIC не требуется ставить номера строк, и единственным исключением из этого является тот случай, когда на эту строку должен быть переход с помощью утверждения GOTO. Получается, что в Small BASIC номера строк являются просто метками. Общая форма записи утверждения GOTO представлена ниже.

```
GOTO номер_строки
```

Когда встречается утверждение GOTO, выполнение продолжается со строки с заданным номером.

Основная сложность, связанная с утверждением GOTO, заключается в том, что переходы должны совершаться как вперед, так и назад. Для того чтобы это можно было эффективно сделать, следует до начала выполнения просканировать программу и учесть в таблице положение каждой метки. Тогда при достижении утверждения GOTO, можно обратиться к таблице распределения меток и легко получить номер строки, на которую необходимо сделать переход.

Коллекция `TreeMap` является идеальной структурой данных для хранения меток и их местоположения, поскольку имеется связь между ключом и соответствующим значением. На коллекцию, в которой содержатся пары метка/индекс, можно обратиться по ссылке, которая является значением переменной `labelTable`. Переменная объявлена следующим образом.

```
// Коллекция для меток.
  private TreeMap labelTable;
```

Метод, с помощью которого можно просканировать программу и получить местонахождение каждой метки, называется `scanLabels()`. Этот метод вызывается в методе `run()` до того, как начнется интерпретация программы. Листинг метода `scanLabels()` приведен ниже. В конце каждой строки должен стоять маркер, для поиска которого используется метод `EOL()`.

```
// Найти все метки.
private void scanLabels() throws InterpreterException
{
  int i;
  Object result;
  // Проверить, является ли первая лексема меткой.
  getToken();
  if(tokType==NUMBER)
    labelTable.put(token, new Integer(progIdx));
  findEOL();
  do {
    getToken();
    if(tokType==NUMBER) {  // Должен быть номер строки.
      result = labelTable.put(token, new Integer(progIdx));
      if(result != null)
        handleErr(DUPLABEL);
    }
    // Найти следующую строку.
    if(kwToken != EOL) findEOL();
  }
  while(!token.equals(EOF));
  progIdx =0; // Установить индекс в начало программы.
}
// Найти начало следующей строки.
private void findEOL()
{
  while(progIdx < prog.length &&
      prog[progIdx] != '\n') ++progIdx;
  if(progIdx < prog.length) progIdx++;
}
```

Метод `scanLabels()` последовательно проверяет первую лексему на каждой строке. Если лексема является номером, то предполагается, что это номер строки, т.е.

метка. Когда метка найдена, она сохраняется в коллекции `labelTable` с помощью метода `put()`. Метод `put()` возвращает ссылку на уже существующие в таблице ключи. Таким образом, если возвращаемое значение не является нулем, то метка уже сохранена в коллекции. Это приводит к ошибке, т.к. в программе не может быть двух одинаковых меток.

Когда в программе встречается утверждение `GOTO`, то вызывается метод `execGoto()`, листинг которого приведен ниже.

```
// Выполнить утверждение GOTO.
private void execGoto() throws InterpreterException
{
  Integer loc;
  getToken(); // Получить метку для перехода.
  // Найти расположение метки.
  loc = (Integer) labelTable.get(token);
  if (loc == null)
    handleErr(UNDEFLABEL);  // Метка не определена.
  else                      // Начать программу с этого места.
    progIdx = loc.intValue(); }
```

Расположение, связанное с меткой, определяется при выполнении следующего утверждения.

```
loc = (Integer)labelTable.get(token);
```

Для коллекции `TreeMap` метод `get()` возвращает значение, связанное с ключом. Как уже говорилось, ключом является метка, а значением является индекс для массива с программой. Если метка найдена, значение присваивается полю `progIdx` и происходит выход из метода. При этом дальнейшее выполнение продолжается с места, на которое указывает значение в `progIdx` (Напомним, что в `progIdx` находится очередной индекс, который указывает на точку выполнения программы). Если метка не найдена, то происходит ошибка.

Утверждение IF

Интерпретатор языка Small BASIC выполняет простейшую форму утверждения `IF`. В Small BASIC нет ключевого слова `ELSE` (Однако если вы хорошо поймете обработку утверждения `IF`, то добавить операции по обработке слова `ELSE` будет не так уж и трудно). Утверждение `IF` записывается следующим образом.

```
IF выражение rel_op выражение THEN утверждение
```

Здесь `rel_op` заменяет один из операторов отношения. Например, `X<10` является допустимым выражением для `IF`. Утверждение за словом `THEN` выполняется только в том случае, если в результате расчета выражения получается значение `true`. Метод `execIf()`, листинг которого приведен ниже, выполняет утверждение `IF`.

```
// Выполнить утверждение IF.
private void execIff) throws InterpreterException
{
  double result;
  result = evaluate)); // Рассчитать значение выражения.
  /* Если результат расчета true (не нуль), то
     выполняется утверждение для IF. В противном случае
     переход к следующей строке программы.  */
  if(result != 0.0) {
```

```
    getToken();
    if(kwToken != THEN) {
      handleErr(THENEXPECTED);
      return;
    }
  }
  else findEOM); // Найти начало следующей строки.
}
```

Метод `execIf()` выполняется следующим образом. Сначала вычисляется значение выражения условия. Если значением выражения будет `true`, то выполняется утверждение за словом THEN, в противном случае вызывается метод `findEOL()` для поиска начала следующей строки. Обратите внимание, что если значение выражения является `true`, то происходит обращение к методу `getToken()`, т.е. происходит очередная итерация основного цикла и считывается следующая лексема. При этом происходит вычисление утверждения, входящего в состав утверждения IF. Если при расчете выражения условия будет получено значение `false`, тогда происходит поиск начала следующей строки перед тем, как продолжить основной цикл.

Цикл FOR

Реализация цикла FOR является стимулом для действительного элегантного решения проблемы. Общая форма записи цикла FOR представлена ниже.

```
FOR управ_переменная = нач_значение TO конеч_значение
  утверждения для повторения
NEXT
```

В версии для языка Small BASIC, в цикле FOR, разрешены только положительные приращения для управляющей переменной, при этом переменная инкрементируется при каждой итерации. Ключевое слово STEP не используется.

В BASIC, как и в Java, циклы могут вкладываться друг в друга. Основной проблемой, возникающей при этом, является сохранение информации, непосредственно связанной с каждый циклом (т.е. каждое заключительное NEXT должно быть связано с соответствующим FOR). Для решения этой проблемы циклы FOR реализуются с использованием механизма на основе стека. В начале каждого цикла информация о состоянии управляющей переменной, конечном значении и расположении вершины цикла в программе, помещается в стек. При каждом появлении слова NEXT эта информация выбирается из стека, обновляется управляющая переменная и сравнивается с конечным значением. Если значение управляющей переменной превышает конечное значение, выполнение цикла прерывается и управление передается на следующую за циклом строку. В противном случае обновленная информация помещается обратно в стек и выполнение цикла продолжается с самого начала. Реализация цикла подобным образом подходит не только для одиночного цикла, но и для вложенных циклов, поскольку самый внутренний NEXT всегда будет связан с самым внутренним FOR. (Последняя информация, помещенная в стек, будет самой первой, извлекаемой из стека.) Когда внутренний цикл выполнится полностью, соответствующая информация будет убрана из стека, и если существует внешний по отношению к данному цикл, то будет извлечена информация, связанная с этим циклом. Таким образом, каждый NEXT автоматически будет связан с соответствующим FOR.

Стек используется для хранения информации, связанной с циклами. Для этих целей Small BASIC использует один из классов коллекций Java: `Stack`. Информация о цикле содержится в объекте типа `ForInfo`. На стек ссылается переменная `fStack`, как показано ниже.

```
// Поддержка цикла FOR.
class ForInfo {
  int var;          // Счетчик.
  double target;    // Конечное значение.
  int   loc;        // Индекс, указывающий положение в исходном коде.
}
// Стек для цикла FOR.
private Stack fStack;
```

В объекте типа `Stack` поддерживаются методы `push()` и `pop()`, которые помещают объекты в стек и извлекают из стека, соответственно.

Два метода `execFor()` и `next()`, используемые при обработке с ключевыми словами FOR и NEXT, показаны ниже.

```
// Выполнить цикл FOR.
private void execFor() throws InterpreterException
{
  ForInfo stckvar = new ForInfo ( );
  double value;
  char vname;
  getToken();           // Считать управляющую переменную.
  vname = token.charAt(0);
  if(!Character.isLetter(vname)) {
    handleErr(NOTVAR);
    return;
  }
  // Схранить индекс в управляющей переменной.
  stckvar.var = Character.toUpperCase(vname) - 'A';
  getToken();           // Считать знак равенства.
  if(token.charAt(0) != '=') {
    handleErr(EQUALEXPECTED);
    return;
  }
  value = evaluated;    // получить начальное значение.
  vars[stckvar.var] = value;
  getToken();           // Считать и отбросить TO.
  if(kwToken != TO) handleErr(TOEXPECTED);
  stckvar.target = evaluate(); // Получить конечное значение.
  /* Если цикл может выполниться по крайней мере один раз,
     поместить в стек. */
  if(value >= vars[stckvar.var]) {
    stckvar.loc = progIdx;
    fStack.push(stckvar);
  }
  else // В противном случае пропустить цикл полностью.
    while (kwToken != NEXT) getToken();
  }

// Выполнить утверждение NEXT.
private void next() throws InterpreterException
{
  ForInfo stckvar;
  try {
    // Извлечь информацию для этого цикла For.
    stckvar = (ForInfo) fStack.pop();
```

```
      vars[stckvar.var]++;    // Инкрементировать управляющую переменную.
      // Если не сделано, возврат.
      if(vars[stckvar.var] > stckvar.target) return;
      // В противном случае восстановить информацию.
      fStack.push(stckvar);
      progIdx = stckvar.loc;  // Цикл.
    }
    catch(EmptyStackException exc) {
      handleErr(NEXTWITHOUTFOR);
    }
  }
```

Вы вполне можете понять все действия, выполняемые в этих методах, в чем помогут комментарии. Поэтому только краткое объяснение. В методе `execFor()` создается объект `ForInfo`, в котором содержатся индекс для управляющей переменной, конечное значение и текущее значение для переменной `progIdx`. Этот объект помещается в `fStack`. Затем продолжается выполнение цикла, пока не встретится ключевое слово NEXT. Когда это произойдет, объект извлекается из стека и производится сравнение текущего значения управляющей переменной с конечным значением. Если конечное значение еще не достигнуто, цикл продолжает выполняться и переменной `progIdx` присваивается значение, сохраняемое в переменной `loc`. Разумеется, это значение указывает на начало цикла. Выполнение продолжается с ключевого слова FOR и процесс повторяется. Если достигнуто конечное значение, то происходит переход на строку, следующую за ключевым словом NEXT.

В том виде, в котором разработан цикл FOR, можно в цикл вставить утверждение GOTO. Однако выход из цикла по утверждению GOTO приведет к разрушению стека, и этого необходимо избегать.

Решения проблем на основе стека для цикла FOR могут быть обобщены. Хотя в Small BASIC не используются циклы других типов, можно применить подобные процедуры к любому типу циклов, включая циклы WHILE или DO/WHILE. Как будет показано в следующем разделе, решения на основе стека могут использоваться с любыми конструкциями языка, которые допускают вложение друг в друга, включая вызов подпрограмм.

Утверждение GOSUB

Хотя язык Small BASIC не поддерживает в полной мере вызов независимых подпрограмм, он разрешает обращаться к фрагментам программы с помощью ключевого слова GOSUB. Для возврата из подпрограммы используется ключевое слово RETURN. Общая форма записи GOSUB и RETURN представлена ниже.

```
GOSUB номер_строки
  ...
  ...
  ...
номер_строки
  код_подпрограммы
RETURN
```

Вызов подпрограммы, даже в том виде, как это реализовано в Small BASIC, требует использования стека. Это объясняется теми же причинами, что и для утверждения FOR. Должна быть реализована возможность применения вложенных вызовов подпрограмм. Поскольку вызов одной подпрограммы может производиться из дру-

гой подпрограммы, то стек необходим для того, чтобы связать ключевое слово RETURN с соответствующим словом GOSUB. Для GOSUB создается объект gStack типа Stack, как показано ниже.

```
// Стек для gosubs.
private Stack gStack;
```

В объекте gStack сохраняются индексы программы. Каждый раз при считывании ключевого слова GOSUB, его индекс помещается в стек gStack. Каждый раз при обработке ключевого слова RETURN индекс положения извлекается из стека.

Листинги методов gosub() и return() приведены ниже.

```
// Выполнить GOSUB.
private void gosub() throws InterpreterException
{
  Integer loc;
  getToken();
  // Найти метку вызова.
  loc = (Integer) labelTable.get(token);
  if(loc == null)
    handleErr(UNDEFLABEL); // Метка не определена.
  else {
    // Сохранить для возврата.
    gStack.push(new Integer(progIdx));
    // Начать программу с этого места.
    progIdx = loc.intValue();
  }
}

// Возврат из GOSUB.
private void greturn() throws InterpreterException
{
  Integer t;
  try {
    // Восстановить индекс программы.
    t = (Integer) gStack.pop();
    progIdx = t.intValue();
  }
  catch(EmptyStackException exc) {
    handleErr(RETURNWITHOUTGOSUB);
  }
}
```

Кратко опишем работу утверждения GOSUB. Когда встречается GOSUB, определяется номер строки и сохраняется в переменной loc. Затем текущее значение переменной progIdx помещается в стек для GOSUB. (Это точка подпрограммы, куда будет произведен возврат из подпрограммы после ее выполнения.) Наконец, индекс, сохраненный в переменной loc, присваивается переменной progIdx. Это приводит к переходу на начало подпрограммы и ее выполнению. Когда встречается ключевое слово RETURN, из стека для GOSUB извлекается помещенное ранее значение и присваивается переменной progIdx, что приводит к выполнению программы начиная со строки, следующей за утверждением GOSUB.

Поскольку адрес возврата сохраняется в стек GOSUB, подпрограммы можно вкладывать друг в друга. Все ранее вызванные подпрограммы будут получать свой индекс, когда встретится утверждение RETURN. (Адрес возврата для всех ранее вызванных подпрограмм будет находиться в вершине стека.) Именно поэтому утверждения GOSUB могут бесконечно вкладываться друг в друга.

Утверждение END

Ключевое слово END указывает на конец программы. Но его можно и не использовать, т.к. реальное окончание последовательности кодов также приведет к прекращению выполнения. Слово END используется только для того, чтобы указать на конец программы до того, как будет достигнут конец файла. Это вызывает возврат из метода `sbInterp()`, что в конечном счете приводит к окончанию выполнения программы.

Использование Small BASIC

Для использования интерпретатора `Sbasic` сначала необходимо создать объект `Sbasic`, определив имя файла, который необходимо интерпретировать. Затем вызывается метод `run()`. Вы должны не забывать перехватывать все исключительные ситуации `InterpreterExceptions`, которые могут произойти.

В листинге ниже приведена программа, которая позволяет обрабатывать любые программы, написанные на языке Small BASIC, имя которой указано в командной строке.

```
// Демонстрация интерпретатора Small BASIC.
class SBDemo {
  public static void main(String args[]) {
    if (args.length  != 1) {
      System.out.println("Usage: sbasic <filename>");
      return;
    }

    try {
      SBasic ob = new SBasic(args[0]);
      ob.run();
    }
    catch(InterpreterException exc) {
      System.out.println(exc);
    }
  }
}
```

Для компиляции SBDemo введите следующую строку.

```
javac SBasic.java SBDemo.Java
```

Выполните SBDemo с именем программы на Small BASIC как первый аргумент командной строки. Например, если интерпретируемая программа называется TEST.BAS, то в командной строке вводится следующая информация.

```
Java SBDemo TEST.BAS
```

Ниже приведена небольшая программа на языке Small BASIC, которую можно использовать.

```
PRINT "Эта программа преобразовывает галлоны в литры."
100 GOSUB 200
  INPUT "Снова? (1 или 0): ", x
  IF x = 1 THEN GOTO 100
  END

200 INPUT "Введите галлоны: ", g
  l = g * 3.7854
  PRINT g; "галлонов равняются "; l; "литрам."
  RETURN
```

При выполнении этой программы на экране могут быть следующие строки.

```
Эта программа преобразовывает галлоны в литры.
Введите галлоны: 10
10.0 галлонов равняются 37.854 литрам.
Снова? (1 или 0): 1
Введите галлоны: 4
4.0 галлонов равняются 15.1416 литрам.
Снова? (1 или 0): 0
```

Еще несколько программ на языке Small BASIC

Ниже приведены листинги программ на языке Small BASIC, которые могут быть выполнены. Обратите внимание, что поддерживается как верхний регистр клавиатуры, так и нижний. Таким образом, ключевые слова и переменные могут вводиться как строчные и как заглавные буквы. В дополнение к программам, приведенным ниже, вы можете написать и свои собственные программы. Попробуйте писать программы с различными ошибками и наблюдайте, какие ошибки выдает интерпретатор Small BASIC.

В следующей программе используются все возможности, заложенные в язык Small BASIC

```
PRINT " Эта программа демонстрирует все возможности."
FOR X = 1 TO 100
  PRINT X; X/2, X; X*X
NEXT
GOSUB 300
PRINT "hello"
INPUT H
IF H<11 THEN GOTO 200
PRINT 12-4/2
PRINT 100
200 A = 100/2
IF A>10 THEN PRINT "Все хорошо."
PRINT A
PRINT A+34
INPUT H
PRINT H
INPUT "Это тест.", Y
PRINT H+Y
END
300 PRINT "Это подпрограмма."
    RETURN
```

Следующая программа демонстрирует вложения подпрограмм.

```
PRINT " Эта программа демонстрирует вложения подпрограмм.
" INPUT "enter a number: ", I
GOSUB 100
END
100 FOR T = 1 TO I
  X = X + I
  GOSUB 150
NEXT
RETURN
150 PRINT X;
    RETURN
```

Следующая программа иллюстрирует использование утверждения INPUT.

```
PRINT " Эта программа подсчитывает объем куба.
INPUT "Enter length of first side ", l
INPUT "Enter length of second side ", w
INPUT "Enter length of third side ", d
t = l * w * d
PRINT "Объем равняется ", t
```

Эта программа демонстрирует вложение циклов FOR.

```
PRINT " Эта программа демонстрирует вложение циклов FOR.
FOR X = 1 TO 100
  FOR Y = 1 TO 10
    PRINT X; Y; X*Y
  NEXT
NEXT
```

Следующая программа выполняет все операторы отношения.

```
PRINT "Эта программа демонстрирует все операторы отношения.
A = 10
B = 20
IF A = B THEN PRINT "A = B"
IF A <> B THEN PRINT "A <> B"
IF A < B THEN PRINT "A < B"
IF A > B THEN PRINT "A > B"
IF A >= B THEN PRINT "A >= B"
IF A <= B THEN PRINT "A <= B"
```

Улучшение и расширение интерпретатора

Довольно легко добавить утверждение в интерпретатор языка Small BASIC. Придерживайтесь общего формата, принятого для утверждений, представленных в этой главе. Для добавление различных типов переменных необходимо создать класс, в котором сохраняются типы и значения переменных, а затем использовать массив этих объектов для хранения переменных.

Создание собственного языка программирования

Улучшая и расширяя возможности языка Small BASIC, вы получите хорошую практику и основательно ознакомитесь со всеми аспектами работы интерпретатора. Хотя можно и не ограничиваться одним языком Small BASIC. Вполне реально использовать все приемы, представленные в данной главе для написания собственного интерпретатора для любого языка, включая простейшее подмножество Java. В можете создать собственный язык программирования, который учитывает ваши вкусы и требования. Действительно, структура интерпретатора, используемая для Small BASIC, является основой для дополнения языка различными структурами, с которыми вы хотите экспериментировать. Например, для того чтобы добавить цикл REPEAT/UNTIL в интерпретатор, необходимо выполнить следующие шаги.

- Добавить ключевые слова REPEAT и UNTIL и определить для них числовые значения.
- Добавить REPEAT и UNTIL в структуру выбора switch основного цикла.

- Описать методы `repeat()` и `until()`, которые будут выполняться с утверждениями `REPEAT` и `UNTIL` (Используйте методы `execFor()` и `next()` как базовые.)

Для тех читателей, которым нравится решать сложные проблемы, можно рекомендовать создать язык сценариев, с помощью которого можно выполнять различные задачи, такие как копирование или удаление файлов, компиляция и т.д. Затем создайте интерпретатор этого языка. Этот язык может стать альтернативой стандартным командным файлам. По существу, вы можете приспособить интерпретатор для создания собственных правил обработки данных.

ГЛАВА

4

Создание менеджера загрузок на Java

Как часто ваша связь с Internet прерывалась еще до того, как вы закончили работу? Если вы подключены к Internet по коммутируемому соединению, то вполне вероятно, что это случалось довольно часто и не приносило вам большой радости. Нарушение связи в момент загрузки оставляло заблокированные треки на диске, а постоянные перезагрузки тоже не доставляют удовольствия.

Но тот факт, что многие прерывания загрузки могут быть восстановлены, мало кому известен. При этом можно продолжить загрузку с той точки, где произошел разрыв, а не повторять все сначала. Однако большинство современных браузеров Internet или вообще не используют эти возможности, или воспользоваться этими возможностями довольно сложно. Но если использовать менеджер загрузки — специальную прикладную программу, которая управляет загрузкой в Internet, — то восстановление прерываний при загрузке будет выполняться очень просто.

Ядро менеджера загрузки построено таким образом, чтобы разбить весь процесс, т.е. загружать отдельные порции файла. При классическом сценарии загрузки файл загружается полностью от начала до конца. Если при этом по какой-либо причине происходит разрыв связи, то процесс загрузки прекращается и вся загруженная часть файла теряется. Напротив, при использовании менеджера загрузки вполне можно продолжить работу с того места, где произошел разрыв, и загрузить только оставшуюся часть. Но не все загрузки происходят одинаково и некоторые из них нельзя будет продолжить загружать после разрыва соединения. Все детали, при которых файлы могут или не могут продолжить загрузку, объяснены в следующем разделе.

Менеджер загрузки является не только полезной утилитой, но и превосходной иллюстрацией встроенных возможностей интерфейса прикладного программирования Java, особенно при их использовании с Internet. В предыдущих двух главах хорошо показана элегантность языка Java, в следующих трех главах будет отражена легкость, с которой программы на Java могут получить доступ к Internet. Поскольку Internet был движущей силой при разработке языка Java, то и не удивительно, что сетевые возможности Java являются незаурядными. Например, попытка создать менеджер загрузок на другом языке, таком как C++, потребует значительно больших усилий, при этом возникнет больше ошибок и трудностей, для преодоления которых необходимо будет затратить дополнительное время.

Как работает менеджер загрузок

Рассмотрим подробнее, как реально происходит загрузка в Internet.

Загрузку в Internet проще представить как обычную транзакцию клиент/сервер. Клиент (это ваш браузер) выдает запрос на загрузку файла с сервера в Internet. Соответствующий сервер отвечает посылкой файла в браузер. Для того чтобы клиент установил связь с сервером, необходимо иметь соответствующий протокол. Наиболее общими протоколами для загрузки файлов являются: протокол передачи файлов (File Transfer Protocol — FTP) и протокол передачи гипертекстовых файлов (Hypertext Transfer Protocol — HTTP). Протокол FTP обычно применяется для обмена файлов между компьютерами, когда как протокол HTTP обычно применяют при передаче Web-страниц и связанных с ними файлов (в том числе графики, звуков и т.д.). По мере

того как всемирная паутина завоевывала популярность, протокол HTTP становился все более значимым, и в настоящее время он используется и при загрузке файлов из Internet, хотя FTP тоже еще не устарел.

Чтобы не усложнять тему, в данной главе будет рассматриваться менеджер загрузок, который использует только загрузки с помощью протокола HTTP. Без сомнения, добавление поддержки для протокола FTP будет превосходным упражнением. Загрузка по протоколу HTTP может происходить в двух режимах: с восстановлением (HTTP 1.1), и без восстановления (HTTP 1.0). Различие между этими режимами состоит в способе, каким файлы могут быть затребованы с сервера. При использовании устаревшего протокола HTTP 1.0 клиент может затребовать у сервера только передачу всего файла, в то время как протокол HTTP 1.1 позволяет клиенту затребовать или передачу всего файла, или передачу только небольшой части файла.

Обзор менеджера загрузок

Менеджер загрузок использует простой, но эффективный графический интерфейс (GUI), находящийся в библиотеках Swing, поставляемых с Java. Окно менеджера загрузок показано на рис. 4.1. Использование библиотек Swing позволяет создать функциональный и современный интерфейс.

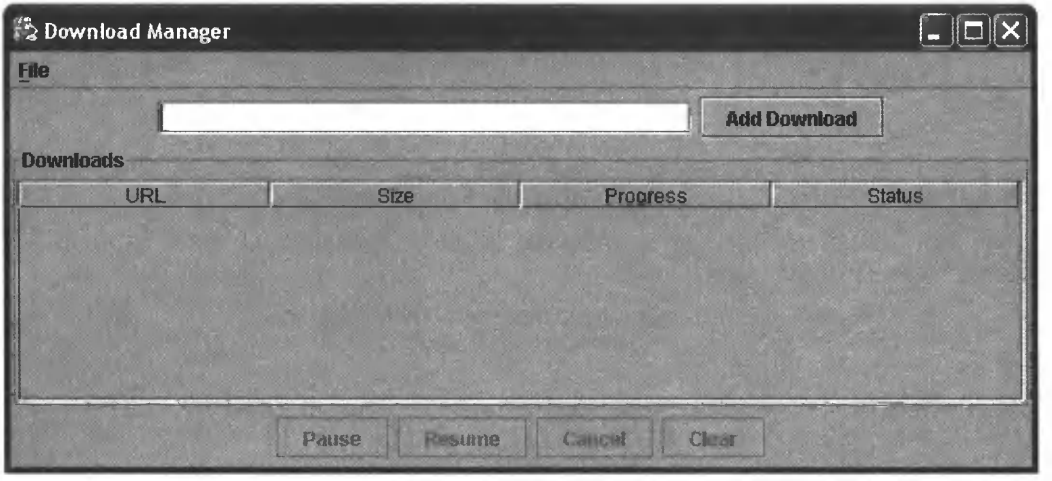


Рис. 4.1. Графический интерфейс менеджера загрузок

С помощью графического интерфейса отображается список текущих загрузок. Для каждой загрузки из списка отображается URL, размер файла в байтах, процент загруженного объема и текущее состояние. Загрузки могут находиться в одном из следующих состояний: загрузка, пауза, завершение, ошибка и отмена. С помощью GUI можно вносить дополнительные загрузки в список и изменять состояние каждой загрузки из списка. Когда выбрана отдельная загрузка, то независимо от ее состояния она может быть приостановлена, повторена, отменена или удалена из списка полностью.

Менеджер загрузок располагается в нескольких классах для естественного выделения функциональных компонентов. Это будут следующие классы: `Download`,

DownloadsTableModel, ProgressRenderer и DownloadManager. Класс Download Manager отвечает за графический интерфейс и предоставляет возможность классам DownloadsTableModel и ProgressRenderer отображать текущий список загрузок. Класс Download обеспечивает управление загрузками и отвечает за процесс загрузки файла. В следующих разделах будут подробно рассмотрены эти классы, описана из работа и взаимодействие.

Класс Download

Класс Download является основным рабочим классом в менеджере загрузок. Его главной задачей является загрузка файла и сохранение его содержимого на диске. Каждый раз, когда новая загрузка добавляется в менеджер загрузок, создается новый объект типа Download для обработки нового файла.

Менеджер загрузок способен загружать несколько файлов одновременно. Для достижения этого необходимо, чтобы каждая новая загрузка производилась независимо от других. Также необходимо иметь возможность управлять состоянием отдельной загрузки таким образом, чтобы ее характеристики отображались в GUI. Все это должен обеспечивать класс Download.

Полностью код для класса Download приведен ниже. Обратите внимание, что класс Download расширяется с помощью класса Observable и реализует интерфейс Runnable. Каждая часть класса подробно исследуется в оставшейся части главы.

```
import java.io.*;
import java.net.*;
import java.util.*;

// Этот класс загружает файл с указанного URL.
class Download extends Observable implements Runnable {
  // Max size of download buffer.
  private static final int MAX_BUFFER_SIZE = 1024;

  // Имена состояния.
  public static final String STATUSES[] = {"Downloading",
    "Paused", "Complete", "Cancelled", "Error"};

  // Коды состояния.
  public static final int DOWNLOADING = 0;
  public static final int PAUSED = 1;
  public static final int COMPLETE = 2;
  public static final int CANCELLED = 3;
  public static final int ERROR = 4;

  private URL url;         // Загрузка URL.
  private int size;        // Размер загрузки в байтах.
  private int downloaded; // Количество загруженных байтов.
  private int status;      // Текущее состояние загрузки.

  // Конструктор для класса Download.
  public Download(URL url) {
    this.url = url;
    size = -1;
    downloaded = 0;
    status = DOWNLOADING;
```

```
    // Начало загрузки.
    download();
  }

  // Получить унифицированный указатель информационного ресурса (URL).
  public String getUrl() {
    return url.toString();
  }

  // Получить размер загрузки.
  public int getSize() {
    return size;
  }

  // Получить процесс загрузки.
  public float getProgress() {
    return ((float) downloaded / size) * 100;
  }

  // Получить состояние загрузки.
  public int getStatus() {
    return status;
  }

  // Приостановить загрузку (пауза).
  public void pause() {
    status = PAUSED;
    stateChanged();
  }

  // Повторить загрузку.
  public void resume() {
    status = DOWNLOADING;
    stateChanged();
    download();
  }

  // Прекратить загрузку.
  public void cancel() {
    status = CANCELLED;
    stateChanged();
  }

  // Отметить загрузку как имеющую ошибку.
  private void error() {
    status = ERROR;
    stateChanged();
  }

  // Начать или повторить загрузки.
  private void download() {
    Thread thread = new Thread(this);
    thread.start();
  }

  // Получить имя файла из URL.
  private String getFileName(URL url) {
    String fileName = url.getFile();
    return fileName.substring(fileName.lastIndexOf('/') + 1);
  }
```

```
// Загрузить файл.
public void run() {
  RandomAccessFile file = null;
  InputStream stream = null;
  try {
    // Открыть соединение с URL.
    HttpURLConnection connection =
      (HttpURLConnection) url.openConnection();

    // Определить часть загруженного файла.
    connection.setRequestProperty("Range",
      "bytes=" + downloaded + "-");

    // Соединиться с сервером.
    connection.connect();

    // Убедиться, что код находится в диапазоне 200.
    if (connection.getResponseCode() / 100 != 2) {
      error();
    }

    // Проверить допустимую длину содержимого.
    int contentLength = connection.getContentLength();
    if (contentLength < 1) {
      error();
    }

    /* Установить размер для загрузки,
       если он еще не установлен. */
    if (size == -1) {
      size = contentLength;
      stateChanged();
    }

    // Открыть файл и найти конец файла.
    file = new RandomAccessFile(getFileName(url), "rw");
    file.seek(downloaded);
    stream = connection.getInputStream();
    while (status == DOWNLOADING) {
      /* Размер буфера в соответствии с размером
         части файла, который осталось загрузить. */
      byte buffer[];
      if (size - downloaded > MAX_BUFFER_SIZE) {
        buffer = new byte[MAX_BUFFER_SIZE];
      } else {
        buffer = new byte[size - downloaded];
      }

      // Считать с сервера в буфер.
      int read = stream.read(buffer);
      if (read == -1)
        break;

      // Записать буфер в файл.
      file.write(buffer, 0, read);
      downloaded += read;
      stateChanged();
    }

    /* Изменить статус завершения, если эта точка уже
       достигнута, поскольку загрузка закончена. */
```

```
      if (status == DOWNLOADING) {
        status = COMPLETE;
        stateChanged();
      }
    } catch (Exception e) {
      error();
    } finally {

      // Закрыть файл.
      if (file != null) {
        try {
          file.close();
        } catch (Exception e) {}
      }

      // Закрыть соединение с сервером.
      if (stream != null) {
        try {
          stream.close();
        } catch (Exception e) {}
      }
    }
  }

  // Уведомить, что статус этой загрузки был изменен.
  private void stateChanged() {
    setChanged();
    notifyObservers();
  }
}
```

Переменные загрузки

Загрузка начинается в объявления нескольких статических переменных, которые определяют различные состояния, используемые классом. Затем объявляются четыре переменные. Переменная `url` содержит URL для файла, который должен быть загружен. Переменная `size` содержит размер загружаемого файла в байтах. Переменная `downloaded` содержит количество байтов, которые уже загружены к этому времени, а переменная `status` содержит текущее состояние загрузки.

Конструктор Download

В конструктор менеджера загрузки передается ссылка на URL, с которого будет происходить загрузка, в форме объекта URL, который связан с переменной `url`. Затем в конструкторе инициализируются переменные и вызывается метод `download()`. Обратите внимание, для переменной `size` устанавливается значение `-1`, свидетельствующее о том, что размер еще не установлен.

Метод download()

В методе `download()` создается новый объект типа `Thread` и ему передается ссылка на вызывающий экземпляр `Download`. Как уже упоминалось, это необходимо сделать для того, чтобы каждая загрузка выполнялась независимо от других загрузок. Для того чтобы класс `Download` работал независимо, он должен выполняться

в своем собственном потоке. Язык Java имеет великолепную встроенную поддержку для потоков и ее использование позволяет легко связываться с потоком. Для использования потоков, класс Download просто реализует поддержку интерфейса Runnable, переопределяя метод run(). После выполнения метода download() будет создан новый экземпляр класса Thread, в который передается класс RunnableDownload, и при этом вызывается метод потока start(). Вызов метода start() приводит к тому, что будет выполняться метод run() объекта Runnable.

Метод run()

При выполнении метода run() происходит действительный процесс загрузки файла. Из-за важности метода и его больших размеров, будем рассматривать его последовательно, строка за строкой. Метод run() начинается следующими строками.

```
RandomAccessFile file = null;
InputStream stream = null;
try {
  // Открыть соединение с URL.
  HttpURLConnection connection =
    (HttpURLConnection) url.openConnection();
```

Сначала в методе устанавливаются переменные для сетевого потока, из которого будет считываться содержимое, и определяется файл, в который будет записываться содержимое. Затем открывается соединение с заданным URL с помощью вызова метода url.openConnection(). Как уже говорилось, менеджер загрузок поддерживает только протокол HTTP, поэтому соединение приводится к типу HttpURLConnection. Приведение соединения к типу HttpURLConnection позволяет использовать многие специфические преимущества соединения HTTP, такие как метод getResponseCode(). Заметьте, что обращение к url.openConnection() в действительности не приводит к соединению сервера с заданным URL, просто при этом создается экземпляр типа URLConnection, связанный с заданным URL, что позже будет использоваться для соединения с сервером.

После создания объекта HttpURLConnection устанавливаются значения для создания запроса с помощью установки свойства, как показано ниже.

```
// Определить часть файла для загрузки.
connection.setRequestProperty("Range", "bytes=" + downloaded + "-");
```

Устанавливая свойства запроса можно послать дополнительную информацию на сервер, с которого загружаются данные. В этом случае устанавливается свойство Range. Это важно, т.к. в свойстве Range указывается диапазон байтов, которые должны быть загружены с сервера. Обычно все байты файла загружаются одновременно. Однако, если загрузка была прервана или приостановлена, то нужно будет продолжить загрузку только оставшейся части. Установка свойства Range является основой работы менеджера загрузок.

Для установки свойства Range используется следующая запись.

```
номер_начального_байта – номер_конечного_байта
```

Например: "0 - 12345". Однако, указывать номер последнего байта не обязательно. Если этот байт не установлен, то он заменяется номером конечного байта для указанного файла. Для метода run() не указывается номер конечного файла,

поскольку загрузка должна производиться по полного выбора всего файла, если не будет паузы или прерывания. Это продемонстрировано в следующих строках.

```
// Подключение к серверу.
connection.connect();
// Убедиться, что код равен 200
if (connection.getResponseCode() / 100 != 2) {
  error(); }

// Проверить допустимую длину.
int contentLength = connection.getContentLength();
if (contentLength < 1) {
  error();
}
```

Метод `connection.connect()` вызывается для создания действительного соединения с сервером. Затем проверяется код ответа, возвращаемый с сервера. В протоколе HTTP указывается список кодов ответа, которые используются сервером для ответа. Все эти коды распределены в диапазоне от 100 до 200 и код 200 говорит о том, что соединение прошло успешно. Код можно получить с помощью метода `connection.getResponseCode()`. После этого результат делится на 100, и если результат деления равен 2, то соединение считается успешным.

Затем в методе `run()` вычисляется длина содержимого загрузки с помощью вызова метода `connection.getContentLength()`. Длина выражается в числе байтов для загружаемого файла. Если длина меньше чем 1, то вызывается метод `error()`. В методе `error()` обновляется переменная `status` и затем вызывается метод `stateChanged()`. Этот метод будет подробно описан несколько позже.

После получения длины содержимого, выполняется код, в котором производится проверка, было ли уже установлено значение для переменной `size`.

```
/* Установить размер загружаемого файла
   если он еще не был установлен. */
if (size == -1) {
  size = contentLength;
  stateChanged();
}
```

Как видим, вместо непосредственного присваивания длины переменной `size`, производится проверка относительно того, было ли уже присвоено значение. Это выполняется потому, что длина содержимого отражает, как много байтов сервер будет передавать. Если значение не соответствует нулю, то устанавливается начальный диапазон. Длина содержимого будет представлять только часть длины всего файла. Сначала для переменной `size` должна быть установлена длина всего загружаемого файла.

В следующих строках кода создается переменная `file` типа `RandomAccessFile`, при этом для имени файла используется переменная адреса URL, которую можно извлечь с помощью метода `getFileName()`.

```
// Открыть файл и установить индекс в конец файла.
file = new RandomAccessFile(getFileName(url), "rw");
file.seek(downloaded);
```

Для метода `RandomAccessFile` используется режим открытия файла `"rw"`, при котором из файла можно как читать, так и записывать в файл. После того как файл будет открыт, в методе `run()` индекс файла устанавливается в конец файла с помощью

метода `file.seek()`, передавая это значение переменной `downloaded`. В конце файла устанавливается его индекс на тот случай, если придется возобновлять загрузку. Если загрузка возобновляется, то будет рассчитано новое количество байтов для загрузки, но это значение не должно переписать уже существующее значение. После подготовки выходного файла будет получен дескриптор выходного потока для открытого с сервером соединения с помощью метода `connection.getInputStream()`, как показано ниже.

```
stream = connection.getInputStream();
```

Основой всех операций является следующий фрагмент с циклом `while`.

```
while (status == DOWNLOADING) {
  /* Размер буфера устанавливается в соответствии с
     остатком тех данных, которые должны быть загружены */
  byte  buffer[];
  if (size - downloaded > MAX_BUFFER_SIZE) {
    buffer = new byte[MAX_BUFFER_SIZE];
  }
  else {
    buffer = new byte[size - downloaded];
  }

  // Считать с сервера в буфер.
  int read = stream.read(buffer);
  if (read == -1)
    break;

  // Переписать буфер в файл.
  file.write(buffer, 0, read);
  downloaded += read;
  stateChanged();
}
```

Этот цикл будет продолжаться до тех пор, пока не изменится статус переменной `DOWNLOADING`. Внутри цикла создается массив для хранения загруженных байтов. Размер буфера устанавливается в соответствии с тем остатком, который необходимо загрузить. Если при этом остаток превышает значение `MAX_BUFFER_SIZE`, то именно это значение используется при установке размера буфера. В противном случае размер буфера устанавливается равным количеству оставшихся для загрузки байтов. После создания буфера нужного размера начинается загрузка с помощью метода `stream.read()`. При этом байты считываются с сервера и помещаются в буфер. Возвращается число реально считанных байтов. Если число считанных байтов равно `-1`, тогда загрузка прекращается и происходит выход из цикла. В противном случае загрузка не прекращается и байты, которые были считаны, переписываются на диск с помощью метода `file.write()`. Затем обновляется значение переменной `downloaded`, где хранится число считанных на данный момент байтов. Наконец, внутри цикла вызывается метод `stateChanged()`. Подробнее об этом будет сказано позже.

После завершения цикла происходит проверка причин завершения цикла в следующем коде.

```
/* Изменить состояние завершения, если достигнут
   этот фрагмент, т.к. загрузка закончена. */
 if (status == DOWNLOADING) {
  status = COMPLETE;
  stateChanged();
}
```

Если состояние загрузки еще определяется значением DOWNLOADING, то это означает , что выход из цикла произошел потому, что загрузка полностью завершена. В противном случае выход из цикла произошел по какой-либо другой причине.

Метод run() завершается блоками catch и finally, как показано ниже.

```
} catch (Exception e) {
  error();
} finally {
  // Закрыть файл.
  if (file != null) {
    try {
      file.close();
    } catch (Exception e) {}
  }
  // Закрыть соединение с сервером.
  if (stream != null) {
    try {
      stream.close();
    } catch (Exception e) {}
  }
}
```

Если в процессе загрузки произошла исключительная ситуация, исключения перехватываются в блоке catch и вызывается метод error(). Блок finally необходим для того, чтобы закрыть файл и поток соединения с сервером независимо от того, возникнет или нет исключительная ситуация.

Метод stateChange()

Для того чтобы менеджер загрузок отображал текущую информацию о каждой из загрузок, ему должны быть известны все изменения в процессе загрузки информации. Для получения информации об изменениях будем использовать шаблоны Observer software design. Шаблоны Observer напоминают список почтовых пересылок, где отдельные люди регистрируют принимаемую информацию. Каждый раз при поступлении нового сообщения отдельный человек занимается его обработкой. Аналогичные процессы происходят и в случае с шаблоном Observer, когда создается и регистрируется отдельный класс для приема изменений.

Класс Download реализует шаблон Observer, дополняя встроенный в Java класс утилит Observer. Дополнения, сделанные в Java при реализации интерфейса Observer, позволяют произвести регистрацию объекта для приема уведомлений. Каждый раз, когда в классе Download возникает необходимость уведомить об изменениях зарегистрированные объекты Observers, вызывается метод stateChanged(). В этом методе сначала вызывается метод setChanged() класса Observable для определения того, есть ли изменения. Затем в методе stateChanged() вызывается метод notifyObservers() класса Observable, который распространяет уведомления об изменении всем зарегистрированным объектам типа Observers.

Действия и методы Accessor

В классе Download предусмотрен значительный набор действий и методов доступа для контроля загрузки и получения данных. Вполне очевидно, что с помощью методов pause(), resume() и cancel() выполняются действия в соответствии

с перечисленными названиями: приостановка, повторение и прерывание загрузки. Аналогично, метод `error()` вызывается при возникновении ошибки, а методы доступа `getUrl()`, `getSize()`, `getProgress()` и `getStatus()` возвращают текущее состояние соответствующих параметров.

Класс ProgressRenderer

Класс `ProgressRenderer` по существу является небольшой утилитой, которая используется для формирования изображения текущего процесса загрузки в графическом интерфейсе на основе объекта `JTable`. Обычно объект `JTable` формирует изображение для каждой из своих ячеек в текстовом виде. Однако в большинстве случаев гораздо лучше отображать данные в более наглядном виде, чем обычный текст. Так, в случае с менеджером загрузки вполне возможно отображать колонку **Progress** как индикатор выполнения. В классе `ProgressRenderer`, представленном ниже, реализована такая возможность. Обратите внимание, что используется класс `JProgressBar` и реализуется интерфейс `TableCellRenderer`.

```
import  java.awt.*;
import  javax.swing.*;
import  javax.swing.table.*;
// Этот класс формирует изображение JProgressBar в отдельной
// ячейке таблицы. Класс ProgressRenderer наследует
// класс JProgressBar и реализует интерфейс TableCellRenderer
class ProgressRenderer extends JProgressBar
    implements TableCellRenderer {
  // Конструктор для ProgressRenderer.
  public ProgressRenderer(int min, int max) {
    super(min, max);
  }
/* Возвращает объект JProgressBar как исходный
   для данной ячейки. */
  public Component getTableCellRendererComponent(
    JTable table, Object value, boolean isSelected,
    boolean hasFocus, int row, int column) {
    // Установить процент завершенности для JProgressBar
    setValue((int) ((Float) value).floatValue());
    return this;
  }
}
```

Класс `ProgressRenderer` использует преимущества класса `JTable` из подмножества классов `Swing`, которые имеют системы формирования изображений и могут использовать сменные модули для прорисовки ячеек таблицы. Для того чтобы использовать эти возможности, в классе `ProgressRenderer` сначала реализуется интерфейс `TableCellRenderer` из библиотеки `Swing`. Затем экземпляр `ProgressRenderer` регистрируется в экземпляре `JTable`, сообщая экземпляру `JTable`, какие ячейки должны быть прорисованы с помощью дополнительного модуля.

Релизация интерфейса `TableCellRenderer` требует, чтобы в классе был переопределен метод `getTableCellRendererComponent()`. Метод `getTableCellRenderer Component()` вызывается каждый раз, когда экземпляр `JTable` должен прорисовать ячейку, для которой этот класс зарегистрирован. В этом методе передается несколько переменных, но в нашем случае используется только переменная `value`.

Переменная value содержит данные для ячейки, которая должна быть прорисована, и она передается в метод JProgressBar's setValue(). Метод getTableCell RendererComponent() завершается возвратом ссылки на необходимый класс. Это срабатывает, потому как класс ProgressRenderer является производным от JProgessbar, который в свою очередь является наследником класса Component библиотеки AWT.

Класс DownloadsTableModel

Класс DownloadsTableModel содержит список загрузок для менеджера загрузок и соответствующие исходные данные для графического интерфейса экземпляра JTable.

Класс DownloadsTableModel приведен ниже. Обратите внимание, что он наследует класс AbstractTableModel и реализует интерфейс Observer.

```
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

// Этот класс управляет загрузкой данных таблицы.
class DownloadsTableModel extends AbstractTableModel
  implements Observer
{
  // Это имена для колонок таблицы.
  private static final String[] columnNames = {"URL", "Size",
    "Progress", "Status"};

  // Это классы для каждого значения колонки.
  private static final Class[] columnClasses = {String.class,
    String.class, JProgressBar.class, String.class};

  // Список для загрузки
  private ArrayList downloadList = new ArrayList();

  // Добавить новую загрузку в таблицу.
  public void addDownload(Download download) {
    // Зарегистрировать для уведомления об изменениях в загрузке.
    download.addObserver(this);
    downloadList.add(download);
    // Ввести уведомление в строку таблицу.
    fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
  }

  // Получить загрузку для указанной строки.
  public Download getDownload(int row) {
    return (Download) downloadList.get(row);
  }

  // Удалить загрузку из списка.
  public void clearDownload(int row) {
    downloadList.remove(row);
    // Удаление уведомления в строке таблицы.
    fireTableRowsDeleted(row, row);
  }

  // Получить счетчик колонок таблицы.
```

```
  public int getColumnCount() {
    return columnNames.length;
  }

  // Получить имя колонки.
  public String getColumnName(int col) {
     return columnNames[col];
  }

  // Получить класс колонки.
  public Class getColumnClass(int col) {
    return columnClasses[col];
  }

  // Получить счетчик строк таблицы.
  public int getRowCount() {
    return downloadList.size();
  }

  // Получить значение для указанной комбинации строки и колонки.
  public Object getValueAt(int row, int col) {
    Download download = (Download) downloadList.get(row);
    switch (col) {
      case 0:                          // URL.
        return download.getUrl();
      case 1:                          // Размер.
        int size = download.getSize();
        return (size == -1) ? "" : Integer.toString(size);
      case 2:                          // Процесс.
        return new Float(download.getProgress());
      case 3:                           // Состояние.
        return Download.STATUSES[download.getStatus()];
    }
    return "";
  }

  /* Обновить при вызове, когда класс Download уведомляет
     экземпляры Observers об изменениях */
  public void update(Observable o, Object arg) {
    int index = downloadList.indexOf(o);
    // Запустить обновление уведомлений для строки таблицы.
    fireTableRowsUpdated(index, index);
  }
}
```

Класс `DownloadsTableModel` по существу является утилитой, используемой экземпляром `JTable` для управления данными в таблице. Когда экземпляр `JTable` инициализируется, ему передается экземпляр `DownloadsTableModel`. Затем в экземпляре `JTable` выполняется вызов нескольких методов из экземпляра `DownloadsTableModel` заполнения данными. Метод `getColumnCount()` вызывается для определения количества колонок в таблице. Аналогично, метод `getColumnName()` возвращает количество строк таблицы. Метод `getColumnName()` возвращает имя колонки, присвоенное идентификатором. Метод `getDownload()` принимает идентификатор строки и возвращает соответствующий экземпляр `Download` из списка. Остальные методы класса `DownLoadsTableModel`, которые являются более сложными для понимания, рассмотрены ниже.

Метод addDownload()

Метод addDownload(), представленный ниже, добавляет новый объект Download в список управляемых загрузок и, соответственно, новую строку в таблицу.

```
// Добавить новый объект download в таблицу.
public void addDownload(Download download) {
// Зарегистрировать для получения уведомления при изменениях в загрузке.
  download.addObserver(this);
  downloadList.add(download);
  // Отправка уведомления в таблицу о вставке строки.
  fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
}
```

Это метод сначала регистрирует себя совместно с новой загрузкой как наблюдатель для приема уведомлений об изменении. Затем экземпляр Download добавляется во внутренний список управляемых загрузок. Наконец, уведомление о вставке новой строки генерируется для отображения в таблице информации о том, что добавлена новая строка.

Метод clearDownload()

Метод clearDownload(), приведенный ниже, удаляет загрузку из списка управляемых загрузок.

```
// Удалить загрузку из списка.
public void clearDownload(int row) {
  downloadList.remove(row);
  fireTableRowsDeleted(row, row);
}
```

После удаления загрузки из внутреннего списка, уведомление об удалении строки генерируется для отображения в таблице.

Метод getColumnClass()

Метод getColumnClass(), приведенный ниже, возвращает тип данных, отображаемых в указанной колонке.

```
// Получить тип колонки.
public Class getColumnClass(int col) {
  return columnClasses[col];
}
```

Все колонки отображаются как текст (т.е. объекты типа String), за исключением колонки Progress, которая отображается как индикатор выполнения (объект типа JProgressBar).

Метод getValueAt()

Метод getValueAt(), приведенный ниже, вызывается для получения текущего значения, которое должно отображаться в отдельной ячейке таблицы.

```
// Получить значение для указанной комбинации строки и колонки.
public Object getValueAt(int row, int col) {
  Download download = (Download) downloadList.get(row);
  switch (col) {
```

```
    case 0:              // URL
      return download.getUrl();
    case 1:              // Размер.
      int size = download.getSize();
      return (size == -1) ? "" : Integer.toString(size) ;
    case 2:              // Процент.
      return new Float(download.getProgress());
    case 3:              // Состояние.
      return Download.STATUSES[download.getStatus()];
  }
  return "";
}
```

В этом методе сначала находится объект `Download`, соответствующий указанной строке. Затем используется указанная колонка для определения необходимого для возвращения значения.

Метод update()

Метод `update()` представлен ниже. Он удовлетворяет требованиям контракта интерфейса `Observer`, позволяя классу `DowloadsTableModel` принимать уведомления от объектов `Download` при их изменениях.

```
/* Обновление происходит, когда объект Download уведомляется
   соответствующим наблюдателем об изменениях. */
public void update(Observable o, Object arg) {
  int index = downloadList.indexOf(o);
  // Отправка уведомления в таблицу об обновлении строки.
  fireTableRowsUpdated(index, index);
}
```

В этот метод передается ссылка на объект `Download`, который был изменен, в виде объекта `Observable`. Затем производится поиск индекса загрузки в списке загрузок и этот индекс используется в дальнейшем для создания уведомления об обновлении строки, которое передается в таблицу. Таблица перерисует строку с заданным индексом, отображая изменения.

Класс DownloadManager

Рассмотрим один из лежащих в основе менеджеров загрузок вспомогательных классов — `DownloadManager`. Класс `DownloadManager` способен создать и запустить графический интерфейс менеджера загрузок. В этом классе объявлен метод `main()`, с которого и начинается выполнение. В методе `main()` создается новый экземпляр класса `DownloadManager` и затем вызывается метод `show()`, который приводит к отображению графического интерфейса на экране.

Класс `DownloadManager` приведен ниже. Обратите внимание, что от наследует класс `Jframe` и реализует интерфейс `Observer`. Раздел посвящен подробному рассмотрению класса `DownloadManager`.

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
```

```
// Менеджер загрузок.
public class DownloadManager extends JFrame
  implements Observer
{
  // Добавить текстовое поле для загрузки.
  private JTextField addTextField;

  // Модель данных таблицы загрузки.
  private DownloadsTableModel tableModel;
  // Таблица для списка загрузок.
  private JTable table;
  // Кнопки для управления выбранной загрузкой.
  private JButton pauseButton, resumeButton;
  private JButton cancelButton, clearButton;
  // Выбранная загрузка.
  private Download selectedDownload;
  // Флаг, указывающий на очистку элемента таблицы.
  private boolean clearing;
  // Конструктор для менеджера загрузок.
  public DownloadManager()
  {
    // Создать заголовок приложения.
    setTitle("Download Manager");
    // Установить размеры окна.
    setSize(640, 480);
    // Обработать события закрытия окна.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        actionExit();
      }
    });
    // Подготовить меню файлов.
    JMenuBar menuBar = new JMenuBar();
    JMenu fileMenu = new JMenu("File");
    fileMenu.setMnemonic(KeyEvent.VK_F);
    JMenuItem fileExitMenuItem = new JMenuItem("Exit",
      KeyEvent.VK_X);
    fileExitMenuItem.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionExit();
      }
    });
    fileMenu.add(fileExitMenuItem);
    menuBar.add(fileMenu);
    setJMenuBar(menuBar);
    // Добавить панель.
    JPanel addPanel = new JPanel();
    addTextField = new JTextField(30);
    addPanel.add(addTextField);
    JButton addButton = new JButton("Add Download");
    addButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionAdd();
      }
    });
    addPanel.add(addButton);
    // Создать таблицу загрузок.
    tableModel = new DownloadsTableModel();
    table = new JTable(tableModel);
    table.getSelectionModel().addListSelectionListener(new
```

```
    ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
      tableSelectionChanged();
    }
  });
  // Разрешить выбирать единовременно только одну строку.
  table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

  // Создать ProgressBar как формирователь изображения для
  // колонки Progress
  ProgressRenderer renderer = new ProgressRenderer(0, 100);
  renderer.setStringPainted(true); // show progress text
  table.setDefaultRenderer(JProgressBar.class, renderer);

  // Установить высоту строк таблицы, достаточной для
  // использования JProgressBar.
  table.setRowHeight(
    (int) renderer.getPreferredSize().getHeight());

  // Создать панель загрузки.
  JPanel downloadsPanel = new JPanel();
  downloadsPanel.setBorder(
    BorderFactory.createTitledBorder("Downloads"));
  downloadsPanel.setLayout(new BorderLayout());
  downloadsPanel.add(new JScrollPane(table),
    BorderLayout.CENTER);

  // Создать панель кнопок.
  JPanel buttonsPanel = new JPanel();
  pauseButton = new JButton("Pause");
  pauseButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionPause();
    }
  });
  pauseButton.setEnabled(false);
  buttonsPanel.add(pauseButton);
  resumeButton = new JButton("Resume");
  resumeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionResume();
    }
  });
  resumeButton.setEnabled(false);
  buttonsPanel.add(resumeButton);
  cancelButton = new JButton("Cancel");
  cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionCancel();
    }
  });
  cancelButton.setEnabled(false);
  buttonsPanel.add(cancelButton);
  clearButton = new JButton("Clear");
  clearButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionClear();
    }
  });
  clearButton.setEnabled(false);
  buttonsPanel.add(clearButton);
```

```
    // Добавить панели для отображения.
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(addPanel, BorderLayout.NORTH);
    getContentPane().add(downloadsPanel, BorderLayout.CENTER);
    getContentPane().add(buttonsPanel, BorderLayout.SOUTH);
  }
  // Выход из программы.
  private void actionExit() {
    System.exit(0);
  }

  // Добавить новую загрузку.
  private void actionAdd() {
    URL verifiedUrl = verifyUrl(addTextField.getText());
    if (verifiedUrl != null) {
      tableModel.addDownload(new Download(verifiedUrl));
      // Переустановить добавленные текстовые поля.
      addTextField.setText("");
    } else {
      JOptionPane.showMessageDialog(this,
        "Invalid Download URL", "Error",
        JOptionPane.ERROR_MESSAGE);
    }
  }

  // Проверить URL загрузки.
  private URL verifyUrl(String url) {
    // Разрешены только HTTP URLs.
    if (!url.toLowerCase().startsWith("http://"))
      return null;

    // Проверить формат URL.
    URL verifiedUrl = null;
    try {
      verifiedUrl = new URL(url);
    } catch (Exception e) {
      return null;
    }

    // Убедиться, что URL указывает на файл.
    if (verifiedUrl.getFile().length() < 2)
      return null;
    return verifiedUrl;
  }

  // Вызов, когда изменяется выбор строки таблицы.
  private void tableSelectionChanged() {
    /* Отменить регистрацию для приема уведомлений
       от последней выбранной загрузки. */
    if (selectedDownload != null)
      selectedDownload.deleteObserver(DownloadManager.this);

    /* Если загрузка не очищается, то
       установить выбранную загрузку и зарегистрировать
       для приема уведомлений от нее. */
    if (!clearing) {
      selectedDownload =
        tableModel.getDownload(table.getSelectedRow());
      selectedDownload.addObserver(DownloadManager.this);
      updateButtons();
```

```
    }
  }

  // Приостановить выбранную загрузку.
  private void actionPause() {
    selectedDownload.pause();
    updateButtons();
  }

  // Возобновить выбранную загрузку.
  private void actionResume() {
    selectedDownload.resume();
    updateButtons();
  }

  // Отменить выбранную загрузку.
  private void actionCancel() {
    selectedDownload.cancel();
    updateButtons();
  }

  // Очистить выбранную загрузку.
  private void actionClear() {
    clearing = true;
    tableModel.clearDownload(table.getSelectedRow());
    clearing = false;
    selectedDownload = null;
    updateButtons();
  }

  /* Обновить состояние кнопки на основе
     статуса текущей выбранной загрузки. */
  private void updateButtons() {
    if (selectedDownload != null) {
      int status = selectedDownload.getStatus();
      switch (status) {
        case Download.DOWNLOADING:
          pauseButton.setEnabled(true);
          resumeButton.setEnabled(false);
          cancelButton.setEnabled(true);
          clearButton.setEnabled(false);
          break;
        case Download.PAUSED:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(true);
          clearButton.setEnabled(false);
          break;
        case Download.ERROR:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
          break;
        default: // COMPLETE or CANCELLED
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(false);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
      }
    } else {
```

```
      // No download is selected in table.
      pauseButton.setEnabled(false);
      resumeButton.setEnabled(false);
      cancelButton.setEnabled(false);
      clearButton.setEnabled(false);
    }
  }

  /* Обновление вызывается, когда загрузка
     уведомляет своего наблюдателя об изменениях. */
  public void update(Observable o, Object arg) {
    // Обновить кнопки, если выбранная загрузка была изменена.
    if (selectedDownload != null && selectedDownload.equals(o))
      updateButtons();
  }

  // Запустить менеджер загрузок.
  public static void main(String[] args) {
    DownloadManager manager = new DownloadManager();
    manager.show();
  }
}
```

Переменные класса DownloadManager

Класс `DownloadManager` начинается с объявления нескольких переменных, большинство из которых содержит ссылки на элементы управления графического интерфейса. Переменная `selectedDownload` содержит ссылку на объект `Download`, отображаемый выбранной строкой таблицы. Наконец, булева переменная `clearing` является флагом, сигнализирующем о том, происходит ли в настоящее время очистка в таблице загрузок.

Конструктор класса DownloadManager

Когда создается экземпляр `DownloadManager`, все элементы управления графического интерфейса инициализируются в конструкторе. Хотя в конструкторе достаточно много кода, большая его часть не представляет трудностей для понимания. Небольшое обсуждение приводится ниже.

Сначала заголовок окна устанавливается с помощью метода `setTitle()`. Затем вызывается метод `setSize()` для задания размеров окна в пикселях. После этого добавляется слушатель для окна с помощью метода `addWindowListener()`, в который передается объект `WindowAdapter`, который переопределяет обработчик события `windowClosing()`. В этом обработчике при закрытии окна приложения вызывается метод `actionExit()`. Следующим в окно приложения добавляется строка меню с названием "File". Затем производится настройка панели **Add**, которая содержит текстовое поле и кнопки. Объект `ActionListener` связывается с кнопкой **Add Download** таким образом, чтобы метод `actionAdd()` вызывался при каждом щелчке на кнопке.

Затем создается таблица загрузок. Объект `ListSelectionListener` добавляется в таблицу таким образом, чтобы при выборе строки в таблице вызывался метод `tableSelectionChanged()`. Режим выбора в таблице также изменяется таким

образом, чтобы только одна строка могла быть выбрана единовременно, при этом устанавливается значение `ListSelectionModel.SINGLE_SELECTION`. Ограничивая выбор только одной строкой, мы тем самым упрощаем логику работы с кнопками, которые должны располагаться на графическом интерфейсе. После чего создается экземпляр класса `ProgressRenderer`, который регистрируется в таблице для обработки колонки "Progress". Высота строк таблицы подгоняется под высоту объекта `ProgressRenderer` с помощью вызова метода `table.setRowHeight()`. После того как таблица сформирована и уточнены все детали, она помещается в объект `JScrollPane` для того, чтобы создать прокрутку, а затем добавляется в панель.

Наконец, создается панель кнопок. На этой панели размещаются кнопки **Pause**, **Resume**, **Cancel** и **Clear**. Для каждой из кнопок создается объект `ActionListener`, который вызывает соответствующий метод действия после щелчка на кнопке. После создания всех панелей они добавляются в окно.

Метод verifyUrl()

Метод `verifyUrl()` вызывается методом `actionAdd()` каждый раз, когда загрузка добавляется в менеджер загрузок. Листинг метода `verifyUrl()` приведен ниже.

```
// Проверить URL загрузки.
private URL verifyUrl(String url) {
  // Разрешены только адреса HTTP.
  if (!url.toLowerCase().startsWith("http://"))
    return null;
  // Проверить формат URL.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  }
  catch (Exception e) {
    return null;
  }
  // Убедиться, что URL указывает на файл.
  if (verifiedUrl.getFile().length() < 2)
    return null;
  return verifiedUrl;
}
```

В этом методе сначала проверяется, что URL представляет HTTP URL, т.к. поддерживается только формат HTTP. Затем URL проверяется на возможность создания экземпляра класса URL.

Если URL неправильно сформирован, то будет сгенерирована исключительная ситуация. Наконец, в этом методе проверяется, что по указанному адресу находится именно файл.

Метод tableSelectionChanged()

Метод `tableSelectionChanged()`, представленный ниже, вызывается каждый раз, когда производится выбор строки в таблице загрузок.

```
// Вызывается при выборе строки.
private void tableSelectionChanged() {
  /* Отменить регистрацию для приема уведомлений
     от последней выбранной загрузки. */
```

```
  if (selectedDownload != null)
    selectedDownload.deleteObserver(DownloadManager.this);
  /* Если загрузка не очищается, то установить выбранную загрузку
     и зарегистрировать для приема уведомлений от нее. */
  if (!clearing) {
    selectedDownload =
      tableModel.getDownload(table.getSelectedRow());
    selectedDownload.addObserver(DownloadManager.this);
    updateButtons();
  }
}
```

Метод начинается с проверки того, что переменная `selectedDownload` не равняется `null`. Если переменная не равняется `null`, то объект `DownloadManager` удаляет себя как наблюдатель за загрузками, чтобы больше не принимать сообщения с уведомлениями. Затем проверяется флаг очистки. Если значение флага не равно `true`, тогда сначала переменная `selectedDownload` обновляется для новой загрузки в соответствии с выбранной строкой. После чего объект `DownloadManager` регистрируется как тип `Observer` для вновь выбранного объекта `Download`. Наконец, вызывается метод `updateButtons()` для обновления состояния кнопок на основе статуса выбранной загрузки.

Метод updateButtons()

Метод `updateButtons()` обновляет состояние всех кнопок на панели кнопок в соответствии со статусом выбранной загрузки. Листинг метода приведен ниже.

```
/* Обновить состояние кнопок в соответствии
   со статусом выбранной загрузки. */
private void updateButtons() {
  if (selectedDownload != null) {
    int status = selectedDownload.getStatus();
    switch (status) {
      case Download.DOWNLOADING:
        pauseButton.setEnabled(true);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(true);
        clearButton.setEnabled(false);
        break;
      case Download.PAUSED:
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(true);
        cancelButton.setEnabled(true);
        clearButton.setEnabled(false);
        break;
      case Download.ERROR:
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(true);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(true);
        break;
      default: // COMPLETE или CANCELLED
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(true);
    }
  }
```

```
    else {
      // В таблице загрузка не выбрана.
      pauseButton.setEnabled(false);
      resumeButton.setEnabled(false);
      cancelButton.setEnabled(false);
      clearButton.setEnabled(false);
    }
  }
```

Если в таблице не выбрано ни одной из загрузок, все кнопки находятся в не активизированном состоянии и отображаются приглушенным цветом. Но как только выбирается одна из загрузок, кнопки становятся активными и состояние каждой кнопки отражает статус объекта `Download`: `DOWNLOADING`, `PAUSED`, `ERROR`, `COMPLETE` или `CANCELLED`.

Обработка событий

Каждый элемент управления из графического интерфейса `DownloadManager` регистрирует объект `ActionListener`, который вызывает соответствующий обработчик события. Объект `ActionListeners` срабатывает каждый раз, когда какое-либо событие происходит с элементом управления. Например, при щелчке на кнопке срабатывает объект `ActionListeners` и при этом уведомляется каждый из зарегистрированных для кнопки объектов `ActionListeners`. Вы можете заметить сходство между работой объекта `ActionListeners` и шаблонами `Observer`, обсуждавшимися ранее. Это происходит потому, что они работают по одной и той же схеме.

Компиляция и запуск менеджера загрузок

Для компиляции программы `DownloadManager` введите следующие строки.

```
javac DownloadManager.Java DownloadsTableModel.Java
ProgressRenderer.Java  Download.Java
```

Для запуска менеджера загрузок введите следующую строку.

```
javaw DownloadManager
```

Менеджер загрузок легко использовать. Сначала в текстовое поле вверху экрана введите URL файла, который вы хотите загрузить. Например, для того чтобы загрузить файл с именем `0072224207_code.zip` с сайта `www.osborne.com`, введите следующую строку.

```
http://www.osborne.com/products/0072224207/0072224207_code.zip
```

Этот файл содержит коды для книги Херба: *Полный справочник по Java 2.*

После добавления загрузки в менеджер загрузок, можно выбрать загрузку из таблицы и управлять ею. После выбора загрузки ее можно приостановить (pause), отменить (cancel), возобновить (resume) и очистить (clear). На рис. 4.2 показан графический интерфейс менеджера загрузок.

Улучшение менеджера загрузок

Менеджер загрузок, представленный на рис. 4.2, является полностью функциональным, с возможностями приостанавливать и возобновлять загрузку. При этом может выполняться несколько загрузок одновременно. Но все же некоторые улуч-

шения вполне возможны. Например, поддержка прокси-сервера, поддержка протоколов FTP и HTTPS или улучшение графического интерфейса пользователя с помощью режима "перетащи и опусти". Особенно привлекательным может быть введение режима планирования, что позволит запускать загрузку в определенное время, возможно, в середине ночи, когда системные ресурсы не перегружены.

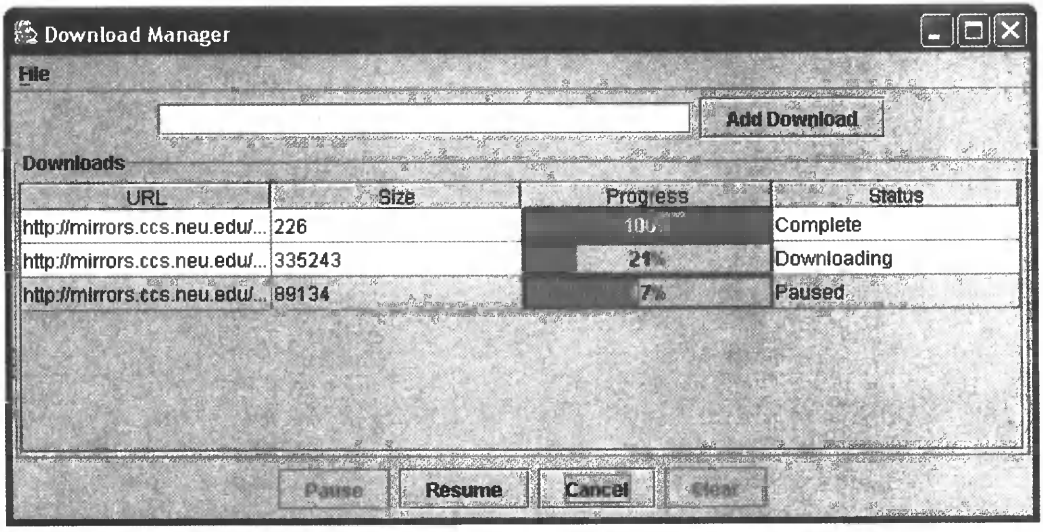


Рис. 4.2. Графический интерфейс менеджера загрузок

Обратите внимание, что технические приемы, продемонстрированные в этой главе, не ограничиваются только решением вопросов загрузки файлов. Можно найти много других способов использовать этот код. Например, многие программы распространяются через Internet в два приема. Сначала передается небольшая часть программы, которая может быть загружена очень быстро. Эта часть содержит небольшой менеджер загрузки для последующей загрузки второй части, которая обычно довольно большая. Это очень удобное решение, т.к. размеры приложений постоянно возрастают и при этом увеличивается вероятность потери данных при передаче. Можно попытаться приспособить разработанный выше менеджер загрузок для этих целей.

ГЛАВА

5

Создание почтового клиента на Java

Как хорошо известно, есть два основных пути использования Internet: просмотр Web-сайтов и электронная почта. Хотя работа с Web-сайтами внешне более привлекательна, электронная почта является именно той услугой, которая необходима большинству пользователей. Поскольку электронная почта предлагает недорогую и высокоскоростную альтернативу обычной почте и намного больше возможностей, чем обычный телефон, она становится средством коммуникации, которым в настоящее время пользуются почти все.

Понимая привлекательность электронной почты и ее важность для Internet, многие программисты охотно занимаются созданием приложений для нее. И в самом деле, способность принимать и отправлять сообщения непосредственно под управлением вашей программы представляется очень привлекательной. Например, рассмотрим приложение, которое производит мониторинг температуры в коммерческом холодильнике. При этом желательно, чтобы приложение автоматически отправляло сообщение по электронной почте директору завода, если температура повысится до плюсовой. Кроме того, возможно, возникнет необходимость в создании специального клиента электронной почты для директора завода, чтобы этот клиент просматривал поступающие сообщения и поднимал сигнал тревоги при поступлении сообщений от холодильника.

В дополнение к перечисленным возможностям, есть и другая причина для прямого управления электронной почтой — безопасность. Возможности электронной почты возросли многократно по сравнению с тем периодом, когда передавались только текстовые сообщения. Сегодня электронная почта поддерживает такие насыщенные форматы, как HTML, с развитой графикой и звуком. Также поддерживается присоединение файлов для передачи данных совместно с сообщением. Неприятность заключается в том, что такие возможности и широкий спектр форматов позволяют недобросовестным программистам рассматривать электронную почту как средство для проведения экспериментов с компьютерными вирусами, "троянскими" программами и т.д.

Хотя сегодня коммерческие программы электронной почты предлагают средства для борьбы с вирусами, появление защитных средств приводит к созданию все более изощренных вирусов. Проблему можно решить, если использовать электронную почту только для приема текстовых сообщений и не обращаться к дополнительным возможностям. Простой клиент электронной почты не будет формировать изображения HTML, не будет отображать графику и не будет позволять просматривать фотографии. Также не будет возможности проигрывать звуковые файлы, что позволит избежать потенциальной угрозы. Конечно, такое решение неприемлемо в целом, но может быть полезно в отдельных случаях, особенно в тех ситуациях, когда необходимо исключить вирусные атаки.

Каковы бы ни были причины, а возможность иметь полный контроль над электронной почтой стоит того, чтобы затратить на это время и хорошо в этом разобраться. Хотя в общем случае разработка приложений электронной почты достаточно сложна, Java со своим интерфейсом JavaMail API значительно облегчают создание таких приложений. В этой главе будет рассмотрен простой клиент электронной почты, принимающий только текстовые сообщения. Создание такого клиента электронной почты преследует две цели. Во-первых, это будет полностью функциональное приложение электронной почты для приема и отсылки текстовой части сообщений.

Никаких действий с остальным содержимым сообщения оно производить не будет. Во-вторых, при его разработке будут продемонстрированы технические приемы, используемые при приеме и отсылке сообщений, что можно будет использовать в ваших дальнейших проектах.

"За кулисами" электронной почты

Если заглянуть "за кулисы" электронной почты, то там можно увидеть немного больше, чем стандартные сети клиент/сервер, когда клиент электронной почты устанавливает связь с сервером для приема и отсылки сообщений. Клиенты электронной почты могут обладать различными возможностями — от отдельного приложения для пользовательского компьютера, подобному тому, который описан в этой главе, до клиента для сотового телефона, который позволяет принимать сообщения электронной почты на сотовый телефон.

Независимо от того, где используется клиент электронной почты, он должен уметь устанавливать соединение с почтовым сервером для приема и передачи сообщений. Соединение с почтовым сервером создается на основе правил, установленных в соответствии с определенным протоколом. Существует два протокола для приема сообщений: POP3 и IMAP. Протокол SMTP в большинстве случаев используется для отправки сообщений. Далее речь пойдет об каждом из этих протоколов.

POP3

Почтовый протокол третьей версии (Post Office Protocol version 3 — POP3) используется в основном для получения почтовых сообщений с почтового сервера в сети Internet. Протокол POP3 является довольно простым и позволяет почтовому клиенту производить доступ только к папке "Inbox", установленной по умолчанию.

IMAP

Протокол доступа к сообщениям в сети Internet (Internet Message Access Protocol — IMAP) является еще одним протоколом для приема сообщений с почтового сервера в Internet. Протокол IMAP имеет более развитые средства обработки сообщений по сравнению с протоколом POP и поддерживает прием сообщений из множества ресурсов и каталогов. Протокол IMAP может использовать открытые папки, в которых размещены сообщения для совместного использования.

SMTP

Простой протокол электронной почты (Simple Mail Transfer Protocol — SMTP) используется для отправки почтовых сообщений в Internet. Когда отправляется сообщение с помощью SMTP, сервер принимает сообщение и затем перенаправляет его по указанному адресу в почтовый сервер получателя. После этого сообщение будет доступно для прочтения с помощью протоколов POP3 или IMAP.

Общая процедура отправки и приема сообщений электронной почты

Как уже упоминалось, прием и отправка сообщений по электронной почте происходят на основании определенных протоколов. Рассмотрим, как происходит типичная связь с почтовым сервером при отправке и приеме сообщений на основе этих протоколов.

Ниже описано, как происходит отправка сообщения на основе протокола SMTP.

1. Почтовый клиент подключается к серверу.
2. Почтовый клиент отправляет сообщение электронной почты на сервер.
3. Почтовый клиент прерывает соединение с сервером.

Отправка сообщений по протоколу SMTP довольно проста. Программа почтового клиента на стороне пользователя устанавливает соединение с SMTP-сервером, передает сообщение на этот сервер и закрывает канал связи. SMTP-сервер принимает сообщение и отправляет его по указанному адресу, выбирая подходящий маршрут.

Прием почтовых сообщений по протоколам POP3 и IMAP выполняется следующим образом.

1. Почтовый клиент подключается к серверу.
2. Почтовый клиент подтверждает свою подлинность на сервере.
3. Почтовый клиент загружает список предназначенных для него сообщений с сервера.
4. Почтовый клиент прерывает соединение с сервером.

Как видим, принять почтовое сообщение немного сложнее, чем отправить его. Сначала программа почтового клиента на стороне пользователя производит соединение с почтовым сервером. В процессе аутентификации следует сообщить о необходимых ресурсах и подтвердить свою подлинность на сервере. После успешной аутентификации почтовый клиент загружает новые сообщения с сервера и прерывает соединение.

Программный интерфейс JavaMail

Разработка кода для соединения с почтовым сервером довольно трудна и не всякий программист сможет выполнить такую работу. С счастью, команда Sun Microsystems JavaSoft хорошо понимала необходимость наличия стандартных библиотек для обработки почтовых сообщений и создала программный интерфейс JavaMail. Библиотеки JavaMail содержат полный набор средств для разработки почтовых приложений на языке Java, включая полную поддержку протоколов POP3, IMAP и SMTP. Почтовый клиент, разработанный в этой главе, широко использует средства JavaMail API для получения мощных и легких в использовании приложений почтового клиента.

Для того чтобы откомпилировать и запустить почтовый клиент, необходимо установить на ваш компьютер библиотеку JavaMail. Эта библиотека поставляется в составе пакета Java 2 Software Development Kit, Enterprise Edition (J2SDKEE). Если у вас не установлен пакет J2SDKEE, то его необходимо загрузить и установить. Хотя вы можете загрузить только библиотеки JavaMail и связанные с ними библиотеки JavaBeans Activation Framework (JAF). Несколько позже, в разделе "Компиляция и запуск почто-

вых сообщений", будет подробно рассказано об использовании библиотеки JavaMail. Ниже указаны адреса Web-сайтов, откуда можно загрузить библиотеки.

```
Java 2 Enterprise Edition    http://java.sun.com/j2ee/
JavaMail           http://java.sun.com/products/javamail/
JavaBeans Activation Framework (JAF)
    http://java.sun.com/products/javabeans/glasgow/jaf.html
```

Использование JavaMail

Библиотеки JavaMail рассчитаны на создание независимого от протокола пользовательского интерфейса для отправки и приема сообщений электронной почты. Программно пользовательский интерфейс состоит из нескольких классов, которые моделируют электронную почту и нескольких абстрактных деталей низкого уровня для получения таких протоколов, как SMTP и POP.С некоторыми подробностями базовых классов библиотеки JavaMail вы познакомитесь позже в этой главе.

Класс javax.mail.Session

Класс `javax.mail.Session` необходим для создания сеанса электронной почты, управления опциями и для аутентификации пользовательской информации при интерактивном взаимодействии с сервером. Этот класс является точкой входа для JavaMail API и используется некоторыми другими классами как внутренний класс.

Класс javax.mail.Store

Класс `javax.mail.Store` используется для сохранения сообщений наподобие учетных записей. Он также предоставляет протокол доступа, который используется при запоминании и извлечении сообщений. Это абстрактные классы, по существу это интерфейсы, используемые при доступе к записям электронной почты по протоколам POPS или IMAP.

Класс jovax.mail.Folder

Класс `javax.mail.Folder` представляет каталог для сообщений электронной почты. Объект типа `Folder` используется для создания иерархии сообщений наподобие дерева и может содержать сообщения, другие каталоги или то и другое одновременно. Также объект `Folder` используется для уничтожения сообщений. Класс `Folder` является абстрактным.

Класс javax.mail.Message

Класс `javax.mail.Message` инкапсулирует сообщения электронной почты и обрабатывает каждое из полей сообщения, такие как отправитель, получатель, дата отправки, предмет и содержимое сообщения. Класс `Message` является абстрактным.

Класс javax.mail.Transport

Класс `javax.mail.Transport` содержит код для отправки сообщений электронной почты. Метод этого класса `send()` используется для отправки сообщений. Класс `Transport` является абстрактным.

Простой клиент электронной почты

Остаток этой главы будет посвящен разработке простого клиента электронной почты, который способен принимать и отправлять текстовые сообщения по электронной почте. Это приложение используется "как есть", когда достаточны самые минимальные возможности электронной почты. При этом основной целью разработки является иллюстрация технических приемом, необходимых при создании электронной почты с помощью библиотек JavaMail API. Оно также может послужить как отправная точка для разработки собственных приложений электронной почты. На рис. 5.1 показано окно клиента электронной почты.

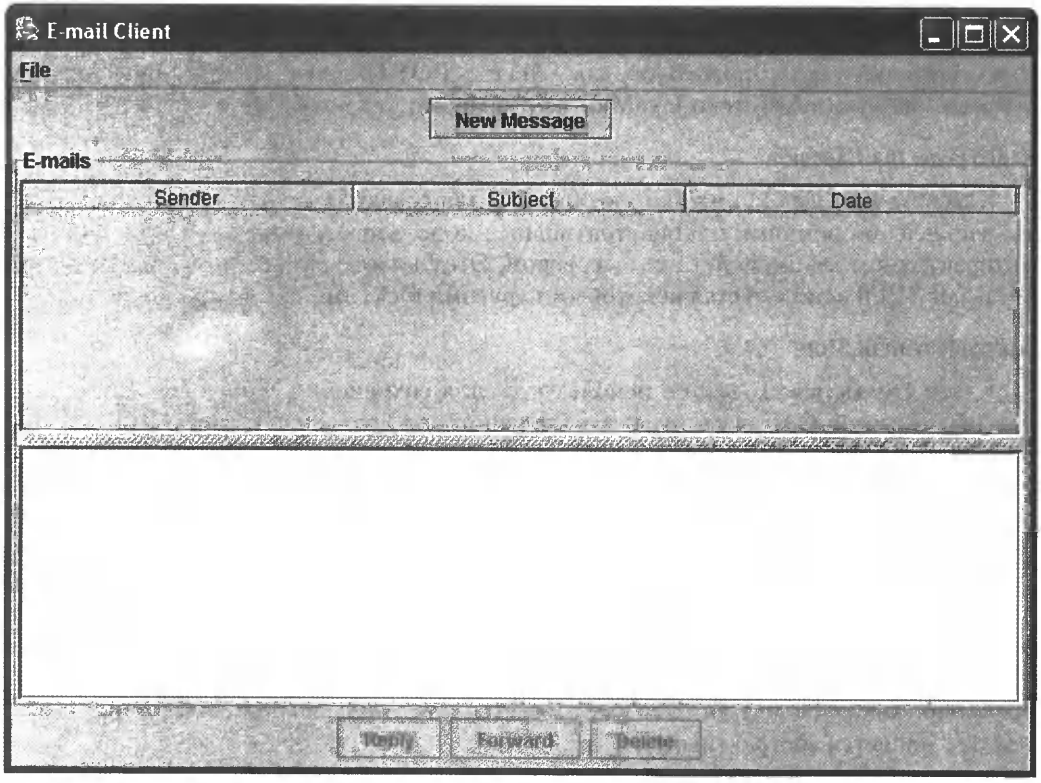


Рис. 5.1. Пользовательский интерфейс клиента электронной почты

В вершине окна находится кнопка **New Message** для создания нового сообщения электронной почты. Центр окна разделен на таблицу с перечнем сообщений и поле для отображения содержимого сообщения. Для каждого сообщения в списке отображается отправитель, предмет и дата. Когда в таблице выбирается сообщение, то при этом содержимое сообщения отображается в поле, расположенном в нижней части окна. В самом низу окна находятся кнопки **Reply**, **Forward** и **Delete** для работы с выбранным сообщением.

Почтовый клиент состоит из классов, перечисленных в представленной таблице.

Класс	Назначение
`EmailClient`	Содержит основную часть приложения, включая код для создания графического интерфейса и код для интерактивного взаимодействия с почтовым сервером
`ConnectDialog`	Отображает диалоговое окно, в котором пользователь вводит данные для соединения
`DownloadingDialog`	Отображает диалоговое окно со строкой "Downloading" во время загрузки сообщений с сервера
`MessageDialog`	Отображает диалоговое окно, используемое при создании сообщений
`MessagesTableModel`	Содержит список сообщений, отображаемых в окне почтового клиента

Основным классом приложения является класс `EmailClient`. Он содержит код, который используется при отправке и приеме сообщений. Остальные четыре класса являются обслуживающими для класса `EmailClient` и выполняют вспомогательную роль. В следующих разделах подробно рассматривается каждый класс, начиная с обслуживающих классов.

Класс ConnectDialog

На рис. 5.2 показано окно, создаваемое классом `ConnectDialog` в момент запуска почтового клиента. В этом диалоговом окне пользователь вводит данные, необходимые для создания соединения с почтовым сервером.

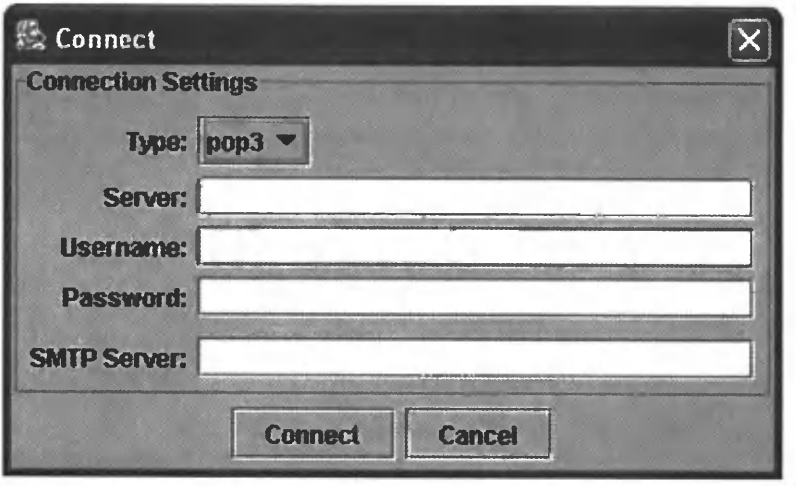


Рис. 5.2. Диалоговое окно для создания соединения

Листинг класса `ConnectDialog` приведен ниже. Обратите внимание на то, что используется родительский класс `JDialog`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
/* Этот класс отображает диалоговое окно для ввода
   данных при создании соединения с почтовым сервером. */
public class ConnectDialog extends JDialog
{
  // Это типы почтовых серверов.
  private static final String[] TYPES = {"pop3", "imap"};

  // Комбинированное окно для типов почтовых серверов.
  private JComboBox typeComboBox;

  // Текстовые поля для сервера, имени пользователя и SMTP-сервера.
  private JTextField serverTextField, usernameTextField;
  private JTextField smtpServerTextField;

  // Текстовое поле для пароля.
  private JPasswordField passwordField;

  // Конструктор для диалогового окна.
  public ConnectDialog(Frame parent)
  {
    // Вызов суперконструктора, определение диалогового окна как модально-
го.
    super(parent, true);

    // Указать заголовок диалогового окна.
    setTitle("Connect");

    // Обработать события при закрытии.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        actionCancel();
      }
    });

    // Установка настроечной панели.
    JPanel settingsPanel = new JPanel();
    settingsPanel.setBorder(
      BorderFactory.createTitledBorder("Connection Settings"));
    GridBagConstraints constraints;
    GridBagLayout layout = new GridBagLayout();
    settingsPanel.setLayout(layout);
    JLabel typeLabel = new JLabel("Type:");
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(typeLabel, constraints);
    settingsPanel.add(typeLabel);
    typeComboBox = new JComboBox(TYPES);
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.WEST;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 0, 5);
    constraints.weightx = 1.0D;
    layout.setConstraints(typeComboBox, constraints);
    settingsPanel.add(typeComboBox);
    JLabel serverLabel = new JLabel("Server:");
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(serverLabel, constraints);
    settingsPanel.add(serverLabel);
```

```
serverTextField = new JTextField(25);
constraints = new GridBagConstraints();
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
constraints.weightx = 1.0D;
layout.setConstraints(serverTextField, constraints);
settingsPanel.add(serverTextField);
JLabel usernameLabel = new JLabel("Username:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(usernameLabel, constraints);
settingsPanel.add(usernameLabel);
usernameTextField = new JTextField();
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.WEST;
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
constraints.weightx = 1.0D;
layout.setConstraints(usernameTextField, constraints);
settingsPanel.add(usernameTextField);
JLabel passwordLabel = new JLabel("Password:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 5, 0);
layout.setConstraints(passwordLabel, constraints);
settingsPanel.add(passwordLabel);
passwordField = new JPasswordField();
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.WEST;
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 5, 5);
constraints.weightx = 1.0D;
layout.setConstraints(passwordField, constraints);
settingsPanel.add(passwordField);
JLabel smtpServerLabel = new JLabel("SMTP Server:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 5, 0);
layout.setConstraints(smtpServerLabel, constraints);
settingsPanel.add(smtpServerLabel);
smtpServerTextField = new JTextField(25);
constraints = new GridBagConstraints();
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 5, 5);
constraints.weightx = 1.0D;
layout.setConstraints(smtpServerTextField, constraints);
settingsPanel.add(smtpServerTextField);

// Установка панели кнопок.
JPanel buttonsPanel = new JPanel();
JButton connectButton = new JButton("Connect");
connectButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionConnect();
  }
});
buttonsPanel.add(connectButton);
JButton cancelButton = new JButton("Cancel");
```

```
    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionCancel();
      }
    });
    buttonsPanel.add(cancelButton);

    // Добавить панель для отображения.
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(settingsPanel, BorderLayout.CENTER);
    getContentPane().add(buttonsPanel, BorderLayout.SOUTH);

    // Оптимизировать размер диалогового окна.
    pack();

    // Центровать диалоговое окно.
    setLocationRelativeTo(parent);
  }

  // Допустимые установки соединения и закрытие диалогового окна.
  private void actionConnect() {
    if (serverTextField.getText().trim().length() < 1
        || usernameTextField.getText().trim().length() < 1
        || passwordField.getPassword().length < 1
        || smtpServerTextField.getText().trim().length() < 1) {
      JOptionPane.showMessageDialog(this,
        "One or more settings is missing.",
        "Missing Setting(s)", JOptionPane.ERROR_MESSAGE);
      return;
    }

    // Закрыть диалоговое окно.
    dispose();
  }

  // Разорвать соединение и выйти из программы.
  private void actionCancel() {
    System.exit(0);
  }

  // Получить тип почтового сервера.
  public String getType() {
    return (String) typeComboBox.getSelectedItem();
  }

  // Получить почтовый сервер.
  public String getServer() {
    return serverTextField.getText();
  }

  // Получить имя пользователя.
  public String getUsername() {
    return usernameTextField.getText();
  }

  // Получить пароль.
  public String getPassword() {
    return new String(passwordField.getPassword());
  }

  // Получить SMTP-сервер.
```

```
  public String getSmtpServer() {
    return smtpServerTextField.getText();
  }
}
```

Класс `ConnectDialog` начинается с объявления статических переменных и типов серверов. Затем объявляются несколько элементов управления графического интерфейса. Остальные элементы класса рассмотрены в разделах ниже.

Конструктор класса ConnectDialog

В конструктор класса `ConnectDialog` передается ссылка на родительский класс `Frame`, который работает совместно с диалоговым окном. Затем в конструкторе передается ссылка на класс `Frame` в конструктор класса `JDialog` с помощью вызова метода `super()`. Вызов конструктора класса `JDialog` необходим для определения диалогового окна как модального. Модальное диалоговое окно блокирует ввод пользователя в другие окна программы. Эта способность особенно полезна в графическом интерфейсе для защиты от возможного случайного взаимодействия пользователя с другими интерактивными элементами интерфейса.

Далее создается панель с пользовательскими установками, после чего инициализируется каждый из элементов управления и добавляется в панель. Обратите внимание, что для размещения элементов используется тип расположения `grid bag layout`. После того как панель с установками для соединения создана, кнопки **Connect** и **Cancel** инициализируются и добавляются в панель кнопок. Обе панели, панель установок и панель кнопок, затем готовятся для отображения. Затем вызывается метод `pack()` для того, чтобы минимизировать размеры окна, которое должно включать только необходимы элементы управления. Наконец, вызывается метод `setLocationRelativeTo()` для центровки окна относительно почтового клиента.

Метод actionConnect()

Метод `actionConnect()`, листинг которого приведен ниже, используется для вывода на экран диалогового окна при щелчке на кнопке **Connect**, проверки вводимых данных и удаления окна с экрана.

```
// Проверка правильности установок и закрытие диалогового окна.
private void actionConnect() {
  if (serverTextField.getText().trim().length()) < 1
      || usernameTextField.getText().trim().length() < 1
      || passwordField.getPassword().length < 1
      || smtpServerTextField.getText().trim().length()) < 1) {
    JOptionPane.showMessageDialog(this,
      "One or more settings is missing.",
      "Missing Setting(s)", JOptionPane.ERROR_MESSAGE);
    return;
  }
  // Закрыть диалоговое окно.
  dispose();
}
```

Установки с помощью диалогового окна требуются для каждого соединения, метод `actionConnect()` производит проверку наличия всех введенных данных. Если одна или более установок ошибочны, то отображается диалоговое окно ошибки для того, чтобы пользователь обратил на это внимание. После того как все будет введено правильно, вызывается метод `dispose()` для закрытия диалогового окна.

Метод actionConcel()

Метод `actionCancel()` вызывается при щелчке на кнопке **Cancel** в диалоговом окне или при закрытии диалогового окна.

```
// Закрытие соединения и выход из программы.
private void actionCancel() {
  System.exit(0);
}
```

Закрытие диалогового окна **Connect** приводит к тому, что программа почтового клиента будет завершена, т.к. вызывается метод `System.exit()`. Почтовый клиент не может функционировать без соединения с сервером.

Методы доступа

Класс `ConnectDialog` содержит несколько методов доступа для извлечения установочных данных, введенных в диалоговом окне. Каждый из методов `getType()`, `getServer()`, `getUsername()`, `getPassword()` и `getSmtpServer()` просто возвращает значение, введенное в соответствующее поле диалогового окна.

Класс DownloadingDialog

Сразу после того как работа с диалоговым окном **Connect** будет завершена, запускается диалоговое окно **Downloading**, как показано на рис. 5.3. Это окно сигнализирует о том, что производится загрузка почтовых сообщений. Дополнительно к этому, поскольку диалоговое окно **Downloading** является модальным, оно не позволяет обращаться к другим окнам, пока не будет закрыто, и этим самым предохраняет от работы с другими элементами почтового клиента во время загрузки.

Рис. 5.3. Диалоговое окно Downloading

Листинг класса `DownloadingDialog` приведен ниже. Данный класс является дочерним для класса `JDialog`.

```
import j ava.awt.*;
import javax.swing.*;
/* Этот класс отображает простое диалоговое окно, сообщая
   пользователю, что производится загрузка сообщений. */
public class DownloadingDialog extends JDialog {
  // Конструктор диалогового окна.
  public DownloadingDialog(Frame parent)
  {
  // Вызов суперконструктора, определение диалогового окна как модального.
  super(parent, true);
  // Указать заголовок диалогового окна.
  setTitle("E-mail Client");
  // Не закрывать окно, когда производится щелчок на
     пиктограмме закрытия "X".
  setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
  // Разместить сообщение с окантовкой в диалоговом окне.
```

```
  JPanel contentPane = new JPanel();
  contentPane.setBorder(
    BorderFactory.createEmptyBorder(5, 5, 5, 5));
    contentPane.add(new JLabel("Downloading messages. .."));
    setContentPane(contentPane);
    // Оптимизировать размеры окна.
    pack();
    // Центровать диалоговое окно в приложении.
     setLocationRelativeTo(parent);
  }
}
```

Структура класса `DownloadingDialog` довольно понятна и состоит только из одного конструктора. Конструктор класса, подобно конструктору класса `ConnectDialog`, начинается вызовом методов `super()` и `setTitle()`. Затем вызывается метод `setDefaultCloseOperation()` для создания определенного поведения диалогового окна и отсутствия реакции на щелчок мыши для закрытия окна. При этом пользователь не сможет закрыть окно преждевременно. После этого добавляется сообщение для отображения с небольшой невидимой границей в 5 пикселей. Затем вызывается метод `pack()` для получения минимально возможных размеров окна при данном содержимом. Наконец, вызывается метод `setLocationRelativeTo()` для размещения окна в центре родительского окна.

Класс MessageDialog

Диалоговое окно `Message`, показанное на рис. 5.4, используется для ввода текста сообщения, которое будет отправлено с помощью класса `EmailClient`. Это окно создается с помощью класса `MessageDialog`. Класс `MessageDialog` используется для создания новых сообщений, а также для ответных сообщений и для перенаправления сообщений. Листинг класса `MessageDialog` приведен ниже. Класс `MessageDialog` наследует класс `JDialog`.

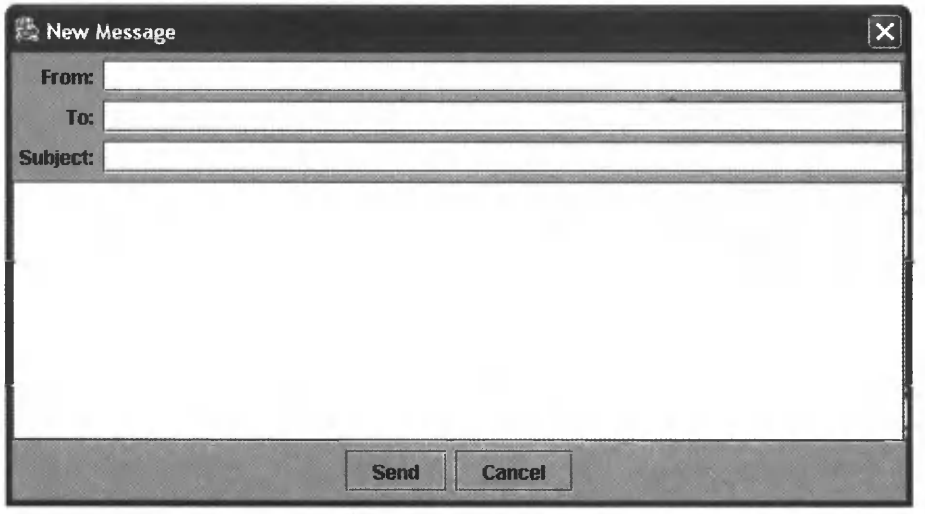


Рис. 5.4. Диалоговое окно Message

```
import java.awt.*;
import java.awt.event.*;
import javax.mail.*;
import javax.swing.*;

// Этот класс отображает диалоговое окно, используемое
// при создании сообщений.
public class MessageDialog extends JDialog
{
  // Идентификаторы сообщений.
  public static final int NEW = 0;
  public static final int REPLY = 1;
  public static final int FORWARD = 2;

  // Текстовые поля для сообщений "от", "для" и "предмет".
  private JTextField fromTextField, toTextField;
  private JTextField subjectTextField;

  // Текстовое поле для содержимого сообщения.
  private JTextArea contentTextArea;

  // Флаг, указывающий о закрытии диалогового окна.
  private boolean cancelled;

  // Конструктор для диалогового окна.
  public MessageDialog(Frame parent, int type, Message message)
    throws Exception
  {
    // Вызов суперконструктора, определение диалогового окна
    // как модального.
    super(parent, true);

    /* Установить заголовок диалогового окна и получить значения
       для полей "to", "subject" и "content" на основе
       типа сообщения. */
    String to = "", subject = "", content = "";
    switch (type) {
      // Ответное сообщение.
      case REPLY:
        setTitle("Reply To Message");

        // Получить значение "to" для сообщения.
        Address[] senders = message.getFrom();
        if (senders != null || senders.length > 0) {
          to = senders[0].toString();
        }
        to = message.getFrom()[0].toString();

        // Получить тему сообщения.
        subject = message.getSubject();
        if (subject != null && subject.length() > 0) {
          subject = "RE: " + subject;
        } else {
          subject = "RE:";
        }

        // Получить содержимое сообщения и добавить
        // запись "REPLIED TO".
        content = "\n----------------- " +
                  "REPLIED TO MESSAGE" +
                  " -----------------\n" +
```

```
                EmailClient.getMessageContent(message);
        break;

      // Перенаправить сообщение.
      case FORWARD:
        setTitle("Forward Message");

        // Получить тему сообщения.
        subject = message.getSubject();
        if (subject != null && subject.length() > 0) {
          subject = "FWD: " + subject;
        } else {
          subject = "FWD:";
        }

        // Получить содержимое сообщения и добавить
        // запись "FORWARDED".
        content = "\n---------------- " +
                  "FORWARDED MESSAGE" +
                  " ----------------\n" +
                  EmailClient.getMessageContent(message);
        break;

      // Новое сообщение.
      default:
        setTitle("New Message");
    }

    // Обработать события при закрытии.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        actionCancel();
      }
    });

    // Установить поля панели.
    JPanel fieldsPanel = new JPanel();
    GridBagConstraints constraints;
    GridBagLayout layout = new GridBagLayout();
    fieldsPanel.setLayout(layout);
    JLabel fromLabel = new JLabel("From:");
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(fromLabel, constraints);
    fieldsPanel.add(fromLabel);
    fromTextField = new JTextField();
    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(fromTextField, constraints);
    fieldsPanel.add(fromTextField);
    JLabel toLabel = new JLabel("To:");
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(toLabel, constraints);
    fieldsPanel.add(toLabel);
    toTextField = new JTextField(to);
    constraints = new GridBagConstraints();
```

```
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 0, 0);
    constraints.weightx = 1.0D;
    layout.setConstraints(toTextField, constraints);
    fieldsPanel.add(toTextField);
    JLabel subjectLabel = new JLabel("Subject:");
    constraints = new GridBagConstraints();
    constraints.insets = new Insets(5, 5, 5, 0);
    layout.setConstraints(subjectLabel, constraints);
    fieldsPanel.add(subjectLabel);
    subjectTextField = new JTextField(subject);
    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 5, 0);
    layout.setConstraints(subjectTextField, constraints);
    fieldsPanel.add(subjectTextField);

    // Установить панель содержимого.
    JScrollPane contentPanel = new JScrollPane();
    contentTextArea = new JTextArea(content, 10, 50);
    contentPanel.setViewportView(contentTextArea);

    // Установить панель кнопок.
    JPanel buttonsPanel = new JPanel();
    JButton sendButton = new JButton("Send");
    sendButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionSend();
      }
    });
    buttonsPanel.add(sendButton);
    JButton cancelButton = new JButton("Cancel");
    cancelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionCancel();
      }
    });
    buttonsPanel.add(cancelButton);

    // Добавить отображения панелей.
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(fieldsPanel, BorderLayout.NORTH);
    getContentPane().add(contentPanel, BorderLayout.CENTER);
    getContentPane().add(buttonsPanel, BorderLayout.SOUTH);

    // Оптимизировать размеры.
    pack();

    // Центровать диалоговое окно в приложении.
    setLocationRelativeTo(parent);
  }

  // Проверка и закрытие диалогового окна.
  private void actionSend() {
    if (fromTextField.getText().trim().length() < 1
        || toTextField.getText().trim().length() < 1
        || subjectTextField.getText().trim().length() < 1
        || contentTextArea.getText().trim().length() < 1) {
      JOptionPane.showMessageDialog(this,
```

```
        "One or more fields is missing.",
        "Missing Field(s)", JOptionPane.ERROR_MESSAGE);
      return;
    }

    // Закрытие диалогового окна.
    dispose();
  }

  // Прервать создание сообщения и закрыть диалоговое окно.
  private void actionCancel() {
    cancelled = true;

    // Закрытие диалогового окна.
    dispose();
  }

  // Показать диалоговое окно.
  public boolean display() {
    show();

    // Возврат, если отображение произошло с ошибкой.
  return !cancelled;
  }

  // Получить сообщение "From" field value.
  public String getFrom() {
    return fromTextField.getText();
  }

  // Получить значения поля "To" для сообщения.
  public String getTo() {
    return toTextField.getText();
  }

  // Получить значения поля "Subject" для сообщения.
  public String getSubject() {
    return subjectTextField.getText();
  }

  // Получить значения поля "content" для сообщения.
  public String getContent() {
    return contentTextArea.getText();
  }
}
```

Переменные класса MessageDialog

Класс `MessageDialog` начинается с объявления трех статических переменных: `NEW`, `REPLY` и `FORWARD`, которые определяют типы сообщений для диалогового окна. Затем объявляются несколько элементов управления графического интерфейса. Наконец, объявляется флаг, указывающий на отмену выполнения, с помощью которого отслеживается работа диалогового окна.

Конструктор класса MessageDialog

Подобно предыдущим двум классам для диалоговых окон, конструктор класса `MessageDialog` начинается с вызова конструктора родительского класса для того, чтобы сделать окно модальным. Затем используется утверждение `switch` для уста-

новки заголовка диалогового окна на основе создаваемого сообщения. В этой конструкции также извлекаются значения полей из оригинального сообщения, которое передается в конструктор как аргумент (типы REPLY и FORWARD). Поля оригинального сообщения используются позже в конструкторе для заполнения полей объекта типа MessageDialog. Обратите внимание, что содержимое сообщений типа REPLY и FORWARD имеют текстовый префикс, определяющий оригинальное сообщение.

После завершение утверждения switch создается панель с полями, при этом каждое поле инициализируется и добавляется в панель. Затем вызывается метод pack() для подгонки границ панели под размеры, которые необходимы только для включения всех элементов управления. Наконец, вызывается метод setLocationRelativeTo() для размещения диалогового окна в центре родительского окна.

Метод actionSend()

Метод actionSend(), листинг которого приведен ниже, подтверждает, что данные введены во все поля, и после щелчка на кнопке **Send** производит закрытие диалогового окна.

```
// Проверяет поля сообщения и закрывает диалоговое окно.
private void actionSend() {
  if (fromTextField.getText().trim().length() < 1
      || toTextField.getText().trim().length() < 1
      || subjectTextField.getText().trim().length() < 1
      || contentTextArea.getText().trim().length() < 1) {
    JOptionPane.showMessageDialog(this,
      "One or more fields is missing.", "Missing Field(s)",
      JOptionPane.ERROR_MESSAGE);
    return;
  }
  // Закрывает диалоговое окно.
  dispose();
}
```

Требуется , чтобы было заполнено каждое поле, поэтому в методе actionSend() производится проверка на наличие данных во всех полях для успешного завершения. Если одно или несколько полей не заполнены, появляется диалоговое окно ошибки для информирования пользователя. Если проверка завершена успешно, то вызывается метод dispose() для закрытия диалогового окна.

Метод actionCancel()

Метод actionCancel(), листинг которого приведен выше, вызывается, когда производится щелчок на кнопке **Cancel** диалогового окна или при закрытии диалогового окна.

```
// Прервать создание сообщения и закрыть диалоговое окно.
private void actionCancel() {
  cancelled = true;
  // Закрыть диалоговое окно.
  dispose();
}
```

Перед закрытием диалогового окна в методе actionCancel() совместно с вызовом метода dispose() устанавливается значение true для флага cancelled.

Флаг cancelled используется методом display() для определения того, что диалоговое окно закрыто.

Метод display()

Обычно диалоговое окно отображается на экране с помощью вызова метода show(). Однако в случае с диалоговым окном Message важно знать, что оно отображается на экране перед его уничтожением. Метод display(), листинг которого приведен ниже, выполняет именно это и замещает метод show().

```
// Отобразить диалоговое окно.
public boolean display() {
  show();
  // Возвратить при отображении диалогового окна.
  return !cancelled;
}
```

Обратите внимание, что возвращается инверсия флага cancelled, в результате чего можно судить об успешном выполнении операции.

Методы доступа

Класс MessageDialog имеет несколько методов доступа для извлечения свойств сообщения, введенных в диалоговом окне. Каждый из методов getFrom(), getTo(), getSubject() и getContent() просто возвращает значение, введенное в соответствующее поле.

Класс MessagesTableModel

Класс MessagesTableModel содержит список сообщений и является источником данных для элемента "Messages" экземпляра класса JTable.

Листинг класса MessagesTableModel приведен ниже. Обратите внимание, что он является наследником класса AbstractTableModel.

```
import java.util.*;
import javax.mail.*;
import javax.swing.*;
import javax.swing.table.*;

// Этот класс управляет данными в таблице почтового клиента.
public class MessagesTableModel extends AbstractTableModel
{
  // Это имена колонок таблицы.
  private static final String[] columnNames = {"Sender",
    "Subject", "Date"};

  // Список сообщений таблицы.
  private ArrayList messageList = new ArrayList();

  // Установить список сообщений таблицы.
  public void setMessages(Message[] messages) {
    for (int i = messages.length - 1; i >= 0; i--) {
      messageList.add(messages[i]);
    }
```

```
    // Создать уведомление для таблицы об изменении данных.
    fireTableDataChanged();
  }

  // Получить сообщение от указанной строки.
  public Message getMessage(int row) {
    return (Message) messageList.get(row);
  }

  // Удалить сообщение из списка.
  public void deleteMessage(int row) {
    messageList.remove(row);

    // Создать уведомление для таблицы об удалении строки.
    fireTableRowsDeleted(row, row);
  }

  // Получить число колонок в таблице.
  public int getColumnCount() {
    return columnNames.length;
  }

  // Получить имена колонок.
  public String getColumnName(int col) {
     return columnNames[col];
  }

  // Получить число строк в таблице.
  public int getRowCount() {
    return messageList.size();
  }

  // Получить комбинацию строки и колонки.
  public Object getValueAt(int row, int col) {
    try {
      Message message = (Message) messageList.get(row);
      switch (col) {
        case 0: // Отправитель.
          Address[] senders = message.getFrom();
          if (senders != null || senders.length > 0) {
            return senders[0].toString();
          } else {
            return "[none]";
          }
        case 1: // Описание.
          String subject = message.getSubject();
          if (subject != null && subject.length() > 0) {
            return subject;
          } else {
            return "[none]";
          }
        case 2: // Данные.
          Date date = message.getSentDate();
          if (date != null) {
            return date.toString();
          } else {
            return "[none]";
          }
      }
    } catch (Exception e) {
      // Ошибка.
```

```
      return "";
    }
    return "";
  }
}
```

Класс `MessagesTableModel` является утилитой, используемой экземпляром "Messages" `JTable` для управления данными в таблице. Когда объект типа `JTable` инициализируется, в него передается объект `MessagesTableModel`. Затем в объекте `JTable` вызываются несколько методов из класса `MessagesTableModel` для получения необходимых данных. Метод `getColumnCount()` вызывается для определения количества колонок в таблице. Аналогично, метод `getRowCount()` используется для определения количества строк в таблице. Метод `getColumnName()` возвращает имена колонок в соответствии с их идентификаторами. Метод `getMessage()` принимает идентификатор строки и возвращает связанный с ним объект `Message` из списка. Остальные методы класса `MessagesTableModel` являются более сложными и будут рассмотрены в отдельных разделах.

Метод setMessages()

Метод `setMessages()`, листинг которого приведен ниже, устанавливает список объектов `Messages`, которые должны отображаться в таблице.

```
// Установить список сообщений таблицы.
public void setMessages(Message[] messages) {
  for (int i = messages.length - 1; i >= 0; i--) {
    messageList.add(messages[i]);
  }
// Создать уведомление для таблицы об изменении в данных.
  fireTableDataChanged();
}
```

В этом методе сначала циклически просматривается массив объектов `Message`, передаваемый как параметр, и каждое сообщение добавляется в список. В соответствии с этим сценарием, массив сообщений должен использоваться как резервный источник данных для модели таблицы. Но поскольку некоторые сообщения могут быть удалены из объекта `MessagesTableModel`, использование объекта `ArrayList` для хранения сообщений будет более удобным. После добавления сообщений в список сообщений, в таблицу посылается уведомление об изменении данных.

Метод deleteMessage()

Метод `deleteMessage()`, листинг которого приведен ниже, удаляет объект `Message` из списка управляемых сообщений.

```
// Удалить сообщение из списка.
public void deleteMessage(int row) {
  messageList.remove(row);
  // Послать в таблицу уведомление об удалении строки.
  fireTableRowsDeleted(row, row);
}
```

После удаления объекта `Message` из внутреннего списка, уведомление об удалении строки посылается в таблицу.

Метод getValueAt()

Метод `getValueAt()`, листинг которого приведен ниже, вызывается для получения текущего значения при отображении каждой ячейки таблицы.

```
// Получить значение для указанной строки и колонки.
public Object getValueAt(int row, int col) {
  try {
    Message message = (Message) messageList.get(row);
    switch (col) {
      case 0: // Отправитель.
        Address[] senders = message.getFrom();
        if (senders != null || senders.length > 0) {
          return senders[0].toString();
        } else {
          return "[none]";
        }
      case 1: // Описание.
        String subject = message.getSubject();
        if (subject != null && subject.length() > 0) {
          return subject;
        } else {
          return "[none]";
        }
      case 2: // Дата.
        Date date = message.getSentDate();
        if (date != null) {
          return date.toString(),-} else {
            return "[none]";
          }
    }
  } catch (Exception e) {
  // Безопасный выход.
  return "";
  }
return " " ;
}
```

В этом методе сначала находится объект `Message` для заданной строки. Затем используется указанная колонка для получения требуемого значения, которое возвращается свойством объекта `Message`. Обратите внимание, что каждое возвращаемое значение проверяется, и если оно равно `null`, то возвращается значение "`[none]`". Эта проверка необходима, потому как некоторые значения объекта могут отсутствовать. Например, сообщение электронной почты может быть послано без описания или обратного адреса. Отметим также, что тело метода заключено в блок `try...catch`. Если в одном из методов объекта `Message` произойдет исключительная ситуация, то будет сделан переход на коды за ключевым словом `caught`, что приведет к возврату пустого значения.

Класс EmailClient

После рассмотрения всех вспомогательных классов программы почтового клиента, подробно обсудим основной класс `EmailClient`. Это класс может создавать и запускать графический интерфейс почтового клиента и обеспечивать соединение с почтовым сервером.

Метод main() находится в классе EmailClient и при запуске на выполнение он будет вызываться первым. Метод main() создает новый объект типа EmailClient и затем вызывает метод show(), с помощью которого объект EmailClient будет отображаться на экране. После отображения почтового клиента на экране, вызывается метод connect() с запросом к пользователю ввести необходимые данные для соединения с почтовым сервером.

Листинг класса EmailClient приведен ниже, а подробное объяснение отдельных деталей будет приведено в последующих разделах. Класс EmailClient является наследником класса JFrame.

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.swing.*;
import javax.swing.event.*;

// Почтовый клиент.
public class EmailClient extends JFrame
{
  // Модель таблицы данных сообщений.
  private MessagesTableModel tableModel;

  // Таблица со списком сообщений.
  private JTable table;

  // Текстовое поле для отображения сообщений.
  private JTextArea messageTextArea;

  /* Панель, разделенная на таблицу сообщений и
     панель для отображения содержимого сообщения. */
  private JSplitPane splitPane;

  // Кнопки для управления выбранным сообщением.
  private JButton replyButton, forwardButton, deleteButton;

  // Выбранное из таблицы сообщение.
 private Message selectedMessage;

  // Флаг для отображения состояния сообщения.
  private boolean deleting;

  // Сеанс JavaMail.
  private Session session;

  // Конструктор для почтового клиента.
  public EmailClient()
  {
    // Установить заголовок приложения.
    setTitle("E-mail Client");

    // Установить размеры окна.
    setSize(640, 480);

    // Обработать события при закрытии окна.
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
```

```
      actionExit();
    }
  });

  // Установить меню файлов.
  JMenuBar menuBar = new JMenuBar();
  JMenu fileMenu = new JMenu("File");
  fileMenu.setMnemonic(KeyEvent.VK_F);
  JMenuItem fileExitMenuItem = new JMenuItem("Exit",
    KeyEvent.VK_X);
  fileExitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionExit();
    }
  });
  fileMenu.add(fileExitMenuItem);
  menuBar.add(fileMenu);
  setJMenuBar(menuBar);

  // Установить панель для кнопок.
  JPanel buttonPanel = new JPanel();
  JButton newButton = new JButton("New Message");
  newButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionNew ();
    }
  });
  buttonPanel.add(newButton);

  // Установить таблицу сообщений.
  tableModel = new MessagesTableModel();
  table = new JTable(tableModel);
  table.getSelectionModel().addListSelectionListener(new
    ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
      tableSelectionChanged();
    }
  });
  // Разрешить выбирать только одну строку.
  table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

  // Установить панель почтового клиента.
  JPanel emailsPanel = new JPanel();
  emailsPanel.setBorder(
    BorderFactory.createTitledBorder("E-mails"));
  messageTextArea = new JTextArea();
  messageTextArea.setEditable(false);
  splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
    new JScrollPane(table), new JScrollPane(messageTextArea));
  emailsPanel.setLayout(new BorderLayout());
  emailsPanel.add(splitPane, BorderLayout.CENTER);

  // Установить вторую панель для кнопок.
  JPanel buttonPanel2 = new JPanel();
  replyButton = new JButton("Reply");
  replyButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionReply();
    }
  });
  replyButton.setEnabled(false);
```

```
    buttonPanel2.add(replyButton);
    forwardButton = new JButton("Forward");
    forwardButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionForward();
      }
    });
    forwardButton.setEnabled(false);
    buttonPanel2.add(forwardButton);
    deleteButton = new JButton("Delete");
    deleteButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionDelete();
      }
    });
    deleteButton.setEnabled(false);
    buttonPanel2.add(deleteButton);

    // Расположение панелей.
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(buttonPanel, BorderLayout.NORTH);
    getContentPane().add(emailsPanel, BorderLayout.CENTER);
    getContentPane().add(buttonPanel2, BorderLayout.SOUTH);
  }

  // Выход из программы.
  private void actionExit() {
    System.exit(0);
  }

  // Создать новое сообщение.
  private void actionNew () {
    sendMessage(MessageDialog.NEW, null);
  }

  // Вызывается при изменении выбранной строки таблицы.
  private void tableSelectionChanged() {
    /* If not in the middle of deleting a message, set
       the selected message and display it. */
    if (!deleting) {
      selectedMessage =
        tableModel.getMessage(table.getSelectedRow());
      showSelectedMessage();
      updateButtons();
    }
  }

  // Ответить на сообщение.
  private void actionReply() {
    sendMessage(MessageDialog.REPLY, selectedMessage);
  }

  // Перенаправить сообщение.
  private void actionForward() {
    sendMessage(MessageDialog.FORWARD, selectedMessage);
  }

  // Удалить выбранное сообщение.
  private void actionDelete() {
    deleting = true;
```

```
    try {
      // Удалить сообщение с сервера.
      selectedMessage.setFlag(Flags.Flag.DELETED, true);
      Folder folder = selectedMessage.getFolder();
      folder.close(true);
      folder.open(Folder.READ_WRITE);
    } catch (Exception e) {
      showError("Unable to delete message.", false);
    }

    // Удалить сообщение из таблицы.
    tableModel.deleteMessage(table.getSelectedRow());

    // Обновить GUI.
    messageTextArea.setText("");
    deleting = false;
    selectedMessage = null;
    updateButtons();
  }

  // Отправить указанное сообщение.
  private void sendMessage(int type, Message message) {
    // Display message dialog to get message values.
    MessageDialog dialog;
    try {
      dialog = new MessageDialog(this, type, message);
      if (!dialog.display()) {
        // Return if dialog was cancelled.
        return;
      }
    } catch (Exception e) {
      showError("Unable to send message.", false);
      return;
    }

    try {
      // Создать новое сообщение со значениями из диалогового окна.
      Message newMessage = new MimeMessage(session);
      newMessage.setFrom(new InternetAddress(dialog.getFrom()));
      newMessage.setRecipient(Message.RecipientType.TO,
        new InternetAddress(dialog.getTo()));
      newMessage.setSubject(dialog.getSubject());
      newMessage.setSentDate(new Date());
      newMessage.setText(dialog.getContent());

      // Отправить новое сообщение.
      Transport.send(newMessage);
    } catch (Exception e) {
      showError("Unable to send message.", false);
    }
  }

  // Отобразить выбранное сообщение в панели для содержимого.
  private void showSelectedMessage() {
    // Показать песочные часы, пока сообщение загружается.
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    try {
      messageTextArea.setText(
        getMessageContent(selectedMessage));
      messageTextArea.setCaretPosition(0);
    } catch (Exception e) {
```

```
      showError("Unabled to load message.", false);
    } finally {
      // Return to default cursor.
      setCursor(Cursor.getDefaultCursor());
    }
  }

  /* Обновить состояние каждой кнопки на основе того,
     выбрано ли сообщение в таблице. */
  private void updateButtons() {
    if (selectedMessage != null) {
      replyButton.setEnabled(true);
      forwardButton.setEnabled(true);
      deleteButton.setEnabled(true);
    } else {
      replyButton.setEnabled(false);
      forwardButton.setEnabled(false);
      deleteButton.setEnabled(false);
    }
  }

  // Показать окно приложения на экране.
  public void show() {
    super.show();

    // Обновить разделение панели с соотношением 50/50.
    splitPane.setDividerLocation(.5);
  }

  // Подключиться к почтовому серверу.
  public void connect() {
    // Отобразить диалоговое окно для соединения.
    ConnectDialog dialog = new ConnectDialog(this);
    dialog.show();

    // Обработать данные для соединения.
    StringBuffer connectionUrl = new StringBuffer();
    connectionUrl.append(dialog.getType() + "://");
    connectionUrl.append(dialog.getUsername() + ":");
    connectionUrl.append(dialog.getPassword() + "@");
    connectionUrl.append(dialog.getServer() + "/");

    /* Отобразить диалоговое окно с фразой о том,
       что почтовые сообщения загружаются с сервера. */
    final DownloadingDialog downloadingDialog =
      new DownloadingDialog(this);
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        downloadingDialog.show();
      }
    });

    // Создать сеанс JavaMail и подключиться к серверу.
    Store store = null;
    try {
      // Инициализировать сеанс JavaMail с SMTP-сервером.
      Properties props = new Properties();
      props.put("mail.smtp.host", dialog.getSmtpServer());
      session = Session.getDefaultInstance(props, null);

      // Подключиться к почтовому серверу.
```

```
      URLName urln = new URLName(connectionUrl.toString());
      store = session.getStore(urln);
      store.connect();
    } catch (Exception e) {
      // Закрыть диалоговое окно.
      downloadingDialog.dispose();

      // Отобразить диалоговое окно ошибки.
      showError("Unable to connect.", true);
    }

    // Загрузить заголовки сообщений с сервера.
    try {
      // Open main "INBOX" folder.
      Folder folder = store.getFolder("INBOX");
      folder.open(Folder.READ_WRITE);

      // Получить список сообщений из каталога.
      Message[] messages = folder.getMessages();

      // Извлечь заголовок каждого сообщения из каталога.
      FetchProfile profile = new FetchProfile();
      profile.add(FetchProfile.Item.ENVELOPE);
      folder.fetch(messages, profile);

      // Поместить сообщение в таблицу.
      tableModel.setMessages(messages);
    } catch (Exception e) {
      // Закрыть диалоговое окно.
      downloadingDialog.dispose();

      // Отобразить диалоговое окно ошибки.
      showError("Unable to download messages.", true);
    }

    // Закрыть диалоговое окно.
    downloadingDialog.dispose();
  }

  // Отбразить диалоговое окно ошибки и выйти из программы.
Show error dialog and exit afterwards if necessary.
  private void showError(String message, boolean exit) {
    JOptionPane.showMessageDialog(this, message, "Error",
      JOptionPane.ERROR_MESSAGE);
    if (exit)
      System.exit(0);
  }

  // Получить содержимое сообщения.
  public static String getMessageContent(Message message)
    throws Exception {
    Object content = message.getContent();
    if (content instanceof Multipart) {
      StringBuffer messageContent = new StringBuffer();
      Multipart multipart = (Multipart) content;
      for (int i = 0; i < multipart.getCount(); i++) {
        Part part = (Part) multipart.getBodyPart(i);
          if (part.isMimeType("text/plain")) {
            messageContent.append(part.getContent().toString());
          }
        }
```

```
        return messageContent.toString();
      } else {
        return content.toString();
      }
    }

    // Запустить программу почтового клиента.
    public static void main(String[] args) {
      EmailClient client = new EmailClient();
      client.show();

      // Отобразить диалоговое окно для соединения.
      client.connect();
    }
}
```

Переменные класса EmailClient

Класс `EmailClient` начинается с объявления нескольких переменных, большинство из которых содержат ссылки на элементы управления. Переменная `selectedMessage` содержит ссылку на объект `Message`, представляющий выбранную строку с таблице сообщений. Булева переменная `deleting` является флагом, который указывает на наличие сообщения в строке или его удаление. Наконец, переменная `session` включает ссылку на сеанс JavaMail, созданный с помощью метода `connect()`.

Конструктор класса EmailClient

При создании объекта типа `EmailClient`, внутри конструктора инициализируются все элементы управления. В конструкторе довольно много строк кода, но понять его не трудно. Далее кратко обсудим все фрагменты кода.

Сначала устанавливается заголовок окна с помощью вызова метода `setTitle()`. Затем вызывается метод `setSize()` для задания высоты и ширины окна в пикселях. После этого добавляется слушатель окна, для чего вызывается метод `addWindowListener()`, в который передается объект `WindowAdapter`, переопределяющий обработчик событий `windowClosing()`. В этом обработчике событий при закрытии окна вызывается метод `actionExit()`. Следующим в окно приложения добавляется строка меню с пунктом **File**. Затем устанавливается панель кнопок, на которой размещается кнопка **New Message**. Для этой кнопки добавляется слушатель `ActionListener` таким образом, чтобы при каждом щелчке на кнопке вызывался метод `actionNew()`.

После этого конструируется таблица сообщений. Слушатель `ListSelectionListener` добавляется в таблицу для того, чтобы при каждом выборе строки вызывался метод `tableSelectionChanged()`. При установке режима выбора строк таблицы используется значение `ListSelectionModel.SINGLE_SELECTION` для того, чтобы можно было выбрать только одну строку за один раз. Ограничение выбора строк связано с упрощением логики для активизации кнопок графического интерфейса при выборе строки (или сообщения).

Затем создается панель почтового клиента, содержащая таблицу сообщений и элемент управления `JTextArea`, в котором будут отображаться выбранные со-

общения. После этого в панель добавляется рамка с заголовком. Затем определяется пространство для отображения сообщения. Обратите внимание, что метод `messageTextArea.setEditable()` вызывается с аргументом `false`. Это защищает текст, отображающийся в текстовом пространстве, от модификации. После этого создается объект `JSplitPane` для разделения панели почтового клиента на таблицу сообщений и текстовое пространство. Объект `JSplitPane` создает разделитель между двумя компонентами таким образом, чтобы для каждой панели можно было установить необходимые размеры. При инициализации для каждого компонента выделяется 50% в методе `show()`, что приводит к разделению точно пополам.

Наконец, создается вторая панель кнопок. На ней размещаются три кнопки: Reply, Forward и Delete. Для каждой кнопки добавляется слушатель `ActionListener`, который вызывает соответствующий метод при щелчке на каждой из кнопок. После создания второй панели кнопок, все созданные панели добавляются в окно.

Метод tableSelectionChanged()

Метод `tableSelectionChanged()`, листинг которого приведен ниже, вызывается каждый раз при выборе строки в таблице сообщений.

```
// Вызывается при выборе строки таблицы.
private void tableSelectionChanged() {
  /* Если сообщение не удаляется, то выбрать сообщение
     и отобразить его. */
  if ( !deleting) {
    selectedMessage =
      tableModel.getMessage(table.getSelectedRow());
    showSelectedMessage();
    updateButtons();
  }
}
```

Этот метод начинается с проверки флага удаления. Если флаг имеет значение `false`, то обновляется переменная `selectedMessage` для соответствующего сообщения из выбранной строки. Затем вызывается метод `showSelectedMessage()` для отображения выбранного сообщения. Наконец, вызывается метод `updateButtons()` для обновления состояния кнопок.

Методы actionNew(), actionForword() и actionReply()

Любой из методов `actionNew()`, `actionForward()` и `actionReply()` обслуживает вызовы метода `sendMessage()`, когда производится щелчок на соответствующих кнопках. В метод `sendMessage()` передается идентификатор, определяющий тип посланного сообщения, и выбранное сообщение `selectedMessage`.

Метод actionDelete()

В том виде, в котором написан почтовый клиент, он автоматически не удаляет сообщения с сервера. Вместо этого от может явно потребовать их удаления. Такая возможность встроена в программу для проведения надежных экспериментов без опасения потери важных сообщений. Если вы явно не удалите сообщения, то они будут

оставаться на сервере до тех пор, пока не будут удалены принудительно или при обращении с почтовому серверу с помощью другого почтового клиента.

Метод `actionDelete()`, листинг которого приведен ниже, удаляет почтовое сообщение с сервера. Он помечает удаляемое сообщение и отдает команду удалить все помеченные сообщения, которые содержатся в указанном каталоге.

В общем случае сообщения могут удаляться из каталогов JavaMail с помощью метода `expunge()` для объекта `Folder`, или при закрытии каталога с установленным флагом удаления (значение `true`).

В библиотеках `JavaMail` определено несколько флагов, которые поддерживаются классом `javax.mail.Flags` и определены в его внутреннем класса `javax.mail.Flags.Flag`. Среди них флаг `DELETED` используется для маркировки сообщения как удаленного.

Поскольку встроенный в `JavaMail` код для протокола POP3 не поддерживает метод `expunge()` для объектов типа `Folder`, то удаление производится при закрытии каталога с помощью установленного флага удаления.

```
// Удаление выбранных сообщений.
private void actionDelete() {
  deleting = true;
  try {
    // Удалить сообщение с сервера.
    selectedMessage.setFlag(Flags.Flag.DELETED, true);
    Folder folder = selectedMessage.getFolder();
    folder.close(true);
    folder.open(Folder.READ_WRITE);
  }
  catch (Exception e) {
    showError("Unable to delete message.", false);
  }
  // Удалить сообщение из таблицы.
  tableModel.deleteMessage(table.getSelectedRow());
  // Обновить графический интерфейс.
  messageTextArea.setText ("");
  deleting = false;
  selectedMessage = null;
  updateButtons();
}
```

Метод `actionDelete()` начинает удаление сообщения с установки значения `true` для переменной `deleting`. Установка флага `deleting` заставляет метод `tableSelectionChanged()` игнорировать изменения в таблице, пока сообщение удаляется. Затем производится действительное удаление сообщения, при котором выполняются следующие шаги. Сначала объект `selectedMessage` маркируется как готовый к удалению, для чего вызывается метод `setFlag()` с аргументом `Flags.Flag.DELETED`. Затем закрывается каталог с объектом `selectedMessage`. Когда в `JavaMail` закрывается каталог, все сообщения в каталоге, имеющие установленный флаг `DELETED`, удаляются. После этого закрытый каталог открывается вновь и все сообщения, содержащиеся в нем, становятся вновь доступны.

После удаления сообщения с сервера, оно удаляется из таблицы сообщений с помощью вызова метода `tableModel.deleteMessage()`. Наконец, вызывается метод `actionDelete()` для перерисовки графического интерфейса, очистки текстового поля и обновления состояния кнопок.

Метод sendMessage()

Метод `sendMessage()`, листинг которого приведен ниже, выполняет непосредственную отправку сообщения. В общих чертах его работа происходит следующим образом. Сначала отображается диалоговое окно Message, затем используется данные, введенные в диалоговом окне Message для создания нового объекта `Message`. Новый объект `Message` затем отправляется с помощью класса `Transport` из библиотеки JavaMail.

```
// Отправить выбранное сообщение.
private void sendMessage(int type, Message message) {
  // Отобразить диалоговое окно для ввода значений.
  MessageDialog dialog;
  try {
    dialog = new MessageDialog(this, type, message);
    if (!dialog.display()) {
      // Return if dialog box was cancelled,
      return;
    }
  }
  catch (Exception e) {
    showError("Unable to send message.", false);
    return;
  }
  try {
    // Создать новое сообщение со значениями из диалогового окна.
    Message newMessage = new MimeMessage(session);
    newMessage.setFrom(new InternetAddress(dialog.getFrom()));
    newMessage.setRecipient(Message.RecipientType.TO,
      new InternetAddress(dialog.getTo()));
    newMessage.setSubject(dialog.getSubject());
    newMessage.setSentDate(new Date());
    newMessage.setText(dialog.getContent());
    // Send new message.
    Transport.send(newMessage);
  }
  catch (Exception e) {
    showError("Unable to send message.", false);
  }
}
```

Сначала в методе `sendMessage()` создается объект типа `MessageDialog`, в который передаются тип сообщения и само сообщение в качестве аргументов. Объект `MessageDialog` выделяет свойства сообщения, такие как адрес `To`, адрес `From` и тема `Subject`. После обработки этих свойств создается новое сообщение — объект `Message` и заполняются его свойства. Новый адрес `From` для объекта `Message` устанавливается с помощью метода `setFrom()`. Аналогично, для заполнения адреса получателя `To` вызывается метод `setRecipient()`. Также отметим, что метод используется для заполнения адресов CC и BCC, определяя тип получателя как `Message.RecipientType.CC` или `Message.RecipientType.BCC` соответственно. Метод `setSentDate()` устанавливает временную метку в сообщение. Метод `setText()` определяет содержимое сообщения.

Наконец, сообщение отправляется с помощью вызова метода `Transport.send()`. Класс `Transport` из библиотеки JavaMail содержит специфический код на основе протокола SMTP для отправки сообщений.

Метод showSelectedMessage()

Метод `showSelectedMessage()`, листинг которого приведен ниже, загружает и затем отображает объект `selectedMessage` в текстовом пространстве графического интерфейса.

```
// Отобразить выбранное сообщение в панели содержимого.
private void showSelectedMessage() {
  // Отобразить песочные часы на время загрузки сообщения.
  setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
  try {
    messageTextArea.setText(
      getMessageContent(selectedMessage));
    messageTextArea.setCaretPosition(0);
  }
  catch (Exception e) {
    showError("Unable to load message.", false);
  }
  finally {
    // Возвратить курсор по умолчанию.
    setCursor(Cursor.getDefaultCursor());
  }
}
```

До того как загружено выбранное сообщение `selectedMessage`, для курсора приложения устанавливается вид `WAIT_CURSOR`, чтобы курсор сигнализировал о том, что приложение занято выполнением задачи. В большинстве операционных систем `WAIT_CURSOR` отображается в виде песочных часов. После установки курсора содержимое выбранного сообщения загружается с помощью метода `getMessageContent()`. Затем содержимое отображается в текстовом поле графического интерфейса и при этом вызывается метод `setCaretPosition()` для того, чтобы установить курсор текстового поля в начальную позицию. Установка курсора текстового поля в начальное положение приводит к тому, что текст прокручивается в начало. Наконец, для курсора приложения устанавливается вид, который определен по умолчанию.

Метод updateButtons()

Метод `updateButtons()`, листинг которого приведен ниже, обновляет состояние всех кнопок на панели кнопок в зависимости от того, выбрано ли сообщение в таблице.

```
/* обновляет состояние всех кнопок на панели кнопок
   в зависимости от того, выбрано ли сообщение. */
private void updateButtons() {
  if (selectedMessage != null) {
    replyButton.setEnabled(true);
    forwardButton.setEnabled(true);
    deleteButton.setEnabled(true);
  }
  else {
    replyButton.setEnabled(false);
    forwardButton.setEnabled(false);
    deleteButton.setEnabled(false);
  }
}
```

Если не выбрано ни одного сообщения, то все кнопки неактивны и отображаются серым цветом. Но если сообщение выбрано, то все кнопки активны и отображаются яркими цветами.

Метод show()

Метод `show()` переопределяет родительский метод в классе `Jframe` таким образом, чтобы панель почтового клиента, разделенная в соотношении 50/50, могла обновляться. По умолчанию объект `JSplitPane` использует предпочтительные размеры входящих компонентов для определения того, где должен проходить разделитель. Если в таблице сообщений нет ни одного сообщения, то предпочтительные размеры таблицы будут минимальны и достаточны только для того, чтобы отобразить заголовки колонок. Обычно разделитель находится в середине панели и метод `show()`, представленный ниже, вызывает метод `setDividerLocation()` для установки разделителя.

```
// Отобразить окно приложения на экране.
public void show() {
  super.show();
  // Обновить разделение панели в соотношении 50/50.
  splitPane.setDividerLocation(.5);
}
```

Метод connect()

Поскольку размеры метода `connect()` довольно велики, разобьем его на отдельные программные фрагменты и будем последовательно их рассматривать. Метод `connect()` начинается такими строками.

```
// Отобразить диалоговое окно Connect.
ConnectDialog dialog = new ConnectDialog(this);
dialog.show();
```

Перед тем как создать соединение с почтовым сервером, должны быть введены необходимые данные для создания соответствующего соединения. Для этого создается объект `ConnectDialog`, который отображает диалоговое окно для ввода данных и сохраняет введенные данные.

После того как введенные данные сохранены в объекте `ConnectDialog`, они используются для создания соединения для заданного URL с помощью библиотек `JavaMail`.

```
// Создание соединения для заданного URL с использованием
   данных из объекта Connect.
StringBuffer connectionUrl = new StringBuffer();
connectionUrl.append(dialog.getType() + "://");
connectionUrl.append(dialog.getUsername() + ":");
connectionUrl.append(dialog.getPassword() + "@");
connectionUrl.append(dialog.getServer() + "/");
```

Соединение JavaMail использует следующую схему `protocol://username:password@hostname`

Например, для соединения с сервером `mailserver.com`, пользователем `johndoe` и паролем `1234` по протоколу POP3, адрес (URL) будет выглядеть следующим образом.

```
pop3://johndoe:pass1234@mailserver.com
```

Затем инициализируется объект DownloadingDialog и информационное сообщение отображается на экране во время загрузки сообщений.

```
/* Отображение диалогового окна на время загрузки
   сообщений с сервера. */
final DownloadingDialog downloadingDialog =
  new DownloadingDialog(this);
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    downloadingDialog.show();
  }
});
```

Когда модальное диалоговое окно, подобное DownloadingDialog, отображается на экране, выполнение текущего потока приостанавливается до тех пор, пока диалоговое окно не будет закрыто. Это стандартное поведение обычно имеет положительный эффект, но оно нежелательно для почтового клиента. В случае с почтовым клиентом, диалоговое окно DownloadingDialog должно отображаться только во время процесса загрузки, но не перед загрузкой. Для того чтобы добиться этого, диалоговое окно DownloadingDialog запускается в отдельном потоке с использованием основного события для запуска потока из библиотеки Swing, для чего вызывается метод SwingUtilities.invokeLater().

После запуска диалогового окна Downloading начинается работа по созданию соединения с почтовым сервером. Сначала инициализируется сеанс JavaMail с помощью следующей последовательности команд.

```
// Запуск сеанса JavaMail и соединение с сервером.
Store store = null; try {
// Инициализация сеана JavaMail для работы с SMTP-сервером.
  Properties props = new Properties();
  props.put("mail.smtp.host", dialog.getSmtpServer());
  session = Session.getDefaultInstance(props, null);
```

Обратите внимание, что в сеанс JavaMail передается несколько свойств, которые составляют адрес SMTP-сервера. Сеанс JavaMail сохраняет настройки и информацию об аутентификации пользователя для взаимодействия с почтовым сервером.

Сохранение этой информации в объекте Session позволяет использовать ее в приложении отдельными классами JavaMai. В случае с почтовым клиентом данные сеанса используются классом Transport и методом sendMessage().

После инициализации сеанса JavaMail, производится соединение с почтовым сервером с помощью следующего кода.

```
  // Подключиться к постовому серверу.
  URLName urln = new URLName(connectionUrl.toString());
  store = session.getStore(urln);
  store.connect();
}
  catch (Exception e) {
    // Close the Downloading dialog box.
    downloadingDialog.dispose();
    // Show error dialog box.
    showError("Unable to connect.", true);
 }
```

В сеансе JavaMail используются объекты Store, содержащие само сообщение и необходимые протоколы доступа для сохранения и извлечения сообщений. Хранение сообщений аналогично почтовым учетным записям, таким образом объекты Store по существу являются интерфейсами для доступа к почтовым сообщениям по определенным протоколам, таким как POP3 или IMAP. В предшествующем коде реальное соединение с почтовым сервером устанавливается с помощью извлечения, сохраняемого в объекте Store протокола, введенного через диалоговое окно Connect и получаемого с помощью вызова метода connect().

Сначала инициализируется объект URLName с использованием адреса, собранного ранее на основе данных, введенных в диалоговом окне Connect. Затем объект URLName передается в метод getStore() для данного сеанса. Метод getStore() возвращает объект Store с протоколом, соответствующим объекту URLName. После того как объект Store получен, вызывается метод connect().

Напомним, что POP3-сервером поддерживают работу только с одним каталогом, тогда как IMAP-серверы могут использовать несколько каталогов. Таким образом, при успешном соединении с почтовым сервером по умолчанию открывается каталог "INBOX" и из него извлекаются сообщения, как показано в следующем коде.

```
// Загрузка заголовков сообщений с сервера.
try {
  // Открыть каталог "INBOX".
  Folder folder = store.getFolder("INBOX");
  folder.open(Folder.READ_WRITE);
  Message[] messages = folder.getMessages();
```

Когда список сообщений извлекается из каталога, как показано в предыдущем листинге, каждый объект Message будет пустым. Метод getMessages() для объекта Folder просто возвращает массив пустых объектов Message, каждый из которых представляет почтовое сообщение из каталога. Объекты JavaMail используют эту технику для того, чтобы разрешить производить загрузку сообщений по требованию, минимизируя таким образом размер данных, которые должны быть загружены с почтового сервера.

После того как почтовый клиент получил данные о сообщениях, он отображает список сообщений в таблице сообщений. В следующем коде производится автоматическое извлечение заголовков сообщений вместо их загрузки по требованию.

```
  // Извлечение заголовков для каждого сообщения в каталоге.
  FetchProfile profile = new FetchProfile();
  profile.add(FetchProfile.Item.ENVELOPE);
  folder.fetch(messages, profile);
  // Записать сообщение в таблицу.
  tableModel.setMessages(messages);
}
  catch (Exception e) {
  // Закрыть диалоговое окно Downloading.
  downloadingDialog.dispose();
  // Отобразить диалоговое окно ошибки.
  showError("Unable to download messages.", true);
}
```

Для того чтобы предварительно загрузить сообщение, создается объект FetchProfile. Отдельные выборки используются для создания порций сообщений, которые должны быть предварительно загружены или выбраны. Объект FetchProfile,

созданный в предыдущем коде, выбирает отдельные порции сообщений. Эти порции объединяются в общие заголовки сообщений, такие как отправитель, получатель и тема.

После предварительного извлечения сообщений из полей сообщений с помощью объекта `FetchProfile`, сообщения добавляются в таблицу сообщений с помощью метода `tableModel.setMessages()`. Таблица сообщений обновляется, включая каждое загруженное с сервера сообщение.

Метод `connect()` завершается закрытием модального диалогового окна `DownloadingDialog`, как показано ниже. Теперь пользователь может вновь использовать почтовый клиент.

```
// Закрыть модальное диалоговое окно.
downloadingDialog.dispose();
```

Метод showError()

Метод `showError()`, листинг которого приведен ниже, отображает диалоговое окно ошибки на экране с заданным сообщением. Заметьте, что метод `showError()` в качестве аргумента получает флаг `exit`, который определяет, должно ли приложение закончить работу после отображения сообщения об ошибке.

```
// Отобразить диалоговое окно ошибок и при необходимости выйти.
private void showError(String message, boolean exit) {
  JOptionPane.showMessageDialog(this , message, "Error",
    JOptionPane.ERROR_MESSAGE);
  if (exit)
    System.exit(0);
}
```

Метод getMessageContent()

Метод `getMessageContent()`, листинг которого приведен ниже, извлекает содержимое сообщения. Обратите внимание, что метод объявлен как `public` и `static`, так что к нему может быть произведен доступ из других классов без требования ссылки на класс `EmailClient`.

```
// Получить содержимое сообщения.
public static String getMessageContent(Message message)
  throws Exception {
  Object content = message.getContent();
  if (content instanceof Multipart) {
    StringBuffer messageContent = new StringBuffer();
    Multipart multipart = (Multipart) content;
    for (int i = 0; i < multipart.getCount(); i++) {
      Part part = (Part) multipart.getBodyPart(i);
      if (part.isMimeType("text/plain")) {
        messageContent.append(part.getContent().toString());
      }
    }
    return messageContent.toString();
  }
  else {
    return content.toString();
  }
}
```

Большинство сообщений электронной почты содержат только текст, но отдельные из сообщений включают и другую информацию. Например, некоторые сообщения содержат дополнения в формате HTML или дополнительные файлы. Когда сообщение включает информацию в формате HTML, обычно оно содержит и текстовую версию этого содержимого. В этом случае содержимое в формате HTML и текстовая версия находятся в разных частях. Сообщения более чем с одной частью называется составным сообщением. Сообщения без дополнений работают точно так же, как и приложения с дополнительной информацией в формате HTML. Дополнение находится в одной из частей сообщения.

Поскольку разработанный выше почтовый клиент обрабатывает только текстовые сообщения, метод `getMessageContent()` производит проверку содержимого сообщения для определения того, является ли оно составным. Если сообщение составное, тогда метод `message.getContent()` возвращает объект `Multipart`. Объект `Multipart` из JavaMail подобен стандартной коллекции Java, к которой инкапсулирован список частей для составного сообщения.

Если метод `message.getContent()` возвращает объект `Multipart`, производится последовательный просмотр объекта и выделяются текстовые части сообщения. Наоборот, если содержимое, возвращаемое методом `message.getContent()` не является составным, тогда метод объекта `toString()` вызывается для возврата обычного текста.

Компиляция и запуск почтового клиента

Как уже упоминалось ранее, на компьютере необходимо иметь установленные библиотеки JavaMail JavaBeans. Следует активизировать библиотеку Framework, поставляемую с пакетом Java, перед тем как запускать программу почтового клиента. Если необходимые библиотеки установлены, нужно добавить их пути в раздел Java classpath. Но этого нельзя делать с помощью модификации переменной окружения CLASSPATH. Как альтернативный способ, можно определить путь при компиляции и запуске почтового клиента. Например, если самым верхним каталогом для JavaMail будет `javamail-1.3`, а каталогом для JavaBeans Activation Framework — `jaf-1.0.2`, тогда можно использовать следующую командную строку для компиляции почтового клиента.

```
javac -classpath.;c:\javamail-1.3\mail.jar;c:\jaf-1.0.2\activation.jar
EmailClient.Java MessagesTableModel.Java ConnectDialog.Java Downloading-
Dialog.Java MessageDialog.Java
```

Файлы `mail.jar` и `activation.jar` содержат классы, используемые JavaMail и JavaBeans Activation Framework.

Для запуска почтового клиента используйте следующую командную строку.

```
javaw -cp.;c:\javamail-1.3\mail.jar;c:\jaf-1.0.2\activation.jar EmailClient
```

Обратите внимание, что пути к файлам `mail.jar` и `activation.jar` необходимо указать явно. Разумеется, если модифицировать переменную `CLASSPATH`, то явно указывать пути уже не надо.

После запуска почтового клиента работа с ним не составляет труда. Сначала на экране появляется диалоговое окно Connect для вода необходимых данных. Сделайте все установки и щелкните на кнопке Connect. После того как будет установлено

соединение с сервером, произойдет загрузка почтовых сообщений и они будут отображены в таблице сообщений. Для просмотра сообщений просто выберите его в таблице, и содержимое будет отображено в текстовом поле. После выбора сообщения оно может быть перенаправлено, удалено, или на него можно ответить, для чего производится щелчок на соответствующих кнопках внизу приложения. Помните два момента. Во-первых, поддерживаются только текстовые сообщения. Во-вторых, ни одно из сообщений не удаляется с сервера, пока вы явно этого не потребуете. На рис. 5.5 показан почтовый клиент в работе.

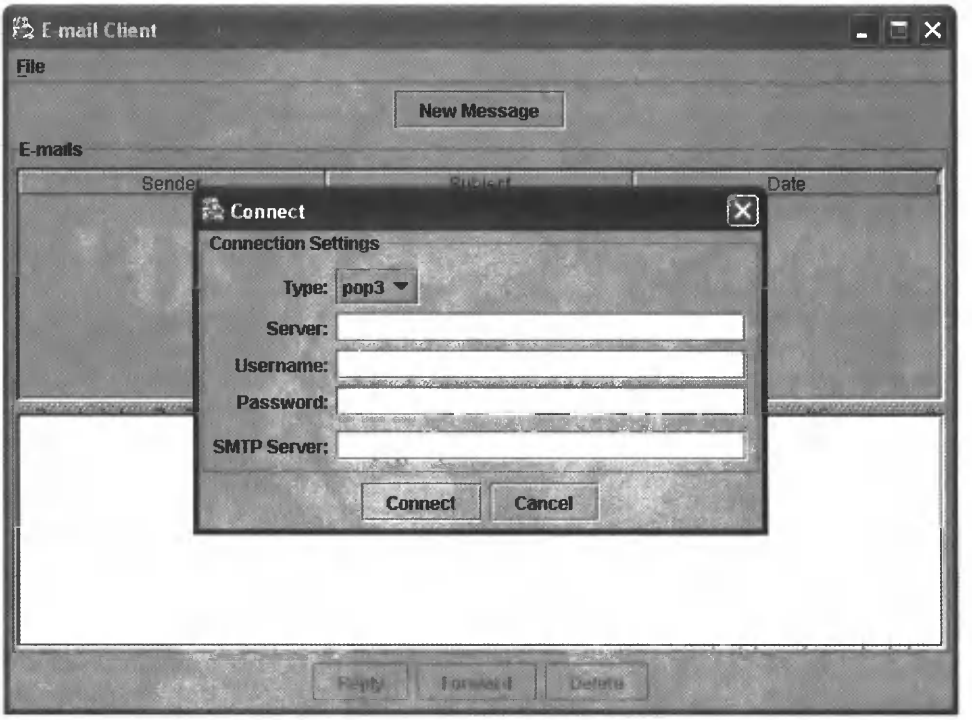


Рис. 5.5. Почтовый клиент в работе

Расширение возможностей почтового клиента

Код, содержащийся в этой главе, предоставляет прекрасную экспериментальную платформу для разработчиков приложений почтового клиента. Он также является отличной отправной точкой при разработке собственных приложений почтового клиента. Ниже перечислено несколько улучшений, которые вы можете добавить в собственные разработки почтового клиента.

- Добавить сохранение настроек соединения для того, чтобы не вводить их каждый раз при запуске.
- Добавить возможность проверять и загружать новые почтовые сообщения с сервера.
- Добавить возможность сохранять все части составного приложения.

- Добавить возможность обрабатывать сообщения в формате HTML.
- Добавить адресную книгу.

Наконец, возможно, вы захотите сократить функциональные возможности. Например, можно встроить в другое приложение только часть возможностей почтового клиента для отправки сообщений. При этом приложение может отправлять сообщения автоматически, при наступлении определенных событий. Хорошо усвоив работу почтового клиента и научившись управлять всеми его возможностями, вполне реально создать много полезных и оригинальных приложений.

ГЛАВА

6

Поиск в Web с помощью Java

Наверно, вы неоднократно восхищались, как поисковые движки Internet, подобные Google или Yahoo, могут почти мгновенно найти в возвратить результирующий список поиска по заданному критерию. Очевидно, что нереально сканировать все сайты Internet для получения результатов при каждом запросе. Вместо поиска в Internet, поисковые движки обращаются к высокооптимизированным базам данных Web-страниц, которые предварительно обработаны и индексированы. Предварительно подготовленные базы данных позволяют поисковым движкам просматривать миллиарды Web-страниц для поиска той или иной информации.

Причем эффективность этого поиска зависит не от самих баз данных Web-страниц, а от того, как эти базы данных создаются. Поисковые движки используют программное обеспечение, известное как Web-червь, который внедряется в Internet и сохраняет каждую из отдельных страниц, встречающихся на пути. Затем поисковые движки используют дополнительное программное обеспечение для индексации каждой из найденных страниц, создавая базы данных, содержащие ключевые слова этих страниц.

Web-червь является существенным компонентом для поисковых движков, однако его использование не ограничивается только созданием баз данных для Web-страниц. На самом деле Web-червь может использоваться во многих случаях. Например, можно использовать Web-червь для поиска разрушенных соединений на коммерческом Web-сайте. Также можно создать червь для поиска изменений на Web-сайте. Для того чтобы сделать это, червь сначала создает запись всех связей, содержащихся на сайте. Через некоторое время червь вновь исследует сайт и сравнивает два набора записей для поиска изменений. Web-червь также может использоваться для архивации содержимого сайта. В общем случае технология, заложенная в основе Web-червя, может использоваться во многих типах приложений, имеющих отношение к Web.

Хотя основные идеи, заложенные в основе Web-червя, довольно хорошо известны (необходимо только прослеживать связи между сайтами), создать его не так просто. Одна из трудностей состоит в том, что список связей постоянно увеличивается по мере прохождения червя по сети, и этот список необходимо поддерживать в рабочем состоянии. Еще она сложность связана с обработкой параллельных связей. К счастью, в Java включены возможности, которые могут быть полезны при создании Web-червя и облегчают эту работу. Во-первых, Java предоставляет средства для работы с сетью и при этом позволяет легко загружать Web-сайты. Во-вторых, в Java имеются стандартные выражения, упрощающие реализацию фрагмента для поиска связей. В-третьих, библиотеки Collection Framework содержат механизм, необходимый при сохранении списка связей.

Web-червь, разработанный в этой главе, называется поисковым (Search Crawler). Он внедряется в Web и ищет сайты, которые содержат страницы, содержащие строки, указанные пользователем. При этом он отображает адреса тех сайтов, которые соответствуют условиям поиска. Хотя поисковый червь сам по себе является полезной утилитой, наибольшую пользу от него можно получить при разработке собственных приложений на основе Web-червя.

Основы построения Web-червя

Несмотря на многообразие приложений для Web-червя, основы построения их ядра одни и те же. Рассмотрим схему работы Web-червя:

- загрузка Web-страницы;
- синтаксический анализ загруженной страницы и извлечение всех связей;
- повторение процесса для каждой из извлеченных связей.

Рассмотрим каждый шаг более подробно.

На первом шаге Web-червь получает URL и загружает страницу из Internet, расположенную по данному адресу. Загруженная страница сохраняется в файле на диске или помещается в базу данных. Сохранение страницы позволяет червю или другому программному обеспечению обратиться к ней позже для совершения определенных операций, таких как индексация слов (как в случае с поисковым движком) или архивация страницы при использовании ее автоматическим архиватором.

На втором шаге Web-червь производит синтаксический анализ загруженной страницы и находит все ссылки на другие страницы. Каждая ссылка определяется по наличию HTML-заголовка, подобного показанному ниже.

```
<AHREF="http://www.host.com/directory/file.html">Link</A>
```

После того как червь извлечет все ссылки данной страницы, каждая ссылка добавляется в список ссылок для дальнейшей обработки Web-червем.

На третьем шаге Web-червь повторяет процесс. Все Черви работают рекурсивно или в цикле, но при этом ссылки могут обрабатываться червем последовательно в глубину или в ширину. Черви, исследующие ссылки в глубину, просматривают каждый возможный путь до его логического завершения, прежде чем перейти к следующему пути. Они начинают работу с первой ссылки на первой странице. Затем обращаются к странице, найденной по первой ссылке, на ней находят первую ссылку, обращаются к следующей странице и т.д., пока не будет достигнута последняя страница. Процесс продолжается до тех пор, пока не будут просмотрены все ветви для всех ссылок.

Черви, работающие в ширину, проверяют каждую ссылку на странице, прежде чем перейти к следующей странице. Таким образом, червь просматривает все ссылки на первой странице, затем просматривает все ссылки на странице для первой ссылки и т.д., пока на каждом уровне не будут просмотрены все ссылки. Выбор между просмотром в глубину и просмотром в ширину зависит от того, в каком приложении червь используется, а также от ваших предпочтений. Например, поисковый червь использует просмотр в ширину, но вы можете это изменить.

Хотя на первый взгляд работа Web-червя кажется довольно простой, необходимо предусмотреть довольно много вещей, чтобы создать законченное приложение. Например, для Web-червя необходимо использовать "протокол робота", о котором речь пойдет в следующем разделе. Web-червь также должен обрабатывать много сценариев исключений, таких как ошибка Web-сервера, переориентирование и т.д.

Протокол робота

Только вообразите себе, как Web-червь может просматривать бесчисленное количество ссылок из ресурсов Web-сервера с многочисленными обратными связями. Обычно одновременно загружается всего несколько страниц, а не сотни или тысячи.

При этом часто Web-сайты имеют некоторые зоны, которые Web-червь не должен просматривать. Для того чтобы исключить все возможные промахи, Web-сайты часто используют протокол робота, который содержит основные принципы, которым должен следовать Web-червь. Со временем протокол станет неписаным законом для Web-червей в Internet.

Протокол робота устанавливает, как Web-сайты ограничивают для Web-червя доступ к определенным зонам или страницам с помощью файла с именем `robots.txt`, размещенного в корневом каталоге Web-сайта. Хорошо разработанный Web-червь будет обращаться в этому файлу и определять, какие части Web-сайта закрыты для прохождения. Такой Web-червь не должен проходить закрытые пространства. В следующем примере приведен образец файла `robots.txt` и объясняется его формат.

```
# robots.txt for http://somehost.com/

User-agent: *
Disallow: /cgi-bin/
Disallow: /registration  # Disallow robots on registration page
Disallow: /login
```

В первой строке примера находится комментарий, который выделяется с помощью символа решетки (#). Комментарии могут размещаться полностью на строке или на строке с утверждением, как показано в 5-й строке примера. При считывании файлов `robots.txt` Web-червь должен игнорировать любые комментарии.

В 3-й строке примера указан *агент пользователя* (User-agent), для которого определены *привила сокрытия* (Disallow), которым необходимо следовать. Термин "агент пользователя" используется для программ, которые производят доступ к Web-сайтам. Например, при доступе к Web-сайту с помощью браузера Microsoft's Internet Explorer, агентом пользователя является программа Mozilla/4.0 (совместимая с MSIE 6.0; Windows NT 5.0) или подобная ей. Каждый браузер имеет уникальный агент пользователя, который он отправляет на Web-сервер при каждом запросе. Web-червь также обычно посылает агента пользователя при каждом обращении к Web-серверу. Использование агента пользователя в файлах `robots.txt` позволяет Web-сайтам устанавливать правила по принципу агент пользователя – агента пользователя. Однако обычно Web-сайты стремятся запретить доступ для всех роботов (или агентов пользователя), и для этого используется символ звездочки (`*`) для агента пользователя. Это говорит о том, что все агенты пользователя не должны использовать правила, которые заложены в них самих. Можно подумать, что использование звездочки запретит всем агентам пользователя доступ к сайту, а также запретит стандартному программному обеспечению браузера работать с определенными разделами Web-сайта. Но это не является проблемой, так как обычно браузеры не используют протокол робота.

В следующих строках агента пользователя вызываются утверждения для закрытия разделов. Эти утверждения определяют пути на Web-сайте, которые Web-червь не должен просматривать. Например, первое утверждение запрещает Web-червю просматривать ссылки, которые начинаются с символов "`/cgi-bin/`". Таким образом, оба адреса

```
http://somehost.com/cgi-bin/
http://somehost.com/cgi-bin/register
```

являются недоступными для Web-червя. Запретительные утверждения указывают пути, а не отдельные файлы, соответственно, любые ссылки, в которых содержится указанный путь, не будут просматриваться.

Обзор поискового червя

Поисковый червь лежит в основе Web-червя для поиска в Web, и он хорошо иллюстрирует фундаментальное построение приложений на основе Web-червя. Для поискового червя можно ввести критерии поиска и производить поиск в реальном времени, просматривая адрес за адресом и производя отбор в соответствии с заданным критерием.

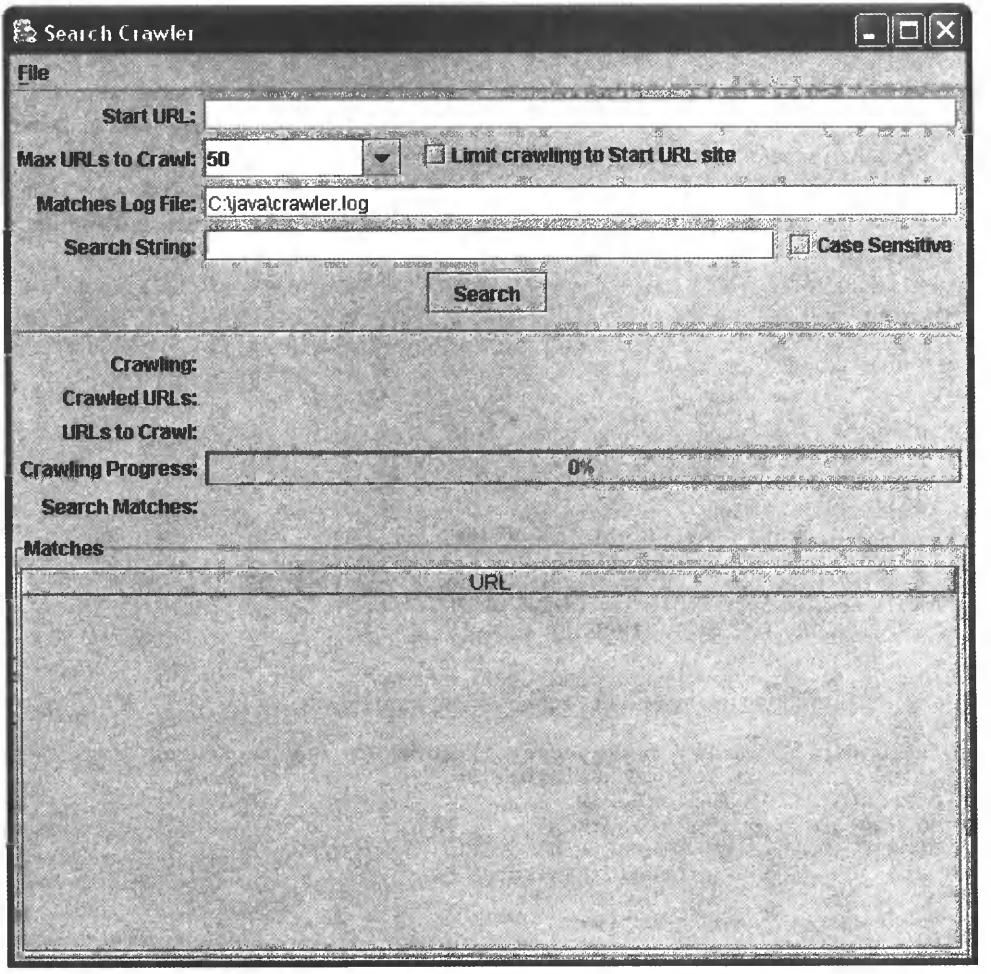


Рис. 6.1. Графический интерфейс пользователя для поискового червя

Графический интерфейс поискового червя показан на рис. 6.1. Он имеет три отдельные секции, на которые в дальнейшем можно будет ссылаться как на Поиск (Search),

Статистику (Stats) и Совпадение (Matches). Секция поиска находится в верхней части окна и содержит элементы управления для ввода критериев поиска, включая начальный адрес поиска, максимальное число адресов для просмотра и строку поиска. Критерий поиска может уточняться с помощью ограничения поиска только на начальном сайте и выбора флага Case Sensitive для строки поиска.

Секция статистики размещена в середине окна и включает элементы управления для отображения текущего состояния поиска. В этой секции также расположен индикатор выполнения для наглядного отображения количества произведенных просмотров.

Секция совпадений находится внизу окна и содержит список всех найденных совпадений в процессе поиска. Это будут адреса Web-страниц, которые содержат строки, заданные для поиска.

Класс SearchCrawler

В классе `SearchCrawler` находится метод `main()`, так что при запуске на выполнение этот метод будет вызван первым. Метод `main()` создает новый объект `SearchCrawler` и затем вызывает метод `show()` для того, чтобы отобразить этот объект.

Листинг класса `SearchCrawler` представлен ниже, а подробный разбор отдельных фрагментов будет приведен в последующих разделах. Обратите внимание, что класс `SearchCrawler` является наследником класса `Jframe`.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.regex.*;
import javax.swing.*;
import javax.swing.table.*;

// Поисковый Web-червь.
public class SearchCrawler extends JFrame
{
  // Максимальное количество раскрывающихся значений URL.
  private static final String[] MAX_URLS =
    {"50", "100", "500", "1000"};

  // Кэш-память для списка ограничений робота.
  private HashMap disallowListCache = new HashMap();

  // Элементы управления графического интерфейса панели Search.
  private JTextField startTextField;
  private JComboBox maxComboBox;
  private JCheckBox limitCheckBox;
  private JTextField logTextField;
  private JTextField searchTextField;
  private JCheckBox caseCheckBox;
  private JButton searchButton;

  // Элементы управления графического интерфейса панели Stats.
  private JLabel crawlingLabel2;
  private JLabel crawledLabel2;
  private JLabel toCrawlLabel2;
  private JProgressBar progressBar;
  private JLabel matchesLabel2;
```

```
// Список соответствий.
private JTable table;

// Флаг отображения состояния поиска.
private boolean crawling;

// Файл журнала для текстового вывода.
private PrintWriter logFileWriter;

// Конструктор для поискового червя.
public SearchCrawler()
{
  // Установка заголовка приложения.
  setTitle("Search Crawler");

  // Установка размеров окна.
  setSize(600, 600);

  // Обработка событий по закрытию окна.
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      actionExit();
    }
  });

  // Установить меню "файл".
  JMenuBar menuBar = new JMenuBar();
  JMenu fileMenu = new JMenu("File");
  fileMenu.setMnemonic(KeyEvent.VK_F);
  JMenuItem fileExitMenuItem = new JMenuItem("Exit",
    KeyEvent.VK_X);
  fileExitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionExit();
    }
  });
  fileMenu.add(fileExitMenuItem);
  menuBar.add(fileMenu);
  setJMenuBar(menuBar);

  // Установить панель поиска.
  JPanel searchPanel = new JPanel();
  GridBagConstraints constraints;
  GridBagLayout layout = new GridBagLayout();
  searchPanel.setLayout(layout);

  JLabel startLabel = new JLabel("Start URL:");
  constraints = new GridBagConstraints();
  constraints.anchor = GridBagConstraints.EAST;
  constraints.insets = new Insets(5, 5, 0, 0);
  layout.setConstraints(startLabel, constraints);
  searchPanel.add(startLabel);

  startTextField = new JTextField();
  constraints = new GridBagConstraints();
  constraints.fill = GridBagConstraints.HORIZONTAL;
  constraints.gridwidth = GridBagConstraints.REMAINDER;
  constraints.insets = new Insets(5, 5, 0, 5);
  layout.setConstraints(startTextField, constraints);
  searchPanel.add(startTextField);
```

```
JLabel maxLabel = new JLabel("Max URLs to Crawl:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(maxLabel, constraints);
searchPanel.add(maxLabel);

maxComboBox = new JComboBox(MAX_URLS);
maxComboBox.setEditable(true);
constraints = new GridBagConstraints();
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(maxComboBox, constraints);
searchPanel.add(maxComboBox);

limitCheckBox =
  new JCheckBox("Limit crawling to Start URL site");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.WEST;
constraints.insets = new Insets(0, 10, 0, 0);
layout.setConstraints(limitCheckBox, constraints);
searchPanel.add(limitCheckBox);

JLabel blankLabel = new JLabel();
constraints = new GridBagConstraints();
constraints.gridwidth = GridBagConstraints.REMAINDER;
layout.setConstraints(blankLabel, constraints);
searchPanel.add(blankLabel);

JLabel logLabel = new JLabel("Matches Log File:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(logLabel, constraints);
searchPanel.add(logLabel);

String file =
  System.getProperty("user.dir") +
  System.getProperty("file.separator") +
  "crawler.log";
logTextField = new JTextField(file);
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(logTextField, constraints);
searchPanel.add(logTextField);

JLabel searchLabel = new JLabel("Search String:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(searchLabel, constraints);
searchPanel.add(searchLabel);

searchTextField = new JTextField();
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.insets = new Insets(5, 5, 0, 0);
constraints.gridwidth= 2;
constraints.weightx = 1.0d;
```

```
    layout.setConstraints(searchTextField, constraints);
    searchPanel.add(searchTextField);

    caseCheckBox = new JCheckBox("Case Sensitive");
    constraints = new GridBagConstraints();
    constraints.insets = new Insets(5, 5, 0, 5);
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    layout.setConstraints(caseCheckBox, constraints);
    searchPanel.add(caseCheckBox);

    searchButton = new JButton("Search");
    searchButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        actionSearch();
      }
    });
    constraints = new GridBagConstraints();
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 5, 5);
    layout.setConstraints(searchButton, constraints);
    searchPanel.add(searchButton);

    JSeparator separator = new JSeparator();
    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 5, 5);
    layout.setConstraints(separator, constraints);
    searchPanel.add(separator);

    JLabel crawlingLabel1 = new JLabel("Crawling:");
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(crawlingLabel1, constraints);
    searchPanel.add(crawlingLabel1);

    crawlingLabel2 = new JLabel();
    crawlingLabel2.setFont(
      crawlingLabel2.getFont().deriveFont(Font.PLAIN));
    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 0, 5);
    layout.setConstraints(crawlingLabel2, constraints);
    searchPanel.add(crawlingLabel2);

    JLabel crawledLabel1 = new JLabel("Crawled URLs:");
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(5, 5, 0, 0);
    layout.setConstraints(crawledLabel1, constraints);
    searchPanel.add(crawledLabel1);

    crawledLabel2 = new JLabel();
    crawledLabel2.setFont(
      crawledLabel2.getFont().deriveFont(Font.PLAIN));
    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 0, 5);
```

```
layout.setConstraints(crawledLabel2, constraints);
searchPanel.add(crawledLabel2);

JLabel toCrawlLabel1 = new JLabel("URLs to Crawl:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(toCrawlLabel1, constraints);
searchPanel.add(toCrawlLabel1);

toCrawlLabel2 = new JLabel();
toCrawlLabel2.setFont(
  toCrawlLabel2.getFont().deriveFont(Font.PLAIN));
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(toCrawlLabel2, constraints);
searchPanel.add(toCrawlLabel2);

JLabel progressLabel = new JLabel("Crawling Progress:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(progressLabel, constraints);
searchPanel.add(progressLabel);

progressBar = new JProgressBar();
progressBar.setMinimum(0);
progressBar.setStringPainted(true);
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(progressBar, constraints);
searchPanel.add(progressBar);

JLabel matchesLabel1 = new JLabel("Search Matches:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 10, 0);
layout.setConstraints(matchesLabel1, constraints);
searchPanel.add(matchesLabel1);

matchesLabel2 = new JLabel();
matchesLabel2.setFont(
  matchesLabel2.getFont().deriveFont(Font.PLAIN));
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 10, 5);
layout.setConstraints(matchesLabel2, constraints);
searchPanel.add(matchesLabel2);

// Установить таблицу совпадений.
table =
  new JTable(new DefaultTableModel(new Object[][]{},
    new String[]{"URL"}) {
  public boolean isCellEditable(int row, int column)
  {
    return false;
```

```
      }
    });

    // Установить панель совпадений.
    JPanel matchesPanel = new JPanel();
    matchesPanel.setBorder(
      BorderFactory.createTitledBorder("Matches"));
    matchesPanel.setLayout(new BorderLayout());
    matchesPanel.add(new JScrollPane(table),
      BorderLayout.CENTER);

    // Отобразить панели на дисплее.
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(searchPanel, BorderLayout.NORTH);
    getContentPane().add(matchesPanel, BorderLayout.CENTER);
  }

  // Выход из программы.
  private void actionExit() {
    System.exit(0);
  }

  // Обработать щелчок на кнопке search/stop.
  private void actionSearch() {
    // Если произведен щелчок на кнопке stop, сбросить флаг.
    if (crawling) {
      crawling = false;
      return;
    }
    ArrayList errorList = new ArrayList();

    // Проверить ввод начального адреса (URL).
    String startUrl = startTextField.getText().trim();
    if (startUrl.length() < 1) {
      errorList.add("Missing Start URL.");
    }
    // Проверить начальный URL.
    else if (verifyUrl(startUrl) == null) {
      errorList.add("Invalid Start URL.");
    }

    /* Проверить, что введено значение для максимально допустимого
       количества адресов и что это число. */
    int maxUrls = 0;
    String max = ((String) maxComboBox.getSelectedItem()).trim();
    if (max.length() > 0) {
      try {
        maxUrls = Integer.parseInt(max);
      } catch (NumberFormatException e) {
      }
      if (maxUrls < 1) {
        errorList.add("Invalid Max URLs value.");
      }
    }

    // Проверить, что файл с журналом совпадений существует.
    String logFile = logTextField.getText().trim();
    if (logFile.length() < 1) {
      errorList.add("Missing Matches Log File.");
    }
```

```
    // Проверить, что введена стока для поиска.
    String searchString = searchTextField.getText().trim();
    if (searchString.length() < 1) {
      errorList.add("Missing Search String.");
    }

    // Показать ошибки, если они есть, и возврат.
    if (errorList.size() > 0) {
      StringBuffer message = new StringBuffer();

      // Объединить ошибки в одно сообщение.
      for (int i = 0; i < errorList.size(); i++) {
        message.append(errorList.get(i));
        if (i + 1 < errorList.size()) {
          message.append("\n");
        }
      }
      showError(message.toString());
      return;
    }

    // Удалить символы "www" из начального URL, если они есть.
    startUrl = removeWwwFromUrl(startUrl);

    // Запустить поискового червя.
    search(logFile, startUrl, maxUrls, searchString);
  }
  private void search(final String logFile, final String startUrl,
    final int maxUrls, final String searchString)
  {
    // Начать поиск в новом потоке.
    Thread thread = new Thread(new Runnable() {
      public void run() {
        // Отобразить песочные часы на время работы поискового червя.
        setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

        // Заблокировать элементы управления поиска.
        startTextField.setEnabled(false);
        maxComboBox.setEnabled(false);
        limitCheckBox.setEnabled(false);
        logTextField.setEnabled(false);
        searchTextField.setEnabled(false);
        caseCheckBox.setEnabled(false);

        // Переключить кнопку поиска в состояние "Stop."
        searchButton.setText("Stop");

        // Переустановить панель Stats.
        table.setModel(new DefaultTableModel(new Object[][]{},
          new String[]{"URL"}) {
          public boolean isCellEditable(int row, int column)
          {
            return false;
          }
        });
        updateStats(startUrl, 0, 0, maxUrls);

        // Открыть журнал совпадений.
        try {
          logFileWriter = new PrintWriter(new FileWriter(logFile));
        } catch (Exception e) {
```

```
            showError("Unable to open matches log file.");
            return;
          }

          // Установить флаг поиска.
          crawling = true;

          // Выполнять реальный поиск.
          crawl(startUrl, maxUrls, limitCheckBox.isSelected(),
            searchString, caseCheckBox.isSelected());

          // Сбросить флаг поиска.
          crawling = false;

          // Закрыть журнал совпадений.
          try {
            logFileWriter.close();
          } catch (Exception e) {
            showError("Unable to close matches log file.");
          }

          // Отметить окончание поиска.
          crawlingLabel2.setText("Done");

          // Разблокировать элементы контроля поиска.
          startTextField.setEnabled(true);
          maxComboBox.setEnabled(true);
          limitCheckBox.setEnabled(true);
          logTextField.setEnabled(true);
          searchTextField.setEnabled(true);
          caseCheckBox.setEnabled(true);

          // Переключить кнопку поиска в состояние "Search."
          searchButton.setText("Search");

          // Возвратить курсор по умолчанию.
          setCursor(Cursor.getDefaultCursor());

          // Отобразить сообщение, если строка не найдена.
          if (table.getRowCount() == 0) {
            JOptionPane.showMessageDialog(SearchCrawler.this,
              "Your Search String was not found. Please try another.",
              "Search String Not Found",
              JOptionPane.WARNING_MESSAGE);
          }
        }
      });
      thread.start();
    }

    // Отобразить диалоговое окно с сообщением об ошибке.
    private void showError(String message) {
      JOptionPane.showMessageDialog(this, message, "Error",
        JOptionPane.ERROR_MESSAGE);
    }

    // Обновить панель stats.
    private void updateStats(
      String crawling, int crawled, int toCrawl, int maxUrls)
    {
      crawlingLabel2.setText(crawling);
```

```
    crawledLabel2.setText("" + crawled);
    toCrawlLabel2.setText("" + toCrawl);

    // Обновить индикатор выполнения.
    if (maxUrls == -1) {
      progressBar.setMaximum(crawled + toCrawl);
    } else {
      progressBar.setMaximum(maxUrls);
    }
    progressBar.setValue(crawled);
    matchesLabel2.setText("" + table.getRowCount());
  }

  // Добавить совпадение в таблицу совпадений и в журнал совпадений.
  private void addMatch(String url) {
    // Добавить URL в таблицу совпадений.
    DefaultTableModel model =
      (DefaultTableModel) table.getModel();
    model.addRow(new Object[]{url});

    // Добавить URL в журнал совпадений.
    try {
      logFileWriter.println(url);
    } catch (Exception e) {
      showError("Unable to log match.");
    }
  }

  // Проверить формат URL.
  private URL verifyUrl(String url) {
    // Разрешить только адреса HTTP.
    if (!url.toLowerCase().startsWith("http://"))
      return null;

    // Проверить формат URL.
    URL verifiedUrl = null;
    try {
      verifiedUrl = new URL(url);
    } catch (Exception e) {
      return null;
    }
    return verifiedUrl;
  }

  // Проверить, если робот разрешает доступ к данному URL.
  private boolean isRobotAllowed(URL urlToCheck) {
    String host = urlToCheck.getHost().toLowerCase();

    // Извлечь список ограничений сайта из кэш-памяти.
    ArrayList disallowList =
      (ArrayList) disallowListCache.get(host);

    // Если в кэш-памяти нет списка, загрузить его.
    if (disallowList == null) {
      disallowList = new ArrayList();
      try {
        URL robotsFileUrl =
          new URL("http://" + host + "/robots.txt");

        // Открыть файл робота заданного URL для чтения.
        BufferedReader reader =
```

```
          new BufferedReader(new InputStreamReader(
            robotsFileUrl.openStream()));

        // Прочитать файл робота, создать список запрещенных путей.
        String line;
        while ((line = reader.readLine()) != null) {
          if (line.indexOf("Disallow:") == 0) {
            String disallowPath =
              line.substring("Disallow:".length());

            /* Просмотреть список запрещенных путей и удалить
               комментарии, если они есть. */
            int commentIndex = disallowPath.indexOf("#");
            if (commentIndex != - 1) {
              disallowPath =
                disallowPath.substring(0, commentIndex);
            }

            // Удалить начальные или конечные пробелы из
            // запрещенных путей.
             disallowPath = disallowPath.trim();

            // Добавить запрещенные пути в список.
            disallowList.add(disallowPath);
          }
        }

        // Добавить новый список в кэш-память.
        disallowListCache.put(host, disallowList);
      }
      catch (Exception e) {
        /* Использовать присвоенного робота после генерации
           исключительной ситуации при отсутствии файла робота. */
        return true;
      }
    }

    /* Просмотр списка запрещенных путей для проверки
       нахождения в нем заданного URL. */
    String file = urlToCheck.getFile();
    for (int i = 0; i < disallowList.size(); i++) {
      String disallow = (String) disallowList.get(i);
      if (file.startsWith(disallow)) {
        return false;
      }
    }
    return true;
  }

  // Загрузить страницу с заданным URL.
  private String downloadPage(URL pageUrl) {
     try {
        // Открыть соединение по заданному URL для чтения.
        BufferedReader reader =
          new BufferedReader(new InputStreamReader(
            pageUrl.openStream()));

        // Считать в буфер.
        String line;
        StringBuffer pageBuffer = new StringBuffer();
        while ((line = reader.readLine()) != null) {
```

```
        pageBuffer.append(line);
      }

      return pageBuffer.toString();
    } catch (Exception e) {
    }
    return null;
}

// Удалить начальные символы "www" из адреса, если они присутствуют.
private String removeWwwFromUrl(String url) {
  int index = url.indexOf("://www.");
  if (index != -1) {
    return url.substring(0, index + 3) +
      url.substring(index + 7);
  }
  return (url);
}

// Произвести синтаксический анализ и возвратить ссылки.
private ArrayList retrieveLinks(
  URL pageUrl, String pageContents, HashSet crawledList,
  boolean limitHost)
{
  // Компилировать ссылки шаблонов совпадений.
  Pattern p =
    Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
      Pattern.CASE_INSENSITIVE);
  Matcher m = p.matcher(pageContents);

  // Создать список совпадающих ссылок.
  ArrayList linkList = new ArrayList();
  while (m.find()) {
    String link = m.group(1).trim();

    // Пропустить пустые ссылки.
    if (link.length() < 1) {
      continue;
    }

    // Пропустить ссылки, которые указывают на заданную страницу.
    if (link.charAt(0) == '#') {
      continue;
    }

    // Пропустить ссылки, которые используются
    // для почтовых отправлений.
    if (link.indexOf("mailto:") != -1) {
      continue;
    }

    // Пропустить ссылки на сценарии JavaScript.
    if (link.toLowerCase().indexOf("javascript") != -1) {
      continue;
    }

    // Восстановить префикс абсолютного или относительного URL.
    if (link.indexOf("://") == -1) {
      // Обработать абсолютный URL.
      if (link.charAt(0) == '/') {
        link = "http://" + pageUrl.getHost() + link;
```

```
        // Обработать относительный URL.
        } else {
          String file = pageUrl.getFile();
          if (file.indexOf('/') == -1) {
            link = "http://" + pageUrl.getHost() + "/" + link;
          } else {
            String path =
              file.substring(0, file.lastIndexOf('/') + 1);
            link = "http://" + pageUrl.getHost() + path + link;
          }
        }
      }

      // Удалить привязки из ссылок.
      int index = link.indexOf('#');
      if (index != -1) {
        link = link.substring(0, index);
      }

      // Удалить начальные символы "www" из URL, если они есть.
      link = removeWwwFromUrl(link);

      // Проверить ссылки и отбросить все неправильные.
      URL verifiedLink = verifyUrl(link);
      if (verifiedLink == null) {
        continue;
      }

      /* Если указано, то использовать только ссылки
         для сайта с начальным URL. */
      if (limitHost &&
          !pageUrl.getHost().toLowerCase().equals(
            verifiedLink.getHost().toLowerCase()))
      {
        continue;
      }

      // Отбросить ссылки, если они уже просмотрены.
      if (crawledList.contains(link)) {
        continue;
      }

      // Добавить ссылку в список.
      linkList.add(link);
    }
    return (linkList);
  }

  /* Определить, есть ли совпадения для строки поиска
     на данной странице. */
  private boolean searchStringMatches(
    String pageContents, String searchString,
    boolean caseSensitive)
  {
    String searchContents = pageContents;

    /* Если учитывается регистр клавиатуры, то преобразовать
       содержимое в нижний регистр для сравнения. */
    if (!caseSensitive) {
      searchContents = pageContents.toLowerCase();
    }
```

```
    // Разделить строку поиска на отдельные термы.
    Pattern p = Pattern.compile("[\\s]+");
    String[] terms = p.split(searchString);

    // Проверки на совпадение каждый терм.
    for (int i = 0; i < terms.length; i++) {
      if (caseSensitive) {
        if (searchContents.indexOf(terms[i]) == -1) {
          return false;
        }
      } else {
        if (searchContents.indexOf(terms[i].toLowerCase()) == -1) {
          return false;
        }
      }
    }
    return true;
  }

  // Выполнить просмотр, производя поиск для заданной строки.
  public void crawl(
    String startUrl, int maxUrls, boolean limitHost,
    String searchString, boolean caseSensitive)
  {

    // Установить список поиска.
    HashSet crawledList = new HashSet();
    LinkedHashSet toCrawlList = new LinkedHashSet();

    // Добавить начальный URL в список поиска.
    toCrawlList.add(startUrl);

    /* Выполнить поиск, последовательно просматривая
       список поиска. */
    while (crawling && toCrawlList.size() > 0)
    {

      /* Проверить, не достигнуто ли максимально
         число разрешенных URL, если это значение задано. */
      if (maxUrls != -1) {
        if (crawledList.size() == maxUrls) {
          break;
        }
      }

      // Получить URL.
      String url = (String) toCrawlList.iterator().next();

      // Удалить URL из списка поиска.
      toCrawlList.remove(url);

      // Преобразовать  строку url в объект URL.
      URL verifiedUrl = verifyUrl(url);

      // Пропустить URL, если по списку робота к нему нет доступа.
      if (!isRobotAllowed(verifiedUrl)) {
        continue;
      }

      // Обновить панель Stats.
```

```
      updateStats(url, crawledList.size(), toCrawlList.size(),
        maxUrls);

      // Добавить страницу в список поиска.
      crawledList.add(url);

      // Загрузить страницу с заданным url.
      String pageContents = downloadPage(verifiedUrl);

      /* Если страница успешно загружена, извлечь из нее
         все ссылки и затем произвести поиск совпадающих строк. */
      if (pageContents != null && pageContents.length() > 0)
      {
        // Извлечь список допустимых ссылок из страницы.
        ArrayList links =
          retrieveLinks(verifiedUrl, pageContents, crawledList,
            limitHost);

        // Добавить ссылки в список поиска.
        toCrawlList.addAll(links);

        /* Проверить на наличие совпадающей строки, и если
           совпадение есть, то записать совпадение. */
        if (searchStringMatches(pageContents, searchString,
            caseSensitive))
        {
          addMatch(url);
        }
      }

      // Обновить панель Stats.
      updateStats(url, crawledList.size(), toCrawlList.size(),
        maxUrls);
    }
  }

  // Запустить поискового червя.
  public static void main(String[] args) {
    SearchCrawler crawler = new SearchCrawler();
    crawler.show();
  }
}
```

Переменные класса SearchCrawler

Класс `SearchCrawler` начинается с объявления нескольких переменных, большинство из которых содержат ссылки на элементы контроля графического интерфейса. Сначала объявляется массив строк `MAX_URLS`, значения которых будут отображаться в комбинированном окне **Max URLs to Crawl**. Затем выделяется кэш-память `disallowListCache` для списка запрещенных путей робота, поэтому из не надо будет специально извлекать из файла при обращении к отдельным URL. После этого объявляются элементы управления графического интерфейса для секций **Search**, **Stats** и **Matches**. После объявления элементов управления, объявляется флаг `crawling`, на основании которого делается вывод о работе червя. Наконец, объявляется переменная `logFileWriter`, которая используется при записи совпадений в журнал.

Конструктор SearchCrawler

При создании объекта типа `SearchCrawler` все элементы управления графического интерфейса инициализируются внутри конструктора. Конструктор содержит много строк кода, но большинство из них довольно прозрачны для понимания. Небольшие пояснения приведены ниже.

Сначала устанавливается заголовок окна с помощью вызова метода `setTitle()`. Затем вызывается метод `setSize()` для получения ширины и высоты окна в пикселях. После этого добавляется слушатель окна, для чего используется метод `addWindowListener()`, в который передается объект `WindowAdapter`, переопределяющий обработчик события для закрытия окна `windowClosing()`. В этом обработчике при закрытии окна приложения вызывается метод `actionExit()`. Затем в окно приложения добавляется меню с пунктом File.

Следующие несколько строк конструктора инициализируют и распределяют элементы управления графического интерфейса. Подобно остальным приложениям, описанным в этой книге, для распределения элементов управления используется класс `GridBagLayout` и связанный с ним класс `CridBagConstraints`. Сначала распределяется секция интерфейса Search, за которой следует секция Stats. В секцию Search включены все элементы управления для ввода критериев поиска и необходимые ограничения. Секция Stats содержит элементы управления отображением текущего состояния работы поискового червя, такие как количество просмотренных URL и количество тех, что еще осталось просмотреть.

В секциях Search и Stats необходимо отметить три важных момента. Во-первых, текстовое поле Matches Log File инициализируется со строкой, содержащей имя файла. В этой строке указывается файл `crawler.log` в том каталоге, из которого запускается приложение, как определено в переменной окружения Java `user.dir`. Во-вторых, для кнопки Search добавляется слушатель `ActionListener` для вызова метода `actionSearch()` при каждом щелчке на кнопке. В-третьих, шрифты для каждой метки, которые используются для отображения результатов, обновляются с помощью вызова метода `setFont()`. Метод `setFont()` используется для отключения поддержки шрифтов метки таким образом, чтобы они были оригинальными для графического интерфейса.

Следующей за секциями Search и Stats идет секция Matches, в которой отображается таблица соответствий, состоящая из URL, по которым были обнаружены совпадающие строки. Таблица совпадений создается с помощью нового подкласса `DefaultTableModel`, который передается в конструктор таблицы. Обычно широкие возможности класса `DefaultTableModel` используются для настройки модели данных, используемых `JTable`. Однако в нашем случае необходимо реализовать только метод `isCellEditable()`. Метод `isCellEditable()` сообщает таблице о том, что ячеек для редактирования нет, если возвращается `false`. В противном случае определяются строка и колонка.

После того как таблица совпадения создана, она добавляется на панель Matches. Наконец, панели Search и Matches добавляются в интерфейс.

Метод actionSearch()

Метод `actionSearch()` вызывается при каждом щелчке на кнопке Search (или Stop). Для вызова метода используются следующие строки.

```
// Если произведен щелчок на кнопке Stop, флаг поиска сбрасывается.
if (crawling) {
  crawling = false;
  return;
}
```

Так как кнопка графического интерфейса Search используется и как кнопка Stop, необходимо знать, в каком режиме на кнопке произведен щелчок. Когда червь находится в режиме поиска, то флаг поиска установлен. Таким образом, если флаг поиска `crawling` установлен при вызове метода `actionsearch()`, то щелчок произведен на кнопке Stop. В соответствии с этим сценарием, флаг `crawling` сбрасывается (т.е. для него устанавливается значение `false`) и в методе `actionSearch()` возврат происходит таким образом, что часть строк кода метода не выполняется. Затем переменная `errorList` типа `ArrayList` инициализируется.

```
ArrayList errorList = new ArrayList();
```

Переменная `errorList` используется для сохранения всех сообщений об ошибках, сгенерированных в следующих строках кода, в которых проверяется соответствие всем данным, введенным в поля панели Search.

Само собой разумеется, что поисковый червь не может работать без указания URL, который определяет адрес, с которого начинается поиск. В следующих строках проверяется, что начальный URL введен и что его написание соответствует определенным правилам.

```
// Убедиться, что начальный URL введен.
String startUrl = startTextField.getText().trim();
if (startUrl.length() < 1) {
  errorList.add("Missing Start URL.");
}
// Проверить начальный URL.
else if (verifyUrl(startUrl) == null) {
  errorList.add("Invalid Start URL.");
}
```

Если при любой проверке возникнет ошибка, то сообщение об ошибке добавляется в список ошибок. Затем проверяется максимальное количество допустимых URL (пункт Max URLs to Crawl).

```
// Убедиться, что значение для максимального
// количества URL пусто или является числом.
int maxUrls = -1;
String max = ((String) maxComboBox.getSelectedItem()).trim();
if (max.length() > 0) {
  try {
    maxUrls = Integer.parseInt(max);
  }
   catch (NumberFormatException e) {
  }
  if (maxUrls < 1) {
    errorList.add("Invalid Max URLs value.");
  }
}
```

Определение максимального количества URL выполнено немного сложнее, чем другие проверки в этом методе, т.к. значение для поля Max URLs to Crawl может содержать положительное число, что говорит о максимально допустимом количестве URL, или может быть пустым, что говорит о том, что нет никаких ограничений на максимальное количество URL. Изначально по умолчанию для `maxUrls` устанавливается значение `-1`, свидетельствующее о том, что максимум не задается. Если пользователь производит ввод в поле Max URLs to Crawl, то введенные данные должны быть допустимым цифровым значением, что проверяется вызовом метода `Integer.parselnt().Integer.parselnt()`, в котором производится преобразование строкового значения в целочисленное. Если такое преобразование не может быть выполнено, то генерируется исключительная ситуация `NumberFormatException` и значение для `maxUrls` не устанавливается. Затем производится сравнение значения `maxUrls` с `1`. Если оно меньше единицы, то сообщение об ошибке добавляется в список ошибок.

Затем проверяются журнал совпадений (Matches Log File) и поле для строки поиска (Search String fields).

```
// Убедиться, что журнал есть совпадений.
String logFile = logTextField.getText().trim();
if (logFile.length)) < 1) {
  errorList.add("Missing Matches Log File.");
  }
// Убедиться, что строка поиска введена.
String searchString = searchTextField.getText().trim();
if (searchString.length() < 1) {
  errorList.add("Missing Search String.");
}
```

Если в эти поля ничего не ввести, то в список ошибок будут добавлены сообщения об ошибках.

В следующем поле производится проверка на наличие записей сообщений об ошибках во время проверки. Если такие записи есть, то все ошибки объединяются в одну строку и это сообщение отображается на экране с помощью метода `showError()`.

```
// Если есть ошибки, отобразить их.
if (errorList.size() > 0) {
  StringBuffer message = new StringBuffer();
    // Объединить ошибки в одно сообщение.
    for (int i = 0; i < errorList.size(); i++) {
      message.append(errorList.get(i));
      if (i + 1 < errorList.size()) {
        message.append("\n");
    }
  }
  showError(message.toString());
  return;
}
```

В целях эффективности объект `StringBuffer` (ссылка на сообщение) используется для сохранения объединенного сообщения. Производится последовательный просмотр списка ошибок, и каждое сообщение об ошибке добавляется в объединенное сообщение. Обратите внимание, что при добавлении каждого сообщения из списка сообщений об ошибках производится проверка, не является ли это сообщение последним. Если сообщение об ошибке не последнее, то добавляется символ новой

строки (\n) для того, чтобы каждое сообщение отображалось в диалоговом окне на отдельной строке. Для отображения используется метод showError().

Наконец, после успешной проверки всех полей вызывается заключительный метод actionSearch() для удаления символов "www" из начала строки с адресом, если эти символы имеются, а затем производится вызов метода search().

```
// Удалить символы "www" из начала адреса, если они есть.
startUrl = removeWwwFromUrl(startUrl);

// Запустить поискового червя.
search(logFile, startUrl, maxUrls, searchString);
```

Метод search()

Метод search() используется для начального запуска процесса поиска. Так как этот процесс может продолжаться значительное количество времени, создается новый поток, для того чтобы код для поискового червя мог работать независимо. При этом используются события для создания потоков из библиотеки Swing, благодаря чему вносятся изменения в интерфейс во время работы поискового червя.

Метод search() начинается с следующих строк кода.

```
// Запустить поискового червя в новом потоке.
Thread thread = new Thread(new Runnable() {
  public void run() {
```

Для запуска кода поиска в новом потоке, создается новый объект Thread с экземпляром Runnable, передаваемым в конструктор в качестве аргумента. Вместо создания отдельного класса для реализации интерфейса Runnable, получается линейный код.

Перед запуском поиска элементы управления графического интерфейса обновляются для отображения того факта, что началась работа поискового червя.

```
// Отобразить песочные часы на время работы поискового червя.
setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
// Блокировать элементы управления поиском.
startTextField.setEnabled(false);
maxComboBox.setEnabled(false);
limitCheckBox.setEnabled(false);
logTextField.setEnabled(false);
searchTextField.setEnabled(false);
caseCheckBox.setEnabled(false) ;
// Переключить кнопку поиска в режим "Stop."
searchButton.setText("Stop");
```

Сначала курсор приложения устанавливается в режим ожидания, т.е. ему присваивается значение WAIT_CURSOR с целью сигнализировать о том, что приложение занято выполнением задачи. В большинстве операционных систем при установке значения WAIT_CURSOR курсор отображается в виде песочных часов. После установки курсора каждый из элементов управления поиском становится неактивным, для чего вызывается метод setEnabled() со сброшенным флагом (значение false). Затем изменяется режим кнопки начала поиска и она переходит в режим остановки поиска (Stop). При этом на ней отображается надпись "Stop". Это изменение происходит потому, что кнопка поиска используется как при запуске поиска, так и при его остановке.

После того как элементы управления поиском были переведены в неактивное состояние, переустанавливается секция Stats, как показано в листинге.

```
// Переустановить секцию Stats.
table.setModel(new DefaultTableModel(new Object[][]{},
new String[]{"URL"}) {
  public boolean isCellEditable(int row, int column)
  {
    return false;
  }
});
updateStats(startUrl, 0, 0, maxUrls);
```

Во-первых, переустанавливаются данные таблицы совпадений с помощью вызова метода setModel() в исходное состояние, как экземпляр типа DefaultTableModel. Во-вторых, вызывается метод updateStats() для обновления индикатора выполнения и меток состояния.

Затем открывается файл журнала совпадений и сбрасывается флаг для переменной crawling.

```
// Открыть журнал совпадений.
try {
  logFileWriter = new PrintWriter(new FileWriter(logFile));
}
catch (Exception e) {
  showError("Unable to open matches log file.");
  return;
}
// Сбросить флаг для переменной поиска.
crawling = true;
```

Файл журнала соответствий открывается путем создания нового экземпляра класса PrintWriter для записи в файл. Если невозможно открыть файл, отображается диалоговое окно ошибки с помощью вызова метода showError(). Устанавливается флаг переменной поиска crawling, что свидетельствует о вызове метода actionSearch() и начале работы поискового червя.

Благодаря следующему коду начинается реальная работа поискового червя с помощью вызова метода crawl().

```
// Начать работу по поиску совпадений.
crawl(startUrl, maxUrls, limitCheckBox.isSelected(), searchString,
  caseCheckBox.isSelected());
```

После завершения работы по поиску совпадений, сбрасывается флаг переменной crawling и закрывается файл журнала совпадений.

```
// Сбросить флаг для переменной crawling.
crawling = false;
// Закрыть файл журнала совпадений.
try {
  logFileWriter.close() ;
}
catch (Exception e) {
  showError("Unable to close matches log file.");
}
```

Значение для переменной crawling устанавливается в состояние false, что свидетельствует о завершении работы поискового червя. Затем закрывается файл журнала совпадений, т.к. уже ничего не будет записываться. Как и при открытии

файла, если в момент закрытия возникнет исключительная ситуация, то отобразится диалоговое окно ошибки после вызова метода `showError()`.

Поскольку работа поискового червя завершена, элементы управления секции Search изменяются в соответствии со следующим кодом.

```
// Отметить завершение поиска.
crawlingLabel2.setText("Done");
// Разблокировать элементы управления секции Search.
startTextField.setEnabled(true);
maxComboBox.setEnabled(true);
limitCheckBox.setEnabled(true);
logTextField. setEnabled (true);
searchTextField.setEnabled(true);
caseCheckBox.setEnabled(true);
// Переключить кнопку поиска в режим "Search."
searchButton.setText("Search");
// Установить курсор по умолчанию.
setCursor(Cursor.getDefaultCursor());
```

Во-первых, поле Crawling обновляется для того, чтобы отобразить слово "Done". Во-вторых, каждый из элементов контроля секции Search повторно устанавливается. В-третьих, кнопка Stop возвращается к предыдущему состоянию, отображая слово "Search".

Наконец, курсор возвращается в то состояние, в котором он находился до начала работы поискового червя, т.е. в состояние по умолчанию для данного приложения.

Если при поиске не найдено никаких совпадений, то отображается диалоговое окно для сообщения об этом факте, для чего используется следующий код.

```
// Отобразить сообщение, если совпадений не найдено.
if (table.getRowCount() ==0) {
  JOptionPane.showMessageDialog(SearchCrawler.this,
    "Your Search String was not found. Please try another.",
    "Search String Not Found",
    JOptionPane.WARNING_MESSAGE);
}
```

Метод `search()` завершается следующими строками кода.

```
  }
});
thread.start();
```

После того как будет описана реализация метода `run()` для интерфейса `Runnable`, поток для работы поискового червя запускается с помощью вызова метода `thread.start()`. При выполнении потока будет вызываться метод `run()` для интерфейса `Runnable`.

Метод showError()

Метод `showError()`, листинг которого приведен ниже, отображает на экране диалоговое окно ошибки с заданным сообщением. Этот метод вызывается, если требуемая для поиски опция пропущена или если возникают проблемы с открытием, записью или закрытием файла журнала совпадений.

```
// Отобразить диалоговое окно с сообщением об ошибке.
private void showError(String message) {
  JOptionPane.showMessageDialog(this, message, "Error",
    JOptionPane.ERROR_MESSAGE);
}
```

Метод updateStats()

Метод `updateStats()`, листинг которого приведен ниже, обновляет значения, отображаемые в секции **Stats** графического интерфейса.

```
// Обновление статистики поиска.
private void updateStats(
  String crawling, int crawled, int toCrawl, int maxUrls)
{
  crawlingLabel2.setText(crawling);
  crawledLabel2.setText("" + crawled);
  toCrawlLabel2.setText("" + toCrawl);
  // Обновить индикатор выполнения.
  if (maxUrls == -1) {
    progressBar.setMaximum(crawled + toCrawl);
  }
  else {
    progressBar.setMaximum(maxUrls);
  }
  progressBar.setValue(crawled);
  matchesLabel2.setText("" + table.getRowCount());
}
```

Во-первых, обновляются результаты поиска для того, чтобы отобразить текущий URL, с которым производится работа, количество просмотренных URL и количество тех URL, что еще осталось просмотреть. Обратите внимание, что количество URL для поля **URLs to Crawl** может ввести в заблуждение. Здесь отображается количество ссылок, которые были собраны и размещены в очереди, а не разность между указанным количеством допустимых URL и количеством URL, которые уже просмотрены. Также отметим, что когда вызывается метод `setText()` с параметрами `crawled` и `toCrawl`, то вместе с ними передаются пустая строка (""), знак плюс (+) и целочисленное значение. Это делается для того, чтобы компилятор Java мог преобразовать целочисленное значение в объект `String`, который требуется для метода `setText()`.

Затем обновляется индикатор выполнения для отображения текущего состояния количества просмотренных URL относительно общего количества URL. Если для поля **Max URLs to Crawl** (максимальное количество URL) не будет введено значения, то переменная `maxUrls` примет значение `-1`. В этом случае максимальным значением для индикатора выполнения будет количество уже просмотренных URL плюс количество URL, находящихся в очереди для просмотра. Если для поля **Max URLs to Crawl** значение указано, то оно и будет являться максимальным значением для индикатора выполнения. После определения максимального значения для индикатора выполнения, вычисляется текущее значение. Класс `JProgressBar` использует максимальное и текущее значения для подсчета процентного соотношения, отображаемого в индикаторе выполнения.

Наконец, обновляется значение для метки **Search Matches**, отображающее количество URL, которые содержат совпадающие строки.

Метод addMatch()

Метод `addMatch()` вызывается в метод `crawl()` каждый раз, когда найдено совпадение для заданной строки. Метод `addMatch()`, листинг которого представлен ниже, добавляет необходимый URL как в таблицу совпадений, так и в файл журнала совпадений.

```
// Произвести запись в таблицу совпадений и файл журнала.
private void addMatch(String url) {
  // Добавить URL в таблицу совпадений.
  DefaultTableModel model =
    (DefaultTableModel) table.getModel();
  model.addRow(new Object[]{url});
  // Добавить URL в файл журнала совпадений.
  try {
    logFileWriter.println(url);
  }
  catch (Exception e) {
    showError("Unable to log match.");
  }
}
```

В этом методе сначала добавляется URL в таблицу соответствий с помощью извлечения модели данных таблицы и вызова метода `addRow()`. Обратите внимание, что метод `addRow()` принимает массив типов `Object` как входной массив. Для того чтобы удовлетворить этим требованиям, строка `url` помещается в оболочку типа `Object`. После добавления URL в таблицу совпадений, URL записывается в файл журнала совпадений с помощью вызова метода `logFileWriter.printIn()`. Этот вызов заключен в блок `try...catch`, и если возникнет исключительная ситуация, то вызывается метод `showError()` для предупреждения пользователя о том, что в процессе записи в файл журнала совпадений возникла ошибка.

Метод verifyUrl()

Метод `verifyUrl()`, листинг которого приведен ниже, используется во многих фрагментах поискового червя для проверки формата URL. Кроме того, этот метод используется для преобразования строкового представления URL в объект URL.

```
// Проверка формата URL.
private URL verifyUrl(String url) {
  // Разрешены только HTTP URL.
  if (!url.toLowerCasef).startsWith("http://"))
    return null;
  // Проверить формат URL.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  }
  catch (Exception e) {
    return null;
  }
  return verifiedUrl;
}
```

В этом методе сначала производится проверка на предмет того, что данный URL указывает только на страницы формата HTTP, которые может обрабатывать поисковый червь. Затем строка URL преобразовывается в объект `URL`. Если написание URL является недопустимым, конструктор для класса `URL` сгенерирует исключительную ситуацию и метод `verifyUrl()` возвратит значение `null`. Это значение будет свидетельствовать о том, что строковое представление URL является недопустимым или что его нельзя проконтролировать.

Метод RobotAllowed()

Метод `isRobotAllowed()` заполняет протокол робота. Для того чтобы полностью проанализировать этот метод, будем последовательно рассматривать строку за строкой.

Метод `isRobotAllowed()` начинается следующими строками кода.

```
String host = urlToCheck.getHost().toLowerCase();
// Извлекает список ограничений из кэш-памяти.
ArrayList disallowList = (ArrayList) disallowListCache.get(host);
// Если в кэш-памяти нет списка, загрузить его и
// поместить в кэш-память.
if (disallowList == null) {
  disallowList = new ArrayList();
```

Для того, чтобы эффективно производить проверку относительно разрешения доступа к определенному URL, поисковый червь помещает в кэш-память список ограничений для данного сайта сразу, как только он считывается. Такой подход значительно повышает производительность поискового червя, поскольку при этом исключается циклическая загрузка списка ограничений для проверки каждого отдельного URL. Вместо загрузки списка, он просто просматривается в кэш-памяти.

В кэш-память со списком ограничений помещается только та часть страниц, которые расположены на данном URL, поэтому метод `isRobotAllowed()` начинается с извлечения списка, для чего используется метод `getHost()` объекта `urlToCheck`. Обратите внимание, что метод `toLowerCase()` подсоединен к методу `getHost()`. Получение всех символов в нижнем регистре гарантирует, что продублированные страницы, входящие в сайт, не будут помещены в список. Также заметьте, что список ограничения для данного сайта не чувствителен к регистру клавиатуры, тогда как строки в кэш-памяти чувствительны к регистру. После извлечения ограничений `urlToCheck`, предпринимается попытка извлечь список ограничений из кэш-памяти. Если в кэш-памяти еще нет списка, возвращается значение `null`, что говорит о необходимости загрузки списка ограничений из сайта в кэш-память. Процесс загрузки списка ограничений из сайта начинается с создания нового объекта `ArrayList`.

Затем содержимое списка ограничений заполняется, для чего используются следующие строки программы.

```
try {
  URL robotsFileUrl = new URL("http://" + host + "/robots.txt");
  // Подключить файл робота данного URL для чтения.
  BufferedReader reader =
    new BufferedReader(new InputStreamReader(robotsFileUrl.openStream()));
```

Как уже упоминалось, владельцы Web-сайта, которые хотят защитить сайт полностью или частично от прохождения его поисковым червем, должны включить файл `robots.txt` в корневой каталог Web-сайта. В объекте `urlToCheck` используется структура адреса с полем `robotsFileUrl`, значение которого указывает на файл `robots.txt`. В конструктор класса `BufferedReader` передается экземпляр объекта `InputStreamReader`, в чей конструктор передается объект `InputStream`, возвращаемый после вызова метода `robotsFileUrl.openStream()`.

В следующей последовательности строк используется цикл `while` для чтения содержимого файла `robots.txt`.

```
// Чтение файла робота, создание списка ограничений для путей.
String line;
while ((line = reader.readLine()) != null) {
  if (line.indexOf("Disallow:") == 0) {
    String disallowPath = line.substring("Disallow:".length());
```

В цикле считывается содержимое файла, строка за строкой, пока метод `reader.readLine()` не возвратит значение `null`, свидетельствующее о том, что все строки считаны. Каждая считанная строка проверяется на наличие утверждения `Disallow` с помощью метода `indexOf()`, определенного в классе `String`. Если строка действительно содержит утверждение `Disallow`, то путь для закрытой страницы извлекается из строки с помощью выбора подстроки, начиная с того места, где заканчивается строка "Disallow:".

Как уже обсуждалось, в файл `robots.txt` могут вставляться комментарии, для чего используется символ решетки (#), за которым следует комментарий. Если символ решетки встречается в утверждении `Disallow`, то комментарий удаляется с помощью следующих строк кода.

```
// Проверка на наличие комментариев и их удаление.
    int commentIndex = disallowPath.indexOf("#");
    if (commentIndex !=-1){
      disallowPath = disallowPath.substring(0, commentIndex);
    }
    // Удалить предыдущие или последующие пробелы из пути.
    disallowPath = disallowPath.trim();
    // Добавить запрещающий путь в список.
    disallowList.add(disallowPath);
  }
}
```

Во-первых, производится поиск по запрещающему пути с целью определить символы решетки. Если символ решетки найден, строка с путем разбивается на две подстроки и вторая подстрока с комментарием до конца строки удаляется. После проверки и возможного удаления комментария из запрещающего пути, вызывается метод `disallowPath.trim()` для удаления всех предшествующих и последующих символов пробела. Подобно комментариям, дополнительные символы пробела будут мешать при сравнении, поэтому они удаляются. Наконец, запрещающий путь добавляется в список ограничений.

После создания списка ограничений, он добавляется в кэш-память, как показано в строках кода ниже.

```
// Добавить новый список ограничений в кэш-память.
    disallowListCache.put(host, disallowList);
  }
  catch (Exception e) {
    /* Предположим, робот существует после исключительной
       ситуации, когда нет файла с протоколом робота. */
    return true;
  }
}
```

Запрещающий путь добавляется в кэш-память таким образом, чтобы последовательные запросы к списку ограничений могли очень быстро обрабатываться без повторной загрузки списка.

Если при открытии входного потока для файла робота или чтении содержимого файла происходит ошибка, то генерируется исключительная ситуация. Если исклю-

чительная ситуация генерируется в том случае, когда не существует файла робота, то предполагается, что робот может использоваться в дальнейшем. Обычно проверка ошибок по такому сценарию должна быть более подробной, однако для упрощения и более компактного изложения было принято решение считать, что работа робота продолжается.

Затем помещены следующие строки кода для проверки относительно возможности производить поиск по указанным путям.

```
/* Проверка списка ограничений для принятия решения о разрешении
   или запрещении производить поиск по данному URL. */
String file = urlToCheck.getFile();
for (int i = 0; i < disallowList.size(); i++) {
  String disallow = (String) disallowList.get(i);
  if (file.startsWith(disallow)) {
    return false;
  }
}
return true;
```

При каждой итерации производится проверка относительно того, что путь, указанный в соответствующем поле объекта `urlToCheck`, совпадает с одним из путей из списка ограничений. Если путь из объекта `urlToCheck` совпадает с одним из путей из списка ограничений, то возвращается значение `false`, что свидетельствует о запрещении поиска по данному URL. Но если при прохождении всего списка ограничение совпадений не найдено, то возвращается значение `true`, что подтверждает разрешение на поиск.

Метод downloadPage()

Предназначение метода `downloadPage()`, листинг которого приведен ниже, вполне понятно из его названия — он загружает Web-страницы с данного URL и возвращает содержимое страницы как одну большую строку.

```
// Загрузить страницу с данного URL.
private String downloadPage(URL pageUrl) {
  try {
    // Открыть соединение с URL для чтения.
    BufferedReader reader =
      new BufferedReader(new InputStreamReader(pageUrl.openStream()));
    // Считать страницу в буфер.
    String line;
    StringBuffer pageBuffer = new StringBuffer();
    while ((line = reader.readLine()) != null) {
      pageBuffer.append(line);
    }
    return pageBuffer.toString();
  }
  catch (Exception e) {}
  return null;
}
```

Загрузка Web-страниц из Internet в Java производится очень просто, как видно из вышеприведенного метода. Сначала создается объект `BufferedReader` для чтения содержимого страницы с указанного URL. В конструктор класса `BufferedReader` передается экземпляр класса `InputStreamReader`, в конструктор которого в свою очередь передается объект `InputStream`, возвращаемый методом `pageUrl.openStream()`.

Затем используется цикл `while` для чтения содержимого страницы, строка за строкой, пока метод `reader.readLine()` не возвратит значение `null`, которое говорит о том, что считаны все строки. Каждая строка, считанная в цикле `while`, добавляется в объект `pageBuffer` типа `StringBuffer`.

После загрузки всех страниц содержимое возвращается как тип `String` с помощью вызова метода `pageBuffer.toString()`.

Если при открытии входного потока для данной страницы URL или во время чтения содержимого Web-страницы произошла ошибка, то генерируется исключительная ситуация. Эта исключительная ситуация будет перехвачена пустым блоком `caught`. Блок `catch` умышленно оставлен незаполненным, чтобы выполнение продолжалось со строки "return null;". Возвращаемое значение `null` из этого метода говорит вызывающему методу о том, что произошла ошибка.

Метод removeWwwFromUrl()

Метод `removeWwwFromUrl()` является простой обслуживающей программой, которая используется для удаления последовательности символов "www" из адреса. Например, если взять URL

```
http://www.osborne.com
```

то после вызова метода `removeWwwFromUrl()` для этого URL получим строку "http://osborne.com".

Поскольку многие Web-сайты используют URL, которые могут содержать или не содержать символы "www", то поисковый червь использует технологию "наименьшего общего знаменателя" для обработки URL. Это позволяет исключить дублирование URL, так как многие Web-сайты могут использовать различное написание для одних и тех же страниц.

Код метода `removeWwwFromUrl()` показан ниже.

```
// Удаление предшествующих "www" из URL, если они есть.
private String removeWwwFromUrl(String url) {
  int index = url.indexOf("://www.");
  if (index !=-1){
    return url.substring(0, index + 3} +
      url.substring(index + 7);
  }
  return (url);
}
```

Метод `removeWwwFromUrl()` начинается с поиска индекса для символов "://www." внутри строки, передаваемой как аргумент для `url`. Символы "://" начинают строку, передаваемую в метод `indexOf()` для того, чтобы акцентировать внимание на том обстоятельстве, что символы "www" должны быть найдены в начале URL, где они помещаются в соответствии с протоколом (например: http://www.osborne.com). Таким образом, в URL, которые включают последовательность "www" как часть адреса, эта последовательность не будет удалена. Если URL содержит строку символов "`://www.`", то символы перед последовательностью "www" и после нее будут объединены и возвращены в составе адреса. В противном случае строка адреса будет возвращена без изменений.

Метод retrieveLinks()

Метод `retrieveLinks()` производит синтаксический анализ содержимого Web-страницы и извлекает все найденные ссылки. Все ссылки для Web-страницы сохраняются в объекте `String`. Но следует отметить, что синтаксический анализ и поиск специфических последовательностей символов будут выглядеть очень неряшливо, если использовать методы класса `String`. К счастью, начиная с версии Java 2, v1.4, язык Java включает в библиотеку API стандартные регулярные выражения, которые значительно облегчают работу по синтаксическому анализу строк.

Эти выражения содержатся в пакете `java.util.regex`. Описание всех регулярных выражений займет слишком много времени и полное обсуждение этих вопросов вынесено за рамки этой книги. Однако поскольку анализ регулярных выражений является ключевым в поисковом черве, то ниже приведено краткое описание регулярных выражений.

Обзор обработки регулярных выражений

Остановимся на терминах, используемых в данном случае. *Регулярное выражение* (regular expression) — это последовательность определенных символов, с помощью которой описывается некоторая символьная последовательность. В самом общем случае это называется *шаблоном* (pattern), который может использоваться для нахождения совпадений в различных символьных последовательностях. В регулярных выражениях могут использоваться групповые символы, наборы символов и различные квалификаторы. Таким образом, можно написать регулярное выражение, которое будет определять общую форму, совпадающую с различными специфическими символьными последовательностями. В Java есть два класса, которые поддерживают использование регулярных выражений: `Pattern` и `Matcher`. Для задания регулярного выражения используется класс `Pattern`. Для поиска соответствия шаблона заданной символьной последовательности используется класс `Matcher`.

В классе `Pattern` нет конструктора. Поэтому шаблон задается с помощью вызова метода `compile()`, а не конструктора. При этом используется следующий формат записи.

```
static Pattern compile(String pattern, int options)
```

Здесь параметр *`pattern`* представляет регулярное выражение, которое необходимо использовать, а вместо параметра *`options`* подставляется одна или несколько опций, которые используются при поиске совпадений. Опцией, используемой поисковым червем, будет `Pattern.CASEJNSENSITIVE`, которая приводит к игнорированию регистра клавиатуры.

Метод `compile()` преобразует строку шаблона в шаблон, который может использоваться как исходный для поиска совпадений в классе `Matcher`. Он возвращает объект `Pattern`, который содержит исходный шаблон.

После создания объекта `Pattern`, он используется при создании объекта `Matcher`. Это делается с помощью вызова метода `matcher()`, для которого задан шаблон, как показано в следующем примере.

```
Matcher matcher(CharSequence str)
```

Здесь параметр *str* представляет строку, с которой будет сравниваться заданный шаблон. Такая строка называется *входной последовательностью* (input sequence). Интерфейс CharSequence добавлен в Java 2, v1.4 и определяет набор символов только для чтения. Этот интерфейс реализован в классе String, как и в некоторых других классах. Таким образом, вполне реально передавать строку в метод matcher().

Можно использовать методы, объявленные в классе Matcher, для выполнения различных операций по сравнению шаблонов. Это могут быть методы find() и group(), используемые в методе retrieveLinks(). Метод find() находит последовательность символов, совпадающую с шаблоном и входящую во входную последовательность. Эта версия используется поисковым червем, как показано ниже.

```
boolean find()
```

Если метод возвращает значение true, то найдено совпадение, в противном случае возвращается false. Это метод может вызываться повторно, позволяя находить все части строки, совпадающие с шаблоном. Каждый повторный вызов метода find() производится при успешном выполнении предыдущего вызова.

Можно получить строку, включающую совпадающую последовательность с помощью вызова метода group(). В поисковом черве используется следующая форма для вызова метода.

```
String group(int which)
```

Здесь параметр *which* определяет последовательность (группу символов), при этом первая группа начинается с 1 и возвращается соответствующая строка.

Синтаксис регулярных выражений

Синтаксис и правила для регулярных выражений подобны тем, что используются в языке Perl 5. Хотя эти правила довольно просты, но они очень многочисленны и полное их обсуждение выходит за рамки этой книги. Однако некоторые наиболее общие конструкции описаны ниже.

В самом общем случае регулярные выражения состоят из *стандартных символов*, *классов символов* (последовательностей символов), *групповых символов* и *квалификаторов*. Стандартные символы рассматриваются один к одному. Таким образом, если шаблон состоит из последовательности символов "xy", то такому шаблону может соответствовать только символьная последовательность "xy". Символы, используемые для перехода на новую строку, или символы табуляции, должны использоваться с символом обратной черты (\). Например, для перехода на новую строку необходимо записать "\n". При описании регулярных выражений стандартные символы также называются *литералами*.

Классом символов называется набор символов. Класс символов определяется с помощью размещения символов класса между квадратными скобками. Например, класс символов [wxyz] используется для поиска совпадений с символами w, x, y или z. Для указания инвертированного набора символов, набору символов должен предшествовать знак циркумфлекс (^). Например, класс символов [^wxyz] может определять любой символ кроме w, x, y и z. Можно указать диапазон символов с помощью дефиса. Например, для задания класса символов в диапазоне цифр от 1 до 9, можно написать [1-9].

Групповым символом является точка (.), которая соответствует любому стандартному символу. Таким образом шаблон, который состоит из одной точки (“.”), будет совпадать с входными последовательностями “A”, “a”, “x” и т.д.

Квалификатор указывает количество повторений, которое должно использоваться при поиске совпадений. Правила использования квалификатора показаны ниже.

+	Одно или больше совпадений
*	Нуль или больше совпадений
?	Нуль или одно совпадение

Например, шаблон “x+” будет совпадать со строками “x”, ”xx”, ”xxx”.

Подробно о методе retrieveLinks()

Метод `retrieveLinks()` использует регулярные выражения API для обнаружения ссылок на странице. Он начинается следующими строками:

```
// Компиляция шаблона для поиска ссылок.
Pattern p =
  Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
    Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher(pageContents);
```

Регулярное выражение используется для получения ссылок, поиск которых производится за несколько шагов, как показано в таблице ниже.

Последовательность символов	*Объяснение*
`<a`	Поиск последовательности “<a”
`\\s+`	Поиск одного или более символов пробела
`Href`	Поиск последовательности “href”
`\\s*`	Поиск ни одного или нескольких символов пробела
`=`	Поиск символа “=”
`\\s*`	Поиск ни одного или нескольких символов пробела
`\"?`	Поиск ни одного или одного символа кавычек
`(.*?)`	Поиск ни одного или нескольких любых символов, пока не встретится следующая часть шаблона, и размещение результата в группе
`[\"|>]`	Поиск символа кавычек или символов, больших символов (">")

Обратите внимание, что аргумент `Pattern.CASE_INSENSITIVE` передается в компилятор шаблона. Как уже упоминалось, это говорит о том, что при сравнении с шаблоном должен игнорироваться регистр клавиатуры. Затем создается список, содержащий ссылки и начинается поиск ссылок, как показано в листинге.

```
// Создать список ссылок.
ArrayList linkList = new ArrayList();
while (m.find()) {
  String link = m.group(1).trim();
```

Для поиска ссылок используется цикл `while`. Метод `find()` из класса `Matcher` возвращает `true` до тех пор, пока больше не будут определяться ссылки. Каждая найденная ссылка извлекается с помощью вызова метода `group()`, описанного в классе `Matcher`. Обратите внимание, что в метод `group()` передается единица как аргумент. Это необходимо для того, чтобы возвращалась первая группа из совпадающих последовательностей. Также заметьте, что метод `trim()` вызывается для возвращения значения из метода `group()`. Таким образом удаляются все ненужные пробелы перед значением и после него.

Многие ссылки, найденные на Web-страницах, не подходят для дальнейшей обработки. В следующем коде производится фильтрация ссылок, которые не подходят для поискового червя.

```
// Отбросить пустые ссылки.
if (link.length() < 1) {
  continue;
}
// Отбросить ссылки, которые связаны с данной страницей.
if (link.charAt(0) == '#') {
  continue;
}
// Отбросить почтовые ссылки.
if (link.indexOf("mailto:") != -1) {
  continue; }
// Отбросить ссылки на JavaScript.
if (link.toLowerCase().indexOf("javascript") != -1) {
  continue;
 }
```

Во-первых, полностью отбрасываются пустые ссылки, чтобы не тратить на них время. Во-вторых, отбрасываются ссылки, которые связаны с обрабатываемой страницей, для чего проверяется первый символ ссылки, который должен быть решеткой (#). Привязанные к странице ссылки позволяют перейти к определенному месту страницы. Для примера рассмотрим следующий URL.

```
http://osborne.com/#contact
```

Этот URL представляет ссылку, связанную с данной страницей, т.е. производится ссылка на раздел "`contact`", расположенный на странице `http://osborne.com`. Таким образом, можно ссылаться на отдельные фрагменты страницы, используя запись "`#contact`". Но так как такие ссылки не являются ссылками на новую страницу, то они отбрасываются.

Затем отбрасываются ссылки для почтовых отправлений. Ссылки для почтовых отправлений используются на Web-страницах для удобной отправки сообщений через Internet. Например, ссылка.

```
mailto:books@osborne.com
```

является почтовой ссылкой. Так как почтовые ссылки не указывают на Web-страницы, они не могут обрабатываться поисковым червем и поэтому должны быть отброшены. Наконец, ссылки на сценарии JavaScript также должны быть отброшены. Сценарии JavaScript описаны с помощью языка сценариев и они вставляются в Web-страницы для получения возможности интерактивного взаимодействия с программой. К таким сценариям можно обратиться через ссылки. Подобно почтовым ссылкам, ссылки на сценарии не могут обрабатываться поисковым червем.

Как только что было показано, ссылки на Web-странице могут иметь различные форматы, среди которых есть почтовые ссылки или ссылки на сценарии JavaScript. Дополнительно к этому, обычные ссылки на Web-странице также могут иметь несколько различных форматов. Ниже представлены три формата, которые могут использоваться с обычными ссылками.

- `http://osborne.com/books/ArtofJava`
- `/books/ArtofJava`
- `books/ArtofJava`

Первая из трех ссылок, представленных выше, является полностью заданным URL. Во втором случае приведен пример сокращенной версии URL, где пропущена часть с указанием узла Internet (host). Обратите внимание на обратную черту в начале URL (/), которая говорит о том, что данный URL является "абсолютным" (absolute). Абсолютными являются такие URL, которые начинаются с корневого каталога Web-сайта. В третьем примере также приведена сокращенная версия URL, в которой пропущена часть с указанием узла Internet. Но в данном случае отсутствует обратная черта в начале URL. Так как обратная черта отсутствует, то данный URL рассматривается как относительный (relative). В данном случае относительными URL считаются такие URL, которые дополняют URL, в котором найдена ссылка.

В ниже приведенных строках производится преобразование абсолютных и относительных URL в полностью заданные URL.

```
// При необходимости добавить префикс в абсолютные и относительные URL.
if (link.indexOf("://") == -1) {
  // Обработать абсолютные URL.
  if (link.charAt(0) == '/') {
    link = "http://" + pageUrl.getHost() + link;
    // Обработать относительные URL.
  }
  else {
    String file = pageUrl.getFile();
    if (file.indexOf('/') == -1) {
      link = "http://" + pageUrl.getHost() + "/" + link;
    }
    else {
      String path =
        file.substring(0, file.lastIndexOf('/') + 1);
      link = "http://" + pageUrl.getHost() + path + link;
    }
  }
}
```

Сначала при проверке ссылки определяется, является ли она полностью заданной, для чего в ссылке выполняется поиск строки "`://`". Если эти три символа присутствуют, то считается, что URL полностью задан. Если эти символы не найдены, то ссылка должна быть преобразована в полностью заданную. Как уже говорилось, ссылки, начинающиеся с обратной черты (/), являются абсолютными, поэтому к ним добавляется строка "`http://`" и адрес узла Internet. Относительные ссылки преобразовываются аналогичным образом.

Для относительных ссылок берется имя файла текущей страницы и проверяется на наличие обратной черты. Обратная черта в имени файла говорит о том, что файл содержится в иерархии каталогов. Например, имя файла может выглядеть как

```
dir1/dir2/file.html
```

или просто как

```
file.html
```

В последнем случае к ссылке добавляются строка "http://" и URL текущей страницы, начиная с корневого каталога Web-сайта, а затем добавляется обратная черта (/). В предыдущем случае извлекается часть пути (или каталог) из полного имени файла для получения полностью заданного URL. При этом добавляется строка "http://", адрес узла Internet, путь и ссылка для записи полностью заданного URL.

После этих действий удаляются ссылка на страницу и строка "www" из полностью заданного URL.

```
// Удалить ссылки на страницу.
int index = link.indexOf('#');
if (index != -1) {
  link = link.substring(0, index);
}
// При наличии удалить предшествующие
// символы "www" из адреса узла Internet.
link = removeWwwFromUrl(link);
```

Простые ссылки на страницу отбрасываются, как и ссылки на страницу, дополняющие адрес. Предшествующие символы "www" удаляются из ссылки, для того чтобы не было дублирования ссылок.

Затем ссылки проверяются на предмет того, что это допустимый адрес.

```
// Проверить ссылку и отбросить, если она неправильная.
URL verifiedLink = verifyUrl(link);
if (verifiedLink == null) {
  continue;
}
```

После проверки того, является ли ссылка правильно сформированным URL, производится также проверка, остался ли адрес узла Internet тем же, что был указан в поле Start URL. И еще одна проверка необходима для подтверждения того, что данная ссылка еще не обрабатывалась.

```
/* Если указано значение для поля START URL, то используются
   ссылки только для этого узла. */
if (limitHost &&
      !pageUrl.getHost().toLowerCase().equals(
        verifiedLink.getHost().toLowerCase()))
{
  continue;
}
// Отбросить ссылку, если она уже обработана.
if (crawledList.contains(link)) {
  continue;
}
```

Оканчивается метод `retrieveLinks()` тем, что каждая ссылка, которую пропустили все фильтры, добавляется в список ссылок.

```
// Добавить ссылку в список.
  linkList.add(link);
}
return (linkList);
```

После того как цикл `while` закончится и все ссылки будут добавлены в список, возвращается список ссылок.

Метод searchStringMatches()

Метод `searchStringMatches()`, листинг которого представлен ниже, используется для поиска совпадений на Web-странице, загруженной в процессе работы поискового червя. С помощью этого метода находятся заданные строки, если они есть в содержимом страницы.

```
/* Определить, есть или нет заданная строка
   среди содержимого данной страницы. */
private boolean searchStringMatches(
    String pageContents, String searchstring,
      boolean caseSensitive)
{
  String searchContents = pageContents;
  /* Если поиск с учетом регистра клавиатуры,
     преобразовать содержимое в нижний регистр перед сравнением. */
  if (!caseSensitive) {
    searchContents = pageContents.toLowerCase();
  }
  // Разбить содержимое на отдельные термы.
  Pattern p = Pattern.compile("[\\s]+");
  String[] terms = p.split(searchstring);
  // Проверить каждый терм на совпадение.
  for (int i = 0; i < terms.length; i++) {
    if (caseSensitive) {
      if (searchContents.indexOf(terms[i]) == -1) {
        return false;
      }
    }
    else {
      if (searchContents.indexOf(terms[i].toLowerCase()) == -1) {
        return false;
      }
    }
  }
  return true;
}
```

Поскольку поиск может производиться с учетом регистра клавиатуры или без него (по умолчанию), метод `searchStringMatches()` начинается с объявления локальной переменной `searchContents`, которая ссылается на строку, в которой производится поиск. По умолчанию значение переменной `pageContents` присваивается переменной `searchContents`. Если поиск производится с учетом регистра, то переменная `searchContents` будет представлять строку, которая является версией строки для переменной `pageContents` в нижнем регистре.

Затем строка поиска разбивается на отдельные термы поиска с помощью методов библиотеки Java для регулярных выражений. Для того чтобы разбить строку, сначала компилируется шаблон регулярного выражения с помощью метода `compile()` из объекта `Pattern`. Шаблон, примененный в данном случае, является строкой "`[\\s]+`". Это говорит о том, что должен обнаруживаться один или несколько пробелов (это символы пробела, табуляции или новой строки). Затем вызывается метод `split()` из класса `Pattern` для обработки данной строки, который возвращает массив `String`, содержащий отдельные выделенные термы.

После разбиения строки производится циклический просмотр отдельных термов с целью определить совпадения в содержимом страницы. Метод `indexOf()`, опи-

санный в классе String, используется для поиска в содержимом переменной searchContents. Если возвращаемое значение равно -1, то соответствующий терм не найден и метод searchStringMatches() возвращает значение false, т.к. должны быть найдены все термы в том порядке, как они заданы в строке поиска. Обратите внимание, что если при поиске не должен учитываться регистр клавиатуры, то при поиске каждый терм берется в нижнем регистре. Это совпадает со значением, присвоенным переменной searchContents в начале метода. Если цикл выполняется полностью, метод searchStringMatches() завершается возвратом значения true. Это говорит о том, что совпадающая строка найдена.

Метод crawl()

Метод crawl() является ядром поискового червя, т.к. именно в этом методе выполняется реальная обработка Web-сайтов. Он начинается следующими строками кода

```
// Открыть списки поиска.
HashSet crawledList = new HashSet();
LinkedHashSet toCrawlList = new LinkedHashSet();
// Добавить начальный URL в объект toCrawllist.
toCrawlList.add(startUrl);
```

Существует несколько способов для обработки Web-сайтов с помощью поискового червя. Наиболее естественно использовать рекурсию, т.к. поиск сам по себе является рекурсивным. Но использование рекурсии может потребовать значительных затрат ресурсов и поэтому в поисковом черве применяется метод очередности. В нашем случае создается объект toCrawlList для хранения очереди ссылок, подлежащих обработке. Затем в объект toCrawlList добавляется начальный URL и запускается процесс обработки Web-сайтов.

После инициализации объекта toCrawlList и добавления начального URL следует цикл while, который выполняется до тех пор, пока не будет сброшен флаг crawling или список объекта toCrawlList будет полностью пройден, как показано в листинге ниже.

```
/* Выполнить реальную обработку в цикле
   для всех элементов списка объекта toCrawlList. */
while (crawling && toCrawlList.size() > 0) {
  /* Производить проверку на достижения максимального
     значения URL, если это значение задано. */
  if (maxUrls !=-1){
    if (crawledList.size() == maxUrls) {
      break;
    }
  }
```

Запомните, что флаг crawling используется для преждевременного прекращения обработки. Если щелкнуть на кнопке Stop графического интерфейса, то переменная crawling примет значение false, т.е. флаг будет сброшен. Поэтому при следующей итерации после проверки условия выполнения цикла произойдет выход из цикла, т.к. результатом вычисления условного выражения будет значение false.

За циклом while идет проверка того, было ли достигнуто заданное максимальное количество URL (значение для переменной maxUrls). Эта проверка выполня-

ется только в том случае, если было установлено значение для переменной maxUrls, которое должно отличаться от -1.

При каждой итерации должен выполняться следующий код

```
// Получить очередной URL из списка.
String url = (String) toCrawlList.iterator().next();
// Удалить URL из списка.
toCrawlList.remove(url) ;
// Преобразовать строку URL в объект URL.
URL verifiedUrl = verifyUrl(url);
// Отбросить URL, если робот запрещает доступ к нему.
if (!isRobotAllowedfverifiedUrl)) {
  continue;
}
```

Во-первых, извлекается очередной URL из конца списка. Таким образом, список работает в режиме "первым вошел – первым вышел" (First In, First Out — FIFO). Так как URL сохраняются в объекте типа LinkedHashSet, то здесь не используется технология, подобная стеку. Вместо этого функциональность метода извлечения симулируется извлечением элемента списка с помощью метода toCrawlList.iterator().next(). Когда URL извлекается из списка, то одновременно он и удаляется из списка с помощью вызова метода toCrawlList.remove(), в который в качестве аргумента передается заданный URL.

После извлечения URL из списка выполняется преобразование строкового представления URL в объект URL с помощью метода verifyUrl(). Затем производится проверка на разрешение обработки данного URL с помощью метода isRobotAllowed(). Если обработка данного URL разрешена, то выполняется утверждение continue для перехода к следующей итерации цикла while.

После извлечения и проверки очередного URL из списка, производится обновление результатов в секции Stats, как показано в листинге.

```
// Обновить статистику.
updateStats(url, crawledList.size(), toCrawlList.size(),maxUrls);
// Добавить страницу в список поиска.
crawledList.add(url);
// Загрузить страницу для данного URL.
String pageContents = downloadPage(verifiedUrl);
```

Выходные данные обновляются с помощью вызова метода updateStats(). Данный URL затем добавляется в список поиска, а это говорит о том, что он был обработан и что последующие ссылки на данный URL должны быть отброшены. Затем по данному URL загружается страница с помощью вызова метода downloadPage().

Если метод downloadPage() успешно загрузил данную страницу, то выполняется следующий код.

```
/* При успешной загрузке страницы извлекаются все ее ссылки
   и производится проверка на совпадение с заданной строкой. */
if (pageContents != null && pageContents.length() > 0) {
  // Извлечь все допустимые ссылки.
  ArrayList links =
    retrieveLinks(verifiedUrl, pageContents, crawledList, limitHost);
  // Добавить ссылки в список поиска.
  toCrawlList.addAll(links);
  /* Проверить содержимое на совпадение с заданной строкой
     и если совпадение есть, записать совпадение. */
  if (searchStringMatches(pageContents, searchString, caseSensitive))
```

```
    {
      addMatch(url);
    }
  }
```

Сначала из содержимого страницы извлекаются все ссылки с помощью метода `retrieveLinks()`. Каждая из ссылок, возвращаемая методом `retrieveLinks()`, добавляется в список поиска. Затем в загруженной странице с помощью метода `searchStringMatches()` определяются строки, совпадающие с заданной строкой. Если совпадающая строка найдена на странице, то страница записывается как совпадающая с помощью метода `addMatch()`.

Метод `crawl()` заканчивается еще одним вызовом метода `updateStats()` в конце цикла `while`.

```
// Обновить статистику.
updateStats(url, crawledList.size(), toCrawlList.size(),
  maxUrls);
}
```

Первый вызов метода `updateStats()`, который происходил ранее в методе, обновлял метки, отображающие URL, который в данный момент обрабатывался. При втором вызове обновляются все другие значения, поскольку они были изменены уже после первого вызова метода `updateStats()`.

Компиляция и запуск поискового червя

Как уже упоминалось, в класс `SearchCrawler` используются дополнительные возможности, предоставляемые новым пакетом Java для работы с регулярными выражениями: `java.util.regex`. Это пакет был введен в версию JDK 1.4, поэтому необходимо использовать версию JDK 1.4 или более позднюю для компиляции и запуска поискового червя. Выполнить компиляцию можно с помощью следующей команды.

```
javac SearchCrawler.Java
```

А запустить поискового червя можно следующим образом.

```
javaw SearchCrawler
```

Интерфейс поискового червя довольно простой, но с достаточным количеством необходимых настроек, которые легко использовать. Сначала в поле Start URL вводится начальный URL, с которого необходимо начинать поиск. Затем задается максимально допустимое количество URL (поле Max URLs to Crawl), которое необходимо просмотреть, если вы не хотите ограничиться только одним Web-сайтом, указанным в поле Start URL. Если вы не хотите ограничивать количество просмотренных Web-сайтов и желаете просмотреть все сайты, которые червь может найти, то оставьте поле Max URLs to Crawl пустым. Но помните, что если не ограничивать количество просмотренных сайтов, то просмотр всех возможных сайтов может занять достаточно много времени.

Затем следует поле Matches Log File. Это поле задано по умолчанию, т.е. файл журнала с именем `crawler.log` будет находиться в каталоге, из которого будет запускаться поисковый червь. При желании изменить путь по умолчанию, в поле Matches Log File вводится тот путь, который необходим. После этого введите строку, которую необходимо искать на Web-страницах, и отметьте, должен ли при поиске

учитываться регистр клавиатуры. Заметьте, что после ввода строки поиска, состоящей из нескольких слов, совпадение должно происходить для всех слов.

После ввода всех критериев поиска и конфигурации критериев поиска, щелкните на кнопке Search. После этого отметьте, что элементы управления секции поиска перестают быть активными, а кнопка Search изменяется на кнопку Stop. После завершения поиска элементы управления панели поиска вновь станут активными, а кнопка Stop вновь станет кнопкой Search. Щелчок на кнопке Stop приведет к завершению просмотра URL и будет прерван просмотр текущего URL. На рис. 6.2 показан поисковый червь в работе.

Несколько замечаний о функционировании поискового червя.

Поддерживаются только ссылки на сайты HTTP, и не поддерживаются форматы HTTPS или FTP.

Не поддерживается переориентация одного URL на другой.

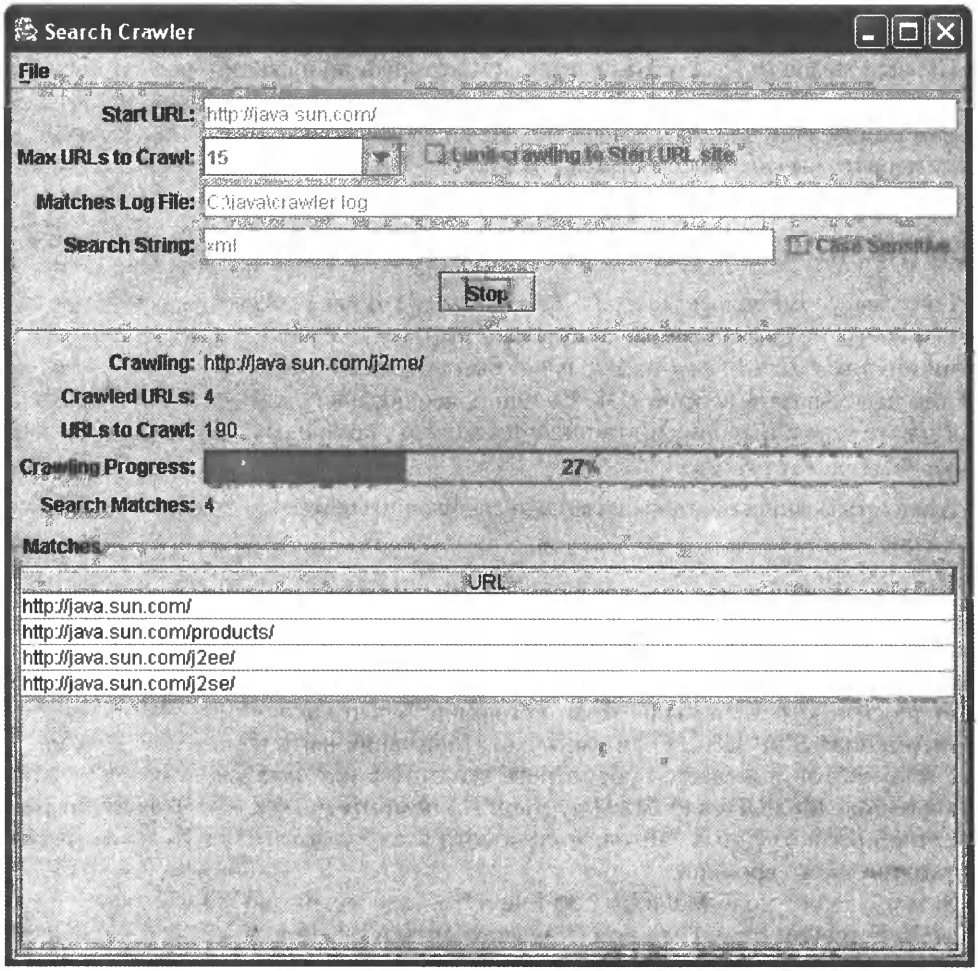


Рис. 6.2. Поисковый червь в работе

Одинаковые ссылки, такие как "http://osborne.com" и "http://osborne.com/" (обратите внимание на обратную черту в конце ссылки), трактуются как отдельные уникальные ссылки. Это происходит потому, что поисковый червь не может обобщать различные записи ссылки.

Возможности поискового червя

Поисковый червь является великолепной иллюстрацией сетевых возможностей языка Java. Он также демонстрирует базовую технологию, связанную с поиском в Web. Как уже упоминалось в начале главы, хотя поисковый червь интересен и сам по себе, набольшую пользу он может принести при разработке улучшенных на его основе программ.

Для начала вы можете несколько усовершенствовать поисковый червь. Постарайтесь изменить способ, которым он отслеживает ссылки. Возможно, поиск в глубину будет эффективнее, чем поиск в ширину. Также постарайтесь добавить поддержку URL, которые перенаправлены на другие URL. Поэкспериментируйте с оптимизацией поиска, возможно, для повышения производительности будет удобнее использовать несколько потоков, загружая несколько страниц одновременно.

При желании разработать свой собственный поисковый червь, можно воспользоваться следующими идеями.

Поиск разрушенных ссылок. Для поиска разрушенных ссылок можно использовать поискового червя и отмечать все ссылки, которые разрушены. Каждая разрушенная ссылка должна быть записана. В конце поиска должен создаваться отчет с указанием страниц, на которых обнаружены разрушенные ссылки, и с указанием каждой разрушенной ссылки. Такое приложение будет особенно полезно для больших Web-сайтов, где находятся сотни и даже тысячи страниц, для которых необходимо производить поиск разрушенных ссылок.

Поиск для сравнения. Поисковый червь для проведения сравнений на нескольких Web-сайтах может быть полезен при анализе некоторых данных, например поиске наименьшей цены на отдельный продукт. Поисковый червь для проведения сравнений может произвести обработку сайтов Amazon.com, Barnes&Noble.com и еще нескольких сайтов по продаже книг в целях выяснения минимальной цены. Такой прием называется "прочесыванием экрана" (screen scraping) и подходит для сравнения цен на многие различные типы вещей в Internet.

Поиск для архивизации. Поисковый червь для архивизации полезен при обработке сайта и архивизации всех его страниц. Есть множество причин, по которым необходимо архивировать сайт, включая возможность просмотра Web-сайта в автономном режиме, создания резервных копий или получения копии страницы. На самом деле поисковые движки используют технологию поискового червя для возможностей архивизации. Поисковый движок использует червя для просмотра всех страниц и сохранения найденных при поиске страниц. После чего производит обработку и индексацию данных для их быстрого поиска.

Это только часть идей, которые связаны с расширением возможностей поискового червя, его использование может принести пользу и в очень многих других приложениях для работы с Web.

This page is intentionally left blank

ГЛАВА

7

Формат HTML и Java

Читателям должно быть хорошо известно, что в основе Web лежит язык HTML. Благодаря механизму гиперссылок, используемому в языке HTML, можно располагать информацию в нелинейном порядке, в отличие от того, как это обычно делается, — текст располагается последовательно сверху вниз. Благодаря удобству использования гиперссылок язык HTML применяют не только в приложениях, рассчитанных на Internet. Например, большинство создаваемых сегодня файлов справки используют язык HTML для удобного представления информации. Поэтому язык HTML широко используется для отображения информации с помощью компьютера, и довольно часто можно столкнуться с ситуацией, когда необходимо или желательно использовать язык HTML. В недавнем прошлом такая задача осложнялась тем, что необходимо было хорошо знать все возможности языка HTML и уметь использовать механизм гиперссылок. С счастью, среда Java помогает упростить этот процесс, хотя не все программисты это знают.

Классы поддержки язык HTML включены в пакет Swing. С помощью этих классов можно отображать тексты, написанные на языке HTML, и обрабатывать уведомления о встречающихся гиперссылках. Для отображения языка HTML используется класс `JEditorPane`. События, связанные с гиперссылками, обрабатываются с помощью класса `HyperlinkEvent` и интерфейса `HyperlinkListener`. Встроенная поддержка языка HTML в среду Java программистами не часто используется.

Эта глава начинается с описания того, как следует отображать тексты, написанные на языке HTML с помощью класса `JEditorPane`, и как обрабатывать события, связанные с гиперссылками. Затем будут продемонстрированы эти возможности при разработке простого Web-браузера. Разумеется, возможность отображения языка HTML не ограничивается только Web-браузером. Эту технику можно использовать во всех случаях, когда производится работа с языком HTML.

Отображение HTML с помощью JEditorPane

С помощью класса `JEditorPane` можно легко производить отображение текстов, написанных на языке HTML. Просто создайте экземпляр класса `JEditorPane` и установите тип содержимого как "`text/html`". Также при отображении формата HTML необходимо запретить возможность редактирования. В следующих строках показано, как это сделать.

```
JEditorPane htmlViewer = new JEditorPane();
htmlViewer.setContentType("text/html");
htmlViewer.setEditable(false);
```

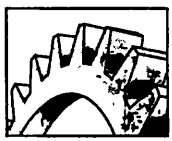
После создания экземпляра `JEditorPane` с именем `htmlViewer`, вызывается метод `setContentType()`, для того чтобы объект `htmlViewer` обрабатывал текст как написанный на языке HTML. Вызов метода `setEditable()` с аргументом `false` запрещает редактирование отображаемого содержимого. При этом будут обрабатываться теги HTML вместо отображения самих тегов. Эти шаги необходимы для того, чтобы подготовить объект `JEditorPane` для отображения формата HTML и получить пример искусного использования языка Java.

После того как объект JEditorPane подготовлен для работы с языком HTML, можно использовать два способа для загрузки (или отображения) страницы HTML, как показано ниже.

```
// Способ 1.
htmlViewer.setText("<html>Hello World!</html>");
// Способ 2.
try {
  htmlViewer.setPage("http://www.dmoz.org/");
}
catch (Exception e) {
  // Обработка ошибок здесь.
}
```

В первом случае вызывается метод setText(), в который передается текст на языке HTML как строка. При этом будет отображен текст, сформатированный в соответствии с тегами языка HTML. Во втором случае вызывается метод setPage(), в который передается адрес страницы для отображения. В примерах этой главы будет использоваться исключительно второй способ.

Во время написания этой книги класс JEditorPane поддерживал только версию 3.2 языка HTML. Это означает, что если будет отображаться страница с более новой версией языка HTML, то отображение может быть некорректным. В последних версиях Java эта должно быть устранено.



Во время написания этой книги класс JEditorPane поддерживал только версию 3.2 языка HTML. Новейшие версии языка HTML не будут отображаться корректно.

Обработка событий для гиперссылок

В дополнение к отображению формата HTML, Java поддерживает возможность перехватывать события, связанные с гиперссылками на Web-страницах, и обрабатывать их соответствующим образом. Для перехвата событий, возникающих при обработке страницы HTML, необходимо для объекта JEditorPane зарегистрировать интерфейс HyperlinkListener. В интерфейсе HyperlinkListener определен только один метод — hyperlinkUpdate(), который объявлен следующим образом.

```
public void hyperlinkUpdate(HyperlinkEvent event)
```

Здесь вместо параметра event подставляется событие, которое было сгенерировано при встрече гиперссылки.

Для того чтобы добавить объект HyperlinkListener, используйте метод addHyperlinkListener(), определенный в классе JEditorPane. Например, ниже показан способ, как можно добавить объект HyperlinkListener в объект htmlViewer, созданный в предыдущем разделе.

```
htmlViewer.addHyperlinkListener(new HyperlinkListener() {
  public void hyperlinkUpdate(HyperlinkEvent event) {

    // Обработка события здесь.
  }
});
```

Каждый раз, когда производится щелчок на гиперссылке, генерируется событие HyperlinkEvent, при этом будет уведомлен каждый зарегистрированный объект HyperlinkListener с помощью вызова метода hyperlinkUpdate(). Обычно реализация метода hyperlinkUpdate() состоит из кода, с помощью которого производится отображение ссылки, на которой был произведен щелчок. Именно таким образом браузер из данного примера производит обработку ссылок.

Для события HyperlinkEvent можно использовать четыре метода для получения информации о ссылке: getDescription(), getURL(), getEventType() и getSourceElement(). Каждый из них возвращает указанный элемент. В дальнейшем в этой главе будет использоваться метод getURL(), который возвращает адрес — объект URL, связанный со ссылкой, на которой был произведен щелчок, и метод getEventType(), который возвращает объект HyperlinkEvent.EventType, указывающий на тип события. Возможны три типа: ACTIVATED, ENTERED и EXITED. В дальнейшем будет использоваться тип ACTIVATED, который говорит о том, что на ссылке был произведен щелчок мыши.

Классы HyperlinkEvent и HyperlinkListener содержатся в пакете javax.swing.event.

Создание мини-Web-браузера

Для иллюстрации встроенных в Java возможностей по обработке языка HTML, остаток этой главы будет посвящен разработке простого Web-браузера, названного мини-Web-браузером. Мини-Web-браузер обеспечивает базовую возможность по обработке страниц HTML, такую как перемещение по странице вперед и назад. Поскольку для отображения страницы используются возможности класса JEditorPane, то поддерживается только язык HTML версии 3.2, поэтому некоторые страницы могут отображаться неправильно, если для из создания использовалась более поздняя версия языка HTML. Хотя мини-Web-браузер имеет только минимальные средства для просмотра HTML-страниц, его возможности легко расширить. Его можно использовать как отправную точку при разработке собственного Web-браузера. Ни рис. 7.1 показано окно мини-Web-браузера.

В верхней части окна есть две кнопки для перемещения по странице, которая отображается в Web-браузере. Также есть текстовое поле для ввода адреса страницы, которую необходимо отобразить в браузере. После ввода адреса в текстовое поле, производится щелчок на кнопке Go и происходит загрузка необходимой страницы.

Класс MiniBrowser

Код мини-Web-браузера содержится в классе MiniBrowser. Класс MiniBrowser включает метод main(), который и будет вызван первым при запуске на выполнение. В методе main() создается новый объект MiniBrowser и затем вызывается метод show(), который приводит к отображению окна.

Класс MiniBrowser, листинг которого приведен ниже, подробно рассмотрен в следующих разделах этой главы. Обратите внимание, что он является наследником класса Jframe и реализует интерфейс HyperlinkListener.

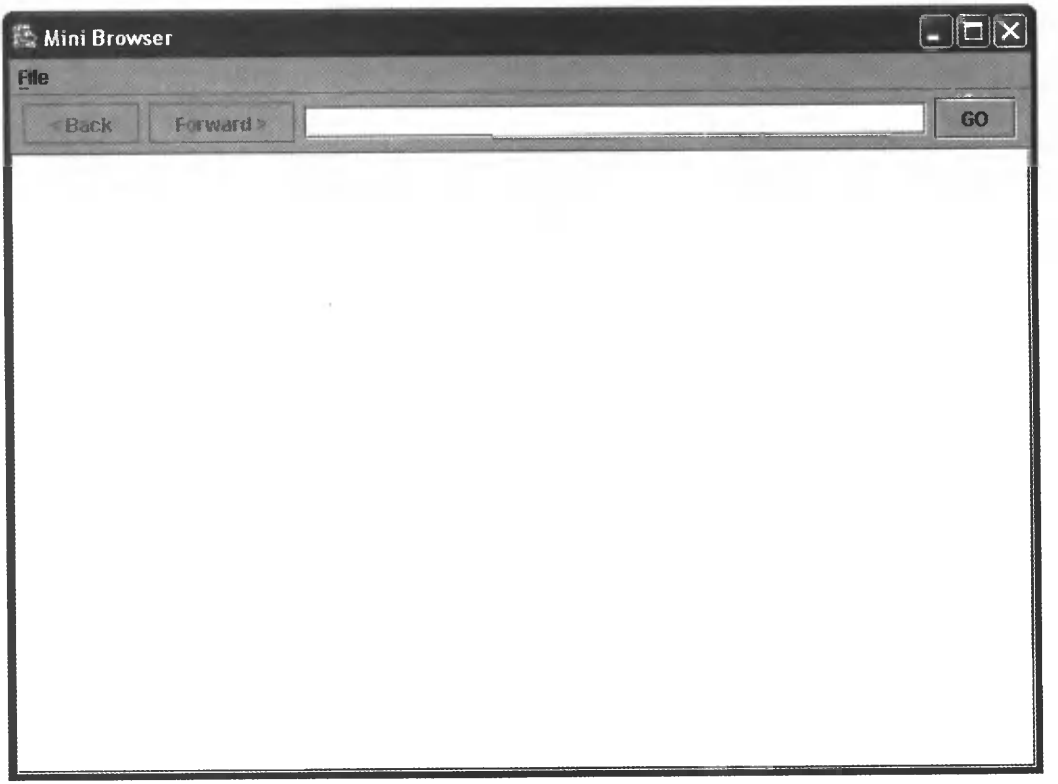


Рис. 7.1. Графический интерфейс мини-Web-браузера

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.html.*;

// Мини-Web-браузер.
public class MiniBrowser extends JFrame
  implements HyperlinkListener
{
  // Кнопки для перемещения по странице.
  private JButton backButton, forwardButton;

  // Поле для адреса страницы.
  private JTextField locationTextField;

  // Панель для отображения страницы.
  private JEditorPane displayEditorPane;

  // Список страниц, которые были посещены.
  private ArrayList pageList = new ArrayList();

  // Конструктор для мини-Web-браузера.
  public MiniBrowser()
```

```
{
  // Установить заголовок.
  super("Mini Browser");

  // Установить размер окна.
  setSize(640, 480);

  // Обработка событий при закрытии.
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      actionExit();
    }
  });

  // Установить меню "File".
  JMenuBar menuBar = new JMenuBar();
  JMenu fileMenu = new JMenu("File");
  fileMenu.setMnemonic(KeyEvent.VK_F);
  JMenuItem fileExitMenuItem = new JMenuItem("Exit",
    KeyEvent.VK_X);
  fileExitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionExit();
    }
  });
  fileMenu.add(fileExitMenuItem);
  menuBar.add(fileMenu);
  setJMenuBar(menuBar);

  // Установить панель кнопок.
  JPanel buttonPanel = new JPanel();
  backButton = new JButton("< Back");
  backButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionBack();
    }
  });
  backButton.setEnabled(false);
  buttonPanel.add(backButton);
  forwardButton = new JButton("Forward >");
  forwardButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionForward();
    }
  });
  forwardButton.setEnabled(false);
  buttonPanel.add(forwardButton);
  locationTextField = new JTextField(35);
  locationTextField.addKeyListener(new KeyAdapter() {
    public void keyReleased(KeyEvent e) {
      if (e.getKeyCode() == KeyEvent.VK_ENTER) {
        actionGo();
      }
    }
  });
  buttonPanel.add(locationTextField);
  JButton goButton = new JButton("GO");
  goButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionGo();
    }
```

```
    });
    buttonPanel.add(goButton);

    // Настроить отображение страницы.
    displayEditorPane = new JEditorPane();
    displayEditorPane.setContentType("text/html");
    displayEditorPane.setEditable(false);
    displayEditorPane.addHyperlinkListener(this);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(buttonPanel, BorderLayout.NORTH);
    getContentPane().add(new JScrollPane(displayEditorPane),
      BorderLayout.CENTER);
  }

  // Выход из программы.
  private void actionExit() {
    System.exit(0);
  }

  // Вернуться к ранее просматриваемой странице.
  private void actionBack() {
    URL currentUrl = displayEditorPane.getPage();
    int pageIndex = pageList.indexOf(currentUrl.toString());
    try {
      showPage(
        new URL((String) pageList.get(pageIndex - 1)), false);
    }
    catch (Exception e) {}
  }

  // Перейти к следующей странице.
  private void actionForward() {
    URL currentUrl = displayEditorPane.getPage();
    int pageIndex = pageList.indexOf(currentUrl.toString());
    try {
      showPage(
        new URL((String) pageList.get(pageIndex + 1)), false);
    }
    catch (Exception e) {}
  }

  // Загрузить и отобразить страницу,
  // адрес которой указан в текстовом поле.
  private void actionGo() {
    URL verifiedUrl = verifyUrl(locationTextField.getText());
    if (verifiedUrl != null) {
      showPage(verifiedUrl, true);
    } else {
      showError("Invalid URL");
    }
  }

  // Показать диалоговое окно ошибки.
  private void showError(String errorMessage) {
    JOptionPane.showMessageDialog(this, errorMessage,
      "Error", JOptionPane.ERROR_MESSAGE);
  }

  // Проверить формат URL.
  private URL verifyUrl(String url) {
```

```
    // Разрешены только HTTP-адреса.
    if (!url.toLowerCase().startsWith("http://"))
      return null;

    // Проверить URL.
    URL verifiedUrl = null;
    try {
      verifiedUrl = new URL(url);
    } catch (Exception e) {
      return null;
    }
    return verifiedUrl;
  }

  /* Показать указанную страницу и добавить ее
     в список страниц. */
  private void showPage(URL pageUrl, boolean addToList)
  {
    // Показать песочные часы на время работы поискового червя.
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    try {
      // Получить адрес страницы для отображения.
      URL currentUrl = displayEditorPane.getPage();

      // Загрузить и отобразить указанную страницу.
      displayEditorPane.setPage(pageUrl);

      // Получить адрес новой страницы для отображения.
      URL newUrl = displayEditorPane.getPage();

      // Добавить страницу в список.
      if (addToList) {
        int listSize = pageList.size();
        if (listSize > 0) {
          int pageIndex =
            pageList.indexOf(currentUrl.toString());
          if (pageIndex < listSize - 1) {
            for (int i = listSize - 1; i > pageIndex; i--) {
              pageList.remove(i);
            }
          }
        }
        pageList.add(newUrl.toString());
      }

      // Обновить текстовое поле.
      locationTextField.setText(newUrl.toString());

      // Обновить кнопки в соответствии с отображаемой страницей.
      updateButtons();
    }
    catch (Exception e)
    {
      // Показать сообщение об ошибке.
      showError("Unable to load page");
    }
    finally
    {
      // Установить курсор по умолчанию.
      setCursor(Cursor.getDefaultCursor());
    }
  }
```

```
  /* Обновить кнопки back и forward в соответствии
     со страницей, которая будет отображаться. */
  private void updateButtons() {
    if (pageList.size() < 2) {
      backButton.setEnabled(false);
      forwardButton.setEnabled(false);
    } else {
      URL currentUrl = displayEditorPane.getPage();
      int pageIndex = pageList.indexOf(currentUrl.toString());
      backButton.setEnabled(pageIndex > 0);
      forwardButton.setEnabled(
        pageIndex < (pageList.size() - 1));
    }
  }

  // Обработать щелчок на гиперссылке.
  public void hyperlinkUpdate(HyperlinkEvent event) {
    HyperlinkEvent.EventType eventType = event.getEventType();
    if (eventType == HyperlinkEvent.EventType.ACTIVATED) {
      if (event instanceof HTMLFrameHyperlinkEvent) {
        HTMLFrameHyperlinkEvent linkEvent =
          (HTMLFrameHyperlinkEvent) event;
        HTMLDocument document =
          (HTMLDocument) displayEditorPane.getDocument();
        document.processHTMLFrameHyperlinkEvent(linkEvent);
      } else {
          showPage(event.getURL(), true);
      }
    }
  }

  // Запустить MiniBrowser.
  public static void main(String[] args) {
    MiniBrowser browser = new MiniBrowser();
    browser.show();
  }
}
```

Переменные класса MiniBrowser

Класс `MiniBrowser` начинается с объявления нескольких переменных, большинство из которых содержат ссылки на элементы управления графического интерфейса. Сначала объявляются экземпляры класса `Jbutton` для создания кнопок **Back** и **Forward**. Эти кнопки используются для перемещения по странице, которую отображает браузер. Затем объявляется объект `locationTextField` типа `JTextField`. Этот объект представляет текстовое поле, в которое вводится адрес (URL) страницы, которая должна отображаться в браузере. После чего объявляется объект `displayEditorPane`. Он представляет экземпляр класса `JEditorPane` и используется для отображения Web-страниц. Наконец, объявляется объект `pageList`, который представляет список страниц, уже просмотренных с помощью браузера.

Конструктор класса MiniBrowser

При создании объекта `MiniBrowser` в конструкторе производится инициализация всех элементов управления графического интерфейса. Хотя конструктор содержит большое количество строк кода, большая часть его не представляет трудностей

для понимания. Ниже произведен краткий обзор всех действий, производимых в конструкторе.

Сначала устанавливается заголовок окна. Затем вызывается метод `setSize()` для задания ширины и высоты окна в пикселях. После чего добавляется слушатель окна с помощью вызова метода `addWindowListener()`, в который передается объект `WindowAdapter` для переопределения обработчика событий `windowClosing()`. Новый обработчик при закрытии окна вызывает метод `actionExit()`. Затем в окно добавляется панель меню с пунктом меню File.

Подобно другим приложениям, описанным в этой книге, следующие несколько строк конструктора инициализируют и распределяют элементы управления графического интерфейса. Панель кнопок содержит кнопки Back, Forward и Go. На этой панели также располагается текстовое поле ввода. Объект `ActionListener` добавляется для каждой кнопки, чтобы вызывался соответствующий метод при каждом щелчке на кнопке. Аналогичным образом для текстового поля ввода добавляется объект `KeyAdapter`. При этом если текстовое поле имеет фокус, то будет вызываться метод `actionGo()` при каждом нажатии клавиши <ENTER>.

Следующей устанавливается панель редактора для отображения заданных страниц. Здесь необходимо обратить внимание на три момента. Во-первых, для типа содержимого устанавливается значение "`text/html`", что заставляет редактор обрабатывать страницы в формате HTML. Во-вторых, в метод `setEditable()` передается значение `false`, что не позволит производить редактирование текста, отображаемого в редакторе. В-третьих, вызывается метод `displayEditorPane.addHyperlinkListener()` с целью зарегистрировать браузер для приема событий `HyperlinkEvents`.

Конструктор класса `MiniBrowser` оканчивается тем, что панель кнопок и редактор добавляются в графический интерфейс. Обратите внимание, что панель редактора помещена в экземпляр класса `JScrollPane`. Это позволяет прокручивать панель.

Метод actionBack()

Метод `actionBack()`, листинг которого приведен ниже, вызывается при каждом щелчке на кнопке Back и используется для перехода к предшествующей странице, которая уже была просмотрена с помощью браузера.

```
// Вернуться к предшествующей странице, просмотренной в браузере.
private void actionBack() {
  URL currentUrl = displayEditorPane.getPage();
  int pageIndex = pageList.indexOf(currentUrl.toString());
  try {
    showPage(
      new URL((String) pageList.get(pageIndex - 1)), false);
  }
  catch (Exception e) {}
}
```

В этом методе сначала извлекается адрес страницы, отображаемой в браузере, с помощью вызова метода `displayEditorPane.getPage()`. Затем извлекается индекс страницы из списка страниц с помощью метода `indexOf()` из класса `ArrayList`. В него передается ссылка на объект и возвращается индекс объекта из списка. Наконец, вызывается метод `showPage()`, для того чтобы отобразить предыдущую страницу. Заметьте, что аргумент `false` передается в метод `showPage()` вместо

второго параметра. Это говорит о том, что отображаемая страница не должна добавляться в список страниц, поскольку она не является новой страницей и уже есть в списке. Вызов метод showPage() заключен в блок try...catch, поскольку конструктор объекта URL может сгенерировать исключительную ситуацию, если во время преобразования URL из строки произойдет ошибка. Так как все страницы в списке страниц уже были проверены с помощью метода verifyUrl(), блок catch можно не заполнять.

Метод actionForward()

Метод actionForward(), листинг которого приведен ниже, вызывается при каждом щелчке на кнопке Forward и приводит к отображению следующей страницы из списка страниц, уже просмотренных с помощью браузера.

```
// Переход к следующей странице из списка страниц,
// просмотренных браузером.
private void actionForward() {
  URL currentUrl = displayEditorPane.getPage();
  int pageIndex = pageList.indexOf(currentUrl.toString());
  try {
    showPage(
      new URL((String) pageList.get(pageIndex + 1)), false);
  }
  catch (Exception e) {}
}
```

Аналогично методу action Back(), в этом методе сначала извлекается адрес страницы, просматриваемой в браузере, с помощью вызова метода displayEditorPane.getPage(). Затем с помощью метода indexOf() извлекается индекс этой страницы в списке страниц. После чего извлекается индекс объекта, соответствующего ссылке на объект. Наконец, вызывается метод showPage() для отображения страницы, следующей за отображаемой страницей. Обратите внимание, что аргумент false передается в метод showPage() вместо второго параметра. Это говорит о том, что отображаемая страница не должна добавляться в список страниц, поскольку она не является новой страницей и уже есть в списке. Вызов метод showPage() заключен в блок try...catch, поскольку конструктор объекта URL может сгенерировать исключительную ситуацию, если во время преобразования URL из строки произойдет ошибка. Так как все страницы в списке страниц уже были проверены с помощью метода verifyUrl(), блок catch можно не заполнять.

Метод actionGo()

Каждый раз при вводе URL в текстовое поле и щелчке на кнопке Go вызывается метод actionGo(). Также этот метод может быть вызван при нажатии клавиши <ENTER>, когда текстовое поле имеет фокус. Листинг метода actionGo() приведен ниже.

```
// Загрузить и отобразить спецификацию страницы в текстовом поле.
private void actionGo() {
  URL verifiedUrl = verifyUrl(locationTextField.getText());
  if (verifiedUrl != null) {
    showPage(verifiedUrl, true);
```

```
    }
    else {
      showError(" Invalid URL");
    }
}
```

Метод `actionGo()` начинается с проверки URL, введенного в текстовое поле, с помощью вызова метода `verifyUrl()`. Метод `verifyUrl()` в случае успешного завершения возвращает объект URL, а также возвращает `null`, если формат URL был неправильным. Если введенный адрес является правильно сформированным, то вызывается метод `showPage()` для отображения страницы. В противном случае произойдет отображение диалогового окна ошибки с помощью вызова метода `showError()`.

Метод showError()

Метод `showError()`, листинг которого приведен ниже, отображает на экране диалоговое окно ошибки с заданным сообщением. Этот метод вызывается в классе `MiniBrowser` каждый раз, когда происходит ошибка.

```
// Отобразить диалоговое окно с сообщением об ошибке.
private void showError(String errorMessage) {
  JOptionPane.showMessageDialog(this, errorMessage,
    "Error", JOptionPane.ERROR_MESSAGE);
}
```

Метод verifyUrl()

Метод `verifyUrl()`, листинг которого приведен ниже, используется в методе `actionGo()` для проверки формата URL. Дополнительно этот метод служит для преобразования строкового представления URL в объект URL.

```
// Проверить формат URL.
private URL verifyUrl(String url) {
  // Разрешены только адреса HTTP-сайтов.
  if (!url.toLowerCase().startsWith("http://"))
    return null;
  // Проверить формат URL.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  }
  catch (Exception e) {
    return null;
  }
  return verifiedUrl;
}
```

В этом методе сначала проверяется, что данный URL является адресом HTTP-сайта, т.к. только такие адреса поддерживаются в классе `MiniBrowser`. Затем производится проверка формата URL и преобразование его в объект URL. Если URL сформирован неправильно, то в конструкторе класса URL генерируется исключительная ситуация и метод возвращает значение `null`. Возвращаемое значение `null` говорит о том, что переданная в объект URL строка не является допустимым адресом.

Метод showPage()

Метод showPage() загружает и отображает страницы в панели редактора мини-Web-браузера. Поскольку в этом методе обрабатывается много действий, производится последовательная проверка строки за строкой. Метод начинается следующими строками кода.

```
private void showPage(URL pageUrl, boolean addToList) {
  // Показать песочные часы на время работы червя.

  setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
```

В этот метод передается URL страницы, которую необходимо отобразить, и флаг, на основании которого делается вывод о добавлении страницы в список страниц. Затем для курсора устанавливается значение WAIT_CURSOR, говорящее о том, что приложение занято. В большинстве операционных систем значение WAIT_CURSOR соответствует отображению песочных часов. После установки курсора указанная страница отображается в панели редактора, как показано в листинге.

```
try {
  // Получить URL страницы для отображения.
  URL currentUrl = displayEditorPane.getPage();
  // Загрузить и отобразить заданную страницу.
  displayEditorPane.setPage(pageUrl);
  // Получить URL новой страницы для отображения.
  URL newUrl = displayEditorPane.getPage();
```

Перед загрузкой новой страницы в панель редактора, URL страницы, которая в это время отображается, сохраняется. Текущая страница записывается, чтобы она могла быть использована позже в этом методе, если она еще не добавлена в список страниц. Затем загружается новая страница. После загрузки новой страницы, ее URL также заносится в список страниц для использования в методе.

В следующих нескольких строках кода производится проверка на предмет того, что страница, передаваемая в метод showPage(), должна быть добавлена в список страниц для последующего использования.

```
// Если список определен, то добавить страницу.
if (addToList) {
  int listSize = pageList.size();
  if (listSize > 0) {
    int pageIndex = pageList.indexOf(currentUrl.toString());
  if (pageIndex < listSize -1) {
    for (int i = listSize -1; i > pageIndex; i--) {
      pageList.remove(i);
    }
  }
}
pageList.add(newUrl.toString());
}
```

Если установлен флаг addToList, то отображаемая страница добавляется в список страниц. Сначала извлекается размер списка страниц. Если в списке есть хоть одна страница, извлекается индекс последней отображаемой страницы. Если индекс меньше чем размер списка, тогда все страницы в списке, расположенные после последней отображаемой страницы, удаляются. Это происходит потому, что впереди не должно быть страниц для отображения.

Затем обновляется графический интерфейс пользователя, как показано ниже.

```
// Записать в текстовое поле URL текущей страницы.
locationTextField.setText(newUrl.toString());
// Обновить кнопки в соответствии с отображаемой страницей.
updateButtons();
```

Сначала обновляется текстовое поле для отображения URL страницы, которая будет отображаться в браузере. Затем вызывается метод `updateButtons()` для получения активного или пассивного состояния кнопок **Back** и **Forward** в зависимости от положения записи о странице в списке страниц.

Если при загрузке новой страницы возникнет исключительная ситуация, выполняется блок `catch`, как показано ниже.

```
}
catch (Exception e)
{
  // Отобразить сообщение об ошибке.
  showError("Unable to load page"); ;
}
```

Если возникает исключительная ситуация, то вызывается метод `showError()` для отображения диалогового окна с сообщением об ошибке.

В методе `showPage()` используются расширенные возможности блока `try...catch` с ключевым словом `finally`. Конструкция `finally` используется для того, чтобы в любом случае возвратить курсор в его состояние по умолчанию.

```
finally
{
  // Возвратить курсор по умолчанию.
  setCursor(Cursor.getDefaultCursor());
}
```

Метод updateButtons()

Метод `updateButtons()`, листинг которого приведен ниже, обновляет состояние кнопок **Back** и **Forward** в панели кнопок на основе положения записи о текущей отображаемой странице в списке страниц. Этот метод вызывается в методе `showPage()` каждый раз, когда производится отображение страницы.

```
/* Обновить состояние кнопок Back и Forward в
   соответствии с отображаемой страницей. */
private void updateButtons() {
  if (pageList.size() < 2) {
    backButton.setEnabled(false);
    forwardButton.setEnabled(false);
  }
  else {
    URL currentUrl = displayEditorPane.getPage();
    int pageIndex = pageList.indexOf(currentUrl.toString());
    backButton.setEnabled(pageIndex > 0);
    forwardButton.setEnabled(
      pageIndex < (pageList.size() - 1));
  }
}
```

Если в списке меньше, чем две страницы, обе кнопки **Back** и **Forward** будут пассивными и будут отображаться в приглушенном сером цвете. Это происходит пото-

му, что нет страниц, на которые можно совершить переход. Однако, если в списке по крайней мере две страницы, то состояние кнопок зависит от того, где расположена запись отображаемой страницы в списке страниц. Если значение переменной `pageIndex` больше чем нуль, то это означает, что в списке есть страницы перед отображаемой страницей, и кнопка **Back** будет активной. Если значение `pageIndex` меньше, чем общее количество страниц минус единица (счет индексов идет с 0), то кнопка **Forward** будет активной.

Метод hyperlinkUpdate()

Листинг метода `hyperlinkUpdate()` приведен ниже. Метод поддерживает контракт с интерфейсом `HyperlinkListener`, позволяя классу `MiniBrowser` принимать уведомления о событиях, связанных со ссылками, таких как щелчок на гиперссылке в тексте, отображаемом в панели редактора.

```
// Обработать щелчок на гиперссылке.
public void hyperlinkUpdate(HyperlinkEvent event) {
  HyperlinkEvent.EventType eventType = event.getEventType();
  if (eventType == HyperlinkEvent.EventType.ACTIVATED) {
    if (event instanceof HTMLFrameHyperlinkEvent) {
      HTMLFrameHyperlinkEvent linkEvent = (HTMLFrameHyperlinkEvent) event;
      HTMLDocument document =
        (HTML/Document) displayEditorPane.getDocument();
        document.processHTMLFrameHyperlinkEvent(linkEvent);
    }
    else {
      showPage(event.getURL(), true);
    }
  }
}
```

В метод `hyperlinkUpdate()` передается объект `HyperlinkEvent`, в котором собрана вся информация относительно события, связанного со ссылкой, которое произошло. Сначала проверяется тип события, которое должно иметь значение `ACTIVATED`. Это говорит о том, что был произведен щелчок на ссылке. Затем событие проверяется на предмет того, является ли оно экземпляром класса `ofHTML FrameHyperlinkEvent`, т.е. произведен ли щелчок в кадре HTML. Если это так, то событие приводится к типу `HTMLFrameHyperlinkEvent` и сохраняется в переменной `linkEvent`. Затем вызывается метод `displayEditorPane.getDocument()` для получения экземпляра документа, связанного в отображаемой страницей. В этом случае документ будет иметь тип `HTMLDocument`. Наконец, вызывается метод `processHTMLFrameHyperlinkEvent()`, в который передается переменная `link Event` как аргумент. В результате всех этих операций можно обрабатывать ссылки внутри кадра HTML.

Если событие отличается от типа `HTMLFrameHyperlinkEvent`, то ссылка обрабатывается как стандартная и возвращается адрес с помощью вызова метода `getURL()`. Затем в панели редактора отображается страница, связанная с этой ссылкой, с помощью вызова метода `showPage()`.

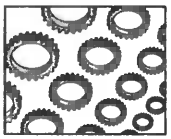
Компиляция и запуск мини-Web-браузера

Для компиляции программы `MiniBrowser` выполните следующую команду:

```
javac MiniBrowser.Java
```

Для запуска `MiniBrowser` введите следующую строку.

```
javaw MiniBrowser
```



Необходимо компилировать и запускать программу `MiniBrowser` с версией JDK, большей, чем 1.4.0. Например, JDK 1.4.2 работает великолепно. Компиляция с JDK 1.4.0 приведет к тому, что многие Web-сайты не будут работать с программой `MiniBrowser`.

Работа с мини-Web-браузером подобна работе с полнофункциональным браузером. Сначала вводится URL страницы, которую необходимо просмотреть, а затем производится щелчок на кнопке **Go**. Это приводит к загрузке указанной страницы в браузер.

После посещения более чем одной страницы с помощью мини-Web-браузера, обратите снимание, что кнопки **Back** и **Forward** становятся активными, т.е. доступными для работы с ними. Каждая из кнопок будет становиться активной с зависимости от положения, которое занимает отображаемая страница в списке страниц. Для перехода к предыдущей странице, произведите щелчок на кнопке **Back**, а для перехода к последующей странице, щелкните на кнопке **Forward**. На рис. 7.2 показан мини-Web-браузер в работе.

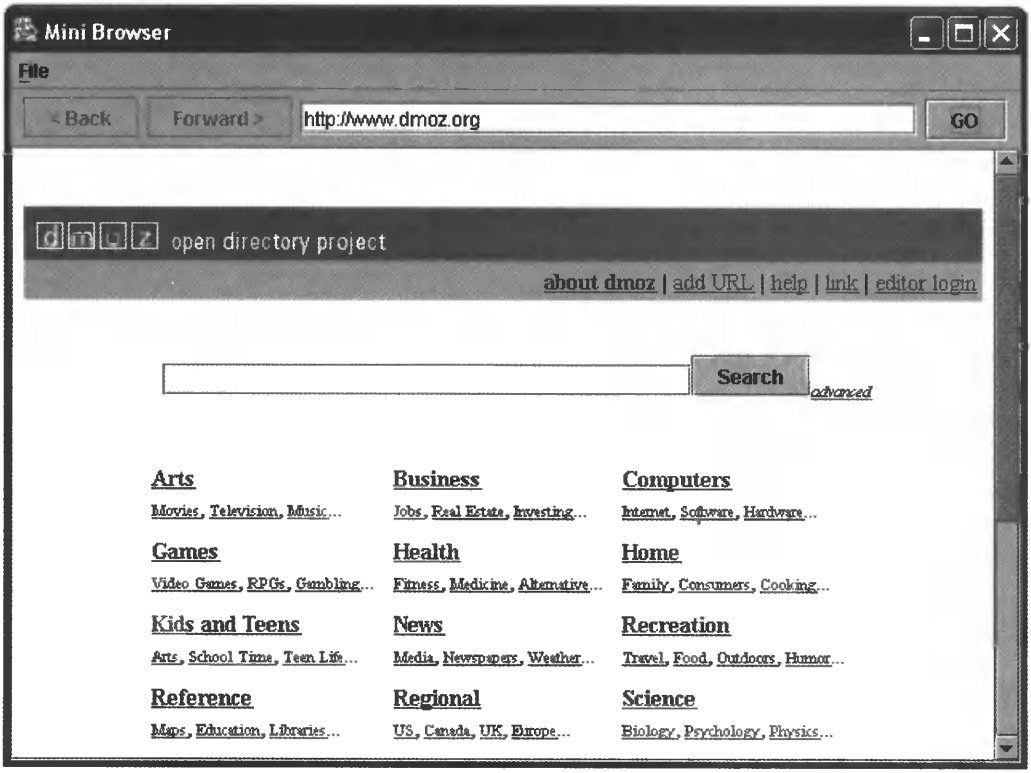


Рис. 7.2. Мини-Web-браузер в работе

Возможности формата HTML

Очень легко найти интересные возможности, которые можно получить с помощью класса `JEditorPane`, обрабатывающего формат HTML. Как уже упоминалось в начале главы, очень эффективно представлять справочную информацию в формате HTML. Пользовательская информация в электронном виде также является одним из кандидатов на использование формата HTML. Например, можно использовать формат HTML для отображения учебного пособия о том, как пользоваться приложением, которое вы разработали. Как справочную информацию, так и учебное пособие можно просматривать с помощью класса `JEditorPane`. Можно найти и другое применение. Например, постарайтесь использовать формат HTML для сообщений об ошибках. Такое сообщение будет содержать гиперссылки на уточняющие разделы.

This page is intentionally left blank

ГЛАВА

8

Статистика, графика и Java

Основное назначение Java — это создавать небольшие программы, такие как аплеты или сервлеты, которые выполняют определенные вычисления и отображают данные. Часто эти данные выражены в числовых значениях, таких как цены на товары или температура воздуха, расписание движения транспорта и т.д. При это необходимо получить и отобразить различные статистические величины, связанные с этими данными, или изобразить эти значения в виде графика. Например, аплет может отображать среднюю долю акционерного капитала за каждый месяц и отображать эти значения графически с помощью гистограммы. Поскольку и статистические вычисления, и вывод графиков в той или иной форме очень часто требуются при программировании на языке Java, то именно эти программы и будут предметом обсуждения настоящей главы.

В главе представлены методы для определения следующих статистических критериев.

- Среднее значение
- Медиана
- Мода
- Среднее отклонение
- Уравнение регрессии (наилучшее приближение)
- Коэффициент корреляции

В этой главе также показаны примеры разработанных программ для отображения графиков в виде гистограмм или отдельных значений. Приведенные примеры можно использовать как дополнения к разрабатываемым вами программам для решения определенных задач.

В главе рассматриваются два важных аспекта Java: математические возможности языка и средства работы с графическим интерфейсом. Надо отметить, что язык Java не оптимизирован для выполнения вычислительных задач, но встроена довольно значительная поддержка математических функций. Хотя в данной главе и нет сложных вычислений, но вполне можно получить иллюстрацию того, как легко Java обращается с числами.

Язык Java хорошо приспособлен для работы с графическим интерфейсом. В Java есть много классов для поддержки интерфейсов на основе окон. Как вам должно быть уже известно, Java поддерживает графический интерфейс двумя способами: с помощью пакета Abstract Window Toolkit (AWT), а также пакета Swing. Поскольку о возможностях Swing уже говорилось в предыдущих главах, в этой главе остановимся на пакете AWT. Вы увидите, что AWT можно эффективно использовать для создания отдельных окон и обрабатывать изменения размеров, перерисовку и многие другие события. Благодаря уникальным возможностям Java, таким как внутренние и адаптерные классы, разработанный код для графического интерфейса значительно нагляднее и меньше по размеру, чем аналогичный код, разработанный на любом другом языке

Отсчеты, генеральные совокупности, распределения и переменные

Перед тем как начать рассматривать определенные задачи, необходимо определить некоторые ключевые термины и основные задачи статистических вычислений. Обычно статистическую информацию связывают с набором специфических отсчетов, над которыми были проведены соответствующие преобразования. *Отсчеты* получаются в результате изучения какого-либо процесса, и все вместе они называются *генеральной совокупностью*. Например, вы можете измерять выходные изделия тарной фабрики в течение года только по средам. В этом случае все отсчеты будут состоять из измерений, выполненных по средам.

Если отсчеты являются достаточно полными, то они равны генеральной совокупности. В случае с тарной фабрикой отсчеты будут являться генеральной совокупностью только в том случае, если измерения будут производиться каждый день в течение года.

Когда же количество измерений меньше генеральной совокупности, то возможны неточные выводы. В настоящей главе мы предполагаем, что количество отсчетов равно генеральной совокупности и что серьезные ошибки исключены.

Статическая информация зависит от *распределения*, связанного с генеральной совокупностью. Возможны различные распределения, но в данной главе будет использоваться только одно из них — *нормальное распределение*, графическое изображение которого напоминает контур колокола. На такой кривой все измерения расположены симметрично относительно середины (или пика) кривой.

В любом статистическом процессе есть *зависимые переменные*, значения которых вычисляются, и *независимые переменные*, которые определяют зависимые переменные. В данной главе в качестве независимой переменной используется время. Это означает, что измерения представляют последовательность событий, связанных с отсчетами времени. Использование времени в качестве независимой переменной является общепринятой практикой. Например, для ведения портфеля заказов можно производить отсчеты каждый день.

Основы статистики

В основе большинства статистических вычислений лежат три величины: *среднее значение*, *медиана* и *мода*. Каждая из них полезна сама по себе, но все вместе они дают довольно полное представление о генеральной совокупности отсчетов.

Во всех статистических методах данной главы предполагается, что элементы, которые составляют отсчеты, сохраняются в массиве типа `double`. Все статические методы являются статическими методами, сохраняемыми в классе с именем `Stats`, который будет полностью приведен в данной главе. Поскольку методы являются статическими, они могут вызываться без создания экземпляра класса `Stats`.

Среднее значение

Среднее значение является наиболее общим статистическим критерием и представляет собой среднее арифметическое для всего набора значений. Таким образом, среднее значение является "центром тяжести" для данных. Для подсчета среднего

значения надо разделить сумму всех элементов генеральной совокупности на количество элементов. Например, сумма последовательности значений

```
1 2 3 4 5 6 7 8 9 10
```

равна 55. Разделим эту сумму на количество элементов, которое равно 10, и получим среднее значение, которое равно 5.5. Таким образом, общая формула для вычисления среднего значения будет иметь вид:

$$M = \frac{1}{N}\sum_{i=1}^{N} Di$$

где D_i представляет элемент данных, а N определяет количество элементов.

Ниже показан метод с именем mean(), в котором производится вычисление среднего значения элементов массива, для указания ссылки на который используется параметр. Метод возвращает среднее значение.

```
// Возвращает среднее значение ряда значений.
Return the average of a set of values,
public static double mean(double[] vals) {
  double avg =0.0;
  for(int i=0; i < vals.length; i++)
    avg += vals[i];
  avg /= vals.length;
  return avg;
}
```

Для обращения к методу mean() просто вызовите метод со ссылкой на массив, содержащий необходимые данные, и получите среднее значение.

Медиана

Медиана является средним значением последовательности возрастающих величин. Например, для последовательности

```
1 2 3 4 5 6 7 8 9
```

медиана будет иметь значение 5. Для последовательности с четным числом элементов медиана будет равна среднему значению от двух центральных значений. Например, если взять последовательность

```
1 2 3 4 5 6 7 8 9 10
```

то для такой последовательности медиана будет иметь значение 5.5. Для последовательностей, которые имеют нормальное распределение, медиана и среднее значение будут одинаковы. Однако, если распределение элементов все больше отклоняется от нормального распределения, то различие между медианой и средним значением будет возрастать. Простейший способ получить медиану для последовательности отсчетов — это отсортировать последовательность и затем взять значение среднего элемента. Этот способ используется в методе median(), листинг которого приведен ниже.

```
// Получение медианы для множества значений.
public static double median(double[] vals) {
  double tempi] = new double[vals.length];
  System.arraycopy(vals, 0, temp, 0, vals.length);
  Arrays.sort(temp);              // Сортировка данных.
  // Возвратить значение среднего элемента.
  if((vals.length)%2==0) {
```

```
    // Если количество элементов четное, вычислить среднее.
    return(temp[temp.length/2] + tempf(temp.length/2)-1])/2;
  }
  else return
  temp[temp.length/2];
}
```

Для получения медианы вызовите метод `median()`, в который передайте ссылку на массив, содержащий необходимые значения. Возвращаемое значение будет медианой для данного массива.

Обратите внимание, что копия массива получена с помощью метода `System.arraycopy()`. Сортируется именно копия, для того чтобы не вносить изменения в оригинальный массив данных. Оригинальный порядок необходим не только для того, чтобы отображать данные в виде графика, но и для проведения других статистических измерений.

Мода

Мода для некоторого множества значений — это наиболее часто встречающееся значение. Например, для выборки

```
1 2 3 3 4 5 6 7 7 7 8 9
```

модой будет значение 7, поскольку именно это значение встречается наибольшее количество раз. Но мода может и не иметь уникального значения. Например, для выборки

```
10 20 30 30 40 50 60 60 70
```

одинаковое количество раз повторяются значения 30 и 60. Каждое из них может считаться модой. Такие множества значений называются бимодальным. Множество, которое имеет только одну моду, называется унимодальным. Для задач, которые рассматриваются в этой книге, если одинаково повторяющихся значений несколько, то в качестве моды будем использовать только первое значение. Если нет ни одного значения, повторяющегося более чаще, чем другие, то такое множество элементов будем считать не имеющим моды.

В следующем примере метода `mode()` находится мода для множества значений.

```
/* Возврат моды для множества значений. Исключительная ситуация
   возникает в том случае, если ни одно из значений не повторяется
   более чаще, чем любое другое. Если два или более значений встречаются
   одинаковое количество раз, то возвращается первое значение */
public static double mode(double[] vals) throws NoModeException {
  double m, modeVal = 0.0;
  int count, oldcount = 0;
  for(int i=0; i < vals.length; i++) {
    m = vals[i];
    count = 0;
    // Подсчет количества вхождений каждого значения.
    for(int j=i+1; j < vals.length; j++)
    if(m == vals[j]) count++;
    /* Если данное значение встречается большее количество раз,
       чем предыдущие, сохранить его */
    if(count > oldcount) {
      modeVal = m;
      oldcount = count;
    }
  }
```

```
  if(oldcount == 0)
    throw new NoModeException();
  else
    return modeVal;
}
```

В методе `mode()` производится подсчет количества вхождений для каждого значения из массива `vals`. Если встречается значение, повторяющееся большее количество раз, чем предыдущие значения, то новое значение сохраняется в переменной `modeVal`. После просмотра всех значений переменная `modeVal` будет содержать значение моды, которое и возвращается из метода. Если есть значения, повторяющиеся одинаковое количество раз, то возвращается первое значение. Если не найдено значения, повторяющегося чаще остальных, то генерируется исключительная ситуация `NoModeException`. Класс `NoModeException` приведен ниже.

```
// Это класс исключения для метода mode().
class NoModeException extends Exception {
  public String toString() {
    return "Set contains no mode.";
  }
}
```

Дисперсия и среднее отклонение

Хотя статистические показатели, состоящие из одного числа, довольно удобны, но надо признать, что они не дают полную картину распределения значений и могут ввести в заблуждение. Например, если последовательность значений представлена отдельными группами, близкими по значению, то это никак не будет отображено при вычислении среднего значения или медианы. Рассмотрим следующий пример:

```
10, 11,9, 1, 0, 2, 3, 12, 11, 10
```

Среднее значение для этой последовательности будет равно `6.9`, но оно никак не выделяет эту последовательность из других, имеющих такое же среднее значение. Дело в том, что среднее значение ни несет никакой информации об относительном расположении и значениях отдельных элементов. Для более четкого представления о разбросе значений отдельных элементов, необходимо знать, как близко каждый элемент находится к среднему значению. Знание дисперсии (рассеивания) помогает лучше интерпретировать среднее значение, медиану и моду.

Для нахождения разброса элементов необходимо подсчитать среднее отклонение. Среднее отклонение — это отклонение от дисперсии. Оба значения говорят о разбросе отдельных значений. Из этих двух показателей, среднее отклонение является более информативным, поскольку оно позволяет представить средний разброс значений элементов последовательности и отклонения от среднего значения.

Дисперсию можно подсчитать по следующей формуле:

$$V = \frac{1}{N}\sum_{i=1}^{N}(Di - M)^2$$

Здесь N соответствует количеству элементов последовательности, M является средним значением и D_i представляет отдельное значение из последовательности. Разности между значением отдельного элемента и средним значением необходимо

возвести в квадрат для получения положительного значения. Если этого не сделать, то результат всегда будет равен нулю.

Среднее отклонение равняется квадратному корню из дисперсии. Таким образом, формула для стандартного отклонения будет такой:

$$std = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(Di - M)^2}$$

Как уже упоминалось, среднее отклонение обычно более информативно, чем дисперсия. Рассмотрим следующую последовательность:

```
11 20 40 30 99 30 50
```

Для подсчета дисперсии сначала подсчитывается среднее значение, которое равно `40`. Затем вычисляется разность расстояний каждого элемента от среднего значения, как показано в таблице 8.1 ниже:

Таблица 8.1. Разность расстояний элементов от среднего значения

D_I	$D_I - M$	$(D_I - M)^2$
11	-29	841
20	-20	400
40	0	0
30	-10	100
99	59	3481
30	-10	100
50	10	100
	Сумма:	5022
	Среднее:	717.43

Возведенные в квадратную степень разности суммируются. Сумма будет равна `5022`. Для получения среднего значения необходимо разделить это число на количество элементов, равное `7`. Получим число `717.43`. Среднее отклонение будет равно квадратному корню из среднего значения, это будет 26.78.

Для интерпретации среднего отклонения посмотрите, чему равна разность значения каждого элемента от среднего значения в этом примере. Среднее отклонение говорит о том, чему приблизительно равна разность между значением каждого элемента и средним значением. Например, если вы владелец кондитерской фабрики и начальник цеха докладывает вам, что за последний месяц каждый день в среднем выпускается 2500 коробок, но при этом известно, что среднее отклонение равно 2000, то желательно повнимательнее присмотреться к этому цеху.

Необходимо использовать следующее правило: предположите, что данные, которые вы обрабатываете, имеют нормальное распределение, и поэтому около 68% данных должны располагаться в пределах среднего отклонения и 95% — в пределах удвоенного среднего отклонения.

В приведенном ниже методе `stdDev()` производится подсчет среднего отклонения для массива значений.

```
// Возвращает среднее отклонение для массива значений.
public static double stdDev(double[] vals) {
  double std = 0.0;
  double avg = mean(vals);
  for(int i=0; i < vals.length; i++)
    std += (vals[i]-avg) * (vals[i]-avg);
  std /= vals.length;
  std = Math.sqrt(std);
  return std;
}
```

Уравнение регрессии

Обычно статистическая информация используется для предсказания "поведения" ряда значений в будущем. Но показанные выше статистические критерии мало подходят для предсказания будущих событий и для этих целей часто проводят анализ тенденции изменения. Возможно, наиболее широко используемым методом для анализа тенденции изменения является уравнение регрессии. Это уравнение описывает прямую линию, которая имеет наилучшее приближение к данным и на которую ссылаются как на линию регрессии.

Перед тем как описать процесс нахождения этой линии, напомним, что линию можно описать с помощью двумерного массива и представить ее соответствующим уравнением:

$$Y = a + bX$$

Здесь X является независимой переменной, а Y — зависимой переменной. Коэффициент a представляет отрезок, отсекаемый на оси Y от начала координат, а коэффициент b определяет наклон лини. Поэтому, чтобы найти уравнение линии, необходимо найти значения коэффициентов a и b.

Для нахождения уравнения регрессии необходимо использовать метод наименьших квадратов. Основная идея заключается в том, чтобы найти такую линию, которая минимизирует сумму квадратов отклонения от действительных данных и этой линии. Получить такое уравнение линии можно за два шага. Сначала подсчитывается коэффициент b по следующей формуле:

$$b = \frac{\sum_{i=1}^{N}(X_i - M_x)(Y_i - M_y)}{\sum_{i=1}^{N}(X_i - M_x)^2}$$

Здесь M_x является средним значением для координаты X, а M_y определяет среднее значение координаты Y. Зная значение b, легко подсчитать значение a по следующей формуле:

$$a = M_y - bM_x$$

В полученное уравнение регрессии можно подставить любое значение X и получить прогнозируемое значение Y для этой точки.

Для облегчения понимания общего смысла и назначения линии регрессии, рассмотрим небольшой пример. Предположим, что рассматривается доля акционерного капитала для корпорации XYZ за прошедшие годы. Полученные данные представлены в таблице 8.2 ниже.

Таблица 8.2. Доля акционерного капитала за прошедшие годы

Год	*Доля*	*Год*	*Доля*
0	68	5	85
1	75	6	82
2	74	7	87
3	80	8	91
4	81	9	94

Уравнение регрессии для этих данных будет следующим:

$$Y = 70.22 + 2.55X$$

Все данные и линия регрессии показаны на рис. 8.1. Как видно из графика, линия регрессии имеет небольшой наклон и имеет тенденцию в возрастанию. Это говорит об увеличении доли капитала. Обратите внимание, что линия регрессии наиболее приближена к данным. Используя этот график, любой может предсказать, что на одиннадцатый год доля капитала может возрасти до 98.27 (это можно легко определить из уравнения регрессии). Конечно такое предсказание является только предсказанием и нет никакой гарантии, что предсказание сбудется.

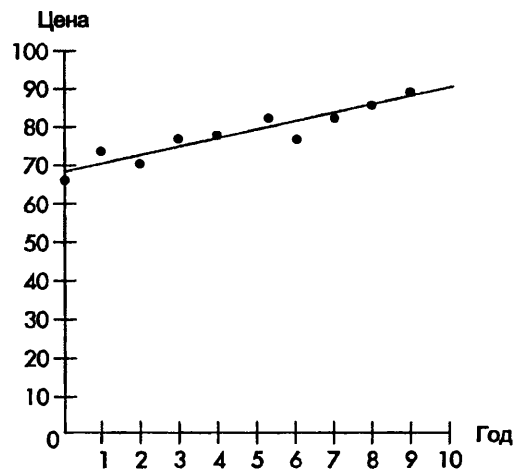


Рис. 8.1. График доли капитала и линия регрессии

Коэффициент корреляции

Хотя линия регрессии на рис. 8.1 показывает тенденцию к возрастанию, нет полной уверенности относительно того, насколько эта линия приближена к реальным данным. Если линия и данные имеют слабую корреляцию, то такая линия не пред-

ставляет особого интереса. Если линия регрессии достаточно приближена к данным, то на ее основе можно делать значимые выводы. Наиболее общим способом определения корреляции данных и линии регрессии является подсчет коэффициента корреляции, который может быть выражен числом от –1 до 1. Коэффициент корреляции отражает размер среднего отклонения относительно линии регрессии. Это может показаться несколько запутанным, но это так. Коэффициент корреляции зависит от разности расстояний каждого значения каждого элемента данных от линии регрессии. Если коэффициент корреляции равен 1, то данные полностью совпадают с линией. Если коэффициент корреляции равен 0, то это означает, что нет никакого совпадения между данными и линией регрессии (в этом случае может подойти любая линия). Знак коэффициента корреляции должен соответствовать знаку для коэффициента b. Если коэффициент корреляции положительный, то это означает, что существует непосредственное соответствие между независимой и зависимой переменными. Если же знак отрицательный, то существует инверсное соответствие.

Формула для подсчета коэффициента корреляции выглядит так:

$$cor = \frac{\frac{1}{N}\sum_{i=1}^{N}(X_i - M_x)(Y_i - M_y)}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - M_x)}\sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - M_y)}}$$

где M_x является средним значением для X, а M_y определяет среднее значение для Y. Знак соответствует знаку наклона линии регрессии. Обычно если абсолютное значение равно 0.81 или больше, то считается, что существует сильная корреляция. Это означает, что около 66% данных приближены к линии корреляции. Для преобразования коэффициента корреляции в процентное соотношение просто определите из него квадратный корень. Такое значение называется коэффициентом смешанной корреляции.

Метод `regress()`, листинг которого приведен ниже, подсчитывает коэффициенты уравнения регрессии и коэффициент корреляции.

```
/* Подсчет коэффициентов уравнения регрессии и коэффициента корреляции
   для массива значений. Значения представлены по координате Y. По
   координате X размещается время с возрастанием на 1. */
public static RegData regress(double[] vals) {
  double a, b, yAvg, xAvg, temp, temp2, cor;
  double vals2[] = new double[vals.length];
  // Создать числовой формат из 2 десятичных чисел.
  NumberFormat nf = NumberFormat.getInstance();
  nf.setMaximumFractionDigits(2);
  // Найти среднее значение для оси Y.
  yAvg = mean(vals);
  // Найти среднее значение для оси X.
  xAvg = 0.0;
  for(int i=0; i < vals.length; i++)
    xAvg + = i;
  xAvg /= vals.length;
  // Найти коэффициент b.
  temp = temp2 = 0.0;
  for(int i=0; i < vals.length; i++) {
    temp + = (vals[i]-yAvg) * (i-xAvg);
    temp2 + = (i-xAvg) * (i-xAvg);
  }
```

```
  b = temp/temp2;
  // Найти коэффициент a.
  a = yAvg - (b*xAvg);
  // Подсчитать коэффициент корреляции.
  for(int i = 0; i < vals.length; i++)
   vals2[i] = i + 1;
  cor = temp/vals.length;
  cor /= stdDev(vals) * stdDev(vals2);
  return new RegData(a, b, cor, "Y = " +
      nf.format(a) +  " + " + nf.format(b) + " * X");
  }
}
```

Очень важно обратить внимание на то, что в методе `regress()` предполагается, что независимой переменной является время и что изменения времени происходят с интервалом, равным единице. Среднее значение для оси X производится с учетом этого предположения.

```
// Найти среднее значение по оси X.
xAvg =0.0;
for(int i=0; i < vals.length; i++)
  xAvg += i;
xAvg /= vals.length;
```

Здесь суммируются значения от 0 до числа, выражающего количество элементов, и затем сумма делится на количество элементов. Таким образом, получается среднее значение для оси X.

Поскольку в качестве независимого переменного по оси X используется время, то часто говорят, что метод `regress()` выполняет анализ временных рядов. Поэтому становится понятным, почему используется только один массив значений. Метод `regress()` легко модифицировать для обработки двух массивов, когда один массив представляет значения по оси Y, а другой представляет значения по оси X, но для решения задач, поставленных в этой главе, в этом нет необходимости.

Метод `regress()` возвращает значения для коэффициентов `a` и `b`, строковое представление уравнения регрессии и коэффициент корреляции, заключенных в объекте `RegData`. Объект `RegData` показан ниже.

```
// Этот класс содержит данные регрессионного анализа.
class RegData {
  public double a, b;
  public double cor;
  public String equation;
  public RegData(double i, double j, double k, String str) {
    a = i;
    b = j;
    cor = k;
    equation = str;
  }
}
```

Полный листинг класса Stats

Все статистические методы могут быть собраны в одном классе, который можно назвать `Stats`, что и показано ниже. В этом же файле удобно расположить классы `RegData` и `NoModeException`.

```
import java.util.*;
import java.text.*;

// Этот класс содержит данные регрессионного анализа.
class RegData {
  public double a, b;
  public double cor;
  public String equation;

  public RegData(double i, double j, double k, String str) {
    a = i;
    b = j;
    cor = k;
    equation = str;
  }
}

// Это класс исключения для метода Mode().
class NoModeException extends Exception {
  public String toString() {
    return "Set contains no mode.";
  }
}

// Класс общего назначения для статистических вычислений.
public class Stats {
  // Возврат среднего значения для массива значений.
  public static double mean(double[] vals) {
    double avg = 0.0;
    for(int i=0; i < vals.length; i++)
      avg += vals[i];
    avg /= vals.length;
    return avg;
  }

  // Возврат медианы для массива значений.
  public static double median(double[] vals) {
    double temp[] = new double[vals.length];
    System.arraycopy(vals, 0, temp, 0, vals.length);
    Arrays.sort(temp); // sort the data
    // Возврат значения среднего элемента.
    if((vals.length)%2==0) {
      // Если число элементов четное, подсчет среднего.
      return (temp[temp.length/2] +
              temp[(temp.length/2)-1]) /2;
    } else return
      temp[temp.length/2];
  }

  /* Возврат моды для массива значений.
     Генерируется исключение NoModeException если на одно
     значение не повторяется чаще, чем другие значения.
     Если одно или несколько значений повторяются одинаковое
     количество раз, то возвращается первое значение. */
  public static double mode(double[] vals)
      throws NoModeException
  {
    double m, modeVal = 0.0;
    int count, oldcount = 0;
    for(int i=0; i < vals.length; i++) {
      m = vals[i];
```

```
      count = 0;

      // Подсчет числа вхождений для каждого значения.
      for(int j=i+1; j < vals.length; j++)
        if(m == vals[j]) count++;

      /* Если это значение повторяется большее количество раз, чем
         все предыдущие, сохранить его. */
      if(count > oldcount) {
        modeVal = m;
        oldcount = count;
      }
    }
    if(oldcount == 0)
      throw new NoModeException();
    else
      return modeVal;
  }

  // Возвратить среднее значение для массива значений.
  public static double stdDev(double[] vals) {
    double std = 0.0;
    double avg = mean(vals);
    for(int i=0; i < vals.length; i++)
      std += (vals[i]-avg) * (vals[i]-avg);
    std /= vals.length;
    std = Math.sqrt(std);
    return std;
  }

  /* Подсчитать уравнение регрессии и коэффициент корреляции
     для массива значений. Значения представлены по оси Y.
     По оси X размещается время с возрастанием на 1. */
  public static RegData regress(double[] vals) {
    double a, b, yAvg, xAvg, temp, temp2, cor;
    double vals2[] = new double[vals.length];

    // Создать числовой формат для 2 десятичных чисел.
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMaximumFractionDigits(2);

    // Найти среднее значение для оси Y.
    yAvg = mean(vals);

    // Найти среднее значение для оси X.
    xAvg = 0.0;
    for(int i=0; i < vals.length; i++)
      xAvg += i;
    xAvg /= vals.length;

    // Найти коэффициент b.
    temp = temp2 = 0.0;
    for(int i=0; i < vals.length; i++) {
      temp += (vals[i]-yAvg) * (i-xAvg);
      temp2 += (i-xAvg) * (i-xAvg);
    }
    b = temp/temp2;

    // Найти коэффициент a.
    a = yAvg - (b*xAvg);
```

```
    // Подсчитать коэффициент корреляции.
    for(int i=0; i < vals.length; i++) vals2[i] = i+1;
    cor = temp/vals.length;
    cor /= stdDev(vals) * stdDev(vals2);
    return new RegData(a, b, cor, "Y = " +
                       nf.format(a) +  " + " +
                       nf.format(b) + " * X");
  }
}
```

Графики данных

Хотя статистические критерии полезны сами по себе, они не всегда дают полное представление о распределении данных. В большинстве случаев гораздо большую пользу приносит отображение данных в виде некоторого графика.

Представление данных в визуальном виде позволяет одновременно отображать как уравнение корреляции, так и данные, при этом легко увидеть аномалии, о которых ничего нельзя сказать, имея только статистические критерии. На графике также хорошо видны изменения в расположении данных. Поскольку графики очень важны для статистического анализа, ниже будут представлены три графических метода.

В дополнение к отображению данных в удобной форме, в нашем примере графические методы предоставляют дополнительные возможности: они иллюстрируют некоторые технические особенности пакета AWT и обработки событий. Как уже должно быть известно, пакет AWT является частью библиотеки классов ядра Java. Он обеспечивает поддержку для ориентированных на окна графических приложений и его можно назвать графическим интерфейсом пользователя (Graphical User Interface — GUI). Графические приложения взаимодействуют с пользователем посредством обработки событий, среди которых нажатия клавиш клавиатуры, выбор пункта меню, изменение размеров окна и т.д. В процессе разработки графических методов вы будете сталкиваться и с некоторыми второстепенными вопросами, связанными с окружением GUI. Например, график должен динамически изменять масштаб в момент изменения размеров окна, содержащего его.

В этой главе разработано три типа графиков. Первый представляет гистограмму, второй отображает отдельные значения и третий, помимо отдельных значений, представляет линию регрессии. Как можно видеть, большая часть кода, такого как масштабирование изображения на экране, является общей для всех трех графиков.

Масштабирование данных

При отображении графика нельзя произвольно устанавливать размеры осей, следует учитывать размеры экрана. Также необходимо производить масштабирование в соответствии с размерами окна, в котором график отображается. Более того, масштаб данных должен динамически устанавливаться каждый раз при перерисовке окна, так как при этом пользователь может изменить размеры окна.

Процесс масштабирования данных связан с необходимостью нахождения соответствующего соотношения между диапазоном данных и физическими размерами окна. После нахождения этого соотношения данные можно размещать с учетом коэффициента соотношения, и все данные для координат будут находиться в пределах окна. Например, формула для масштабирования по координате Y будет такой:

$$Y' = Y * (\text{ширина}) / (\max - \min)$$

где `Y'` представляет масштабированное значение, которое указывает положение в окне.

Хотя эта формула и очень простая, но при использовании окружения графического интерфейса возникают сложности. Например, необходимо вычислять ширину окна каждый раз, когда происходит перерисовка окна, так как эта перерисовка может быть связана с изменением размеров окна. Более того, необходимо учитывать ширину рамки окна, которую необходимо вычитать из общих размеров окна. Также должна учитываться ширина и высота цифр, используемых для отображения числовых значений. Таким образом, процесс масштабирования данных требует учета всех нюансов и этапов.

Класс Graphs

Все графические методы собраны в классе `Graphs`. Класс `Graphs` расширяет возможности класса `Frame`. Таким образом, графики содержатся в окне верхнего уровня, и это позволяет изменять размеры и в определенной степени оставаться независимыми от приложения, в котором они используются. Например, можно отобразить график и затем минимизировать окно с графиком без минимизации остальной части приложения.

Листинг класса `Graphs` приведен ниже. В следующих разделах каждая часть класса объясняется подробно.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;

// Графический класс общего назначения.
public class Graphs extends Frame {
  // Константы для типов графиков.
  public final static int BAR = 0;
  public final static int SCATTER = 1;
  public final static int REGPLOT = 2;
  private int graphStyle;

  /* Здесь устанавливается размер пространства, которое
     должно оставаться между данными и рамкой окна. */
  private final int leftGap = 2;
  private final int topGap = 2;
  private final int bottomGap = 2;
  private int rightGap; // Это значение рассчитывается.

  /* В этих переменных сохраняются максимальное и
     минимальное значения данных. */
  private double min, max;

  // Ссылка на данные.
  private double[] data;

  // Цвета, используемые графиком.
  Color gridColor = new Color(0, 150, 150);
  Color dataColor = new Color(0, 0, 0);

  /* Различные переменные, используемые для масштабирования
     и отображения данных. */
```

```
private int hGap;      // Расстояние между данными.
private int spread;    // Расстояние между значениями min и max.
private double scale; // Коэффициент масштабирования.
private int baseline; // Вертикальная координата базовой линии.

// Размещение пространства данных в окне.
private int top, bottom, left, right;

public Graphs(double[] vals, int style) {
  // Обработка закрытия окна.
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
      setVisible(false);
      dispose();
    }
  });

  // Обработка изменения размеров.
  addComponentListener(new ComponentAdapter() {
    public void componentResized(ComponentEvent ce) {
      repaint();
    }
  });
  graphStyle = style;
  data = vals;

  // Сортировка данных для нахождения значений min и max.
  double t[] = new double[vals.length];
  System.arraycopy(vals, 0, t, 0, vals.length);
  Arrays.sort(t);
  min = t[0];
  max = t[t.length-1];
  setSize(new Dimension(200, 120));
  switch(graphStyle) {
    case BAR:
      setTitle("Bar Graph");
      setLocation(25, 250);
      break;
    case SCATTER:
      setTitle("Scatter Graph");
      setLocation(250, 250);
      break;
    case REGPLOT:
      setTitle("Regression Plot");
      setLocation(475, 250);
      break;
  }
  setVisible(true);
}
public void paint(Graphics g) {
  Dimension winSize = getSize(); // Размер окна.
  Insets ins = getInsets();      // Размер рамки.

  // Получить размер текущего символа.
  FontMetrics fm = g.getFontMetrics();

  // Подсчитать правый интервал.
  rightGap = fm.stringWidth("" + data.length);

  // Подсчитать размер области данных.
  left = ins.left + leftGap + fm.charWidth('0');
```

```
top = ins.top + topGap + fm.getAscent();
bottom = ins.bottom + bottomGap + fm.getAscent();
right = ins.right + rightGap;

/* Если минимальное значение положительное, то использовать 0
   как начальную точку графика.
   Если максимальное значение отрицательное, используйте 0. */
if(min > 0) min = 0;
if(max < 0) max = 0;

/* Подсчитать расстояние между максимальным и
   минимальным значениями. */
spread = (int) (max - min);

// Подсчитать коэффициент масштабирования.
scale = (double) (winSize.height - bottom - top) / spread;

// Найти расположение базовой линии.
baseline = (int) (winSize.height - bottom + min * scale);

// Подсчитать расстояние между данными.
hGap = (winSize.width - left - right) / (data.length-1);

// Установить цвет сетки.
g.setColor(gridColor);

// Нарисовать базовую линию.
g.drawLine(left, baseline,
           left + (data.length-1) * hGap, baseline);

// Нарисовать ось Y.
if(graphStyle != BAR)
  g.drawLine(left, winSize.height-bottom, left, top);

// Отобразить значения min, max и 0.
g.drawString("0", ins.left, baseline+fm.getAscent()/2);
if(max != 0)
  g.drawString("" + max, ins.left, baseline -
               (int) (max*scale) - 4);
if(min != 0)
  g.drawString("" + min, ins.left, baseline -
               (int) (min*scale)+fm.getAscent());

// Отобразить номера значений.
g.drawString("" + data.length,
             (data.length-1) * (hGap) + left,
             baseline + fm.getAscent());

// Установить цвет данных.
g.setColor(dataColor);

// Отобразить данные.
switch(graphStyle) {
  case BAR:
    bargraph(g);
    break;
  case SCATTER:
    scatter(g);
    break;
  case REGPLOT:
    regplot(g);
```

```
        break;
      }
    }

    // Отобразить гистограмму.
    private void bargraph(Graphics g) {
      int v;
      for(int i=0; i < data.length; i++) {
        v = (int) (data[i] * scale);
        g.drawLine(i*hGap+left, baseline,
                   i*hGap+left, baseline - v);
      }
    }

    // Отобразить график в виде точек.
    private void scatter(Graphics g) {
      int v;
      for(int i=0; i < data.length; i++) {
        v = (int) (data[i] * scale);
        g.drawRect(i*hGap+left, baseline - v, 1, 1);
      }
    }

    // Отобразить точки графика и линию регрессии.
    private void regplot(Graphics g) {
      int v;
      RegData rd = Stats.regress(data);
      for(int i=0; i < data.length; i++) {
        v = (int) (data[i] * scale);
        g.drawRect(i*hGap+left, baseline - v, 1, 1);
      }

      // Нарисовать линию регрессии.
      g.drawLine(left, baseline - (int) ((rd.a)*scale),
                 hGap*(data.length-1)+left+1,
                 baseline - (int) ((rd.a+(rd.b*(data.length-1)))*scale));
    }
}
```

Графические константы и переменные

Класс `Graphs` начинается с объявления следующих переменных.

```
// Константы для типов данных.
public final static int BAR = 0;
public final static int SCATTER = 1;
public final static int REGPLOT = 2;
private int graphStyle;
```

Первые три переменные, объявленные как `final static`, называются `BAR`, `SCATTER` и `REGPLOT`. Они используются для указания вида графика, который необходимо отобразить. Вид графика сохраняется в переменной `graphStyle`.

Затем объявляются переменные, которые определяют промежуток между рамкой окна и пространством, в котором отображается график.

```
/* Эти переменные определяют промежуток между
   данными и рамкой окна. */
private final int leftGap = 2;
private final int topGap = 2;
```

```
private final int bottomGap = 2;
private int rightGap; // Это значение рассчитывается.
```

Три из них объявлены как неизменяемые (final), а значение переменной rightGap будет зависеть от текущей ширины символа и количества отображаемых данных. Затем объявляются следующие переменные.

```
// Переменные для хранения минимального и максимального значений.
private double min, max;
// Ссылка на данные.
private double[] data;
```

Переменные min и max содержат минимальное и максимальное значения данных. На массив, в котором хранятся исходные данные, производится обращение с помощью переменной data.

Цвета, используемые при отображении графика, сохраняются в переменных gridColor и dataColor.

```
// Переменные для цветов, используемых при отображении графика.
Color gridColor = new Color(0, 150, 150);
Color dataColor = new Color(0, 0, 0);
```

Цвет сетки будет отображаться светло-зеленым цветом, а данные будут изображаться черным цветом, хотя это легко можно изменить и задать те цвета, которые вам больше нравятся.

После этого идут объявления переменных, которые имеют отношение к масштабированию данных.

```
// Переменные, используемые при масштабировании и отображении данных.
private int hGap;     // Расстояние между данными.
private int spread;   // Расстояние между значениями min и max.
private double scale; // Коэффициент масштабирования.
private int baseline; // Вертикальная координата базовой линии.
```

Расстояние между точками вдоль оси X сохраняется в переменной hGap. Число, являющееся разностью между максимальным и минимальным значениями, сохраняется в переменной spread. Коэффициент масштабирования хранится в переменной scale. Вертикальное положение базовой линии (т.е. оси X) содержится в переменой baseline. И в конце объявляются переменные top, bottom, left и right.

```
// Размещение пространства данных в окне.
private int top, bottom, left, right;
```

Эти переменные определяют положение пространства для отображения графика относительно включающего его окна.

Конструктор класса Graphs

Конструктор класса Graphs имеет два параметра. Первый параметр используется для задания ссылки на данные, которые должны отображаться. С помощью второго параметра задается значение, которое определяет вид графика. Этими значениями могут быть Graph.BAR, Graph.SCATTER или Graph.REGPLOT.

Конструктор Graphs() начинается с добавления слушателя событий, возникающих при закрытии окна.

```
// Обработка закрытия окна.
addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent we) {
```

```
    setVisible(false);
    dispose();
  }
});
```

Слушатель добавляется с помощью вызова метода `addWindowListener()`, в который передается объект `WindowAdapter`, переопределяющий обработчик события `windowClosing()`. Вспомните, что класс адаптеров предусматривает пустые реализации всех методов для соответствующих интерфейсов слушателей событий. В этом специфическом случае объект `WindowAdapter` реализует методы интерфейса `WindowListener`. Поскольку для всех методов интерфейса `WindowListener` в объекте `WindowAdapter` сделаны пустые реализации, то необходимо только переопределить те из них, которые нас интересуют. В нашем случае это `windowClosing()`.

При закрытии графическое окно удаляется с экрана с помощью вызова метода `setVisible(false)` и затем удаляется из памяти, для чего вызывается метод `dispose()`. Эти методы имеются в классе `Frame` из пакета AWT.

Затем добавляется слушатель для обработки изменения размеров окна. Для этого вызывается метод `addComponentListener()`, в который передается объект `ComponentAdapter`. В этом объекте переопределяется обработчик события `componentResized()`, как показано в следующем фрагменте.

```
// Обработка изменения размеров окна.
addComponentListener(new ComponentAdapter() {
  public void componentResized(ComponentEvent ce) {
    repaint();
  }
});
```

Когда изменяются размеры окна, метод `componetResized()` вызывает метод `repaint()`, в котором в свою очередь вызывается метод `paint()`. Как далее можно будет видеть, в методе `paint()` производится масштабирование с учетом размеров текущего окна. Таким образом, когда изменяются размеры окна, метод `paint()` просто вновь перерисовывает график, используя новые размерности.

Затем переменной `graphStyle` присваивается вид графика, а переменной `data` присваивается ссылка на массив данных. Затем создается временная копия данных и производится сортировка данных. Из отсортированных данных определяются максимальное и минимальное значения. Строки кода, в которых производятся все эти действия, показаны ниже.

```
graphStyle = style;
data = vals;
// Сортировать данные для получения значений min и max.
double t[] = new double[vals.length];
System.arraycopy(vals, 0, t, 0, vals.length);
Arrays.sort(t);
min = t [0];
max = t[t.length-1];
```

Конструктор `Graphs` заканчивается следующими строками кода.

```
setSize(new Dimension(200, 120));
switch(graphStyle) {
  case BAR:
    setTitle("Bar Graph");
    setLocation(25, 250);
    break;
```

```
  case SCATTER:
    setTitle("Scatter Graph");
    setLocation(250, 250);
    break;
  case REGPLOT:
    setTitle("Regression Plot");
    setLocation(475, 250);
    break;
}
setVisible(true);
```

Начальному размеру окна для графика задаются значения 200 и 120 пикселей с помощью вызова метода `setSize()`. Затем, используя переменную `graphStyle`, указывается заголовок окна, для чего используется метод `setTitle()`, и задается положение окна с помощью вызова метода `setLocation()`. Наконец, производится отображение окна на экране, для чего вызывается метод `setVisible(true)`. При этом вызывается метод `paint()`, который реально прорисовывает окно.

Метод paint()

Большую часть работы по отображению графика на экране берет на себя метод `paint()`. Он выполняет следующие основные задачи:

- определяет размер окна и размеры рамки;
- получает размер выбранного шрифта;
- рассчитывает размер области отображения графика, которая должна быть меньше размера окна на размер рамки и некоторой зоны допуска;
- подсчитывает коэффициент масштабирования;
- подсчитывает координату Y базовой линии, которая является осью X;
- определяет размер между отсчетами данных;
- отображает базовую линию и ось Y;
- отображает максимальные и минимальные значения по осям X и Y;
- вызывает подходящий метод для прорисовки графика.

Комментарии, приведенные в методе `paint()`, поясняют многие действия и позволяют понять те места программы, которые не являются очевидными. Однако, ввиду важности этого метода, рассмотрим все операции строка за строкой. Метод `paint()` начинается со следующих объявлений.

```
Dimension winSize = getSize(); // Размер окна.
Insets ins = getInsets();      // Размер рамки.
```

Визуальное окно класса `Frame` состоит из двух частей: рамка, включающая панель названия и панель меню (дополнительно), и пространство для отображения данных. Размеры окна можно получить с помощью вызова метода `getSize()`. При этом возвращаются внешние размеры окна в виде объекта `Dimension`. Ссылка на этот объект сохраняется в переменной `winSize`. Объект `Dimension` включает два поля: `width` и `height`. Таким образом, внешние размеры окна можно получить, обратившись к переменным `winSize.width` и `winSize.height`.

В целях определения размеров пространства для отображения данных, необходимо уменьшить внешние размеры окна, вычитая из них размеры рамки. Для того чтобы это сделать, необходимо вызвать метод `getInsets()`. Этот метод возвращает размеры вложенных объектов типа `Insets`, которые содержат поля `left`, `right`, `top` и `bottom`. Ссылка на этот объект сохраняется в переменной `ins`. Вспомните, что координаты отсчитываются от верхнего левого угла, в котором они имеют значение 0,0. Таким образом, положение верхнего левого угла определяется переменными `ins.left`, `ins.top`, а положение нижнего правого угла будет определяться значениями `winSize.width-right` и `winSize.height-bottom`.

Затем с помощью метода `paint()` получаем метрику текущего шрифта, как показано ниже.

```
// Получить размеры шрифта.
FontMetrics fm = g.getFontMetrics();
```

Информация, сохраняемая в переменной `fm`, будет использоваться при подсчете ширины и высоты символов, используемых для отображения диапазона данных.

Хотя можно использовать все пространство для отображения данных, но по эстетическим соображениям не будем заполнять его полностью. Лучше оставить небольшой зазор между данными и рамкой. Чтобы это сделать, необходимо произвести уменьшение пространства данных на небольшие величины `leftGap`, `topGap`, `bottomGap` и `rightGap`. Первые три из них содержат значение, равное 2, а значение для переменной `rightGap` необходимо рассчитать исходя из ширины строки, которая содержит номера элементов, как показано ниже.

```
// Подсчитать зазор справа.
rightGap = fm.stringWidth("" + data.length);
```

Поскольку строка, в которой отображаются номера элементов, заходит за правый край базовой линии, пространство для отображения данных должно быть уменьшено на размер этой строки. Этим и объясняется необходимость использовать метрику шрифтов, которую можно получить с помощью метода `getFontMetrics()`. Используя метод `stringWidth()`, можно получить ширину строки, которая добавляется к значению переменной `rightGap`.

Затем рассчитываются зазоры для каждой стороны на основе полученных значений и результирующие значения присваиваются переменным `left`, `top`, `right` и `bottom`, как показано ниже.

```
// Рассчитать размеры пространства для отображения данных.
left = ins.left + leftGap + fm.charWidth('0');
top = ins.top + topGap + fm.getAscent();
bottom = ins.bottom + bottomGap + fm.getAscent();
right = ins.right + rightGap;
```

Обратите внимание на пространство слева для отображения диапазона данных. В следующих строках производится подсчет коэффициента масштабирования.

```
/* Если минимальное значение положительное, то в
   качестве начальной точки графика использовать 0.
   Если максимальное значение отрицательное,
   использовать 0. */
if(min > 0) min = 0 ;
if(max < 0) max = 0;
/* Рассчитать разность между максимальным
   и минимальным значениями. */
```

```
spread = (int) (max - min);
// Подсчитать коэффициент масштабирования.
scale = (double) (winSize.height - bottom - top) / spread;
```

Подсчет коэффициента масштабирования начинается с нормализации значений для переменных `min` и `max`. Все графики должны начинаться с координат 0, 0. Поэтому, если минимальное значение больше нуля, то оно устанавливается в 0. Если максимальное значение меньше 0, то максимальное значение устанавливается в 0. Затем подсчитывается разность между переменными `min` и `max`. Это значение используется для получения коэффициента масштабирования, который сохраняется в переменной `scale`.

После подсчета коэффициента масштабирования можно определить положение базовой линии, масштабируя значение переменной `min`, как показано ниже.

```
// Найти положение базовой линии.
baseline = (int) (winSize.height - bottom + min * scale);
```

Если переменная `min` равняется нулю, базовая линия размещается у нижней грани окна. В противном случае она может размещаться в середине поля, а если все значения отрицательные, то в верхней части окна.

Расстояние между данными определяется с помощью деления длины, занимаемой данными, на количество данных, как показано ниже.

```
// Подсчитать расстояние между данными.
hGap = (winSize.width - left - right) / (data.length-1);
```

Затем устанавливается цвет для объекта `gridColor` и отображаются диапазоны по осям X и Y. Это выполняется с помощью следующего кода.

```
// Установить цвет сетки.
g.setColor(gridColor);
// Нарисовать базовую линию.
g.drawLine(left, baseline,
  left + (data.length-1) * hGap, baseline);
// Нарисовать ось Y.
if(graphStyle != BAR)
g.drawLine(left, winSize.height-bottom, left, top);
// Отобразить значения min, max и 0.
g.drawstring("0", ins.left, baseline+fm.getAscent()/2);
if(max != 0)
  g.drawstring("" + max, ins.left, baseline -
    (int) (max*scale) - 4);
if(min != 0)
  g.drawstring("" + min, ins.left, baseline -
    (int) (min*scale)+fm.getAscent());
// Отобразить количество значений.
g.drawstring("" + data.length,
  (data.length-1) * (hGap) + left,
  baseline + fin.getAscent());
```

Обратите внимание, что для гистограммы ось Y не отображается. Также отметьте, что максимальное и минимальное значения отображаются, если только они не равны нулю. И обратите внимание, как высота символов влияет на точность расположения, когда отображаются диапазоны данных.

Наконец, показанный ниже код устанавливает цвета для объекта `dataColor` и вызывает необходимый метод для реальной прорисовки графика.

```
// Установить цвет отображаемых данных.
g.setColor(dataColor);
```

```
// Display the data,
switch(graphStyle) {
  case BAR:
    bargraph(g);
    break;
  case SCATTER:
    scatter(g);
    break;
  case REGPLOT:
    regplot(g) ;
    break;
```

Метод bargraph()

Метод `bargraph()` масштабирует каждый элемент массива данных, на который производится ссылка с помощью переменной `data`, и затем отображает линии, длина которых пропорциональна значениям данных. Эти линии рисуются от базовой линии. Листинг метода `bargraph()` приведен ниже.

```
// Отобразить гистограмму.
private void bargraph(Graphics g) {
  int v;
  for(int i=0; i < data.length; i++) {
    v = (int) (data[i] * scale);
    g.drawLine(i*hGap+left, baseline,
    i*hGap-t-left, baseline - v);
  }
}
```

Поскольку большая часть работы уже сделана в методе `paint()`, то в методе `bargraph()` остается только масштабировать каждый элемент с помощью коэффициента масштабирования и нарисовать линии. Линии начинаются от базовой линии, которой является ось X. Рассчитывается конец линии, для чего производится вычитание масштабированного значения из положения базовой линии. Вспомните, что верхний левый угол имеет координаты 0, 0. Поэтому наименьше значение для оси Y размещается выше в окне, чем наибольшее значение. Следовательно, переменная `v` должна вычитаться из положения базовой линии. Зазор между отсчетами задается с помощью переменной `hGap`. Положение каждого отсчета на оси X определяется путем умножения индекса элемента на величину зазора и добавления смещения.

Метод scatter()

Метод `scatter()` выполняет такие же задачи, что и метод `bargraph()`, за исключением того, что отображаются точки вместо линий, как показано в следующем фрагменте.

```
// Отобразить точечный график.
private void scatter(Graphics g) {
  int v;
  for(int i=0; i < data.length; i++) {
    v = (int) (data[i] * scale);
    g.drawRect(i*hGap+left, baseline - v, 1, 1);
  }
}
```

Метод scatter() масштабирует каждый элемент массива, на который ссылается переменная data, и затем отображает точки, значения которых по оси Y пропорциональны данным.

Метод regplot()

Подобно методу scatter(), метод regplot(), листинг которого приведен ниже, отображает график в виде точек. Различие состоит в том, что в этом методе дополнительно отображается линия регрессии, для чего вызывается метод regress().

```
// Отобразить точечный график совместно с линией регрессии.
private void regplot(Graphics g) {
  int v;
  RegData rd = Stats.regress(data);
  for(int i=0; i < data.length; i++) {
    v = (int) (data[i] * scale);
    g.drawRect(i*hGap+left, baseline - v, 1, 1);
  }
  // Нарисовать линию регрессии.
  g.drawLine(left, baseline - (int) ((rd.a)*scale),
      hGap*(data.length-1)+left+1,
      baseline - (int) ((rd.a+(rd.b*(data.length-1)))*scale));
}
```

Обратите внимание, как рисуется линия регрессии. При обращении к методу drawLine() конечная точка линии рассчитывается на основе значений rd.a и rd.b, которые соответствуют точке пересечения линией оси Y (a) и наклону линии (b) в уравнении регрессии.

Приложение для статистического анализа

Используя классы Stats и Graphs, можно создать простое, но достаточно эффективное приложение. Основное окно приложения создается с помощью класса StatsWin, листинг которого приведен ниже.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.*;

// Обработка и отображение статистических данных.
public class StatsWin extends Frame
    implements ItemListener, ActionListener
{
  NumberFormat nf = NumberFormat.getInstance();
  TextArea statsTA;
  Checkbox bar = new Checkbox("Bar Graph");
  Checkbox scatter = new Checkbox("Scatter Graph");
  Checkbox regplot = new Checkbox("Regression Line Plot");
  Checkbox datawin = new Checkbox("Show Data");
  double[] data;
  Graphs bg;
  Graphs sg;
  Graphs rp;
  DataWin da;
  RegData rd;
```

```
public StatsWin(double vals[]) {
  data = vals; // Сохранить ссылку на данные.
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
      shutdown();
    }
  });

  // Создать меню file.
  createMenu();

  // Изменить расположение, разместить по центру.
  setLayout(new FlowLayout(FlowLayout.CENTER));
  setSize(new Dimension(300, 240));
  setTitle("Statistical Data");
  rd = Stats.regress(data);

  // Установить цифровой формат с двумя десятичными цифрами.
  nf.setMaximumFractionDigits(2);

  // Создать выходную строку.
  String mstr;
  try {
    // Получить режим, если он установлен.
    mstr = nf.format(Stats.mode(data));
  } catch(NoModeException exc) {
    mstr = exc.toString();
  }
  String str = "Mean: " +
               nf.format(Stats.mean(data)) + "\n" +
               "Median: " +
               nf.format(Stats.median(data)) + "\n" +
               "Mode: " + mstr + "\n" +
               "Standard Deviation: " +
               nf.format(Stats.stdDev(data)) + "\n\n" +
               "Regression equation: " + rd.equation +
               "\nCorrelation coefficient: " +
               nf.format(rd.cor);

  // Разместить выход в текстовом пространстве.
  statsTA = new TextArea(str, 6, 38, TextArea.SCROLLBARS_NONE);
  statsTA.setEditable(false);

  // Добавить компоненты в окно.
  add(statsTA);
  add(bar);
  add(scatter);
  add(regplot);
  add(datawin);

  // Добавить слушателей компонентов.
  bar.addItemListener(this);
  scatter.addItemListener(this);
  regplot.addItemListener(this);
  datawin.addItemListener(this);
  setVisible(true);
}

// Обработать пункт меню "Close".
public void actionPerformed(ActionEvent ae) {
  String arg = (String)ae.getActionCommand();
```

```
    if(arg == "Close") {
      shutdown();
    }
  }

  // Пользователь изменил состояние флажка.
  public void itemStateChanged(ItemEvent ie) {
    if(bar.getState()) {
      if(bg == null) {
        bg = new Graphs(data, Graphs.BAR);
        bg.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent we) {
            bar.setState(false);
            bg = null;
          }
        });
      }
    }
    else {
      if(bg != null) {
        bg.dispose();
        bg = null;
      }
    }
    if(scatter.getState()) {
      if(sg == null) {
        sg = new Graphs(data, Graphs.SCATTER);
        sg.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent we) {
            scatter.setState(false);
            sg = null;
          }
        });
      }
    }
    else {
      if(sg != null) {
        sg.dispose();
        sg = null;
      }
    }
    if(regplot.getState()) {
      if(rp == null) {
        rp = new Graphs(data, Graphs.REGPLOT);
        rp.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent we) {
            regplot.setState(false);
            rp = null;
          }
        });
      }
    }
    else {
      if(rp != null) {
        rp.dispose();
        rp = null;
      }
    }
    if(datawin.getState()) {
      if(da == null) {
        da = new DataWin(data);
```

```
        da.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent we) {
            datawin.setState(false);
            da = null;
          }
        });
      }
    }
    else {
      if(da != null) {
        da.dispose();
        da = null;
      }
    }
  }

  // Создать пункты меню для работы с файлами.
  private void createMenu()
  {
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);
    Menu file = new Menu("File");
    MenuItem close = new MenuItem("Close");
    file.add(close);
    mbar.add(file);
    close.addActionListener(this);
  }

  // Закрыть окно.
  private void shutdown() {
    if(bg != null) bg.dispose();
    if(sg != null) sg.dispose();
    if(rp != null) rp.dispose();
    if(da != null) da.dispose();
    setVisible(false);
    dispose();
  }
}
```

Класс `StatsWin` наследует возможности класса `Frame` для получения всех возможностей окна верхнего уровня, в котором отображается статистическая информация. В него также включены переключатели, с помощью которых можно устанавливать режимы отображения данных в различных графических форматах. В классе `StatsWin` также реализованы интерфейсы `ItemListener` и `ActionListener`.

Класс `StatsWin` начинается с создания объекта типа `NumberFormat`. С помощью класса `NumberFormat` можно форматировать различные цифровые данные. В классе `StatsWin` он используется для задания количества десятичных цифр, которые будут отображаться в окне со статистическими данными.

Далее в классе `StatsWin` объявляются несколько переменных, которые содержат ссылки на различные элементы управления графического интерфейса, используемые в классе. Это текстовое пространство, четыре флажка и три объекта `Graphs`. Ссылка на объект `DataWin` сохраняется в переменной `da.DataWin`. В классе `DataWin` представляется окно, в котором отображаются цифровые данные, о чем будет сказано позже. Ссылка на данные, которые необходимо анализировать, сохраняется в переменной `data`, а ссылка на данные для регрессии сохраняется в переменной `rd`.

Далее будут последовательно рассматриваться методы класса `StatsWin`.

Конструктор класса StatsWin

В метод `StatsWin()` должна передаваться ссылка на данные, которые необходимо анализировать. Затем создается объект, который производит статистический анализ данных. Большинство кода в теле конструктора не представляет трудностей для понимания, поэтому только кратко остановимся на всех выполняемых операциях.

Метод `StatsWin()` начинается с сохранения ссылки на данные. Затем добавляется слушатель окна, который обрабатывает события по закрытию окна. Когда это происходит, то вызывается метод `shutdown()`, который удаляет все окна, открытые объектом типа `StatsWin`.

Затем в методе `StatsWin()` создается меню для обработки файлов с помощью вызова метода `CreateMenu()`. В этом меню создается только один пункт: **Close**. Когда производится щелчок на этом пункте, то приложение заканчивает свою работу.

После этого изменяется расположение элементов управления, для чего применяется менеджер расположения `FlowLayout`. Это необходимо сделать, т.к. по умолчанию для окна класса `Frame` используется менеджер `BorderLayout`.

Далее изменяются размеры, записывается заголовок окна и извлекаются данные для регрессии. Затем устанавливается числовой формат для отображения данных, при котором определяются только два места для десятичный цифр. Для этого задается следующая строка кода.

```
nf.setMaximumFractionDigits(2);
```

Как уже говорилось ранее, переменная `nf` ссылается на объект `NumberFormat`. Это объект используется для создания формата отображаемых данных. Метод `setMaximumFractionDigits()` определяет максимальное число цифр, которые будут отображаться после десятичной точки. В классе `StatsWin` переменная `nf` используется для задания формата для всех числовых данных. Если нужно получить больше цифр после десятичной точки, то просто измените значение параметра в методе `setMaximumFractionDigits()`.

В следующем фрагменте кода создается строка, которая содержит статистические критерии для данных.

```
// Создать выходной объект.
String mstr;
try {
  // Получить режим, если он задан.
  mstr = nf.format(Stats.mode(data));
}
catch(NoModeException exc) {
  mstr = exc.toString();
}
String str = "Mean: " +
    nf.format(Stats.mean(data)) + "\n" + "Median: " +
    nf.format(Stats.median(data)) + "\n" +
    "Mode: " + mstr + "\n" + "Standard Deviation: " -t-
    nf.format(Stats.stdDevfdata)) + "\n\n" +
    "Regression equation: " + rd.equation +
    "\nCorrelation coefficient: " + nf.format(rd.cor);
```

Обратите внимание на создание строки для режима. Вспомните, что метод `mode()` генерирует исключение, когда режим не задан. Таким образом, строковая переменная `mstr` будет содержать или строку с названием необходимого типа графика, или сообщение, свидетельствующее о том, что тип графика не задан.

После того как выходная строка str будет полностью сформирована, она помещается в текстовое пространство TextArea объекта, на который ссылается переменная statsTA. Затем для этого объекта устанавливается режим только для чтения с помощью вызова метода setEditable(false). Поэтому отображаемые в текстовом пространстве данные нельзя будет редактировать, что в нашем случае и должно быть.

Затем в окно добавляются различные элементы управления с соответствующими слушателями. Но поскольку текстовое пространство имеет режим только для чтения, ему не требуется слушатель. Наконец, окно отображается на экране.

Обработчик события itemStateChanged()

Большинство действий класса StatsWin происходят в методе itemStateChanged(). Этот метод обрабатывает изменения для четырех флажков. Каждый раз, когда пользователь устанавливает флажок, отображается окно, связанное с этим флажком. После того как пользователь сбрасывает флажок, окно закрывается. Для того чтобы лучше понять, как это происходит, рассмотрим последовательность кодов (фрагмент приведен ниже), с помощью которых производится обработка изменений во флажках.

```
if(bar.getState()) {
  if(bg == null) {
    bg = new Graphs(data, Graphs.BAR);
    bg.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
         bar.setState(false);
         bg = null;
      }
    )};
  }
  else {
    if(bg != null) {
      bg.dispose();
      bg = null;
  }
}
```

Состояние переменной bar, которая содержит ссылку на флажок, определяется с помощью вызова метода getState(). Если возвращаемое значение равно true, то флажок установлен, в противном случае он сброшен. Если флажок установлен и переменная bg равна null, то это означает, что никаких графических окон в настоящее время не отображается. Если переменной bg присваивается ссылка на объект Graphs, то отображается графическое окно. В противном случае, если значение bg не равно null, то окно с гистограммой отображается и никаких других действий не предпринимается.

Если создается новое графическое окно, то добавляется слушатель для мониторинга окна. Слушатель обрабатывает события по закрытию, создаваемые графическим окном. Это означает, что объект типа StatsWin будет принимать уведомления, когда графическое окно закрывается. Когда принимается уведомление о закрытии, флажок сбрасывается и переменная bg принимает значение null.

Если переменная bar сбрасывается при изменении и если состояние переменной bg не равно null, то окно, содержащее гистограмму, удаляется с помощью вызова метода dispose() и переменная bg устанавливается в состояние null. Такой механизм используется при работе с любым из четырех флажков.

Метод actionPerformed()

Метод actionPerformed() обрабатывает щелчок на пункте меню Close из меню File. Он просто вызывает метод shutdown() для прекращения работы программы.

Метод shutdown()

Когда окно StatsWin закрывается, вызывается метод shutdown(). Он удаляет все окна, открытые в объекте StatsWin, включая основное окно и все графические окна или окна данных. Поэтому даже графические окна, отображаемые в окне верхнего уровня, удаляются в момент прекращения работы основного приложения.

Метод createMenu()

Метод createMenu() создает меню File. Он начинается с создания объекта типа MenuBar с именем mbar. Затем в нем создается объект типа Menu с именем file, в который добавляется объект MenuItem с именем close. Затем объект file добавляется в объект mbar. Наконец, добавляется объект StatsWin как слушатель событий для меню. Таким образом, когда пользователь производит щелчок на пункте меню Close, то создается событие, которое обрабатывается методом actionPerformed(), описанным ранее.

Класс DataWin

Класс StatsWin использует объект типа DataWin для отображения ряда цифровых данных, которые анализируются. Листинг класса DataWin приведен ниже.

```
import java.awt.event.*;
import java.awt.*;

// Отобразить массив цифровых данных.
class DataWin extends Frame {
  TextArea dataTA;
  DataWin(double[] data) {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        setVisible(false);
        dispose();
      }
    });
    dataTA = new TextArea(10, 10);
    dataTA.setEditable(false);
    for(int i=0; i < data.length; i++)
      dataTA.append(data[i]+"\n");
    setSize(new Dimension(100, 140));
    setLocation(320, 100);
```

```
    setTitle("Data");
    setResizable(false);
    add(dataTA);
    setVisible(true);
  }
}
```

Класс `DataWin` расширяет возможности класса `Frame` и является окном верхнего уровня. В конструктор класса `DataWin` передается ссылка на массив данных, подлежащих отображению. В нем создается объект `TextArea` для отображения данных. Текст, отображаемый в текстовом пространстве, не может изменяться пользователем, и нельзя изменять его размеры. Однако окно можно минимизировать.

Классы Stats и Graphs

В следующей программе продемонстрировано использование классов `Stats` и `Graphs`.

```
// Демонстрация использования классов Stats и Graphs.
import java.io.*;
import java.awt.*;

class DemoStat {
  public static void main(String args[])
    throws IOException
  {
    double nums[] = { 10, 10, 11, 9, 8, 8, 9,
                      10, 10, 13, 11, 11, 11,
                      11, 12, 13, 14, 16, 17,
                      15, 15, 16, 14, 16 };
    new StatsWin(nums);
  }
}
```

Для компиляции программы используйте следующую командную строку.

```
javac DemoStat.Java DataWin.Java StatsWin.Java Stats.Java Graphs.Java
```

Для запуска программы наберите следующую строку.

```
javaw DemoStat
```

Обратите внимание, что программа `javaw` (в отличие от `Java`) используется для запуска приложения, при этом исключается требование работы в окне консоли. Использование `javaw` приводит к корректному закрытию приложения при закрытии основного окна. Для Java 2 версии 1.4 и более поздних версий можно использовать и программу `Java`, но для более ранних версий используйте только `javaw`. На рис. 8.2–8.4 показаны окна, создаваемые классами в работе.

Одна из полезных функций, которая делает это приложение привлекательным, — это возможность изменять размеры окон, содержащих графики. Размеры окон можно изменять динамически и при этом будут автоматически выравниваться размеры графиков. Также окно можно минимизировать. Это позволяет убирать их с экрана, но не удалять из программы.

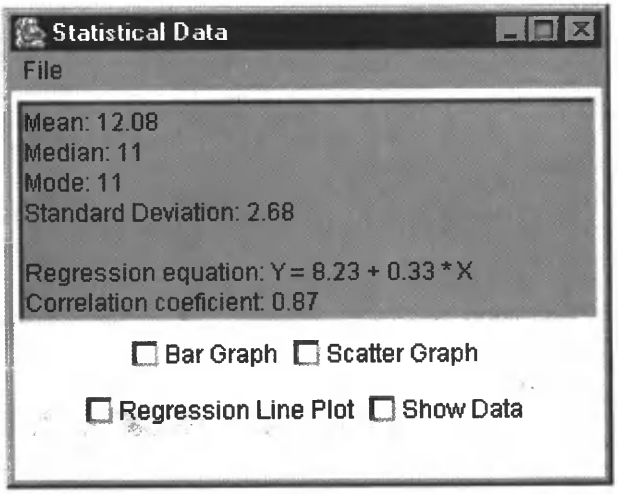


Рис. 8.2. Основное окно, создаваемое классом StatsWin

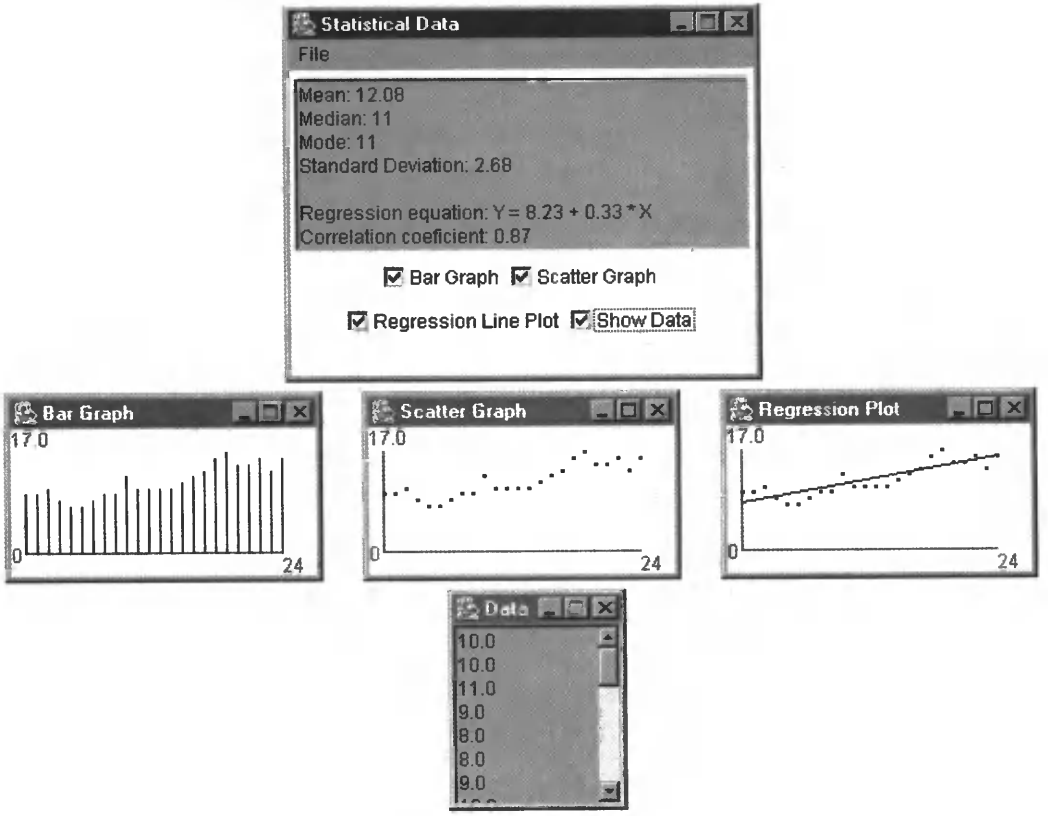


Рис. 8.3. Графические окна

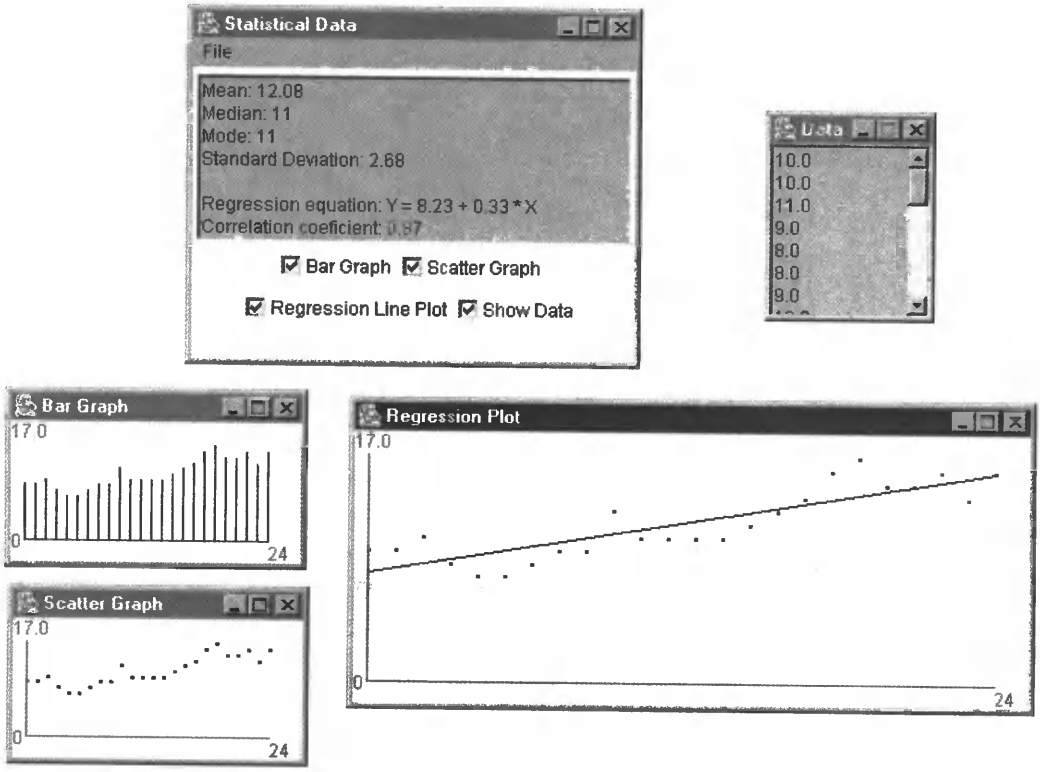


Рис. 8.4. Изменение размеров окон

Создание простого аплета для статистического анализа

В предыдущем разделе было создано отдельное приложение с использованием классов `Stats` и `Graphs`, но использование этих классов не ограничивается только подобными приложениями. Их можно легко использовать как в аплетах, так и в сервлетах. Для этого рассмотрим следующий простой аплет. В нем используются классы `Stats` и `Graphs` для отображения статистической информации о переданных данных.

```
// Демонстрация аплета с использованием классов Stats and Graphs.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
/*
  <applet code="StatApplet" width=120 height=50>
  <param name=data value="1.2, 3.6, 5.7, 4.4, 7.1, 4.4,
                          6.89, 8.9, 10.3, 9.45">
  </applet>
*/
public class StatApplet extends Applet implements ActionListener {
  StatsWin sw;
  Button show;
```

```
  ArrayList al = new ArrayList();
  public void init() {
    StringTokenizer st = new
      StringTokenizer(getParameter("data"), ", \n\r");
    String v;

    // Получить значения из HTML.
    while(st.hasMoreTokens()) {
      v = st.nextToken();
      al.add(v);
    }
    show = new Button("Display Statistics");
    add(show);
    show.addActionListener(this);
  }
  public void actionPerformed(ActionEvent ae) {
    if(sw == null) {
      double nums[] = new double[al.size()];
      try {
        for(int i=0; i<al.size(); i++)
          nums[i] = Double.parseDouble((String)al.get(i));
      } catch(NumberFormatException exc) {
        System.out.println("Error reading data.");
        return;
      }
      sw = new StatsWin(nums);
      show.setEnabled(false);
      sw.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent we) {
          sw = null;
          show.setEnabled(true);
        }
      });
    }
  }
}
```

Обратите внимание, что данные передаются в объект `StatApplet` через HTML-параметр с именем `data`. Эта строка содержит список значений, разделяемых точкой с запятой. Аплет `StatApplet` использует объект `StringTokenizer` для извлечения каждого отдельного значения из строки. Как только значение считано, оно добавляется в список `ArrayList`, на который ссылается переменная `al`. Список `ArrayList` представляет класс коллекций, который поддерживает динамические массивы, размеры которых можно изменять по мере необходимости.

Когда пользователь щелкает на кнопке Display Statistics, выполняется метод `actionPerformed()`. Поскольку для объекта `StatsWin` требуется массив типа `double`, а не объект `ArrayList`, то строка для переменной `al` должна быть конвертирована в тип `double` и скопирована в массив. После выполнения этих действий создается объект `StatsWin` и отображаются статистические данные.

Пример работы аплета показан на рис. 8.5. Подобные аплеты будут хорошим дополнением на любом Web-узле.

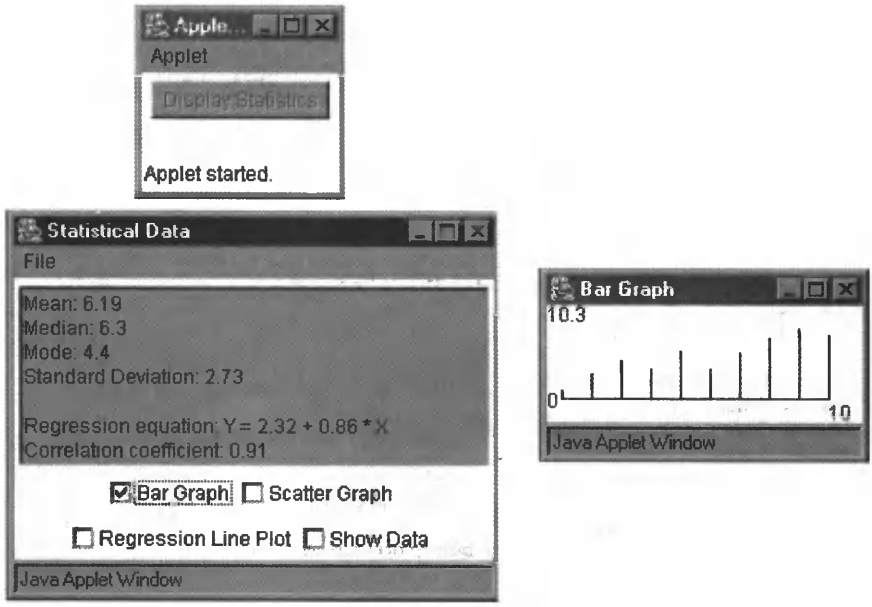


Рис. 8.5. Пример работы аплета `StatApplet`

Что еще можно сделать

Хочется предложить несколько вполне реальных идей. Как уже говорилось, графические методы и метод `regress()` могут работать только с одним набором данных, значения которого отображаются по оси Y. По оси X отображается время. Можно создать аналогичные методы, которые принимают в качестве аргументов два массива данных, причем значения второго массива будут отображаться по оси X. Возможно, вам будет интересно позволить пользователям производить вращение осей графика, что можно делать даже в реальном времени. Может быть, вы захотите изменять ширину отдельных элементов гистограммы или изменять вид точек графика.

Наконец, вы можете провести небольшую экспериментальную работу и сделать встроенные графические окна в классе `StatsWin` вместо отдельных окон. Если вы это сделаете, то сможете использовать флажки, задающие вид графика для каждого окна.

ГЛАВА

9

Финансовые аплеты и сервлеты

Несмотря на то, что разработаны сложные и многофункциональные, рассчитанные на широкое использование приложения, такие как текстовые процессоры, все же наибольшее распространение имеют базы данных и пакеты для финансовых вычислений, среди которых еще можно найти небольшие, но удобные в использовании и довольно популярные приложения. С их помощью выполняются различные финансовые вычисления, например, платеж по займу, предполагаемая выгода от капиталовложений или остаток от ссуды. Никакое из этих приложений не является очень сложным и не требует много кода, даже если в них включена справочная информация.

Как уже говорилось выше, язык Java изначально был ориентирован на создание небольших, переносимых программ. В оригинале эти программы разрабатываются в виде аплетов, хотя позднее были добавлены сервлеты. (Аплеты запускаются на локальном компьютере, а сервлеты выполняются на сервере.) Поскольку размеры программ для финансовых вычислений невелики, большинство финансовых калькуляторов общего назначения подходят для разработки как в виде аплетов, так и в виде сервлетов. К тому же, включая финансовый аплет (сервлет) в Web-страничку, можно рассчитывать на большее внимание и благодарность пользователей, которые чаще будут заглядывать на такие странички.

В этой главе разработаны аплеты, которые выполняют следующие финансовые вычисления.

- Регулярные уплаты по ссуде
- Остаточный баланс по ссуде
- Прибыль с инвестиций
- Начальные капиталовложения для получения расчетной прибыли
- Ежегодный доход с инвестиций
- Начальные капиталовложения, необходимые для получения расчетного ежегодного дохода

Эти аплеты можно использовать "как есть" или доработать для учета специфических требований. В конце главы будет показано, как преобразовать аплет в сервлет.

Подсчет выплат по ссуде

Возможно одним из самых популярных финансовых калькуляторов является такой калькулятор, который производит подсчет регулярных выплат по ссуде. Выплаты по ссуде можно найти с помощью следующей формулы:

$$\text{Payment} = (\text{intRate} * (\text{principal} / \text{payPerYear})) / (1 - ((\text{intRate} / \text{payPerYear}) + 1)^{-\text{payPerYear} * \text{numYears}})$$

где переменная `intRate` определяет ставку процента, переменная `principal` содержит начальный баланс, переменная `payPerYear` задает количество платежей за год и переменная `numYears` определяет количество лет, на которые выдана ссуда.

Аплет, листинг которого приведен ниже, называется `RegPay`, и в нем производятся расчеты на основе вышеприведенной формулы. Обратите внимание, что класс

RegPay расширяет возможности класса Applet и реализует интерфейс ActionListener.

```
// Простой аплет для расчета выплат по ссуде.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.text.*;
/*
  <applet code="RegPay" width=280 height=200>
  </applet>
*/

public class RegPay extends Applet
  implements ActionListener {

  TextField amountText, paymentText, periodText,
            rateText;
  Button doIt;

  double principal; // Начальный баланс.
  double intRate;   // Ставка процента.
  double numYears;  // Количество лет.

  /* Количество платежей за год. Это значение
     можно разрешить вводить пользователю. */
  final int payPerYear = 12;
  NumberFormat nf;
  public void init() {
    // Используем расположение GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);
    Label heading = new
          Label("Compute Monthly Loan Payments");
    Label amountLab = new Label("Principal");
    Label periodLab = new Label("Years");
    Label rateLab = new Label("Interest Rate");
    Label paymentLab = new Label("Monthly Payments");
    amountText = new TextField(16);
    periodText = new TextField(16);
    paymentText = new TextField(16);
    rateText = new TextField(16);

    // Поле Payment только для чтения.
    paymentText.setEditable(false);
    doIt = new Button("Compute");

    // Задать размеры сетки.
    gbc.weighty = 1.0; // Использовать коэфициент 1.
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.anchor = GridBagConstraints.NORTH;
    gbag.setConstraints(heading, gbc);

    // Привязать большинство компонентов к правой стороне.
    gbc.anchor = GridBagConstraints.EAST;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(amountLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(amountText, gbc);
    gbc.gridwidth = GridBagConstraints.RELATIVE;
```

```
    gbag.setConstraints(periodLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(periodText, gbc);
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(rateLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(rateText, gbc);
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(paymentLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(paymentText, gbc);
    gbc.anchor = GridBagConstraints.CENTER;
    gbag.setConstraints(doIt, gbc);

    // Добавить все компоненты.
    add(heading);
    add(amountLab);
    add(amountText);
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(paymentLab);
    add(paymentText);
    add(doIt);

    // Зарегистрировать для приема уведомлений о событиях.
    amountText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    doIt.addActionListener(this);
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажимает клавишу Enter при активном текстовом
     поле или производит щелчок на кнопке Compute. */
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // Отобразить результат, если все поля заполнены.
  public void paint(Graphics g) {
    double result = 0.0;
    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    try {
      if(amountStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0) {
        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        intRate = Double.parseDouble(rateStr) / 100;
        result = compute();
        paymentText.setText(nf.format(result));
      }
      showStatus(""); // Удалить предыдущие сообщения об ошибках.
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
```

```
      paymentText.setText("");
    }
  }

  // Подсчитать размер платежа по ссуде.
  double compute() {
    double numer;
    double denom;
    double b, e;
    numer = intRate * principal / payPerYear;
    e = -(payPerYear * numYears);
    b = (intRate / payPerYear) + 1.0;
    denom = 1.0 - Math.pow(b, e);
    return numer / denom;
  }
}
```

Внешний вид аплета, отображаемого при выполнении программы, показан на рис. 9.1. Для использования аплета просто введите начальный баланс ссуды, срок, на который выдана ссуда и ставку процента. Предполагается, что выплаты будут производиться ежемесячно. После вводе информации щелкните на кнопке Compute для расчета ежемесячной выплаты.

В следующих разделах будет подробно рассмотрен код класса `RegPay`. Поскольку все аплеты в этой главе используют одну и ту же базовую оболочку, большая часть объяснений, приведенная в этом разделе, будет применима и к остальным аплетам, описанным в этой главе.

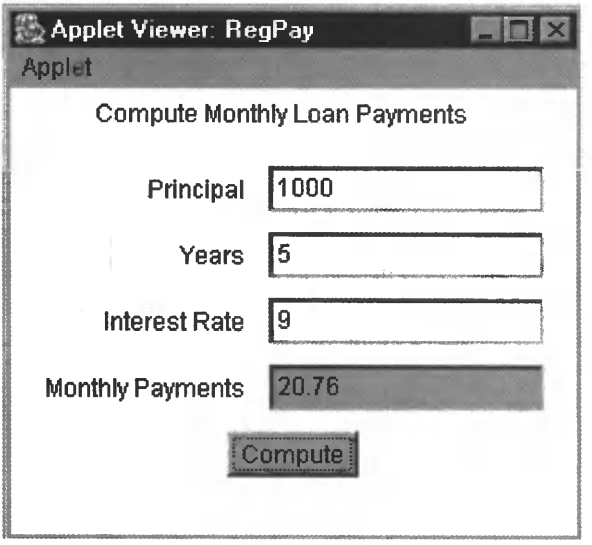


Рис. 9.1. Аплет `RegPay`

Поля аплета RegPay

Аплет `RegPay` начинается с объявления нескольких переменных, которые содержат ссылки на текстовые поля, в которые вводится информация о ссуде. Затем объявляется переменная `dolt`, которая содержит ссылку на кнопку Compute.

В RegPay также объявлены три переменные типа double, которые содержат информацию о ссуде. Исходная сумма сохраняется в переменной principal, ставка процента сохраняется в переменной intRate и срок выплат находится в переменной numYears. Эти три значения вводятся пользователем в текстовые поля. Затем объявляется неизменяемая целочисленная переменная payPerYear, которая инициализируется значением 12. Таким образом, в течение года выплаты производятся 12 раз, т.е. один раз в месяц, как и принято в большинстве случаев.

Но можно позволить вводить это значение пользователю, для чего необходимо создать еще одно текстовое поле ввода.

Последней переменной, объявленной в классе RegPay, является переменная nf, представляющая ссылку на объект типа NumberFormat, который используется для задания числового формата.

Метод init()

Как и во всех других аплетах, метод init() вызывается, когда аплет вызывается на выполнение первый раз. Этот метод выполняет следующие основные задачи:

- изменяет тип расположения на GridBagLayout;
- инициализирует различные компоненты ;
- добавляет компоненты в таблицу;
- добавляет слушателей для компонентов.

Рассмотрим метод init() строка за строкой. Метод начинается со следующих строк кода.

```
// Использовать расположение GridBagLayout.
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
```

Для большинства небольших аплетов наиболее подходит тип расположения FlowLayout, который и устанавливается по умолчанию. Но поскольку в финансовых аплетах требуется, чтобы пользователь вводил исходные значения, необходимо обеспечить больший контроль за компонентами, расположенными в окне аплета. Это удобно сделать с помощью типа расположения GridBagLayout, для получения которого используется класс GridBagLayout. Преимущество использования этого класса заключается в том, что можно задать относительное расположение компонентов в окне. Расположение компонентов происходит по линиям сетки, при этом каждая строка может иметь различное число столбцов и каждый компонент может иметь различные размеры. Именно поэтому такое расположение называется *многосеточным* (grid bag). Это коллекция отдельных сеток, соединенных вместе.

Расположение и размеры каждого компонента на сетке определяется набором ограничителей, связанных с ним. Ограничители содержатся в объекте типа GridBagConstraints. Ограничители содержат высоту и ширину компонента, выравнивание компонента и точку его привязки.

Затем в методе init() создаются надписи, текстовые поля и кнопка Compute, как показано во фрагменте ниже.

```
Label heading = new
Label("Compute Monthly Loan Payments");
Label amountLab = new Label("Principal");
Label periodLab = new Label("Years");
Label rateLab = new Label("Interest Rate");
Label paymentLab = new Label("Monthly Payments");
amountText = new TextField (16);
periodText = new TextField(16);
paymentText = new TextField(16);
rateText = new TextField(16);
// Поле для отображения платежа.
paymentText.setEditable(false);
dolt = new Button("Compute");
```

Обратите внимание, что текстовое поле, которое предназначено для отображения ежемесячных выплат, установлено в режим только для чтения с помощью вызова метода `setEditable(false)`. Это приводит к тому, что визуально поле отображается в сером цвете.

Затем для каждого компонента задаются ограничители, как показано во фрагменте ниже.

```
// Задать сетку.
gbc.weighty = 1.0; // Использовать коэффициент строки 1.
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);
// Привязать компоненты к правой стороне.
gbc.anchor = GridBagConstraints.EAST;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);
gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(dolt, gbc);
```

На первый взгляд это кажется довольно запутанным, но на самом деле все просто. Давайте вспомним, что каждая строка сетки задается отдельно. В приведенных строках и производится последовательная работа. Сначала коэффициент каждой строки, сохраняемый в переменной `gbc.weighty`, устанавливается в `1`. Это указание менеджеру `GridBagLayout` равномерно распределить дополнительное пространство по вертикали, которое остается после размещения компонентов. Затем переменной `gbc.gridwidth` присваивается значение `REMAINDER`, а переменной `gbc.anchor` присваивается значение `NORTH`. В помощью метода `setConstraints()` для объекта `gbag` добавляется надпись, на которую ссылается переменная `heading`. Такая последовательность устанавливает положение заголовка вверху сетки (north) и остав-

ляет остаток строки. Таким образом, после выполнения этой последовательности строк заголовок будет располагаться вверху окна и на строке.

Затем добавляются четыре текстовых поля и соответствующие надписи. Сначала присваивается значение EAST для переменной gbc.anchor. Это приводит к выравниванию всех элементов управления по правой стороне. Затем переменной gbc.gridWidth присваивается значение RELATIVE и добавляются надписи. После этого переменной присваивается значение REMAINDER и добавляются текстовые поля. Таким образом, каждое текстовое поле и надпись занимают по одной строке. Этот процесс повторяется до тех пор, пока все четыре текстовых поля и надписи не будут добавлены. Наконец, в центр последней строка добавляется кнопка Compute.

После задания всех ограничителей, компоненты добавляются в окно с помощью следующего кода.

```
// Добавить все компоненты.
add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(paymentLab);
add(paymentText);
add(dolt);
```

Затем для трех текстовых полей и кнопки Compute регистрируются слушатели, как показано в следующем коде.

```
// Регистрация для приема уведомлений о событиях.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
dolt.addActionListener(this);
```

Наконец, устанавливается числовой формат для вывода двух десятичных цифр.

```
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
```

Метод actionPerformed()

Метод actionPerformed() вызывается, когда пользователь нажимает клавишу <ENTER> при активном текстовом поле или при щелчке на кнопке Compute. Этот метод просто вызывает метод repaint(), в котором в конечном итоге вызывается метод для прорисовки изображения paint().

Метод paint()

Метод paint() выполняет три основные задачи: получает информацию о ссуде, вводимую пользователем, вызывает метод compute() для подсчета выплат по ссуде, и отображает результат расчета. Рассмотрим подробнее метод paint().

После объявления переменных в методе paint() заполняются строки из трех полей ввода, как показано ниже.

```
String amountStr = amountText.getText();
String periodStr = periodText.getText();
String rateStr = rateText.getText();
```

Затем выполняется блок `try` и производится проверка, что все три поля содержат данные, для чего используются следующие строки.

```
try {
  if(amountStr.length() != 0 &&
     periodStr.length() != 0 && rateStr.length() != 0) {
```

Вспомните, что пользователь должен ввести начальный размер ссуды, срок выплаты ссуды и ставку процента. Если все три поля содержат информацию, то длина строки должна быть больше нуля.

Если заполнены все три поля, то определяются числовые значения, соответствующие этим строкам, и сохраняются в отдельных переменных. Затем вызывается метод `compute()` для расчета выплат по ссуде и результат отображается в текстовом поле только для чтения, на которое ссылается переменная `paymentText`, как показано ниже.

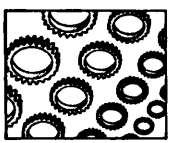
```
principal = Double.parseDouble(amountStr);
numYears = Double .parseDouble (periodStr);
intRate = Double.parseDouble(rateStr) / 100;
result = compute();
paymentText.setText(nf.format(result));
```

Если пользователь ввел не число в одном из полей, то в методе `Double.parseDouble()` будет сгенерирована исключительная ситуация `NumberFormatException`. В этом случае в строке состояния будет отображаться сообщение об ошибке и текстовое поле будет очищено, как показано ниже.

```
showStatus("");     //  Удалить все предыдущие сообщения.
}
catch (NumberFormatException exc) {
  showStatus("Invalid  Data");
  paymentText.setText( " " );
}
```

Метод compute()

Расчет выплаты по ссуде происходит в методе `compute()`. Здесь просто реализовано вычисление по формуле, описанной выше, при этом используются переменные `principal`, `intRate`, `numYears`, и `payPerYear`. Возвращается результат расчета.



Базовый каркас, используемый в классе `RegPay`, применяется во всех аплетах, представленных в этой главе.

Расчет будущей стоимости инвестиций

Другим часто используемым финансовым расчетом, является расчет будущей стоимости инвестиций на основе начального капиталовложения, нормы прибыли, значения сложного процента за год и количества лет, на которые рассчитаны инвестиции. Например, вы можете узнать, какова будет стоимость через 12 лет, если в на-

стоящее время она составляет 98 тыс. дол. и средний годовой уровень составляет 6%. Аплет `FutVal`, разработанный для этих целей, легко даст ответ на этот вопрос. Подсчитать будущую стоимость можно по следующей формуле:

$$\text{Future Value} = \text{principal} * ((\text{rateOfRet} / \text{compPerYear}) + 1)^{\text{compPerYear} * \text{numYears}}$$

где переменная `rateOfRet` определяет норму прибыли, переменная `principal` содержит начальное значение суммы инвестиций, переменная `compPerYear` определяет периоды расчета сложного процента за год и переменная `numYears` задает количество лет, определяющее срок инвестиций. Если вы используете ежегодную норму прибыли для переменной `rateOfRet`, то число периодов расчета сложного процента будет равно 1.

В приведенном ниже аплете `FutVal` эта формула используется для расчета будущей стоимости инвестиций. Визуальное отображение аплета показано на рис. 9.2.

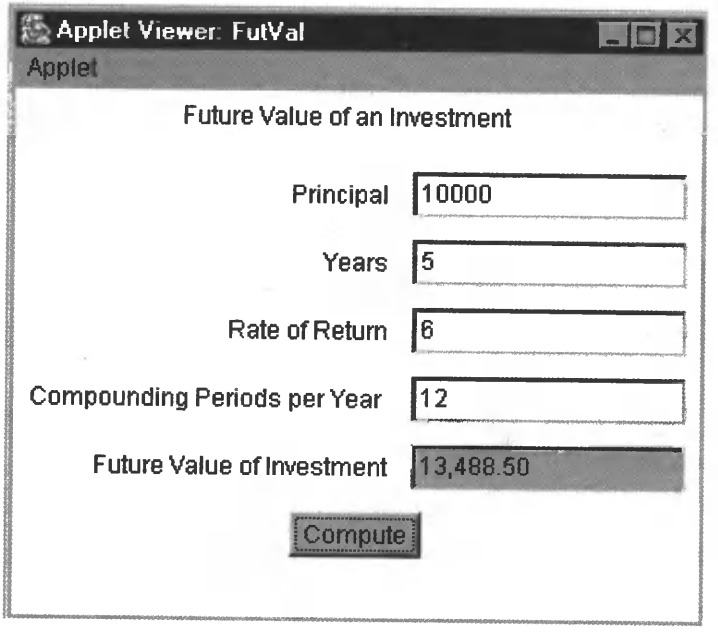


Рис. 9.2. Аплет `FutVal`

Исключая различия в расчетных формулах в методах `compute()`, аплет аналогичен аплету `RegPay`, описанному в предыдущем разделе.

```
// Расчет будущей стоимости инвестиций.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.text.*;
/*
  <applet code="FutVal" width=340 height=240>
  </applet>
*/

public class FutVal extends Applet
  implements ActionListener {
```

```
TextField amountText, futvalText, periodText,
          rateText, compText;
Button doIt;

double principal; // Начальный капитал.
double rateOfRet; // Норма прибыли.
double numYears;  // Срок инвестиций в годах.
int compPerYear;  // Число периодов расчета сложного процента.
NumberFormat nf;
public void init() {
  // Use a grid bag layout.
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);
  Label heading = new
        Label("Future Value of an Investment");
  Label amountLab = new Label("Principal");
  Label periodLab = new Label("Years");
  Label rateLab = new Label("Rate of Return");
  Label futvalLab =
          new Label("Future Value of Investment");
  Label compLab =
          new Label("Compounding Periods per Year ");
  amountText = new TextField(16);
  periodText = new TextField(16);
  futvalText = new TextField(16);
  rateText = new TextField(16);
  compText = new TextField(16);
  // Поле для отображения будущей стоимости инвестиций.
  futvalText.setEditable(false);
  doIt = new Button("Compute");

  // Задать менеджер расположения.
  gbc.weighty = 1.0; // Использовать коэффициент 1
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);

  // Привязка компонентов к правой стороне.
  gbc.anchor = GridBagConstraints.EAST;
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(amountLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(amountText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(periodLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(periodText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(rateLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(rateText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(compLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(compText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(futvalLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(futvalText, gbc);
  gbc.anchor = GridBagConstraints.CENTER;
  gbag.setConstraints(doIt, gbc);
```

```
    add(heading);
    add(amountLab);
    add(amountText);
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(compLab);
    add(compText);
    add(futvalLab);
    add(futvalText);
    add(doIt);

    // Регистрация для приема уведомлений о событиях.
    amountText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    compText.addActionListener(this);
    doIt.addActionListener(this);
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал клавишу Enter при активном
     текстовом поле или щелкнул на кнопке Compute. */
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }
  public void paint(Graphics g) {
    double result = 0.0;
    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();
    try {
      if(amountStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0 &&
         compStr.length() != 0) {
        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        compPerYear = Integer.parseInt(compStr);
        result = compute();
        futvalText.setText(nf.format(result));
      }
      showStatus(""); // Удалить предыдущие сообщения.
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      futvalText.setText("");
    }
  }
  // Подсчитать будущее значение.
  double compute() {
    double b, e;
    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;
    return principal * Math.pow(b, e);
  }
}
```

Определение размера начальных капиталовложений для достижения необходимой будущей стоимости

Иногда надо знать, какие начальные капиталовложения необходимо сделать для того, чтобы получить требуемую будущую стоимость. Например, если вы собираетесь откладывать деньги на оплату учебы детей в колледже и знаете, что вам нужно иметь 75 тыс. дол. на пять лет, то какую сумму вам необходимо инвестировать под 7%, чтобы получить необходимые средства? Аплет `InitInv`, листинг которого приведен ниже, поможет ответить на этот вопрос. Формула для расчета начальных вложений такова:

$$\text{Initial Investment} = \text{target Value} / (((\text{rateOfRet} / \text{compPerYear}) + 1)^{\text{corapPerYear} * \text{numYears}})$$

где переменная `rateOfRet` определяет норму прибыли, переменная `targetValue` содержит остаток на начало периода, переменная `compPerYear` определяет периоды расчета сложного процента за год и переменная `numYears` задает срок инвестиций в годах. Если вы используете ежегодную норму прибыли для переменной `rateOfRet`, то число периодов расчета сложного процента будет равно 1.

В приведенном ниже аплете `InitInv` используется приведенная формула для расчета начальных инвестиций и получения необходимой суммы в будущем. Работа аплета показана на рис. 9.3.

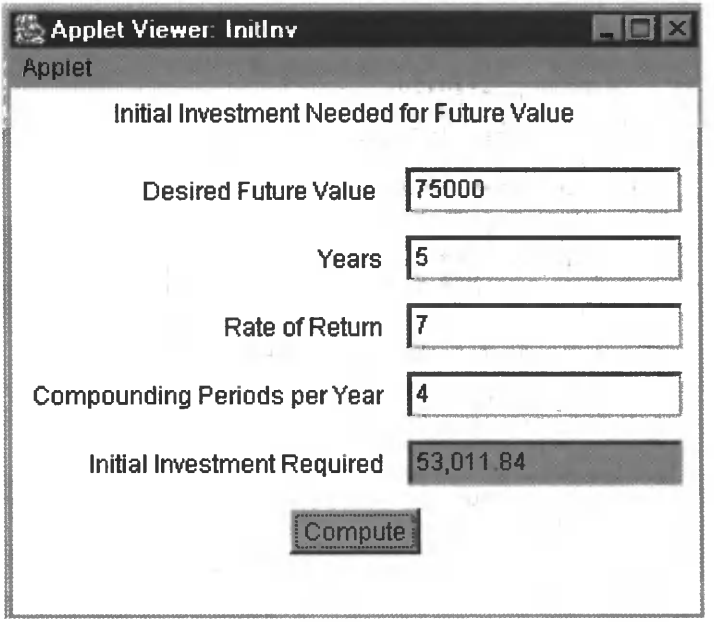


Рис. 9.3. Аплет `InitInv`

```
/* Расчет начальных инвестиций для получения
   необходимой суммы в будущем */
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
import java.text.*;
/*
  <applet code="InitInv" width=340 height=240>
  </applet>
*/
public class InitInv extends Applet
    implements ActionListener {
  TextField targetText, initialText, periodText,
            rateText, compText;
  Button doIt;
  double targetValue; // Остаток на начало периоде.
  double rateOfRet;   // Норма прибыли.
  double numYears;    // Срок инвестиций в годах.
  int compPerYear;    // Количество расчетов сложного процента.
  NumberFormat nf;
  public void init() {
    // Используем менеджер расположения GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);
    Label heading = new
          Label("Initial Investment Needed for " +
                "Future Value");
    Label targetLab = new Label("Desired Future Value ");
    Label periodLab = new Label("Years");
    Label rateLab = new Label("Rate of Return");
    Label compLab =
          new Label("Compounding Periods per Year");
    Label initialLab =
          new  Label("Initial Investment Required");
    targetText = new TextField(16);
    periodText = new TextField(16);
    initialText = new TextField(16);
    rateText = new TextField(16);
    compText = new TextField(16);

    // Поле для отображения начального значения.
    initialText.setEditable(false);
    doIt = new Button("Compute");

    // Задать сетку.
    gbc.weighty = 1.0; // Используем коэффициент 1
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbc.anchor = GridBagConstraints.NORTH;
    gbag.setConstraints(heading, gbc);

    // Привязка компонентов к правой стороне.
    gbc.anchor = GridBagConstraints.EAST;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(targetLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(targetText, gbc);
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(periodLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(periodText, gbc);
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(rateLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(rateText, gbc);
```

```
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(compLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(compText, gbc);
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(initialLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(initialText, gbc);
    gbc.anchor = GridBagConstraints.CENTER;
    gbag.setConstraints(doIt, gbc);

    // Добавить все компоненты.
    add(heading);
    add(targetLab);
    add(targetText);
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(compLab);
    add(compText);
    add(initialLab);
    add(initialText);
    add(doIt);

    // Зарегистрировать для приема уведомлений о событиях.
    targetText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    compText.addActionListener(this);
    doIt.addActionListener(this);
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал клавишу Enter при
     активном текстовом поле или щелкнул на кнопке Compute. */
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }
  public void paint(Graphics g) {
    double result = 0.0;
    String targetStr = targetText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();
    try {
      if(targetStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0 &&
         compStr.length() != 0) {
        targetValue = Double.parseDouble(targetStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        compPerYear = Integer.parseInt(compStr);
        result = compute();
        initialText.setText(nf.format(result));
      }
      showStatus(""); // Удалить предыдущие сообщения.
    } catch (NumberFormatException exc) {
```

```
      showStatus("Invalid Data");
      initialText.setText("");
    }
  }

  // Подсчитать требуемые начальные капиталовложения.
  double compute() {
    double b, e;
    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;
    return targetValue / Math.pow(b, e);
  }

}
```

Определение начальных капиталовложений для получения требуемого ежегодного дохода

Еще одним часто используемым финансовым расчетом является расчет суммы денег, которые должны быть инвестированы, чтобы получить желаемый ежегодный доход. Например, вы хотите получать 5 тыс. дол. в месяц в течение 20 лет после ухода на пенсию. Поэтому нужно знать, какую сумму необходимо инвестировать, чтобы получать такие деньги. На этот вопрос можно ответить с помощью следующей формулы:

$$\text{Initial Investment} = ((\text{regWD} * \text{wdPerYear}) / \text{rateOfRet}) * (1-(1 / (\text{rateOfRet} / \text{wdPerYear} + 1)^{\text{wdPerYear} * \text{numYear}}))$$

где переменная `rateOfRet` определяет норму прибыли, переменная `regWD` содержит величину требуемых регулярных выплат денег, переменная `wdPerYear` задает количество выплат за год и переменная `numYears` определяет срок выплат.

Аплет `Annuity`, листинг которого приведет ниже, подсчитывает размер начальных вложений для получения требуемых выплат. На рис. 9.4 показано окно рабочего аплета.

```
/* Подсчет размера начальных вложений для
   получения требуемых выплат. Другими словами,
   определяется размер начальных капиталовложений, на
   основе которых можно будет регулярно получать
   требуемую сумму в течение заданного срока  */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.text.*;
/*
  <applet code="Annuity" width=340 height=260>
  </applet>
*/
public class Annuity extends Applet
  implements ActionListener {
  TextField regWDText, initialText, periodText,
            rateText, numWDText;
  Button doIt;
  double regWDAmount; // Размер выплат.
  double rateOfRet;   // Норма прибыли.
  double numYears;    // Срок выплат в годах.
  int numPerYear;     // Количество выплат за год.
```

```
NumberFormat nf;
public void init() {

  // Используем менеджер GridBagLayout.
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);
  Label heading = new
        Label("Initial Investment Needed for " +
              "Regular Withdrawals");
  Label regWDLab = new Label("Desired Withdrawal");
  Label periodLab = new Label("Years");
  Label rateLab = new Label("Rate of Return");
  Label numWDLab =
         new Label("Number of Withdrawals per Year ");
  Label initialLab =
         new Label("Initial Investment Required");
  regWDText = new TextField(16);
  periodText = new TextField(16);
  initialText = new TextField(16);
  rateText = new TextField(16);
  numWDText = new TextField(16);

  // Поле для отображения начальных вложений.
  initialText.setEditable(false);
  doIt = new Button("Compute");

  // Задать сетку.
  gbc.weighty = 1.0; // Использовать коэффициент 1
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);

  // Привязка компонентов к правой стороне.
  gbc.anchor = GridBagConstraints.EAST;
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(regWDLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(regWDText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(periodLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(periodText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(rateLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(rateText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(numWDLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(numWDText, gbc);
  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(initialLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(initialText, gbc);
  gbc.anchor = GridBagConstraints.CENTER;
  gbag.setConstraints(doIt, gbc);

  // Добавить все компоненты.
  add(heading);
  add(regWDLab);
  add(regWDText);
```

```
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(numWDLab);
    add(numWDText);
    add(initialLab);
    add(initialText);
    add(doIt);

    // Зарегистрировать для приема уведомлений о событиях.
    regWDText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    numWDText.addActionListener(this);
    doIt.addActionListener(this);
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  // User pressed Enter on a text field.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }
  public void paint(Graphics g) {
    double result = 0.0;
    String regWDStr = regWDText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();
    try {
      if(regWDStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0 &&
         numWDStr.length() != 0) {
        regWDAmount = Double.parseDouble(regWDStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        numPerYear = Integer.parseInt(numWDStr);
        result = compute();
        initialText.setText(nf.format(result));
      }
      showStatus(""); // Удалить предыдущие сообщения.
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      initialText.setText("");
    }
  }

  // Подсчитать требуемое начальное вложение.
  double compute() {
    double b, e;
    double t1, t2;
    t1 = (regWDAmount * numPerYear) / rateOfRet;
    b = (1 + rateOfRet/numPerYear);
    e = numPerYear * numYears;
    t2 = 1 - (1 / Math.pow(b, e));
    return t1 * t2;
  }
}
```

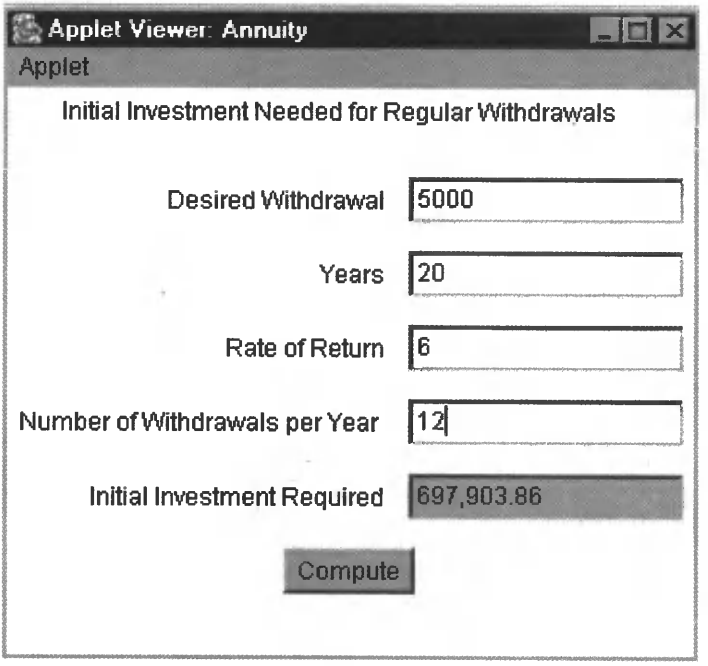


Рис. 9.4. Аплет `Annuity`

Определение ежегодного дохода при заданных начальных вложениях

С помощью еще одного финансового калькулятора можно рассчитать максимальный ежегодный доход (при регулярном снятии денег), который можно получить при заданном начальном вложении капитала через указанный промежуток времени. Например, если вы имеете 500 тыс. дол. на пенсионном счету, то какую сумму можно получать каждый месяц в течение 20 лет, предполагая, что норма прибыли составляет 6%? Формула для подсчета этого значения будет такая:

$$\text{Maximum Withdrawal} = \text{principal} * (((\text{rateOfRet} / \text{wdPerYear}) / (-1 + ((\text{rateOfRet} / \text{wdPerYear}) + 1)^{\text{wdPerYear} * \text{numYears}})) + (\text{rateOfRet} / \text{wdPerYear}))$$

где переменная `rateOfRet` определяет норму прибыли, переменная `principal` содержит начальное значение суммы инвестиций, переменная `wdPerYear` задает количество выплат за год и переменная `numYears` определяет срок выплат.

Аплет `MaxWD`, листинг которого приведен ниже, подсчитывает максимальное значение регулярных выплат, которые могут быть получены исходя из срока выплат при предполагаемой норме прибыли. Работа аплета показана на рис. 9.5.

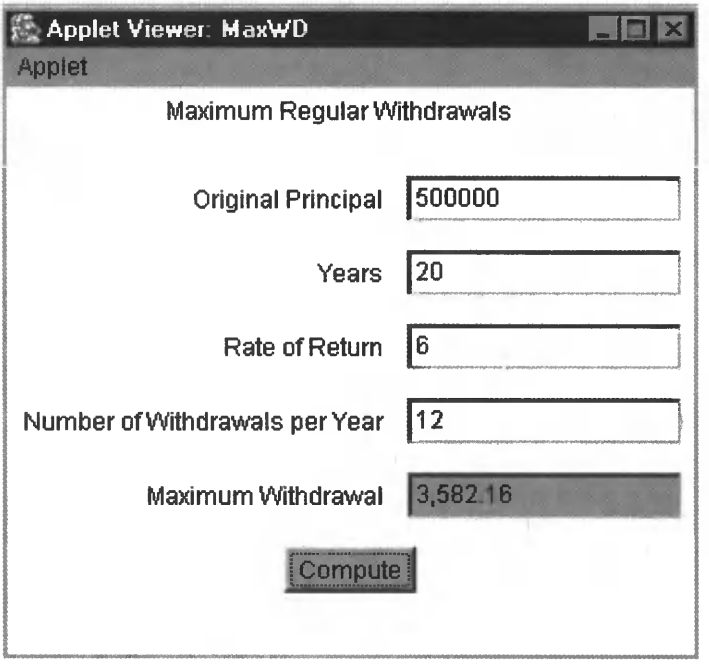


Рис. 9.5. Аплет MaxWD

```
/* подсчитывает максимальное значение регулярных выплат,
   которые могут получены исходя из срока выплат при
   предполагаемой норме прибыли.  */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.text.*;
/*
  <applet code="MaxWD" width=340 height=260>
  </applet>
*/

public class MaxWD extends Applet
  implements ActionListener {
  TextField maxWDText, orgPText, periodText,
            rateText, numWDText;
  Button doIt;
  double principal; // Начальные вложения.
  double rateOfRet; // Ежегодная норма прибыли.
  double numYears;  // Срок выплат.
  int numPerYear;   // Количество выплат в год.
  NumberFormat nf;
  public void init() {
    // Использовать расположение GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);
    Label heading = new
          Label("Maximum Regular Withdrawals");
    Label orgPLab = new Label("Original Principal");
    Label periodLab = new Label("Years");
```

```
Label rateLab = new Label("Rate of Return");
Label numWDLab =
       new Label("Number of Withdrawals per Year");
Label maxWDLab = new Label("Maximum Withdrawal");
maxWDText = new TextField(16);
periodText = new TextField(16);
orgPText = new TextField(16);
rateText = new TextField(16);
numWDText = new TextField(16);

// Поле для отображения величины выплат.
maxWDText.setEditable(false);
doIt = new Button("Compute");

// Задать сетку.
gbc.weighty = 1.0; // Использовать коэффициент 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Привязать компоненты к правой стороне окна
gbc.anchor = GridBagConstraints.EAST;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(orgPLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(orgPText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numWDText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(maxWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(maxWDText, gbc);
gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавить все компоненты.
add(heading);
add(orgPLab);
add(orgPText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(numWDLab);
add(numWDText);
add(maxWDLab);
add(maxWDText);
add(doIt);

// Зарегистрировать для приема уведомлений о событии.
orgPText.addActionListener(this);
periodText.addActionListener(this);
```

```
    rateText.addActionListener(this);
    numWDText.addActionListener(this);
    doIt.addActionListener(this);
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал клавишу Enter при активном
     текстовом поле или щелкнул на кнопке Compute. */
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }
  public void paint(Graphics g) {
    double result = 0.0;
    String orgPStr = orgPText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();
    try {
      if(orgPStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0 &&
         numWDStr.length() != 0) {
        principal = Double.parseDouble(orgPStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        numPerYear = Integer.parseInt(numWDStr);
        result = compute();
        maxWDText.setText(nf.format(result));
      }
      showStatus(""); // Удалить предыдущие сообщения.
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      maxWDText.setText("");
    }
  }

  // Подсчитать размер регулярных выплат.
  double compute() {
    double b, e;
    double t1, t2;
    t1 = rateOfRet / numPerYear;
    b = (1 + t1);
    e = numPerYear * numYears;
    t2 = Math.pow(b, e) - 1;

    return principal * (t1/t2 + t1);
  }
}
```

Определение остаточного баланса по займу

Если вы хотите узнать остаточный баланс по займу, то это легко рассчитать при знании размера начальных вложений, ставки процента, срока займа и количества платежей. Для определения остаточного баланса необходимо суммировать платежи, вычитая из каждого платежа размер процента, а затем вычесть результат из величины основной суммы.

Аплет RemBal, показанный ниже, определяет остаточный баланс по займу. Работа аплета показана на рис. 9.6.

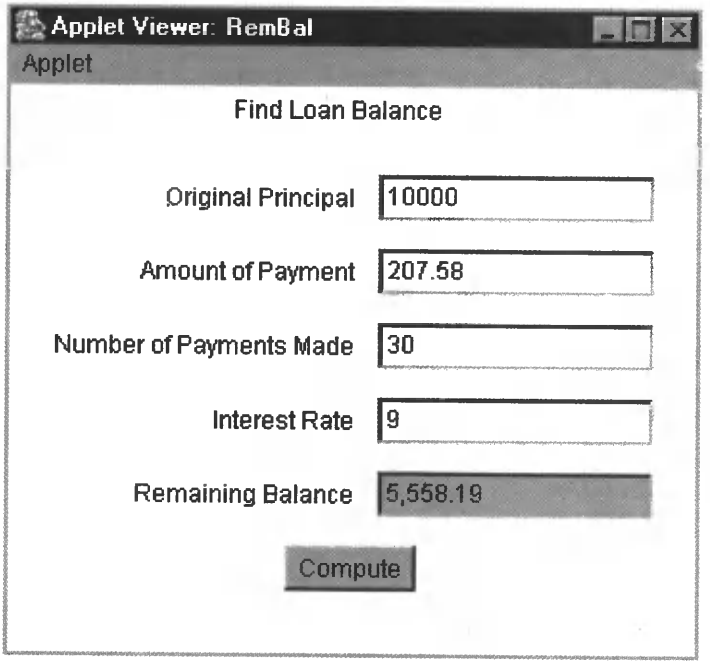


Рис. 9.6. Аплет RemBal

```
// Определить остаточный баланс по займу.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.text.*;
/*
  <applet code="RemBal" width=340 height=260>
  </applet>
*/

public class RemBal extends Applet
  implements ActionListener {
  TextField orgPText, paymentText, remBalText,
            rateText, numPayText;
  Button doIt;
  double orgPrincipal; // Начальный капитал.
  double intRate;      // Ставка процента.
  double payment;      // Размер каждого платежа.
  double numPayments;  // Количество сделанных платежей.

  /* Количество платежей за год. Это значение
     можно разрешить вводить пользователю. */
  final int payPerYear = 12;
  NumberFormat nf;
  public void init() {
    // Использовать расположение GridBagLayout.
    GridBagLayout gbag = new GridBagLayout();
```

```
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
Label heading = new
      Label("Find Loan Balance ");
Label orgPLab = new Label("Original Principal");
Label paymentLab = new Label("Amount of Payment");
Label numPayLab = new Label("Number of Payments Made");
Label rateLab = new Label("Interest Rate");
Label remBalLab = new Label("Remaining Balance");
orgPText = new TextField(16);
paymentText = new TextField(16);
remBalText = new TextField(16);
rateText = new TextField(16);
numPayText = new TextField(16);

// Поле для отображения платежей.
remBalText.setEditable(false);
doIt = new Button("Compute");

// Задать сетку.
gbc.weighty = 1.0; // use a row weight of 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Привязка компонентов с правой стороны.
gbc.anchor = GridBagConstraints.EAST;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(orgPLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(orgPText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numPayLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numPayText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(remBalLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(remBalText, gbc);
gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавить все компоненты.
add(heading);
add(orgPLab);
add(orgPText);
add(paymentLab);
add(paymentText);
add(numPayLab);
add(numPayText);
add(rateLab);
add(rateText);
add(remBalLab);
```

```
    add(remBalText);
    add(doIt);

    // Зарегистрировать для приема уведомлений о событиях.
    orgPText.addActionListener(this);
    numPayText.addActionListener(this);
    rateText.addActionListener(this);
    paymentText.addActionListener(this);
    doIt.addActionListener(this);
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

/* Пользователь нажал клавишу Enter при активном
   текстовом поле или щелкнул на кнопке Compute. */
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }
  public void paint(Graphics g) {
    double result = 0.0;
    String orgPStr = orgPText.getText();
    String numPayStr = numPayText.getText();
    String rateStr = rateText.getText();
    String payStr = paymentText.getText();
    try {
      if(orgPStr.length() != 0 &&
         numPayStr.length() != 0 &&
         rateStr.length() != 0 &&
         payStr.length() != 0) {
        orgPrincipal = Double.parseDouble(orgPStr);
        numPayments = Double.parseDouble(numPayStr);
        intRate = Double.parseDouble(rateStr) / 100;
        payment = Double.parseDouble(payStr);
        result = compute();
        remBalText.setText(nf.format(result));
      }
      showStatus(""); // Удалить предыдущие сообщения.
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      remBalText.setText("");
    }
  }

  // Подсчитать остаточный баланс по займу.
  double compute() {
    double bal = orgPrincipal;
    double rate = intRate / payPerYear;
    for(int i = 0; i < numPayments; i++)
      bal -= payment - (bal * rate);
    return bal;
  }
}
```

Создание финансовых сервлетов

Хотя аплеты легко создавать и использовать, это только половина задач, которые Java выполняет с помощью Internet. Другой половиной являются сервлеты. Сервлеты выполняются на стороне сервера для отдельного соединения и для решения не-

которых задач они больше подходят, чем аплеты. Поскольку многие читатели могут предпочесть использовать сервлеты вместо аплетов для создания своих коммерческих приложений, то оставшаяся часть этой главы будет посвящена вопросу преобразования аплета в сервлет.

Поскольку все аплеты для финансовых расчетов разработаны по одному и тому же принципу, рассмотрим преобразование только одного аплета `RegPay`. Аналогичный способ можно использовать для преобразования всех других аплетов. И как вы сможете скоро убедиться, сделать это несложно.

Еще одно замечание: в этой книге предполагается, что читатель понимает основы архитектуры и жизненный цикл сервлета. Если вам необходимо освежить свои знания о сервлетах, то можно обратиться к книге *Java 2: The Complete Reference by Herbert Schild (Berkeley, CA: McGraw-Hill/ Osborne, 2002).*

Использование Tomcat

Перед разработкой сервлета, будет полезно познакомиться с процедурой для компиляции и запуска сервлета. Создание сервлета требует несколько больших усилий, чем создание аплета.

Для компиляции и тестирования сервлета, который будет разработан, необходимо установить на компьютер среду разработки сервлетов. Часто для этих целей используют программу Tomcat версии 4.1, которая поддерживает спецификацию сервлетов 2.3. (Полная спецификация сервлетов доступна для всех на сервере `java.sun.com`. Хотя как текущая версия Tomcat, так и спецификация сервлетов могут быть модифицированы, пока вы читаете эту книгу.) Среду Tomcat нужно использовать вместо устаревшей среды разработки Java Servlet Development Kit (JSDK), ранее разработанной фирмой Sun. Tomcat является свободно распространяемым продуктом с исходным кодом (open-source) но с ограничениями по его использованию как коммерческого продукта, которые установлены группой Jakarta Project организации Apache Software Foundation. В него входят библиотеки классов, документация и программы поддержки, необходимые при разработке и тестировании сервлета. Продукт Tomcat можно загрузить с Web-сервера фирмы Sun Microsystems: `atjava.sun.com`

По умолчанию размещение Tomcat в окружении Windows будет находиться по следующему адресу.

```
C:\Program Files\ApacheGroup\Tomcat4.1\
```

В примере, рассмотренном в данной главе, предполагается именно этот адрес. Если вы установите Tomcat в другой каталог, то необходимо будет сделать соответствующие изменения в примере. Вы можете задать переменную окружения `JAVA_HOME` в каталоге верхнего уровня, где установлена среда разработки Java Software Developers Kit. Для Java 2 версии 1.4 по умолчанию это будет каталог `C:\j2sdk1.4.0`, но прежде необходимо убедиться в том, что все именно так. Каталог

```
C:\Program Files\Apache Group\Tomcat 4.1\common\lib\
```

содержит файл `servlet.jar`. Этот файл с расширением `.jar` содержит классы и интерфейсы, которые необходимы при разработке сервлета. Для доступа к этим файлам, обновите переменную окружения `CLASSPATH` таким образом, чтобы она включала следующую строку.

```
C:\Program FilesXApache Group\Tomcat 4.1\common\lib\servlet.jar
```

Как альтернативное решение, вы можете указать этот файл при компиляции сервлета. Например, с помощью следующей команды производится компиляция сервлета с именем `MyServlet`.

```
javac MyServlet.Java -classpath "C:\Program Files\Apache Group\Tomcat
4.1\common\lib\servlet.jar"
```

После компиляции сервлета необходимо будет скопировать файл с классами в сервлет, который будет использоваться, или использовать среду Tomcat для тестирования сервлета, как это описано ниже.

Тестирование сервлета

Для тестирования или экспериментов можно использовать среду Tomcat, запуская сервлет непосредственно на компьютере без выхода в Internet. Для того чтобы сделать это, необходимо скопировать файл с классами сервлета в каталог, который среда Tomcat использует для примеров файла классов. Для наших целей это будет следующий каталог.

```
C:\Program FilesXApache Group\Tomcat 4.1\webapps\examples\WEB-INF\classes
```

После того как сервлет скопирован в указанный каталог, необходимо запустить Tomcat. Для этого можно выбрать пункт меню **Start Tomcat** в меню **Start⇨Programs** или использовать командный файл `startup.bat` из следующего каталога.

```
C:\Program Files\Apache Group\Tomcat 4.1\bin\
```

После тестирования сервлета надо будет закрыть программу `Tomcat` с помощью выбора команды меню **Stop Tomcat** в меню **Start⇨Programs** или использовать командный файл `shutdown.bat`.

Для тестирования сервлета запустите Web-браузер и введите URL, подобный показанному ниже.

```
http://localhost:8080/examples/servlet/MyServlet
```

Конечно, вы должны использовать реальное имя сервлета вместо имени `MyServlet`. Указывая хост `localhost:8080`, вы задаете управляющий компьютер.

Преобразование аплета RegPay в сервлет

Теперь рассмотрим, как преобразовать аплет `RegPay` в сервлет. Сначала сервлет должен импортировать пакеты `javax.servlet` и `javax.servlet.http`. Он также должен расширять возможности класса `HttpServlet`, а не класса `Applet`. Затем необходимо удалить весь код, связанный с графическим интерфейсом. После чего надо добавить код, с помощью которого можно получить параметры, передаваемые в сервлет страницей HTML, на которой произошло обращение к сервлету. Наконец, сервлет должен послать код, который отображает результат. Основные финансовые расчеты остаются теми же, изменяется только способ получения и отображения данных.

Сервлет RegPayS

В следующем листинге показан класс RegPayS, который является сервлетом, созданным на основе аплета RegPay.

```
// Простой сервлет для расчета ссуды.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.*;

public class RegPayS extends HttpServlet {
  double principal; // Начальные вложения.
  double intRate;   // Ставка процента.
  double numYears;  // Срок ссуды в годах.

  /* Количество платежей за год. Можно разрешить
     пользователю вводить это значение. */
  final int payPerYear = 12;
  NumberFormat nf;
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    String payStr = "";
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);

    // Получить параметры.
    String amountStr = request.getParameter("amount");
    String periodStr = request.getParameter("period");
    String rateStr = request.getParameter("rate");
    try {
      if(amountStr != null && periodStr != null &&
         rateStr != null) {
        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        intRate = Double.parseDouble(rateStr) / 100;
        payStr = nf.format(compute());
      }
      else { // one or more parameters missing
        amountStr = "";
        periodStr = "";
        rateStr = "";
      }
    } catch (NumberFormatException exc) {
      // Никаких действий для этого исключения.
    }

    // Установить тип содержимого.
    response.setContentType("text/html");

    // Получить выходной поток.
    PrintWriter pw = response.getWriter();

    // Отобразить в формате HTML.
    pw.print("<html><body> <left>" +
             "<form name=\"Form1\"" +
             " action=\"http://localhost:8080/" +
             "examples/servlet/RegPayS\">" +
             "<B>Enter amount to finance:</B>" +
```

```
            " <input type=textbox name=\"amount\"" +
            " size=12 value=\"");
    pw.print(amountStr + "\">");
    pw.print("<BR><B>Enter term in years:</B>" +
            " <input type=textbox name=\"period\""+
            " size=12 value=\"");
    pw.println(periodStr + "\">");
    pw.print("<BR><B>Enter interest rate:</B>" +
            " <input type=textbox name=\"rate\"" +
            " size=12 value=\"");
    pw.print(rateStr + "\">");
    pw.print("<BR><B>Monthly Payment:</B>" +
            " <input READONLY type=textbox" +
            " name=\"payment\" size=12 value=\"");
    pw.print(payStr + "\">");
    pw.print("<BR><P><input type=submit value=\"Submit\">");
    pw.println("</form> </body> </html>");
  }

  // Выполнить расчеты.
  double compute() {
    double numer;
    double denom;
    double b, e;
    numer = intRate * principal / payPerYear;
    e = -(payPerYear * numYears);
    b = (intRate / payPerYear) + 1.0;
    denom = 1.0 - Math.pow(b, e);
    return numer / denom;
  }
}
```

Первое, на что нужно обратить внимание, что сервлет `RegPayS` содержит только два метода: `doGet()` и `compute()`. Метод `compute()` тот же самый, что использовался в аплете. Метод `doGet()` определен в классе `HttpServlet`, который является базовым для класса `RegPayS`. Этот метод вызывается сервлетом в том случае, когда сервлет должен отвечать на запрос GET. Обратите внимание, что с него передаются ссылки на объекты `HttpServletRequest` и `HttpServletResponse`, связанные с этим запросом.

Из параметра `request` сервлет получает аргументы, связанные с запросом. Для этого вызывается метод `getParameter()`. Возвращаемый параметр имеет строковый тип. Таким образом, цифровые значения должны быть преобразованы в двоичный формат. Если параметр недоступен, то возвращается значение `null`.

Из объекта `response` сервлет получает поток, в который должна быть записана выходная информация. Ответ затем возвращается в браузер с помощью вывода его в поток. До того, как создать объект `PrintWriter` для выходного потока, тип выхода должен быть установлен в значение `text/html` с помощью вызова метода `setContentType()`.

Можно обратиться к сервлету `RegPayS` не задавая параметры. Если сервлет вызван без параметров, то будет отображена пустая форма для расчета ссуды. Если же указать необходимые параметры, сервлет `RegPayS` рассчитает выплаты по ссуде и повторно отобразит форму с заполненным полем для выплат, как показано на рис. 9.7.

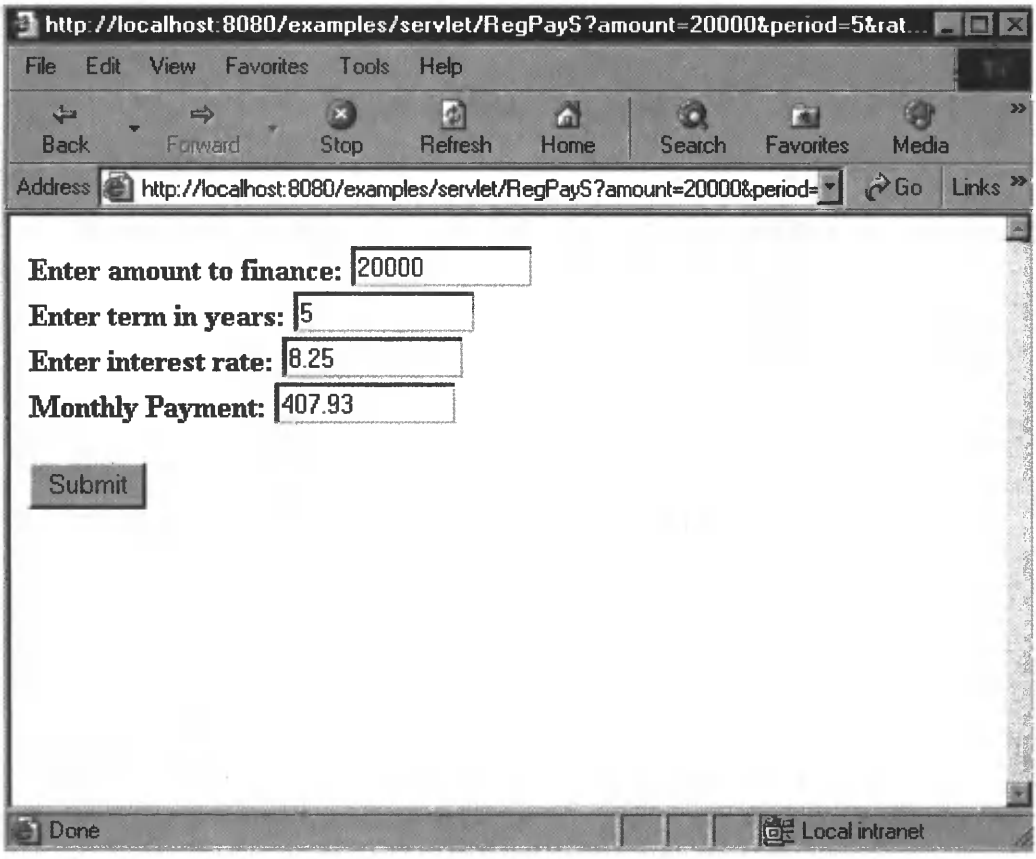


Рис. 9.7. Сервлет RegPayS в работе

Простейшим способом для обращения к сервлету RegPayS будет соединение по заданному адресу без передачи параметров. Например, при использовании среды Tomcat можно ввести следующую строку.

```
<A HREF = "http: //localhost:8080/examples/servlet/RegPayS">Loan Calcula-
tor</A>
```

Это приводит к отображению ссылки с именем Loan Calculator на сервлет RegPayS, находящийся в каталоге для примеров Tomcat. Обратите внимание, что не задается никаких параметров. Это приводит к возврату кода HTML для отображения пустой формы.

Вы также можете вызвать сервлет RegPayS, создав пустую форму вручную. Как это сделать, показано ниже, при этом опять используется каталог Tomcat.

```
<html>
<body>
<form name="Form1"
action="http://localhost:8080/examples/servlet/RegPayS">
<B>Enter amount to finance:</B>
<input type=textbox name="amount" size=12 value="">
<BR>
<B>Enter term in years:</B>
```

```
<input type=textbox name="period" size=12 value="">
<BR>
<B>Enter interest rate:</B>
<input type=textbox name="rate" size=12 value="">
<BR>
<B>Monthly Payment:</B>
<input READONLY type=textbox name="payment" size=12 value="">
<BR><P>
<input type=submit value="Submit">
</form>
</body>
</html>
```

Что еще можно сделать

Первое, что вы захотите еще сделать, — это преобразовать остальные финансовые аплеты с сервлеты. Поскольку все финансовые аплеты построены по одному и тому же принципу, то просто выполните те же действия, что и при преобразовании аплета `RegPay`.

Существует много других финансовых калькуляторов, которые можно реализовать с виде аплетов или сервлетов. Возможно, вы захотите создать большое приложение, в котором будут объединены все финансовые калькуляторы, представленные в этой главе или разработанные вами самими, а пользователи будут выбирать необходимый калькулятор из предлагаемых в меню.

This page is intentionally left blank

ГЛАВА 10

Поиск решений

В завершение этой книги будет рассмотрена интереснейшая дисциплина программирования: искусственный интеллект (ИИ). Как уже неоднократно упоминалось, при написании этой книги была поставлена задача показать мощь и богатство возможностей языка Java. Уже были показаны некоторые отличительные особенности языка, но, возможно, наилучшей демонстрацией потрясающих возможностей Java будет реализация приложений искусственного интеллекта. Мощные возможности Java по обработке строк и класс `Stack` делают приложения искусственного интеллекта наглядными и завершенными. Объектная модель Java со средствами уборки мусора позволяет получить хорошо понимаемый код. В этой последней главе показано, что язык Java является наиболее подходящим языком для разработки программ искусственного интеллекта.

Область искусственного интеллекта включает несколько самостоятельных разделов, но в основе большинства приложений лежит проблема принятия решения. В основном используются два способа решения проблемы. Первый способ — это решение задачи путем применения ряда детерминистических процедур, что всегда гарантирует успех, например, расчет синуса угла или извлечение квадратного корня. Для такого типа задач легко разработать алгоритм, который будет однозначно выполняться. Но в реальной ситуации не все задачи допускают такое прямолинейное решение. Некоторые задачи можно решить только методом поиска решения. Для решения таких задач и используются алгоритмы искусственного интеллекта.

Для того чтобы понять, почему поиск так важен в ИИ, рассмотрим следующие моменты. Одним из первых исследований в области ИИ была разработка общего способа поиска решения. Такая программа поиска общего решения может найти решение для различных проблем, которые не имеют специфического, уже разработанного решения. И получение решений для таких проблем является крайне желательным. К сожалению, реализация программы поиска общего решения является крайне трудной, и несмотря на всю ее привлекательность, она по-прежнему остается проблематичной.

Некоторые трудности сопряжены с большими объемами и сложностью многих реальных ситуаций. Поскольку решение задачи поиска общего решения будет довольно сложным и комплексным, в настоящее время приоритетными являются отдельные специфические задачи поиска решения. И в этой главе нет амбициозной попытки найти решение задачи поиска общего решения для всех случаев, будут только рассмотрены некоторые проблемы поиска, которые имеют довольно широкое применение.

Введение и терминология

Предположим, что вы потеряли свои ключи от автомобиля. Наверняка вы знаете только то, что они где-то в вашем доме, схематический план которого показан ниже.

Вы начинаете поиск от входной двери (на плане обозначена значком X). Сначала вы просматриваете гостиную. Затем вы проходите в холл и ищете в первой спальне, возвращаетесь в холл и смотрите во второй спальне, и опять через холл заходите в главную спальную комнату. Ничего не найдя, вы идете обратно через гостиную и продолжаете поиск в кухне, где и находите ключи. Такую ситуацию легко представить с помощью графа, как показано на рис. 10.2. Представление проблемы поиска в графическом виде помогает решить задачу поиска и быстрее найти подходящее решение.

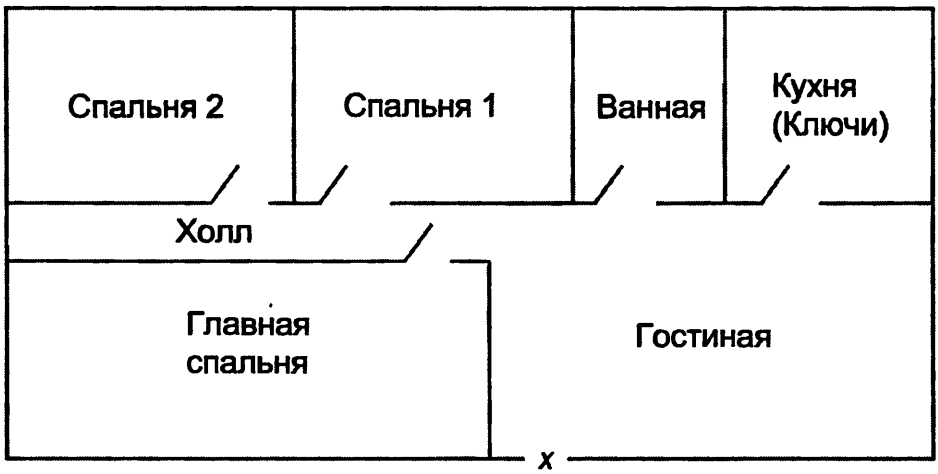


Рис. 10.1. Схематический план дома

Основываясь на предыдущих данных, введем термины, которые будем использовать на протяжении этой главы.

Узел	Отдельная точка
Конечный узел	Узел, в котором заканчивается путь
Пространство поиска	Множество всех узлов
Цель	Узел, который является объектом поиска
Эвристика	Предварительная информация об узле, имеющем преимущество перед другими узлами
Путь поиска	Ориентированный граф для узлов, приводящих к цели

В примере с поиском потерянных ключей, каждая комната в доме является узлом, весь дом является пространством поиска, целью поиска является кухня, а путь поиска показан с помощью графа (см. рис. 10.2). Спальни, кухня и ванная являются конечными узлами. Эвристика не представлена на графе, т.к. это только предположение, помогающее вам выбрать наилучший путь.

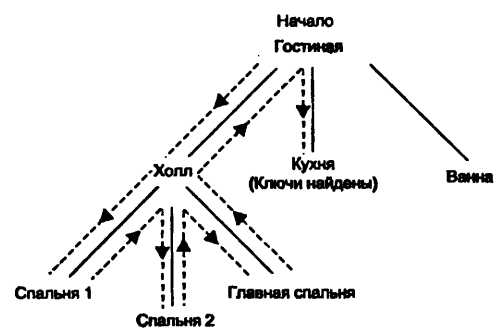


Рис. 10.2. Путь поиска потерянных ключей

Комбинаторный взрыв

Рассматривая предыдущий пример, можно подумать, что поиск решения довольно простая штука — вы просто начинаете последовательный поиск и продолжаете его до тех пор, пока не будет достигнут результат. В крайне простом случае с поиском ключей так и происходит, потому как пространство поиска очень маленькое. Но при решении большинства задач, особенно тех, для решения которых вы будете использовать компьютер, число узлов будет значительно больше и пространство поиска также будет большим, что приведет к огромному количеству путей поиска. Трудность заключается в том, что часто добавление всего одного узла к пространству поиска приводит к появлению многих дополнительных путей поиска. В результате количество потенциальных путей поиска цели возрастает нелинейно по сравнению с увеличением пространства поиска. В этом случае очень быстро увеличивается количество путей поиска.

Например, рассмотрим всего три объекта — A, B и C, различные взаимные расположения которых показаны в таблице ниже. Возможны шесть перестановок.

A	B	C
A	C	B
B	C	A
B	A	C
C	B	A
C	A	B

Вы можете легко убедиться, что располагая объекты A, B и C различными способами относительно друг друга, получаются только эти шесть перестановок. Однако, можно не делать перестановки, анализируя различные ситуации, а получить число различных перестановок с помощью формулы, которая используется в области математики, называемой *комбинаторикой*, где изучаются различные способы комбинирования объектов и подсчет количества комбинаций. В соответствии с этой формулой, количество способов, какими могут быть упорядочены N объектов, равно N! (N-факториал). Факториал подсчитывается путем перемножения последовательности чисел, каждое из которых на единицу меньше предыдущего, начиная с числа, равного количеству объектов, и кончая единицей. В нашем случае для трех объектов это будет 3*2*1, что равняется 6. Если взять четыре объекта, то это будет уже 24 (4*3*2). Для пяти объектов это будет 120, а для шести — 720. Для 1000 объектов это будет огромное число. На рис. 10.3 показан график роста количества перестановок. Такая ситуация называется комбинаторным взрывом. Поэтому если количество различных объектов превышает 10, то становится очень трудно анализировать все комбинации, а при значительном количестве объектов даже подсчет количества перестановок займет очень много времени.

Именно такая ситуация может случиться, когда при незначительном увеличении пространства поиска и количество путей поиска становится очень большим. Это будет комбинаторный взрыв, и поэтому только в простейших случаях допустимо использовать эвристический анализ. При эвристическом поиске производится обход всех узлов. Такой способ часто называют методом "грубой силы". Метод "грубой си-

лы" работает всегда, но часто его нельзя использовать, так как решение займет слишком много времени или потребует значительных машинных ресурсов, а обычно и того, и другого. Именно по этой причине стремительно развиваются технологии искусственного интеллекта.

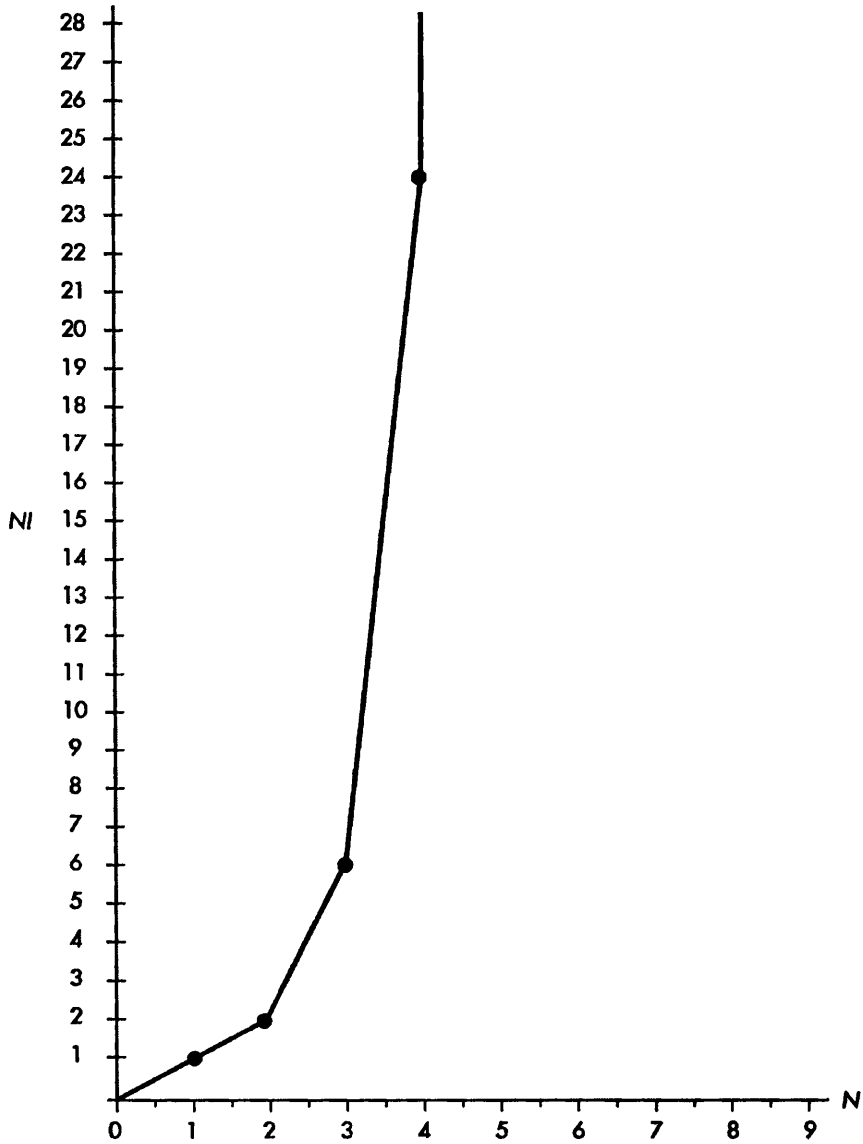


Рис. 10.3. Комбинаторный взрыв при расчете факториала

Метод поиска

Поиск решения можно выполнить несколькими путями. Основными из них являются:

- поиск вглубь;
- поиск в ширину;
- поиск экстремума;
- наименьшая стоимость.

Далее каждый из этих методов будет рассмотрен подробно.

Оценка поиска

Сравнительный анализ производительности различных методов поиска довольно сложен. В нашем случае будем использовать только два измерения.

- Как быстро может быть найдено решение?
- Насколько оптимально решение?

Существуют некоторые типы задач, для которых любое действие может рассматриваться как решение, в других случаях решение может быть найдено с минимальными усилиями. В этих случаях первое измерение особенно важно. В других ситуациях более важно качество решения.

Скорость поиска зависит как от размера пространства поиска, так и от числа узлов, просмотренных при нахождении решения. Поскольку последовательный перебор с возвратом из конечного узла является неэффективным, необходимо исключать такие варианты.

При поиске с использованием методов искусственного интеллекта может быть найдено наилучшее решение, а может быть, просто хорошее решение. Определение наилучшего решения требует полного последовательного поиска, потому что иногда это будет единственный способ узнать, что это решение наилучшее. Определение хорошего решения, наоборот, означает определение решения при ограниченном поиске. При этом не анализируется наличие наилучшего решения.

Как вы увидите, все технологии поиска, описанные в этой главе, в одних ситуациях работают лучше, чем в других, поэтому достаточно сложно сказать, что один метод всегда лучше другого. Но об отдельных методах можно сказать, что они наилучшие в большинстве случаев. Также четкая постановка задачи часто может помочь в выборе наилучшего метода.

Задача

Рассмотрим задачу, для решения которой будем использовать различные методы поиска. Предположим, что вы агент бюро путешествий и вам попался неприятный клиент, который хочет совершить полет из Нью-Йорка в Лос-Анджелес только воздушными линиями компании XYZ. Вы стараетесь объяснить клиенту, что эта компания не имеет прямых рейсов из Нью-Йорка в Лос-Анджелес, но клиент настаивает, что он доверяет только этой компании. Таким образом, вы

должны подобрать воздушный путь между Нью-Йорком и Лос-Анджелесом, который будет состоять из маршрутов, совершаемых этой компанией. Маршруты воздушных путей указаны в таблице 10.1 ниже.

Таблица 10.1. Маршруты воздушных путей

Путь	*Расстояние*
Нью-Йорк – Чикаго	900 миль
Чикаго – Денвер	1000 миль
Нью-Йорк – Торонто	500 миль
Нью-Йорк – Денвер	1800 миль
Торонто – Калгари	1700 миль
Торонто – Лос-Анджелес	2500 миль
Торонто – Чикаго	500 миль
Денвер – Урбана	1000 миль
Денвер – Хьюстон	1000 миль
Хьюстон – Лос-Анджелес	1500 миль
Денвер – Лос-Анджелес	1000 миль

Даже при быстром взгляде на таблицу можно понять, что клиент может долететь из Нью-Йорка в Лос-Анджелес маршрутами этих компаний. Задача состоит в том, чтобы написать программу на языке Java, которая выполнит то, что вы только что "сделали в своей голове".

Графическое представление

Информация о расписании полетов компаний XYZ может быть выражена с помощью направленного графа, показанного на рис. 10.4. Направленный граф — это обычный граф, в котором все узлы соединены с помощью линий со стрелками, указывающими направление движения. В направленном графе нельзя совершать движение против направления, указываемого стрелкой.

Для того, чтобы более наглядно представить всю картину, можно перерисовать граф в виде дерева, как показано на рис. 10.5. Именно на этот граф будем ссылаться в оставшейся части главы. Конечная точка пути — Лос-Анджелес (обведен линией). Обратите внимание, что отдельные города могут находиться в нескольких местах графа, что упрощает его исследование. Таким образом, представление графа в виде дерева не является двоичным деревом. Это сделано для наглядного представления.

Теперь все готово для начала разработки программ, которые будут искать путь между Нью-Йорком и Лос-Анджелесом.

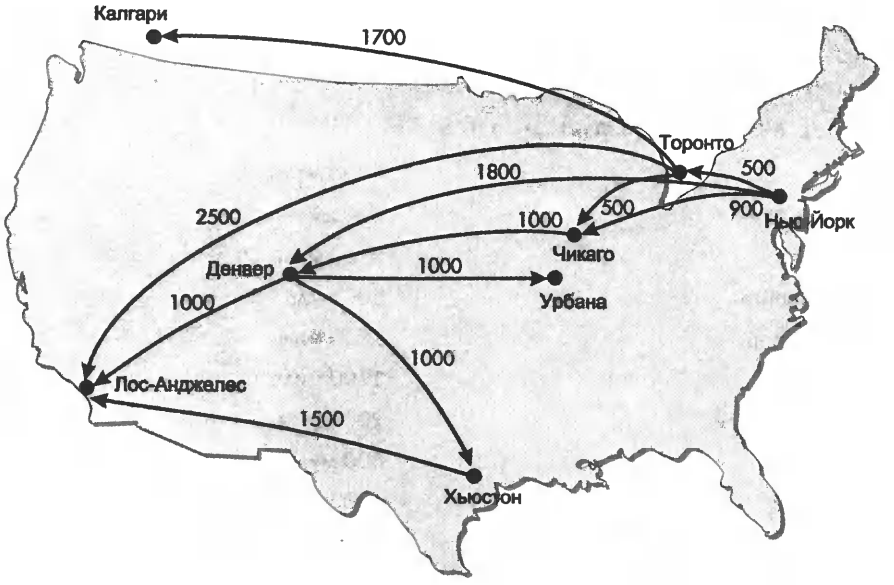


Рис. 10.4. Направленный граф расписания полетов компаний XYZ

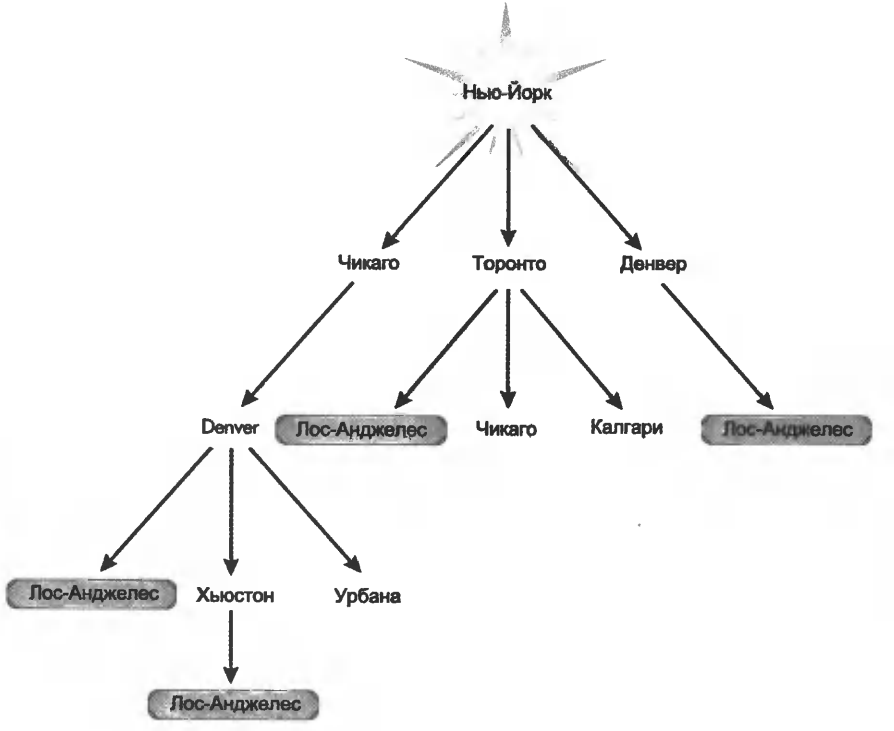


Рис. 10.5. Граф маршрутов в виде дерева

Класс FlightInfo

Написание программы поиска пути от Нью-Йорка до Лос-Анджелеса требует наличия базы данных с информацией о полетах компании XYZ. Каждый элемент базы данных должен содержать города отправления и прибытия, расстояния между ними и флаг, который указывает на возврат. Эта информация содержится в классе `FlightInfo`, показанном ниже.

```
// Информация о полетах.
class FlightInfo {
  String from;
  String to;
  int distance;
  boolean skip; // Используется для указания возврата.
  FlightInfo(String f, String t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
}
```

Этот класс будет использоваться во всех программах поиска, описанных в оставшейся части главы.

Поиск вглубь

При поиске вглубь выполняется прохождение каждого возможного пути до самого его конца, перед тем как будет сделан переход к следующему пути. Для более полного понимания работы этого метода, рассмотрите следующее дерево. В данном случае целью является узел F. При поиске вглубь прохождение графа будет выполнено в следующем порядке: ABDBEBACF. Если вы довольно хорошо знакомы с анализом деревьев, вы увидите, что в данном случае производился поиск с помощью симметричного обхода дерева. При этом обход совершается до тех пор, пока не будет обнаружен искомый узел или не будут достигнуты конечны узел. Если достигнут конечный узел, производится возврат на один уровень, затем производится переход вправо, а затем влево, пока искомый или конечный удел не будет достигнут. Эта процедура повторяется до тех пор, пока не будет исследовано все пространство поиска.

Как можно видеть, при поиске вглубь обязательно будет обнаружен искомый узел, поскольку производится последовательный обход всех возможных путей и в наихудшем случае это сводится к полному обходу всех возможных путей. В нашем случае наихудшим вариантом будет тот, когда искомым узлом будет узел, полученный при завершении полного поиска. То есть, в нашем случае наихудшим вариантом будет поиск узла G.

Поиск вглубь реализован в классе `Depth`, листинг которого представлен ниже.

```
class Depth {
  final int MAX = 100; // Максимальное количество соединений.
  // Этот массив содержит информацию о полетах.
  FlightInfo flights!] = new FlightInfo[MAX];
  int numFlights =0;   // Количество элементов массива.
  Stack btStack = new Stack(); // Стек возвратов.
  public static void main(String args[]) {
```

```
    String to, from;
    Depth ob = new Depth();
    BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in));
    ob.setup();
    try {
      System.out.print("From? ");
      from = br.readLine();
      System.out.print("To? "};
      to = br.readLine();
      ob.isflight(from, to);
      if(ob.btStack.size() != 0)
        ob.route(to);
    }
    catch (IOException exc) {
      System.out.println("Error on input.");
    }
  }
```

Создается массив объектов `FlightInfo` с именем `flights`, в котором сохраняется информация о полетах. Для задания размера этого массива используется неизменяемая переменная `MAX`. Количество маршрутов, реально сохраняемых в массиве, содержится в переменной `numFlights`. Необходимо отметить, что информацию о маршрутах можно сохранять и в одном из классов коллекций Java, например таком, как `ArrayList`. Это позволит получать списки неограниченного размера. Однако, поскольку нам необходимо сохранять информацию только о нескольких полетах, мы используем массив для получения более простого и понятного решения.

Объект типа `Stack` создается для получения информации о возвратах, и ссылка на него сохраняется в переменной `btStack`. Далее будет ясно, что стек возвратов является важным элементом в технологиях поиска.

Внутри метода `main()` происходит обращение к методу `setup()`, в котором производится инициализация информации о полетах. Затем вызывается метод `isflight()` для нахождения маршрута или создания пути между двумя городами, которыми в нашем примере будут являться города Нью-Йорк и Лос-Анджелес. Наконец, если путь между двумя городами найден, это отображается на экране. Сейчас рассмотрим отдельные фрагменты программы.

Метод `setup()` последовательно обращается к методу `addFlight()`, который добавляет маршрут в массив `flights`. Значение переменной `numFlights` инкрементируется с добавлением каждого нового маршрута. Таким образом, после окончания работы метода `setup()` значение переменной `numFlights` будет соответствовать количеству маршрутов в базе данных.

Методы `setup()` и `addFlight()` показаны ниже.

```
// Инициализировать базу данных.
void setup()
{
  addFlight("Нью-Йорк", "Чикаго", 900);
  addFlight("Чикаго", "Денвер", 1000);
  addFlight("Нью-Йорк", "Торонто", 500);
  addFlight("Нью-Йорк", "Денвер", 1800);
  addFlight("Торонто", "Калгари", 1700);
  addFlight("Торонто", "Лос-Анджелес", 2500);
  addFlight("Торонто", "Чикаго", 500);
  addFlight("Денвер", "Урбана", 1000);
  addFlight("Денвер", "Хьюстон", 1000);
```

```
  addFlight("Хьюстон", "Лос-Анджелес", 1500);
  addFlight("Денвер", "Лос-Анджелес", 1000); }
  // Поместить маршрут в базу данных.
  void addFlight(String from, String to, int dist)
  {
    if(numFlights < MAX) { flights[numFlights] =
        new FlightInfo(from, to, dist);
      numFlights++;
    }
    else System.out.println("Flight database full.\n");
}
```

Для определения пути между Нью-Йорком и Лос-Анджелесом, необходимо применять несколько дополнительных методов. Во-первых, это метод `match()`,который определяет, есть ли связь между двумя городами. Если связи между городами нет, возвращается значение нуль, если связь есть, возвращается расстояние между двумя городами. Листинг этого метода приведен ниже.

```
/* Если связь между городами есть, то возвращается расстояние
   между двумя городами, в противном случае
   возвращается значение 0. */
int match(String from, String to) {
  for(int i=numFlights-1; i > -1; i--) {
    if(flights[i].from.equals(from) &&
        flights[i].to.equals(to) && !flights[i].skip)
    {
      // Предупреждение повторного использования.
      flights[i].skip = true;
      return flights[i].distance;
    }
  }
  return 0; // Нет связи.
}
```

Следующий метод называется `find()`. Зная город отправления, в методе `find()` производится поиск в базе данных для создания маршрута. Если маршрут найден, возвращается объект `FlightInfo`, связанный с этим маршрутом. В противном случае возвращается `null`. Таким образом, различие между методами `match()` и `find()` заключается в том, что метод `match()` определяет наличие маршрута между двумя городами, тогда как метод `find()` определяет только наличие маршрута из данного города. Листинг метода `find()` приведен ниже.

```
// Найти маршрут из данного города.
FlightInfo find(String from)
{
  for(int i=0; i < numFlights; i++) {
    if(flights[i].from.equals(from) &&
        !flights[i].skip)
    {
      FlightInfo f = new FlightInfo(flights[i].from,
        flights[i].to, flights[i].distance);
      flights[i].skip = true; // Запретить повторное использование.
      return f;
    }
  }
  return null;
}
```

Как в методе `match()` и в методе `find()`, маршрут, в котором поле `skip` установлено в единицу, пропускается. А если маршрут найден, то соответствующему полю `skip` присваивается единица. Это необходимо для организации возврата из тупиковых ветвей и возможного повторного прохода уже найденных маршрутов.

Сейчас рассмотрим код, который действительно находит маршрут. Он включен в метод `isflight()` — ключевой алгоритм поиска маршрута между двумя городами. Метод вызывается с названиями городов отправления и назначения.

```
// Определение наличия маршрута между городами.
void isflight(String from, String to)
{
  int dist;
  FlightInfo f;
  // Проверить пункт назначения.
  dist = match(from, to);
  if(dist != 0) {
    btStack.push(new FlightInfo(from, to, dist));
    return;
  }
  // Попробовать другой маршрут.
  f = find(from); if(f != null) {
  btStack.push(new FlightInfo(from, to, f.distance));
  isflight(f.to, to);
  }
  else
    if (btStack.sizeO > 0) {
    // Вернуться и попробовать другой маршрут.
    f = (FlightInfo)btStack.pop();
    isflight(f.from, f.to);
  }
}
```

Рассмотрим этот метод более подробно. Сначала с помощью метода `match()` проверяется база данных маршрутов на наличие маршрута между пунктами `from` and `to`. Если маршрут есть, то цель поиска достигнута, маршрут помещается в стек и происходит возврат из стека. В противном случае используется метод `find()` для нахождения маршрута между пунктом отправления `from` и любым другим пунктом. Метод `find()` возвращает объект `FlightInfo`, в котором описывается найденный маршрут или значение `null`, если маршрута не существует. Если маршрут найден, то он сохраняется в переменной `f`, текущий маршрут помещается в стек и производится рекурсивный вызов метода `isflight()`, в который передается новый город отправления для параметра `f.to`. В противном случае перебор путей продолжается. Предыдущий узел удаляется из стека и производится рекурсивный вызов метода `isflight()`. Этот процесс продолжается до тех пор, пока не будет найдет полный маршрут от одного города до другого.

Например, если метод вызывается для городов Нью-Йорк и Чикаго, то при первом же определении маршрута метод `isflight()` закончит работу, поскольку между Нью-Йорком и Чикаго существует прямое воздушное сообщение. Ситуация будет более сложной, если для метода `isflight()` задать города Нью-Йорк и Калгари. В этом случае метод `isflight()` возвратит значение `null`, поскольку между городами Нью-Йорк и Калгари нет прямого сообщения. При этом делается попытка найти прямое сообщение между Нью-Йорком и каким-либо другим городом. Для чего вызывается метод `find()`, который находит маршрут между Нью-Йорком и Чикаго.

Этот маршрут будет помещен в стек возврата и будет произведен рекурсивный вызов метода `isflight()` с городом отправления (Чикаго). От Чикаго до Калгари также нет прямого сообщения и будет произведено несколько ошибочных поисков. В конце концов, после нескольких рекурсивных вызовов метода `isflight()` и возвратов, будет найден маршрут между Нью-Йорком и Торонто, а затем найден и маршрут между Торонто и Калгари. При этом будет производиться несколько рекурсивных возвратов из метода `isflight()` и будет составлен полный маршрут. Можно добавить метод `println()` для получения полной картины поиска в методе `isflight()`. Пустой стек говорит об отсутствии сообщения между заданными городами, с противном случае в стеке содержится решение задачи (помещен полный маршрут).

В общем, использование возвратов является основным элементом поиска во всех поисковых технологиях. Возвраты связаны с использованием рекурсии и стека возвратов. Все это похоже на обычные операции со стеком, когда первый помещенный в стек элемент извлекается последним. При поиске узлы последовательно помещаются в стек и при достижении конечной точки последний узел удаляется из стека, а также определяется новый путь от предыдущего узла. Этот процесс продолжается до тех пор, пока не будет найден нужный маршрут или не будут проверены все пути поиска.

Для полного решения задачи необходим еще один метод — `route()`. Этот метод отображает маршрут и общую дистанцию. Листинг метода `route()` приведен ниже.

```
// Показать маршрут и дистанцию.
void route(String to)
{
  Stack rev = new Stack();
  int dist = 0;
  FlightInfo f;
  int num = btStack.size();
  // Сделать реверс стека для отображения маршрута.
  for(int i=0; i<num; i++)
    rev.push(btStack.pop());
  for(int i=0; i<num; i++) {
    f = (FlightInfo)rev.pop();
    System.out.print(f.from + " to ");
    dist += f.distance;
  }
  System.out.println(to);
  System.out.println("Distance is " + dist);
}
```

Обратит внимание, что используется еще один стек с именем `rev`. Полученное решение сохраняется в стеке `btStack` в обратном порядке — в вершине стека содержатся последние маршруты, а внизу размещаются первые маршруты. Таким образом, необходимо использовать обратный путь для получения правильного порядка маршрутов. Для получения решения в прямом порядке, маршруты последовательно извлекаются из стека `btStack` и помещаются в стек `rev`.

Полная программа поиска вглубь представлена ниже.

```
// Нахождение пути при поиске вглубь.
import java.util.*;
import java.io.*;

// Информация о полетах.
class FlightInfo {
  String from;
```

```
  String to;
  int distance;
  boolean skip; // Используется для возврата.

  FlightInfo(String f, String t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
}

class Depth {
  final int MAX = 100;

  // Этот массив содержит информацию о полетах.
  FlightInfo flights[] = new FlightInfo[MAX];
  int numFlights = 0; // Количество элементов массива.
  Stack btStack = new Stack(); // Стек возвратов.
  public static void main(String args[])
  {
    String to, from;
    Depth ob = new Depth();
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    ob.setup();
    try {
      System.out.print("From? ");
      from = br.readLine();
      System.out.print("To? ");
      to = br.readLine();
      ob.isflight(from, to);
      if(ob.btStack.size() != 0)
        ob.route(to);
    } catch (IOException exc) {
      System.out.println("Error on input.");
    }
  }

  // Инициализировать базу данных о маршрутах.
  void setup()
  {
    addFlight("Нью-Йорк", "Чикаго", 900);
    addFlight("Чикаго", "Денвер", 1000);
    addFlight("Нью-Йорк", "Торонто", 500);
    addFlight("Нью-Йорк", "Денвер", 1800);
    addFlight("Торонто", "Калгари", 1700);
    addFlight("Торонто", "Лос-Анджелес", 2500);
    addFlight("Торонто", "Чикаго", 500);
    addFlight("Денвер", "Урбана", 1000);
    addFlight("Денвер", "Хьюстон", 1000);
    addFlight("Хьюстон", "Лос-Анджелес", 1500);
    addFlight("Денвер", "Лос-Анджелес", 1000);
  }

  // Поместить маршруты в базу данных.
  void addFlight(String from, String to, int dist)
  {
    if(numFlights < MAX) {
      flights[numFlights] =
        new FlightInfo(from, to, dist);
```

```
      numFlights++;
    }
    else System.out.println("Flight database full.\n");
  }

  // Показать маршрут и общее расстояние.
  void route(String to)
  {
    Stack rev = new Stack();
    int dist = 0;
    FlightInfo f;
    int num = btStack.size();

    // Реверс стека для отображения маршрута.
    for(int i=0; i < num; i++)
      rev.push(btStack.pop());
    for(int i=0; i < num; i++) {
      f = (FlightInfo) rev.pop();
      System.out.print(f.from + " to ");
      dist += f.distance;
    }
    System.out.println(to);
    System.out.println("Distance is " + dist);
  }

  /* Если маршрут найден, то возвратить расстояние
     от города отправления до города прибытия,
     В противном случае возвратить 0. */
  int match(String from, String to)
  {
    for(int i=numFlights-1; i > -1; i--) {
      if(flights[i].from.equals(from) &&
         flights[i].to.equals(to) &&
         !flights[i].skip)
      {
        flights[i].skip = true; // Запретить повторное использование.
        return flights[i].distance;
      }
    }

    return 0; // Не найдено.
  }

  // Найти любой маршрут.
  FlightInfo find(String from)
  {
    for(int i=0; i < numFlights; i++) {
      if(flights[i].from.equals(from) &&
         !flights[i].skip)
      {
        FlightInfo f = new FlightInfo(flights[i].from,
                       flights[i].to, flights[i].distance);
        flights[i].skip = true; // Запретить повторное использование.
        return f;
      }
    }
    return null;
  }

  // Определить, есть ли маршрут между from и to.
  void isflight(String from, String to)
```

```
  {
    int dist;
    FlightInfo f;

    // Определение города прибытия.
    dist = match(from, to);
    if(dist != 0) {
      btStack.push(new FlightInfo(from, to, dist));
      return;
    }

    // Попробовать другой маршрут.
    f = find(from);
    if(f != null) {
      btStack.push(new FlightInfo(from, to, f.distance));
      isflight(f.to, to);
    }
    else if(btStack.size() > 0) {
      // Возвратиться и попробовать другой маршрут.
      f = (FlightInfo) btStack.pop();
      isflight(f.from, f.to);
    }
  }
}
```

Обратите внимание, что в методе `main()` производится запрос как для города отправления, так и для города прибытия. Это означает, что программу можно использовать для поиска маршрута между любыми городами. Однако, в оставшейся части главы предполагается, что Нью-Йорк является городом отправления, а Лос-Анджелес — городом прибытия.

Когда производится запуск программы с городом отправления Нью-Йорк и с городом прибытия Лос-Анджелес, отображается следующий маршрут.

```
From? Нью-Йорк
To? Лос-Анджелес
Нью-Йорк to Чикаго to Денвер to Лос-Анджелес
Distance is 2900
```

Как можно видеть из рис. 10.6, это решение действительно может быть найдено при поиске вглубь. Это также будет и наилучшим решением.

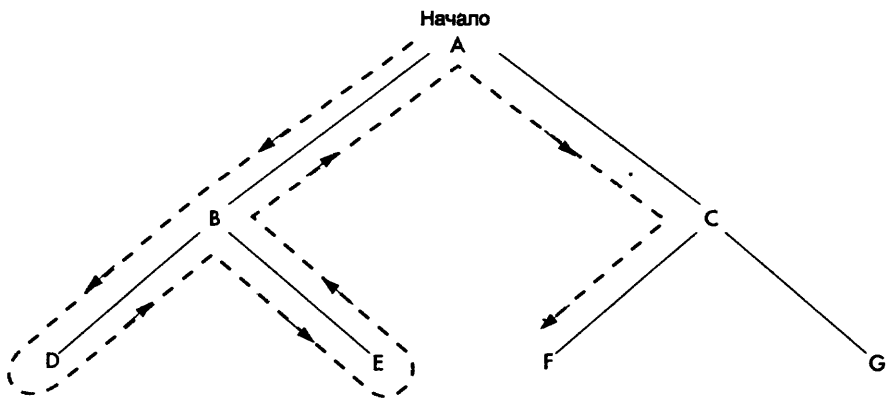


Рис. 10.6. Решение при поиске вглубь

Анализ поиска вглубь

Поиск вглубь обеспечил определение хорошего решения. Причем в нашем специфическом случае поиск вглубь позволил найти решение с первой попытки, без использования возвратов. Это очень удачно, но при любом другом расположении городов, решение может быть найдено только при неоднократном использовании возвратов. Таким образом, нельзя делать обобщающие выводы только на основании этого решения. Производительность метода поиска вглубь может быть очень низкой, когда производится поиск с возвратом по очень длинным ветвям. В этом случае производится возврат не только по тем ветвям, где нет города прибытия, но и по ветви с городом прибытия.

Поиск в ширину

Вместо метода поиска вглубь можно использовать *метод поиска в ширину*. При этом методе поиска каждый узел одного и того же уровня проверяется перед переходом к узлам более глубокого уровня. Именно такой метод поиска приведен ниже (рис. 10.7), где узел C является целевым узлом.

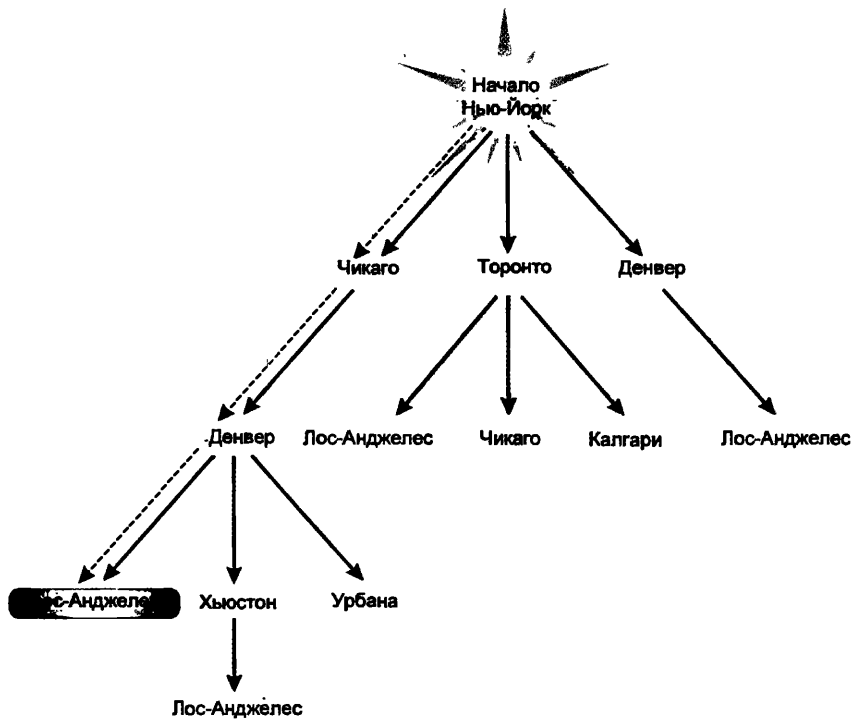


Рис. 10.7. Определение маршрута при поиске в ширину

Используя термины двоичных структурированных деревьев, легко описать последовательность действий при поиске в ширину. Но многие пространства поиска, включая и

наш пример с маршрутами полетов, не являются двоичными деревьями. Поэтому точнее будет сказать, что подход к реализации метода поиска в ширину в нашем случае будет несколько субъективным. При определении нужного маршрута полета по методу поиска в ширину, производится последовательная проверка всех возможных маршрутов для города отправления к городам прибытия. Другими словами, перед тем как перейти на более глубокий уровень производится проверка всех возможных маршрутов текущего уровня.

Для того чтобы создать программу метода поиска в ширину, необходимо разработать альтернативный метод `isflight()`, как показано ниже.

```
/* Определить, есть ли путь между from и to, используя
   поиск в ширину. */
void isflight(String from, String to)
{
  int dist, dist2;
  FlightInfo f;
  // Этот стек необходим при поиске в ширину.
  Stack resetStck = new Stack();
  // Просмотр наличия пункта назначения.
  dist = match(from, to);
  iffdist != 0) {
    btStack.push(new FlightInfo(from, to, dist));
    return;
  }

  /* Фрагмент ниже является первой модифицированной частью
     для поиска в ширину. Здесь производится проверка всех
     маршрутов из заданного узла. */
  while((f = find(from)) != null) {
    resetStck.push(f);
    if((dist = match(f.to, to)) != 0) {
      resetStck.push(f.to);
      btStack.push(new FlightInfo(from, f.to, f.distance))
      btStack.push(new FlightInfo(f.to, to, dist));
      return;
    }
  }
  /* В следующем коде производится переустановка полей skip,
     установленных в предыдущем цикле. Эта часть также
     модифицирована для поиска в ширину. */
  int i = resetStck.size();
  for(; i!=0; i--)
  resetSkip((FlightInfo) resetStck.pop());
  // Попробовать другой маршрут.
  f = find(from);
  if(f != null) {
    btStack.push(new FlightInfo(from, to, f.distance));
    isflight(f.to, to);
  }
  else if (btStack.sizeO > 0) {
    // Вернуться и попробовать другой маршрут.
    f = (FlightInfo) btStack.pop();
    isflight(f.from, f.to);
  }
```

Были сделаны два изменения. Во-первых, в цикле `while` производится проверка всех маршрутов из города отправления (`from`) до города прибытия. Во-вторых, если нужный маршрут не найден, то поля `skip` для этих маршрутов сбрасываются с помощью метода `resetSkip()`. Маршрут, который необходимо переустановить, со-

храняется в стеке `resetStck`, который является локальным для метода `isflight()`. Сбрасывание флагов необходимо для получения альтернативных путей, которые могут использовать эти маршруты.

Метод приведен ниже.

```
// переустановка полей skip указанного маршрута.
void resetSkip(FlightInfo f) {
  for(int i=0; i<numFlights; i++)
    if(flights[i].from.equals(f.from) &&
               flights[i].to.equals(f.to))
      flights[i].skip = false;
}
```

Для получения программы для определения пути по методу поиска в ширину на основе программы поиска вглубь, замените метод `isflight()` на модифицированный и добавьте метод `resetSkip()`. После запуска программы будет получено следующее решение.

```
From? Нью-Йорк
To? Лос-Анджелес
Нью-Йорк to Торонто to Лос-Анджелес
Distance is 3000
```

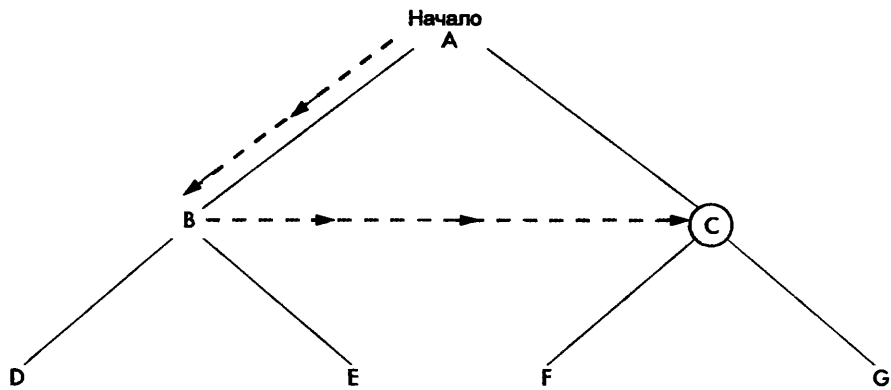


Рис. 10.8. Определение пути с помощью метода поиска в ширину

Анализ поиска в ширину

В описанном примере определение пути с помощью метода поиска в ширину выполняется довольно успешно, при этом находится довольно приемлемый маршрут. Но как и для предыдущего метода поиска, нельзя делать обобщения только на основе этого примера, так как качество работы метода зависит от физической организации информации. Довольно часто пути, найденные по методу поиска вглубь и поиска в ширину, оказываются различными несмотря на одно и то же пространство поиска.

Метод поиска в ширину работает хорошо, когда узел назначения находится на одном из верхних уровней, и выполняет поиск довольно долго, если для достижения узла назначения приходится проходить значительное количество уровней. В этом случае расходуются значительные временные ресурсы на проход маршрутов и возвраты.

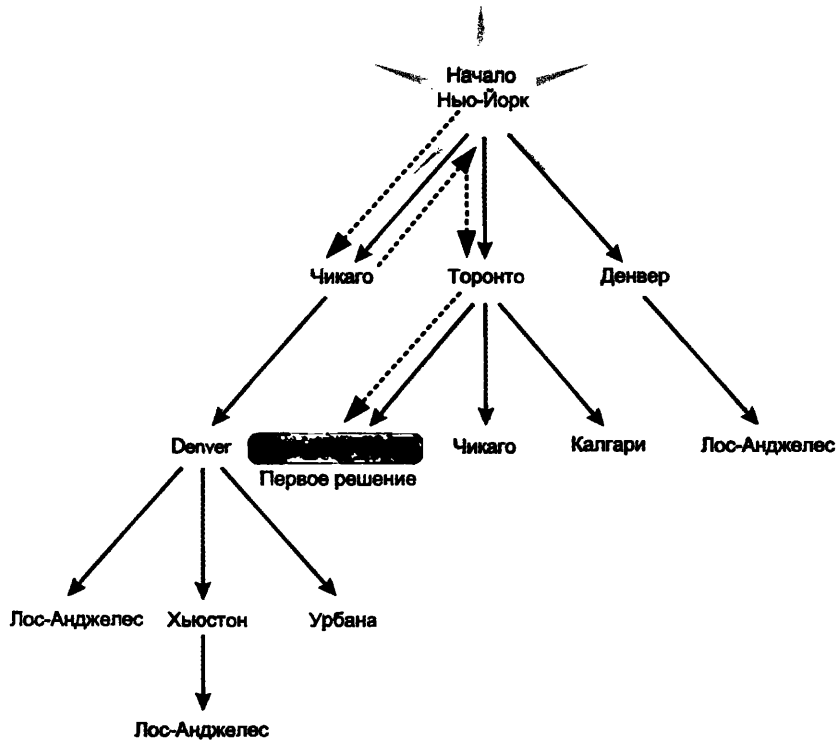


Рис. 10.9. Определение решения с помощью метода поиска в ширину

Добавляем эвристику

Ни в методе поиска вглубь, ни в методе поиска в ширину не делается никаких предположений относительно того, почему один узел предпочтительнее другого и почему лучше начинать поиск с этого узла, а не с другого. Вместо этого просто производится последовательный перебор узлов до тех пор, пока не будет обнаружен узел назначения. Это может быть наилучший способ поиска при отсутствии дополнительной информации, но часто пространство поиска может содержать информацию, которую можно использовать для получения наиболее предпочтительного пути поиска и тем самым значительно повысить скорость поиска. Чтобы использовать все преимущества, которые может дать такая предварительная информация, необходимо добавить блок обработки эвристической информации.

Эвристическая информация — это просто набор правил, на основании которых возрастает вероятность поиска в нужном направлении. Например, вообразите, что вы потерялись в лесу и вам сильно хочется пить. Лес довольно густой и нет возможности просмотреть окрестности на большое расстояние. Но вы должны знать, что реки, ручейки и ключи, вероятнее всего, находятся в низине, что животные часто прокладывают тропы к источникам воды и что недалеко от воды можно почувствовать свежесть воздуха или услышать журчание воды. Поэтому вы выбираете направление поиска вниз по склону, поскольку, вероятнее всего, вода находится имен-

но там. Затем вы находите тропинку, которая также ведет вниз, и стараетесь ее придерживаться, так как она может привести к воде. Вы начинаете прислушиваться к различным шумам в надежде услышать шум бегущей воды. Вы также пытаетесь почувствовать повышение влажности воздуха и наконец находите воду, которую можно пить. В этой ситуации вы использовали эвристическую информацию для поиска воды. И хотя не было абсолютной уверенности, что вы найдете воду, эта вероятность значительно возросла. В общем случае наличие эвристической информации быстрее приводит к цели поиска. Чаще всего эвристические методы поиска основаны на поиске максимумов или минимумов некоторых критериев. В нашем случае поиска маршрута между Нью-Йорком и Лос-Анджелесом пассажир может использовать два критерия для минимизации. Первый критерий — это уменьшение количества пересадок, и второй критерий — уменьшение длины маршрута. При этом кратчайший маршрут не всегда будет с минимальным количеством пересадок и наоборот. В данном разделе рассмотрим два метода поиска с использованием эвристических данных. Сначала минимизируем число пересадок, а затем выберем самый кратчайший путь. Оба эвристических метода будем использовать при поиске вглубь.

Метод поиска экстремума

Алгоритм поиска, при котором производится попытка сократить количество пересадок, предполагает, что чем больше длина одного маршрута, тем больше вероятность того, что количество пересадок будет меньше. Или (используя термины искусственного интеллекта) это будет пример поиска экстремума.

В алгоритме поиска экстремума очередной шаг по поиску следующего узла производится из принципа его максимального приближения к цели (в нашем случае это будет самый длинный маршрут). Можно провести некоторую аналогию с путешественником, который затерялся в темноте на полпути в лагерю, который находится на вершине холма. Каждый шаг путешественника вверх по холму ведет его в правильном направлении.

Обрабатывая только информацию, содержащуюся в расписании полетов, можно реализовать метод поиска экстремума в программе поиска пути между двумя городами. При поиске экстремума выбирается маршрут, который как можно дальше удален от пункта отправления (в предположении, что при этом будет достигнуто наилучшее приближение к оптимальному маршруту). Для того чтобы это сделать, модифицируется метод `find()`, как показано ниже.

```
// Поиск наиболее длинного маршрута.
FlightInfo find(String from)
{
  int pos = -1;
  int dist = 0;
  for(int i=0; i<numFlights; i++) {
    if(flights[i].from.equals(from) && !flights[i].skip) {
      // Использовать наиболее протяженный маршрут.
      if(flights[i].distance > dist) {
        pos = i;
        dist = flights[i].distance;
      }
    }
  }
  if(pos != -1) {
```

```
    flights[pos].skip = true; // Запретить повторное использование.
    FlightInfo f = new FlightInfo(flights[pos].from,
        flights[pos].to, flights[pos].distance);
    return f;
  }
  return null;
}
```

Метод `find()` производит поиск в базе данных, определяя наиболее удаленный маршрут от города отправления.

Полностью программа определения пути по методу поиска экстремума будет такой.

```
// Определение пути по методу поиска экстремума.
import java.util.*;
import java.io.*;

// Информация о полетах.
class FlightInfo {
  String from;
  String to;
  int distance;
  boolean skip; // Используется при возвратах.

  FlightInfo(String f, String t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
}

class Hill {
  final int MAX = 100;

  // Этот массив содержит информацию о полетах.
  FlightInfo flights[] = new FlightInfo[MAX];
  int numFlights = 0;          // Количество элементов массива.
  Stack btStack = new Stack(); // Стек возвратов.
  public static void main(String args[])
  {
    String to, from;
    Hill ob = new Hill();
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    ob.setup();
    try {
      System.out.print("From? ");
      from = br.readLine();
      System.out.print("To? ");
      to = br.readLine();
      ob.isflight(from, to);
      if(ob.btStack.size() != 0)
        ob.route(to);
    } catch (IOException exc) {
      System.out.println("Error on input.");
    }
  }

  // Инициализация базы данных о маршрутах.
  void setup()
  {
```

```
    addFlight("Нью-Йорк", "Чикаго", 900);
    addFlight("Чикаго", "Денвер", 1000);
    addFlight("Нью-Йорк", "Торонто", 500);
    addFlight("Нью-Йорк", "Денвер", 1800);
    addFlight("Торонто", "Калгари", 1700);
    addFlight("Торонто", "Лос-Анджелес", 2500);
    addFlight("Торонто", "Чикаго", 500);
    addFlight("Денвер", "Урбана", 1000);
    addFlight("Денвер", "Хьюстон", 1000);
    addFlight("Хьюстон", "Лос-Анджелес", 1500);
    addFlight("Денвер", "Лос-Анджелес", 1000);
  }

  // Разместить маршрут в базе данных.
  void addFlight(String from, String to, int dist)
  {
    if(numFlights < MAX) {
      flights[numFlights] =
        new FlightInfo(from, to, dist);
      numFlights++;
    }
    else System.out.println("Flight database full.\n");
  }

  // Показать маршрут и суммарную дистанцию.
  void route(String to)
  {
    Stack rev = new Stack();
    int dist = 0;
    FlightInfo f;
    int num = btStack.size();

    // Сделать реверс стека для отображения пути.
    for(int i=0; i < num; i++)
      rev.push(btStack.pop());
    for(int i=0; i < num; i++) {
      f = (FlightInfo) rev.pop();
      System.out.print(f.from + " to ");
      dist += f.distance;
    }
    System.out.println(to);
    System.out.println("Distance is " + dist);
  }

  /* Если между пунктами from и to существует маршрут,
     то возвратить расстояние.
     В противном случае возвратить 0. */
  int match(String from, String to)
  {
    for(int i=numFlights-1; i > -1; i--) {
      if(flights[i].from.equals(from) &&
         flights[i].to.equals(to) &&
         !flights[i].skip)
      {
        flights[i].skip = true; // Запретить повторное использование.
        return flights[i].distance;
      }
    }
    return 0; // Не найдено.
  }
```

```
  // Выбрать наиболее длинный маршрут.
  FlightInfo find(String from)
  {
    int pos = -1;
    int dist = 0;
    for(int i=0; i < numFlights; i++) {
      if(flights[i].from.equals(from) &&
         !flights[i].skip)
      {
        if(flights[i].distance > dist) {
          pos = i;
          dist = flights[i].distance;
        }
      }
    }
    if(pos != -1) {
      flights[pos].skip = true; // Запретить повторное использование.
      FlightInfo f = new FlightInfo(flights[pos].from,
                                    flights[pos].to,
                                    flights[pos].distance);
      return f;
    }
    return null;
  }

  // Проверить наличие маршрута между from и to.
  void isflight(String from, String to)
  {
    int dist;
    FlightInfo f;

    // Проверить наличие пункта прибытия.
    dist = match(from, to);
    if(dist != 0) {
      btStack.push(new FlightInfo(from, to, dist));
      return;
    }

    // Попробовать другой маршрут.
    f = find(from);
    if(f != null) {
      btStack.push(new FlightInfo(from, to, f.distance));
      isflight(f.to, to);
    }
    else if(btStack.size() > 0) {
      // Возврат и попробовать другой маршрут.
      f = (FlightInfo) btStack.pop();
      isflight(f.from, f.to);
    }
  }
}
```

После запуска программы на выполнение получим следующее решение.

```
From? Нью-Йорк
To? Лос-Анджелес
Нью-Йорк to Денвер to Лос-Анджелес
Distance is 2800
```

Это очень хорошее решение. Полный маршрут содержит минимальное количество пересадок и является наиболее коротким маршрутом. Таким образом, найдено наилучшее решение.

Однако, если не существует маршрута между Денвером и Лос-Анджелесом, решение не будет таким хорошим. Полный маршрут будет от Нью-Йорка в Денвер, затем в Хьюстон и только затем в Лос-Анджелес, и суммарное расстояние будет 4300 миль. В этом случае будет найдет ложный пик, поскольку маршрут в Хьюстон не приводит нас ближе к цели. На рис. 10.10 показано как первое решение, так и ложный пик.

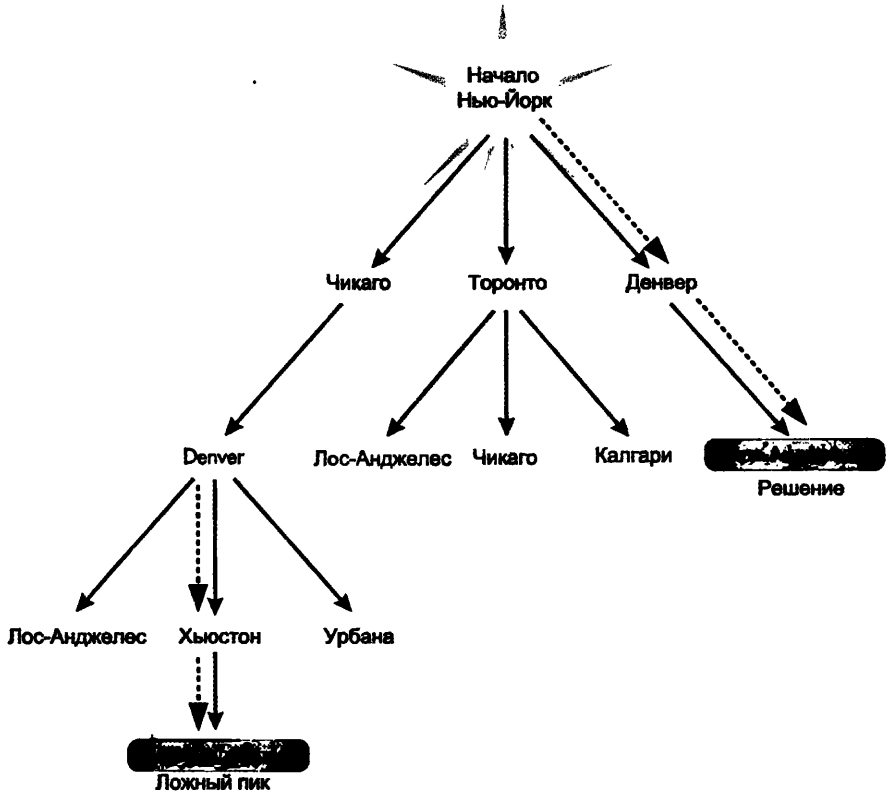


Рис. 10.10. Решение задачи с помощью нахождения максимума и ложный максимум

Анализ метода поиска экстремума

Действительно, метод поиска экстремума дает удачное решение во многих случаях, поскольку при этом уменьшается количество узлов, которое необходимо пройти для поиска решения. Однако ему присущи и некоторые слабые места. Во -первых, это проблема ложного пика, о которой только что было сказано. Во-вторых, это проблема плоского участка, когда все последующие шаги одинаково хороши (равновероятны). В этом случае нахождение решения с использованием поиска экстремума не лучше, чем обычный поиск вглубь. Последняя проблема заключается в том, что использование метода поиска экстремума не всегда обеспечивает высокую производительность, так как возникает несколько пересекающихся маршрутов при возвратах. Но несмотря на эти потенциальные трудности, метод поиска экстремума часто повышает вероятность поиска хорошего решения.

Поиск по критерию наименьшей стоимости

В противоположность методу поиска экстремума, существует метод маршрутизации по критерию наименьшей стоимости. Стратегию критерия наименьшей стоимости можно почувствовать, находясь на дороге, спускающейся с холма, когда вы одеваете роликовые коньки. Вы хорошо понимаете, что спускаться с холма значительно легче, чем подниматься на него. Другими словами, при маршрутизации по критерию наименьшей стоимости выбирается маршрут с наименьшими трудностями.

Применяя метод маршрутизации по критерию наименьшей стоимости к проблеме поиска пути, в любом случае выбирается кратчайший маршрут, и поэтому велика вероятность того, что полный маршрут будет самым коротким. В отличие от метода поиска экстремума, когда производится попытка минимизировать количество пересадок, при маршрутизации по критерию наименьшей стоимости минимизируется количество миль.

Для использования метода маршрутизации по критерию наименьшей стоимости необходимо вновь изменить метод `find()`, как показано в листинге.

```
// Поиск кратчайшего маршрута.
FlightInfo find(String from)
{
  int pos = -1;
  int dist = 10000; // Длиннее, чем самый длинный маршрут.
  for(int i=0; i<numFlights; i++) {
    if(flights[i].from.equals(from) &&
      !flights[i].skip)
    {
      // Использовать кратчайший маршрут.
       if(flights[i].distance < dist) {
         pos = i;
         dist = flights[i].distance;
       }
    }
  }
  if(pos != -1) {
    flights[pos].skip = true; // Запретить повторное использование.
    FlightInfo f = new FlightInfo(flights[pos].from,
    flights[pos].to, flights[pos].distance);
    return f;
  }
  return null;
}
```

Используя эту версию метода `find()`, получим следующее решение.

```
From? Нью-Йорк
To? Лос-Анджелес
Нью-Йорк to Торонто to Лос-Анджелес
Distance is 3000
```

Как можно видеть, найдено хорошее решение, но не самое лучшее. На рис. 10.11 показан поиск с использованием метода маршрутизации по критерию наименьшей стоимости.

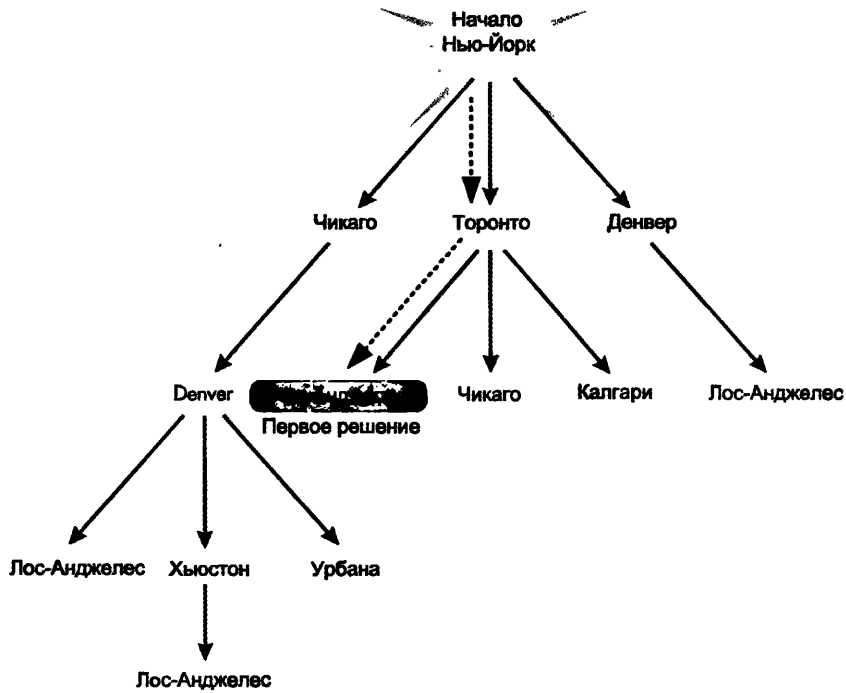


Рис. 10.11. Получение решения с использованием метода маршрутизации по критерию наименьшей стоимости

Анализ поиска по критерию наименьшей стоимости

Поиск по критерию наименьшей стоимости имеет те же преимущества и недостатки, что и метод поиска экстремума, но с противоположным эффектом. Здесь также могут быть ложные долины, низины и узкие места. В нашем специфическом случае поиск по критерию наименьшей стоимости работает так же хорошо, как и метод поиска экстремума.

Поиск кратных решений

Иногда полезно найти несколько решений одной и той же проблемы. Это не будет определением всех решений (полный поиск). Вместо этого несколько решений позволят иметь репрезентативный набор для заданного пространства поиска.

Существует несколько способов для создания кратных решений, но только два из них будут исследоваться ниже. Первый способ — это удаление маршрутов, второй — удаление узлов.

Как следует из названий способов, получение нескольких решений без избыточности требует, чтобы уже найденное решение было удалено. Напомню, что ни в одном из этих способов не делается попытка найти все решения. Нахождение всех решений, это совсем другая задача, для выполнения которой нужно провести полный обход всех возможных путей.

Удаление маршрутов

При использовании метода удаления маршрутов можно получить несколько решений, причем после получения первого решения удаляются узлы из базы данных, которые формировали полученный маршрут и делается попытка найти следующее решение. По существу, при методе удаления маршрутов отсекаются отдельные ветви дерева. Для получения решений с использованием метода удаления маршрутов, необходимо только изменить метод `main()` для поиска вглубь, как показано на листинге ниже, и изменить имя класса для поиска на `PathR`.

```
public static void main(String args[]) {
  String to, from;
  PathR ob = new PathR() ;
  BufferedReader br = new
  BufferedReader(new inputStreamReader(System.in));
  boolean done = false;
  ob.setup();
  try {
    System.out.print("From? ");
    from = br.readLine();
    System.out.print("To? ");
    to = br.readLine(); do {
      ob.isflight(from, to);
      if(ob.btStack.size() == 0) done = true;
      else {
        ob.route(to);
        ob.btStack = new Stack();
      }
    } while(!done);
  } catch (IOException exc) {
    System.out.println("Error on input.");
  }
}
```

Как видно из листинга, дополнительно добавлен цикл do для проведения итераций, пока стек возвратов не станет пустым. Вспомните, что когда стек возвратов станет пустым, никаких решений (в нашем случае дополнительных маршрутов) не может быть найдено. Больше не нужно выполнять никаких модификаций, поскольку все маршруты, являющиеся частью решения, будут иметь установленное поле `skip`. Поэтому такой маршрут больше не будет найден с помощью метода `find()` и не будет частью следующего решения. Разумеется, новый стек возвратов должен быть создан для получения следующего решения.

Используя метод удаления маршрутов, можно получить следующие решения.

```
From? Нью-Йорк
To? Лос-Анджелес
Нью-Йорк to Чикаго to Денвер to Лос-Анджелес
Distance is 2900
Нью-Йорк to Торонто to Лос-Анджелес
Distance is 3000
Нью-Йорк to Денвер to Хьюстон to Лос-Анджелес
Distance is 4300
```

При этом поиске найдено три пути. Обратите внимание, что ни одно из этих решений не является наилучшим.

Удаление узлов

Второй способ нахождения дополнительных решений — удаление узлов. При этом удаляется последний узел текущего решения и производится повторный поиск. Для этого изменяется метод `main()` таким образом, чтобы в нем удалялся последний маршрут из базы данных, сбрасывались все поля `skip` и создавался новый, пустой стек возвратов для следующего решения. Последний маршрут предыдущего решения удаляется из базы данных полетов с помощью нового метода с именем `remove()`. Все поля `skip` переустанавливаются в помощью еще одного вновь введенного метода `resetAllSkip()`. Измененный метод `main()` с вновь введенными методами `remove()` и `resetAllSkip()` приведен в листинге.

```
public static void main(String args[])
{
  String to, from;
  NodeR ob = new NodeR();
  BufferedReader br = new
  BufferedReader(new InputStreamReader(System.in));
  boolean done = false;
  FlightInfo f;
  ob.setup();
  try {
    System.out.print("From? ");
    from = br.readLine();
    Systern.out.print("To? ");
    to = br.readLine();
    do {
      ob.isflight(from, to);
      if(ob.btStack.size() == 0)
        done = true;
      else {
        // Сохранить маршрут в стеке.
        f = (FlightInfo) ob.btStack.peek();
        ob.route(to); // Отобразить текущий путь.
        ob. resetAHSkip(); // Переустановить все поля skip.

        /* Удалить последний маршрут предыдущего пути
           из базы данных маршрутов. */
        ob.remove(f);

        // Переустановить стек возвратов.
        ob.btStack = new Stack();
      }
    } while(!done);
  }
  catch (IOException exc) {
    System.out.println("Error on input.");
  }
}

// Переустановить все поля skip.
void resetAllSkip() {
  for(int i=0; i<numFlights; i++)
    flights[i].skip = false;
}

// Удалить маршрут.
void remove(FlightInfo f) {
```

```
  for(int i=0; i<numFlights; i++)
    if(flights[i].from.equals(f.from) &&
            flights[i].to.equals(f.to))
      flights[i].from = "";
}
```

Как можно видеть, удаление маршрута выполняется с помощью присваивания пустой строки для названия города отправления. Поскольку требуется достаточно много изменений, программа поиска решений с удалением узлов приведена ниже полностью.

```
// Нахождение нескольких решений с помощью удаления узлов.
import java.util.*;
import java.io.*;

// Информация о полетах.
class FlightInfo {
  String from;
  String to;
  int distance;
  boolean skip; // Используется при возвратах.
  FlightInfo(String f, String t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
}

class NodeR {
  final int MAX = 100;

  // Этот массив содержит информацию о полетах.
  FlightInfo flights[] = new FlightInfo[MAX];
  int numFlights = 0; // Количество элементов массива.
  Stack btStack = new Stack(); // Стек возвратов.
  public static void main(String args[])
  {
    String to, from;
    NodeR ob = new NodeR();
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    boolean done = false;
    FlightInfo f;
    ob.setup();
    try {
      System.out.print("From? ");
      from = br.readLine();
      System.out.print("To? ");
      to = br.readLine();
      do {
        ob.isflight(from, to);
        if(ob.btStack.size() == 0) done = true;
        else {
          // Сохранить маршрут в стеке.
          f = (FlightInfo) ob.btStack.peek();
          ob.route(to); // Отобразить текущий путь.
          ob.resetAllSkip(); // Переустановить все поля skip.
          /* Удалить последний маршрут предыдущего пути
             из базы данных маршрутов. */
          ob.remove(f);
          // Переустановить стек возвратов.
```

```
        ob.btStack = new Stack();
      }
    } while(!done);
  } catch (IOException exc) {
    System.out.println("Error on input.");
  }
}

// Инициализировать базу данных полетов.
void setup()
{
  addFlight("Нью-Йорк", "Чикаго", 900);
  addFlight("Чикаго", "Денвер", 1000);
  addFlight("Нью-Йорк", "Торонто", 500);
  addFlight("Нью-Йорк", "Денвер", 1800);
  addFlight("Торонто", "Калгари", 1700);
  addFlight("Торонто", "Лос-Анджелес", 2500);
  addFlight("Торонто", "Чикаго", 500);
  addFlight("Денвер", "Урбана", 1000);
  addFlight("Денвер", "Хьюстон", 1000);
  addFlight("Хьюстон", "Лос-Анджелес", 1500);
  addFlight("Денвер", "Лос-Анджелес", 1000);
}

// Разместить маршруты в базе данных.
void addFlight(String from, String to, int dist)
{
  if(numFlights < MAX) {
    flights[numFlights] =
      new FlightInfo(from, to, dist);
    numFlights++;
  }
  else System.out.println("Flight database full.\n");
}

// Показать путь и общее расстояние.
void route(String to)
{
  Stack rev = new Stack();
  int dist = 0;
  FlightInfo f;
  int num = btStack.size();

  // Выполнить реверс стека для отображения пути.
  for(int i=0; i < num; i++)
    rev.push(btStack.pop());
  for(int i=0; i < num; i++) {
    f = (FlightInfo) rev.pop();
    System.out.print(f.from + " to ");
    dist += f.distance;
  }
  System.out.println(to);
  System.out.println("Distance is " + dist);
}

/* Если существует маршрут между from и to,
   возвратить расстояние,
   в противном случае возвратить 0. */
int match(String from, String to)
{
  for(int i=numFlights-1; i > -1; i--) {
```

```
    if(flights[i].from.equals(from) &&
       flights[i].to.equals(to) &&
       !flights[i].skip)
    {
      flights[i].skip = true;
      return flights[i].distance;
    }
  }

  return 0; // Не найдено.
}

// Найти любой маршрут.
FlightInfo find(String from)
{
  for(int i=0; i < numFlights; i++) {
    if(flights[i].from.equals(from) &&
       !flights[i].skip)
    {
      FlightInfo f = new FlightInfo(flights[i].from,
                                    flights[i].to,
                                    flights[i].distance);
      flights[i].skip = true; // Запретить повторное использование.
      return f;
    }
  }
  return null;
}

// Есть ли маршрут между from и to?
void isflight(String from, String to)
{
  int dist;
  FlightInfo f;

  // Есть ли пункт прибытия?
  dist = match(from, to);
  if(dist != 0) {
    btStack.push(new FlightInfo(from, to, dist));
    return;
  }

  // Попробовать другой маршрут.
  f = find(from);
  if(f != null) {
    btStack.push(new FlightInfo(from, to, f.distance));
    isflight(f.to, to);
  }
  else if(btStack.size() > 0) {
    // Вернуться и попробовать другой маршрут.
    f = (FlightInfo) btStack.pop();
    isflight(f.from, f.to);
  }
}

// Переустановить все поля skip.
void resetAllSkip() {
  for(int i=0; i< numFlights; i++)
    flights[i].skip = false;
}
```

```
  // Удалить маршрут.
  void remove(FlightInfo f) {
    for(int i=0; i< numFlights; i++)
      if(flights[i].from.equals(f.from) &&
         flights[i].to.equals(f.to))
           flights[i].from = "";
  }
}
```

После выполнение эта программа найдет следующие пути.

```
From? Нью-Йорк
To? Лос-Анджелес
Нью-Йорк to Чикаго to Денвер to Лос-Анджелес
Distance is 2900
Нью-Йорк to Чикаго to Денвер to Хьюстон to Лос-Анджелес
Distance is 4400
Нью-Йорк to Торонто to Лос-Анджелес
Distance is 3000
```

В этом случае наихудшим решением будет второй путь, но два других найденных пути довольно удачные. Обратите внимание, что пути, найденные по методу с удалением узлов, отличаются от путей, найденных по методу удаления маршрутов. Различные подходы к получению нескольких решений дают различные результаты.

Поиск "оптимального" решения

Все предыдущие технологии поиска были ориентированы в основном на поиск любого решения, даже далеко не самого лучшего. С помощью эвристического поиска можно было увеличить скорость поиска и повысить вероятность нахождения хорошего решения. Но не было никаких попыток удостовериться, что найдено оптимальное решение. Однако в некоторых случаях необходимо иметь только "оптимальное" решение. В нашем случае под словом "оптимальный" будем понимать то наилучшее решение, которое может быть получено из нескольких решений, полученных как кратные решения. На самом деле это может и не быть наилучшим решением (Определение наилучшего решения требует значительного времени на полное прохождения всех возможных путей.)

Перед тем как закончить с примерами по определению пути между двумя городами, рассмотрим программу, которая находит оптимальный маршрут при минимизации расстояния. Для того чтобы сделать это, в программе реализован метод удаления маршрутов для получения нескольких решений и используется маршрутизация по критерию наименьшей стоимости для минимизации расстояния. Ключевым условием для поиска оптимального решения будет выбор кратчайшего расстояния из всех полученных.

Полностью программа получения "оптимального" решения приведена ниже. Обратите внимание, что в программе создается дополнительный стек с именем `optimal`, в котором содержатся оптимальные решения, а также создается дополнительная переменная с именем `minDist`, которая содержит расстояние. Также изменен метод `route()` и несколько модифицирован метод `main()`.

```
/* Нахождение "оптимального" решения при использовании
   критерия наименьшей стоимости. */
import java.util.*;
import java.io.*;
```

```
// Информация о полетах.
class FlightInfo {
  String from;
  String to;
  int distance;
  boolean skip; // Используется при возвратах.

  FlightInfo(String f, String t, int d) {
    from = f;
    to = t;
    distance = d;
    skip = false;
  }
}

class Optimal {
  final int MAX = 100;

  // Этот массив содержит информацию о полетах.
  FlightInfo flights[] = new FlightInfo[MAX];
  int numFlights = 0; // Число элементов массива.
  Stack btStack = new Stack(); // Стек возвратов.
  Stack optimal; // Содержит оптимальные решения.
  int minDist = 10000;
  public static void main(String args[])
  {
    String to, from;
    Optimal ob = new Optimal();
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    boolean done = false;
    FlightInfo f;
    ob.setup();
    try {
      System.out.print("From? ");
      from = br.readLine();
      System.out.print("To? ");
      to = br.readLine();
      do {
        ob.isflight(from, to);
        if(ob.btStack.size() == 0) done = true;
        else {
          ob.route(to);
          ob.btStack = new Stack();
        }
      } while(!done);

      // Отобразить оптимальное решение.
      if(ob.optimal != null) {
        System.out.println("Optimal solution is: ");

        int num = ob.optimal.size();
        for(int i=0; i < num; i++) {
          f = (FlightInfo) ob.optimal.pop();
          System.out.print(f.from + " to ");
        }
        System.out.println(to);
        System.out.println("Distance is " + ob.minDist);
      }
    } catch (IOException exc) {
      System.out.println("Error on input.");
```

```
    }
  }

  // Инициализировать базу данных полетов.
  void setup()
  {
    addFlight("Нью-Йорк", "Чикаго", 900);
    addFlight("Чикаго", "Денвер", 1000);
    addFlight("Нью-Йорк", "Торонто", 500);
    addFlight("Нью-Йорк", "Денвер", 1800);
    addFlight("Торонто", "Калгари", 1700);
    addFlight("Торонто", "Лос-Анджелес", 2500);
    addFlight("Торонто", "Чикаго", 500);
    addFlight("Денвер", "Урбана", 1000);
    addFlight("Денвер", "Хьюстон", 1000);
    addFlight("Хьюстон", "Лос-Анджелес", 1500);
    addFlight("Денвер", "Лос-Анджелес", 1000);
  }

  // Расположить маршруты в базе данных.
  void addFlight(String from, String to, int dist)
  {
    if(numFlights < MAX) {
      flights[numFlights] =
        new FlightInfo(from, to, dist);
      numFlights++;
    }
    else System.out.println("Flight database full.\n");
  }

  // Save shortest route.
  void route(String to)
  {
    int dist = 0;
    FlightInfo f;
    int num = btStack.size();
    Stack optTemp = new Stack();
    for(int i=0; i < num; i++) {
      f = (FlightInfo) btStack.pop();
      optTemp.push(f); // save route
      dist += f.distance;
    }

    // Если кратчайший путь, то сохранить.
    if(minDist > dist) {
      optimal = optTemp;
      minDist = dist;
    }
  }

  /* If there is a flight between from and to,
     return the distance of flight;
     otherwise, return 0. */
  int match(String from, String to)
  {
    for(int i=numFlights-1; i > -1; i--) {
      if(flights[i].from.equals(from) &&
         flights[i].to.equals(to) &&
         !flights[i].skip)
      {
        flights[i].skip = true; // prevent reuse
```

```
        return flights[i].distance;
      }
    }

    return 0; // not found
  }

  // Given from, find any connection using least-cost.
  FlightInfo find(String from)
  {
    int pos = -1;
    int dist = 10000; // longer than longest route
    for(int i=0; i < numFlights; i++) {
      if(flights[i].from.equals(from) &&
         !flights[i].skip)
      {
        if(flights[i].distance < dist) {
          pos = i;
          dist = flights[i].distance;
        }
      }
    }
    if(pos != -1) {
      flights[pos].skip = true; // prevent reuse
      FlightInfo f = new FlightInfo(flights[pos].from,
                           flights[pos].to,
                           flights[pos].distance);
      return f;
    }
    return null;
  }

  // Determine if there is a route between from and to.
  void isflight(String from, String to)
  {
    int dist;
    FlightInfo f;

    // See if at destination.
    dist = match(from, to);
    if(dist != 0) {
      btStack.push(new FlightInfo(from, to, dist));
      return;
    }

    // Try another connection.
    f = find(from);
    if(f != null) {
      btStack.push(new FlightInfo(from, to, f.distance));
      isflight(f.to, to);
    }
    else if(btStack.size() > 0) {
      // Backtrack and try another connection.
      f = (FlightInfo) btStack.pop();
      isflight(f.from, f.to);
    }
  }
}
```

После выполнения будет получен следующий результат.

```
From? Нью-Йорк
To? Лос-Анджелес
Optimal solution is:
Нью-Йорк to Чикаго to Денвер to Лос-Анджелес
Distance is 2900
```

В этом случае "оптимальное" решение не является самым лучшим, но оно все же довольно хорошее. Как уже говорилось, когда используется поиск с помощью методов искусственного интеллекта, наилучшие решения, найденные с помощью некоторой технологии поиска, не всегда будут самыми лучшими решениями. Можно попытаться применить другую технологию поиска для предыдущей программы и посмотреть, какое "оптимальное" решение будет найдено в этом случае.

Одной из неудачных логических конструкций этой программы является то, что все пути "проходятся" полностью, до самого конца. Можно повысить эффективность работы программы, если прекращать поиск, как только полученное расстояние будет превышать заданный максимум. Попытайтесь улучшить программу, введя это правило.

Вернемся к потерянным ключам

В заключение главы о поиске решения, рассмотрим улучшенную программу поиска ключей, рассматриваемую в начале книги. Соответствующий код реализует технологию, используемую при определении пути между двумя городами, поэтому программа приведена без подробного описания.

```
// Найти потерянные ключи!
import java.util.*;
import java.io.*;

// Информация о комнатах.
class RoomInfo {
  String from;
  String to;
  boolean skip;
  RoomInfo(String f, String t) {
    from = f;
    to = t;
    skip = false;
  }
}

class Keys {
  final int MAX = 100;

  // Этот массив содержит информацию о комнатах.
  RoomInfo room[] = new RoomInfo[MAX];
  int numRooms = 0; // Число комнат.
  Stack btStack = new Stack(); // Стек возвратов.
  public static void main(String args[])
  {
    String to, from;
    Keys ob = new Keys();
    ob.setup();
    from = "front_door";
    to = "keys";
    ob.iskeys(from, to);
```

```
    if(ob.btStack.size() != 0)
      ob.route(to);
  }

  // Инициализировать базу данных о комнатах.
  void setup()
  {
    addRoom("front_door", "lr");
    addRoom("lr", "bath");
    addRoom("lr", "hall");
    addRoom("hall", "bd1");
    addRoom("hall", "bd2");
    addRoom("hall", "mb");
    addRoom("lr", "kitchen");
    addRoom("kitchen", "keys");
  }

  // Разместить комнаты в базе данных.
  void addRoom(String from, String to)
  {
    if(numRooms < MAX) {
      room[numRooms] = new RoomInfo(from, to);
      numRooms++;
    }
    else System.out.println("Room database full.\n");
  }

  // Показать путь и полное расстояние.
  void route(String to)
  {
    Stack rev = new Stack();
    RoomInfo r;
    int num = btStack.size();

    // Выполнить реверс стека для отображения пути.
    for(int i=0; i < num; i++)
      rev.push(btStack.pop());
    for(int i=0; i < num; i++) {
      r = (RoomInfo) rev.pop();
      System.out.print(r.from + " to ");
    }
    System.out.println(to);
  }

  /* Если есть маршрут между from и to,
     возвратить true,
     в противном случае возвратить false. */
  boolean match(String from, String to)
  {
    for(int i=numRooms-1; i > -1; i--) {
      if(room[i].from.equals(from) &&
         room[i].to.equals(to) &&
         !room[i].skip)
      {
        room[i].skip = true; // Запретить повторное использование.
        return true;
      }
    }
    return false; // Не найдено.
  }
```

```
  // Найти любой путь.
  RoomInfo find(String from)
  {
    for(int i=0; i < numRooms; i++) {
      if(room[i].from.equals(from) &&
         !room[i].skip)
      {
        RoomInfo r = new RoomInfo(room[i].from,
                                  room[i].to);
        room[i].skip = true; Запретить повторное использование.
        return r;
      }
    }
    return null;
  }

  // Определить, есть ли путь между from to.
  void iskeys(String from, String to)
  {
    int dist;
    RoomInfo r;

    // Есть ли пункт прибытия.
    if(match(from, to)) {
      btStack.push(new RoomInfo(from, to));
      return;
    }

    // Попробовать другой маршрут.
    r = find(from);
    if(r != null) {
      btStack.push(new RoomInfo(from, to));
      iskeys(r.to, to);
    }
    else if(btStack.size() > 0) {
      // Вернуться и попробовать другой маршрут.
      r = (RoomInfo) btStack.pop();
      iskeys(r.from, r.to);
    }
  }
}
```

Предметный указатель

Научно-популярное издание

Герберт Шилдт, Джеймс Холмс

Искусство программирования на Java

Литературный редактор *С.Г. Татаренко*
Верстка *М.А. Удалов*
Художественный редактор *С.А. Черноозинский*
Корректоры *З.В. Александрова, Т.А. Гордиенко, Л.В. Черноозинская*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 14.02.2005. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 27,09. Уч.-изд. л. 17,1.
Тираж 3000 экз. Заказ № 604.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

ИСКУССТВО программирования на JAVA

ПОДНЯТЬ СВОЙ УРОВЕНЬ ПРОГРАММИРОВАНИЯ НА JAVA ЕЩЕ НА ОДНУ СТУПЕНЬ

Помогут вам высочайшие авторитеты по языкам программирования Герберт Шилдт и Джеймс Холмс. Вместе они раскроют секреты профессиональных программистов и опишут все этапы создания профессиональных программ. Герберт и Джеймс используют язык Java для создания мощных приложений, причем в каждом приложении демонстрируются различные возможности Java и применение различных технологий. Диапазон рассматриваемых приложений простирается от интерпретаторов языка, Web-червей и почтовых систем до синтаксических анализаторов выражений, прикладных статистических программ и финансовых аплетов. Вы даже узнаете, как с помощью Java реализовать возможности искусственного интеллекта! Каждое приложение может использоваться самостоятельно или как базовое приложение, на основе которого вы сможете разрабатывать собственные программы. Здесь вы найдете изящные фрагменты кода, не описанные больше нигде, которые можно использовать для создания сетевых приложений или программ синтаксического анализа

БЕСПЛАТНЫЙ КОД В INTERNET

БОНУС! WWW.OSBORNE.COM

В ЭТОЙ КНИГЕ:

- исследование богатейших возможностей языка Java
- создание синтаксического анализатора для целочисленных выражений
- построение Web-червя
- структура и реализация интерпретатора языка программирования
- разработка завершенной почтовой системы
- создание диспетчера загрузок для упрощения выкачки файлов из Internet
- программа для статистических вычислений, таких как среднее значение, медиана, мода, стандартное отклонение и др.
- написание финансовых аплетов и сервлетов, с помощью которых можно подсчитать необходимые выплаты, ссуды, инвестиции и многое другое
- описание работы алгоритмов поиска, которые используются при реализации программ искусственного интеллекта
- генерация динамических HTML-страниц на языке Java

The McGraw-Hill Companies

Вильямс McGraw-Hill OSBORNE

ISBN 5-8459-0786-1

9 785845 907868 05056

JAVA / ПРОГРАММИРОВАНИЕ

www.williamspublishing.com
www.osborne.com

ГЕРБЕРТ ШИЛДТ Популярный автор книг по языкам программирования Java, C, C++ и C#, а также эксперт по программированию для Windows. По всему миру разошлось более трех миллионов экземпляров его книг; они переведены на все основные языки.

ДЖЕЙМС ХОЛМС Признанный лидер в программировании на языке Java, который разрабатывал серверные приложения для очень ответственных применений. Заказчиками Джеймса были оргкомитет летних олимпийский игр в Атланте, а также компания IBM. Он был признан лучшим специалистом 2002 года журналом *Oracle Magazine Java Developer*. В настоящее время он является независимым консультантом по языку Java.

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с Web-сервера издательства по адресу: http://www.williamspublishing.com