Вадим Будилов

bhv

Интернет-программирование на Java

- Разработка Web-приложений
- Создание клиентских и серверных приложений
- Разработка сервлетов и JSP-страниц
- Серверные компоненты EJB и среда J2EE

МАСТЕР_ПРОГРАММ_

Вадим Будилов

Интернет-программирование на Java

Санкт-Петербург

«БХВ-Петербург»

2003

УДК 681.3.06
ББК 32.973.26-018.1
Б90

Будилов В. А.

Б90 Интернет-программирование на Java. — СПб.: БХВ-Петербург, 2003. — 704 с.: ил.

ISBN 5-94157-272-7

В книге подробно рассматриваются методы создания интернет-приложений на языке Java, в том числе Web-приложений, апплетов, серверных приложений, использование серверных страниц Java, конструирование и программирование пользовательских библиотек ярлыков Java, а также разработка приложений с применением современных технологий, реализованных в пакете J2EE. Подробно описано функционирование сервера Blazix. Внимание акцентировано на раскрытие наиболее существенных сторон создания клиентских и серверных приложений. Многочисленные примеры делают изложенный материал весьма наглядным и помогают его лучшему усвоению. Книга рассчитана на читателя, знакомого с программированием и имеющего некоторый опыт создания программ на любом языке.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Консультант	*Ильдар Хабибуллин*
Редактор	*Ирина Радченко*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 23.05.03.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 56,76.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-272-7

© Будилов В. А., 2003
© Оформление, издательство "БХВ-Петербург", 2003

Содержание

Виктории Будкевич

Введение

Данная книга посвящена программированию на языке Java. Этот язык изначально создавался для программирования в сети Интернет, поэтому он является наиболее приспособленным к нуждам сети механизмом, использование прогрессивных методов которого позволяет реализовать наиболее эффективные, надежные, многократно используемые приложения, создаваемые на основе компонентов. Современные технологии программирования распределенных приложений основаны на детально разработанных концепциях, воплощенных средствами Java. Разработка компонентов не будет составлять большого труда, если задача сформулирована четко, а менеджер и разработчик знакомы со средствами, которые предоставляет язык Java.

В книге рассматриваются ставшие классическими подходы, используемые для создания серверных приложений — сервлетов и серверных страниц JSP (Java Server Pages). Для того чтобы сделать материал наиболее наглядным и легко воспринимаемым, читателю предлагается разбор примеров. Сама же книга посвящена изучению общих принципов рассматриваемых технологий без акцента на конкретной реализации той или иной технологии. Для облегчения изложения примеры работают с сервером Blazix, выбор этого сервера обусловлен тем, что он достаточно прост в использовании, поэтому не потребуется большого времени для изучения его устройства. Сервер Blazix представляет собой сервер, который исполняет функции Web-сервера и который работает с компонентами EJB (компонентами Enterprise JavaBeans), сервлетами и JSP. Это полнофункциональный сервер, поддерживающий возможности Java Application Server.

Книга рассчитана на читателя, знакомого с программированием и имеющего некоторый опыт создания программ на любом языке программирования. Книга может быть полезна студентам, а также тем, кто желает расширить свой кругозор и познакомиться с методами создания интернет-приложений на языке Java. Предполагается, что читатель знаком с основами программирования для сети Интернет. В современных интернет-технологиях широко применяются языки XML (eXtensible Markup Language — расширяемый язык разметки) — язык разметки, позволяющий дополнительно определять теги

(разметку), а также связанные с ними языки описания типов документов и языки стилей, например, XML Schema и XSL (eXtensible Stilesheets Language — расширяемый язык стилей). Все, кто связан с интернет-программированием, в той или иной степени знакомы с этими языками. Базовых знаний о них достаточно для работы с предлагаемой книгой. Для более детального знакомства с перечисленными языками советуем обратиться к специальной литературе. Java-сервера используют платформонезависимые форматы описания конфигурации как самого сервера, так и размещаемых на нем приложений. Основным используемым форматом является формат XML. Описатель размещения Web-приложений и компонентов EJB представляет собой XML-документ.

Тот факт, что конфигурация серверов Java-приложений представлена в формате XML, накладывает ряд требований. Современный язык Java имеет набор средств для работы с XML, а также с протоколами и технологиями, основанными на XML, например, с SOAP (Simple Objects Access Protocol — простой протокол доступа к объектам). Большое значение во всех серверах уделяется описанию служб Web, доступ к которым может быть осуществлен различными способами. Один из универсальных подходов — это использование протокола SOAP, который является легковесным протоколом, а сообщения на основе SOAP могут быть переданы поверх различных протоколов более низкого уровня, в том числе с использованием протокола HTTP (HyperText Transfer Protocol) или SMTP (Simple Mail Transfer Protocol — простой протокол передачи почты). Протокол SOAP естественным образом предполагает наличие SOAP-сервиса, расположенного на соответствующем сервере. Ввиду того, что протокол SOAP прост, его реализация собственными силами не представляет сложную задачу. Существуют и готовые реализации SOAP. Доступ к службам SOAP осуществляется по протоколу SOAP. В Интернете могут существовать различные службы, в том числе службы, доступ к которым основан на SOAP. Существует необходимость задания описания различных служб. В настоящее время консорциум W3C (World Wide Web Consortium) ведет работу по разработке серии языков нового поколения, которые сделают работу в сети Интернет более структурированной функционально. Выполнение частей приложений на различных компьютерах сети станет более естественным. Для этого, в частности, создается язык описания служб сети WSDL (Web Services Description Language — язык описания Web-служб).

Новые возможности сети Интернет становятся реальностью только тогда, когда есть механизмы их реализации. Высокоуровневое взаимодействие ресурсов сети Интернет друг с другом, которое не зависит ни от типа программного обеспечения, ни от архитектуры аппаратного обеспечения, возможно только тогда, когда существуют эффективные средства решения стоящих перед разработчиком задач на более низком уровне. Именно изучению таких методов посвящена предоставляемая вашему вниманию книга.

Она состоит из пяти глав и пяти приложений. *Первая глава* посвящена работе с апплетами — небольшими программами, которые работают на клиентском браузере в составе HTML-страницы (страницы, написанной на HyperText Markup Language — языке разметки гипертекста). Во *второй главе* рассмотрены вопросы сериализации и работы с сетью. *Третья глава* посвящена изучению технологии серверных страниц JSP (с использованием описаний на основе XML), рассмотрены способы создания пользовательских ярлыков JSP и создания классов обработчиков этих ярлыков. В этой же главе дано описание сервера Blazix. В *четвертой главе* более подробно рассмотрена технология создания сервлетов, а также устройство сервлета, его жизненный цикл, методы вызова контейнером сервлета в течение его жизненного цикла. *Пятая глава* посвящена изучению серверных компонентов EJB. После изучения этой главы читатель сможет ориентироваться в технологии создания компонентов EJB, разрабатывать и использовать компоненты EJB в собственных приложениях.

Глава 1

Интернет и Java

So, naturalists observe, a flea
Hath smaller fleas that on him prey;
And these have smaller still to bite'em;
And so proceed ad infinitum.

Sir Andrew Larsh

Java — это алгоритмический язык, который является разработкой компании Sun Microsystems и предназначен для программирования в сети Интернет, или, другими словами, язык, предназначенный для создания приложений, работающих, в частности, в сети WWW (World Wide Web). Наиболее компактными Java-приложениями являются *апплеты*. Апплеты — это небольшие программы, которые, как правило, создаются для работы в браузере. Апплеты могут существенно обогатить внешний вид HTML-страницы. При работе с ними удобно использовать пакеты, предоставляющие возможность работы с графикой и звуком. Значительное место в апплетах может занимать код обработки событий.

Для того чтобы научиться создавать апплеты, необходимо иметь минимальный навык программирования на языке Java.

1.1. Первое приложение на языке Java

Для того чтобы написать, скомпилировать и запустить приложение на языке Java, необходимо кроме файла с программой иметь среду выполнения программ Java. Для платформы, с который вы работаете, например, для Windows, следует получить пакет разработчика jdk1.3 или jdk1.4, который можно найти на сайте производителя **http://java.sun.com**. Пакет следует установить на компьютере в соответствии со всеми инструкциями, в частности необходимо указать значение переменной `path`, соответствующее папке, где установлен пакет Java. После того как пакет установлен, можно приступить к созданию простого кода (листинг 1.1).

Листинг 1.1. Файл VsemPrivet.java

```
class VsemPrivet {
 public static void main(String[] args) {
   System.out.println("Vsem Privet I S Dobrym Utrom!");
 }
}
```

Что здесь важно? Файл содержит класс, имя которого должно совпадать с именем файла. Класс этот содержит метод `main()`. Метод `main()` должен присутствовать всегда.

После того как файл с классом создан, вызовем компилятор javac из командной строки (рис. 1.1):

```
E:\temp>javac vsemprivet.java
```

Рис. 1.1. Вызов компилятора из командной строки

Вызов компилятора необходимо осуществить из той папки, где расположен файл VsemPrivet.java. Если компиляция прошла успешно, то сообщений об ошибках не будет. В папке появится новый файл с именем VsemPrivet.class. Этот файл содержит скомпилированный Java-код. Для выполнения классов Java-кода необходима среда выполнения Java или виртуальная машина Java. А для того чтобы среда выполнения классов Java была правильно инициализирована, необходимо указать путь к файлам классов в переменной `classpath`. Из командной строки это делается так:

```
set classpath=%classpath%;путь_к_папке_файла_класса
```

Теперь можно запустить программу (рис. 1.2), выполнив команду

```
java VsemPrivet
```

при этом расширение файла в имени класса не указывается.

Рис. 1.2. Запуск программы

Фактически при вызове среды выполнения Java в качестве аргумента указывается класс, а не файл. Класс хранится в файле, имя которого соответствует имени класса. Следует помнить, что Java различает строчные и прописные буквы. Если изменить регистр букв при обращении к классу, то Java выведет сообщения об ошибках (рис. 1.3).

Рис. 1.3. Появление ошибок вызвано неверным написанием имени класса

1.2. Апплеты

Апплет можно определить как небольшую программу, написанную на языке Java и работающую в браузере. В действительности апплет — это неполная, несамостоятельная программа, к тому же она вовсе не обязана быть маленькой. Небольшой ее создают только в целях удобства, по причине того, что класс программы, как правило, передается по сети, да и ресурсы браузера бывают ограничены. Кроме того, существуют способы использовать апплет не только в браузере. Просто для апплета типично быть маленькой программой,

выполняемой в браузере. Итак, наиболее корректным будет следующее определение: апплет — это программа, созданная на основе класса `java.applet.Applet` или на основе одного из подклассов этого класса.

Апплет — наследник графического интерфейса пользователя, это графический компонент, который может быть показан в окнах, например, в окне браузера или в другой программе. При этом апплет занимает некоторую часть окна браузера, и внутри апплета могут быть расположены другие графические элементы (кнопки, текст, поля и т. п.), отображены рисунки, линии, апплет может реагировать на события (например, на щелчки мыши), включать в себя другие элементы.

Сам по себе класс `Applet` не будет для нас слишком интересен. В окне он может создать пустую прямоугольную область, в которой ничего нет, область, которая не реагирует ни на какие действия. При создании апплета мы задаем класс на основе класса `Applet`, который содержит в себе дополнительные возможности, "оживляющие" апплет. В классе `Applet` определено несколько методов, которые необходимо переопределить во время создания апплета, изначально эти методы ничего не выполняют.

Отметим, что в апплете нет функции `main()`, это происходит потому, что апплет не является самостоятельной программой, он должен работать в составе другого приложения. Задачей разработчика является создание "ответов" на "запросы" "системы", то есть ответов на запросы той программы, с которой работает апплет.

Рассмотрим некоторые важнее моменты, часто используемые при создании апплетов. Один из описанных в классе `Applet` методов, которые, однако, изначально не приспособлены к каким-либо действиям, — метод `paint()`. Этот метод вызывается, когда необходимо что-либо нарисовать. В апплете метод `paint()` можно вызывать при необходимости вывода графических элементов, например, прямоугольников, линий, текста. Метод `paint()` определяется следующим образом:

```
public void paint(Graphics g) {
  // функции рисования
}
```

В качестве аргумента функции указывается параметр `g`, который имеет тип `Graphics`. Большинство задач по отображению в языке Java решаются с использованием объектов `Graphics`. Для работы с этими объектами существует множество методов.

1.2.1. Первый апплет

Создадим простой апплет, пусть этот апплет выводит текст "`Privet Vsem, a ne Hello World!`". Эта строка будет выводиться с использованием метода `paint()`. Чтобы не использовать полные имена методов, включая имена

пакетов (`Applet` вместо `java.applet.Applet` и `Graphics` вместо `java.awt.Graphics`), импортируем в начале файла основные пакеты (листинг 1.2).

Листинг 1.2. Файл PrivetVsemApplet.java

```
import java.awt.*;
import java.applet.*;
public class PrivetVsemApplet extends Applet {
  // Апплет для вывода строки текста
  public void paint(Graphics g) {
    g.drawString("Privet Vsem, a ne Hello World!", 15, 45);
  }
}
```

Сейчас необходимо скомпилировать класс. В действительности, рисование как таковое осуществляет метод `drawString()`, определенный в классе `Graphics`. В качестве параметров этого метода указывается строка, которая будет выведена, а также точка (координаты) апплета, где эта строка будет расположена. Теперь можно использовать апплет, создав новый объект, например, с помощью инструкции

```
Applet na = new PrivetVsemApplet();
```

Так можно поступать, если существует необходимость вставить апплет в окно другой программы. Но чаще всего апплеты используются в браузере. Для этого необходимо создать HTML-страницу, например, так, как это сделано в файле `PrivetVsemApplet.html` (листинг 1.3).

Листинг 1.3. Файл PrivetVsem.html

```
<applet code="PrivetVsemApplet.class" width=300 height=150>
</applet>
```

Здесь мы пренебрегли (для сохранения предельной простоты) правилами хорошего тона, создав простейший файл HTML, в котором не указаны основные его элементы. Это не страшно. Браузер автоматически восполнит недостающее. Поместим этот файл в ту же папку, где расположен класс апплета. Загрузим HTML-страницу в браузер. Если в браузере включена поддержка Java, то в окне получим отображенный в апплете текст (рис. 1.4).

Этот текст отображается в прямоугольнике размером 300×150 пикселов. Этот прямоугольник не видно, так как его цвет совпадает с цветом фона браузера.

Рис. 1.4. Отображение апплетом текста в окне браузера

В классе `Applet` существует метод `init()`, который выполняется сразу после того, как апплет будет создан, но до того, как будет отображен в окне. В этот момент можно инициализировать апплет. Разработчику не надо беспокоиться о вызове этого метода, он вызывается автоматически, необходимо лишь описать этот метод, создать нужные функции для него (листинг 1.4).

Листинг 1.4. Метод `init()`

```
public void init() {
  // Функции инициализации апплета
}
```

Инициализация апплета может быть осуществлена также с использованием функции конструктора. Однако при вызове конструктора недоступным будет размер апплета, в то время как функция `init()` позволяет использовать параметры размера апплета. Как правило, инициализация апплета осуществляется с применением метода `init()`.

В качестве примера использования метода `init()` перепишем наш пример и зададим цвет фона (листинг 1.5).

Листинг 1.5. Файл VsemPrivetApplet2.java

```
import java.awt.*;
import java.applet.*;
  public class VsemPrivetApplett2 extends Applet {
  public void init() {
    // инициализируем апплет, цвет фона — зеленый
    // цвет основной — красный.
    setBackground(Color.green);
    setForeground(Color.red);
   }
  public void paint(Graphics g) {
    g.drawString("Vsem Privet, a ne Hello World!", 30, 90);
  }
 }
```

Рис. 1.5. Инициализация апплета в методе `init()`

Сейчас прямоугольник апплета стал видимым и зеленым, а текст выведен красным цветом (рис. 1.5).

1.2.2. Апплеты AWT и апплеты Java 2

В примерах, рассмотренных выше, мы использовали пакет `java.awt.*`. Недостатки, обнаруженные в этом пакете, привели к созданию пакета `Swing`, явившегося составной частью версии Java 2 начиная с jdk1.2. Существует возможность создания апплетов на основе пакета `Swing`. Однако при этом нужно иметь в виду, что некоторые версии браузеров не будут поддерживать такие апплеты, к тому же апплеты, основанные на пакете `Swing`, отнимают большие ресурсы, чем апплеты, основанные на AWT (Abstract Window Toolkit, абстрактный оконный интерфейс).

Пакет `javax.swing` содержит класс `JApplet`. Класс `javax.swing.JApplet` можно использовать в качестве базового класса при создании апплетов. Класс `JApplet` является подклассом по отношению к классу `Applet`, т. е. апплеты `Japplets` являются полноценными апплетами, но возможности использовать `Swing` у них встроены. Рисование при помощи апплетов `JApplet` несколько более усложнено. Для класса, основанного на `JApplet`, не нужно писать метод `paint()`, зато здесь необходим метод инициализации апплета `init()`. Рисование же осуществляется при помощи вставки компонентов.

Первый апплет JApplet

Рассмотрим пример апплета, в котором проявляются основные идеи использования пакета `Swing` и программирования графического интерфейса. Программирование элементов графики предполагает использование готовых компонентов, например, таких как кнопки и т. п., посредством которых апплет может взаимодействовать с пользователем. В примере (листинг 1.6) апплет состоит только из кнопки (листинг 1.7).

Листинг 1.6. Код апплета находится в файле Knopka.java

```
import javax.swing.*;  // классы Swing GUI
import java.awt.event.*; // классы обработки событий
  public class Knopka extends JApplet implements ActionListener {
  public void init() {
    // метод вызывается после создания апплета,
    // но перед тем, как апплет будет отображен в окне
    JButton knopka = new JButton("Nazhmi Menya!");
    knopka.addActionListener(this);
    getContentPane().add(knopka);
```

```
  }
  public void actionPerformed(ActionEvent evt) {
  // Метод используется при наступлении события.
  // В данном случае возможно только одно событие –
  // нажатие кнопки.
  // В ответ отображается диалоговое окно
  // с информацией и кнопкой OK,
  // нажатие которой закрывает новое диалоговое окно.
  String title = "Priovetstvuyu vas";  // заголовок диалогового окна
  String message = "Privet vam ot biblioteki Swing.";
  JOptionPane.showMessageDialog(null, message, title,
  JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Этот апплет вызывается из файла Knopka.html.

Листинг 1.7. Файл Knopka.html

```
<p align=center>
  <applet code="Knopka.java" width=250 height=140 >
  </applet>
</p>
```

При загрузке этого апплета в окно браузера появляется кнопка размером 250 × 140 пикселов, нажатие которой создает окно с информацией и кнопкой **OK** (рис. 1.6).

Кнопка создается на основе класса кнопок `JButton` (класс `javax.swing.JButton`). После создания апплета кнопка также должна быть создана и вставлена в апплет. Эта задача входит в перечень задач, выполняемых при инициализации апплета. Кнопка создается при помощи следующей функции:

```
JButton knopka = new JButton("Nazhmi Menya!");
```

В качестве параметра конструктору передается строка, которая будет отображена на кнопке. Кнопка должна быть вставлена в контекст панели апплета с помощью метода

```
getContentPane().add(knopka);
```

Нажатие кнопки создает событие, тип которого `ActionEvent`. Для обработки событий этого типа создаем метод

```
public void actionPerformed(ActionEvent evt) { ... }
```

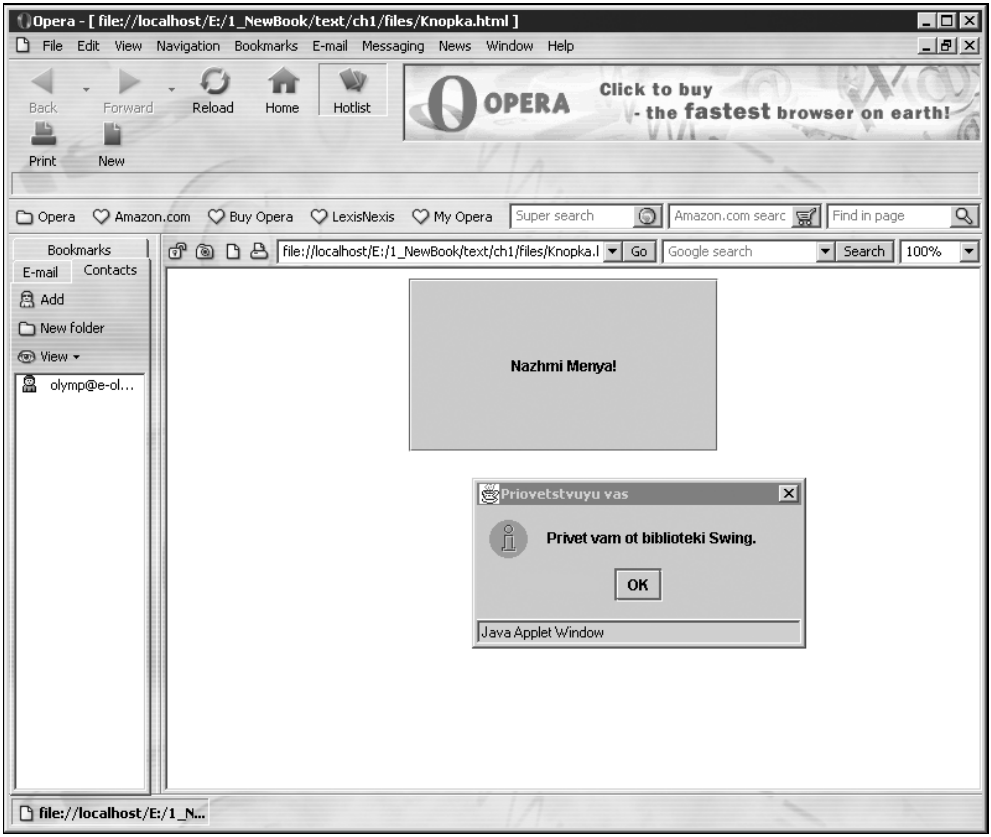


Рис. 1.6. Функционирование апплета Swing

Кнопка создается на основе класса кнопок `JButton` (класс `javax.swing.JButton`). После создания апплета кнопка также должна быть создана и вставлена в апплет. Эта задача входит в перечень задач, выполняемых при инициализации апплета. Кнопка создается при помощи следующей функции:

```
JButton knopka = new JButton("Nazhmi Menya!");
```

В качестве параметра конструктору передается строка, которая будет отображена на кнопке. Кнопка должна быть вставлена в контекст панели апплета с помощью метода

```
getContentPane().add(knopka);
```

Нажатие кнопки создает событие, тип которого `ActionEvent`. Для обработки событий этого типа создаем метод

```
public void actionPerformed(ActionEvent evt) { ... }
```

Кнопке необходимо сообщить, что апплет прослушивает связанные с ней события. Это делается при помощи метода `addActionListener()`, который используется внутри метода `init()`.

Что происходит внутри метода `actionPerformed()`? При нажатии кнопки должно появиться окно с информацией. Это легко осуществимо средствами пакета `Swing`. Класс `swing.javax.JOptionPane` содержит статический метод `showMessageDialog()`, с помощью него и решается поставленная задача.

В этом примере апплет сам заботится о том, чтобы прослушивать сообщения кнопки. Однако это не самый лучший способ. Для прослушивания можно выделить специальный объект, который будет отвечать за обработку событий. Это осуществляется следующим образом (листинг 1.8).

Листинг 1.8. Файл knopka2.java

```
import javax.swing.*;
import java.awt.event.*;
  public class Knopka2 extends JApplet {
    public void init() {
      JButton knopka = new JButton("Nazhmi Menya!");
      knopka.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
          // метод вызывается при нажатии кнопки
          String title = "Privet";  // название кнопки
          String message = "Esche raz privet ot Swing.";
          JOptionPane.showMessageDialog(null, message, title,
          JOptionPane.INFORMATION_MESSAGE);
        }
      }
    );
    getContentPane().add(knopka);
  }
```

1.3. Кратко о HTML

Наиболее распространенное применение апплетов — использование их в HTML-страницах. *HTML-страница* — это файл, содержащий в себе код, созданный с использованием языка HTML, HyperText Markup Language (язык разметки гипертекста). При помощи этого языка описывается содержимое Web-страниц, которое загружается в клиентскую программу-браузер. Сам по себе HTML-код выглядит совсем не так, как будет представлена

страница с использованием HTML-кода, загруженного в браузер. Помимо самого текста, HTML-код будет содержать инструкции, которые определят структуру текста, внешний вид страницы, динамические элементы, вставленные в страницу, рисунки и т. п. При помощи HTML в страницу могут быть вставлены и Java-апплеты.

Создать HTML-страницу можно при помощи простого текстового редактора, вводя вручную все инструции разметки. Однако существует большое количество программ, которые помогают создавать HTML-страницы без необходимости вникать в детали HTML-кода. При этом создание HTML-страницы становится столь же простым делом, как создание текстовой страницы в текстовом процессоре, например, в программе MS Word. Однако использование программ автоматического редактирования HTML-страниц не позволяет воспользоваться всеми возможностями, предоставляемыми языком HTML, к тому же динамические элементы требуют задания определенных параметров, которые необходимо указывать в контексте потока HTML-кода. Редакторы HTML-страниц порой генерируют чрезмерно причудливый HTML-код, который требует редактирования вручную. Все это говорит о том, что знание языка HTML просто необходимо.

В этом разделе мы рассмотрим основы языка HTML.

Команды языка HTML представляют собой теги. Структура тега такова:

```
<имя_тега [модификаторы-атрибуты]>
```

Имя тега — это слово. Существует ограниченный набор стандартных имен тегов. Модификаторы-атрибуты — это определенные свойства, в каждом теге их может быть указано несколько или же может не быть ни одного. Для каждого тега существует свой набор атрибутов. Модификаторы имеют такой вид:

```
Имя_атрибута = значение
```

Как правило, значение заключается в кавычки. Это особенно важно в том случае, если строка значения состоит из нескольких слов. Помните, что HTML не чувствителен к регистру, то есть заглавные и строчные буквы несут один и тот же смысл. Вот пример тега: `<HR>` — он используется для создания горизонтальной черты-линейки, проходящей слева направо (или, если хотите, справа налево) через всю страничку. Тег `<HR>` может иметь несколько атрибутов, например, атрибуты `WIDTH` и `ALIGN`. Для того чтобы создать короткую линейку, расположенную по центру странички, используем следующий тег:

```
<HR align=center width="33%">
```

Здесь указана ширина линейки в процентах от ширины всей странички (рис. 1.7). Вместо относительной ширины можно задать значение ширины линейки в пикселах. Атрибут `ALIGN` может принимать значения `CENTER` (или `CENTRE`), `LEFT`, `RIGHT`.

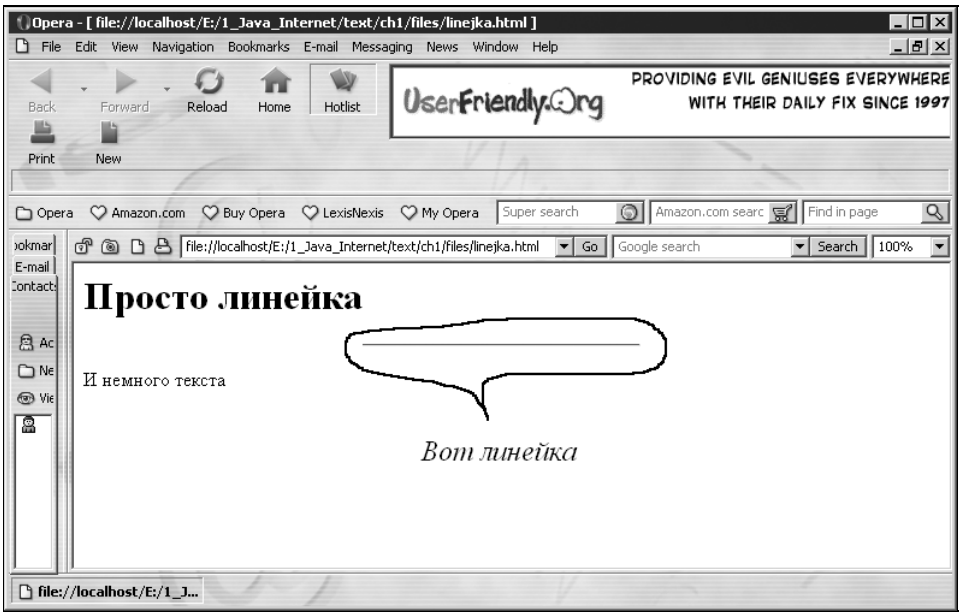


Рис. 1.7. Простая линейка и текст HTML

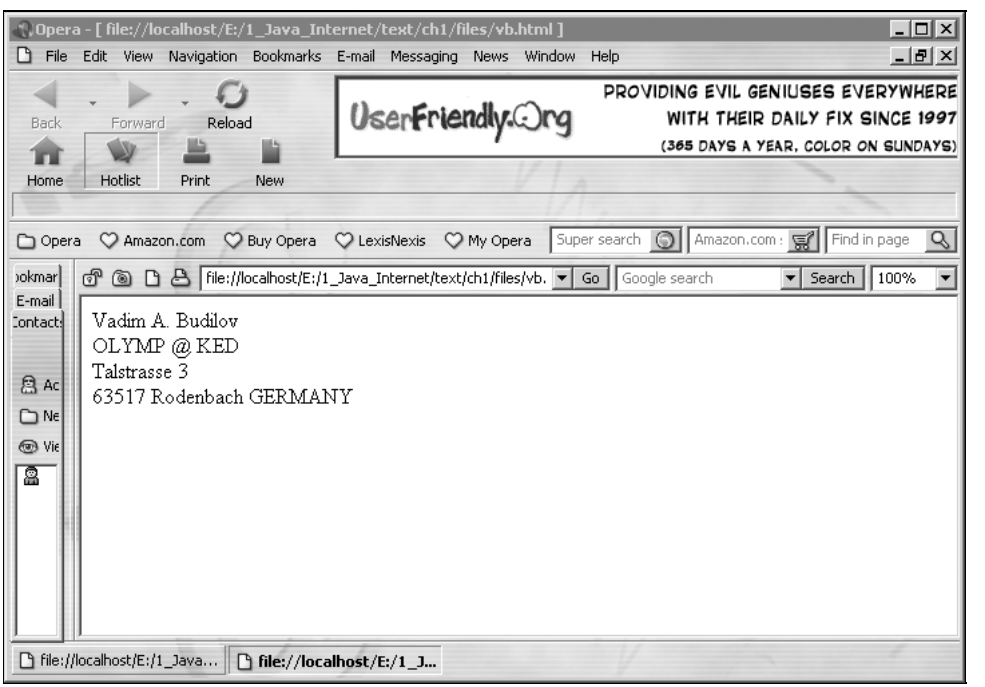


Рис. 1.8. Перенос строки

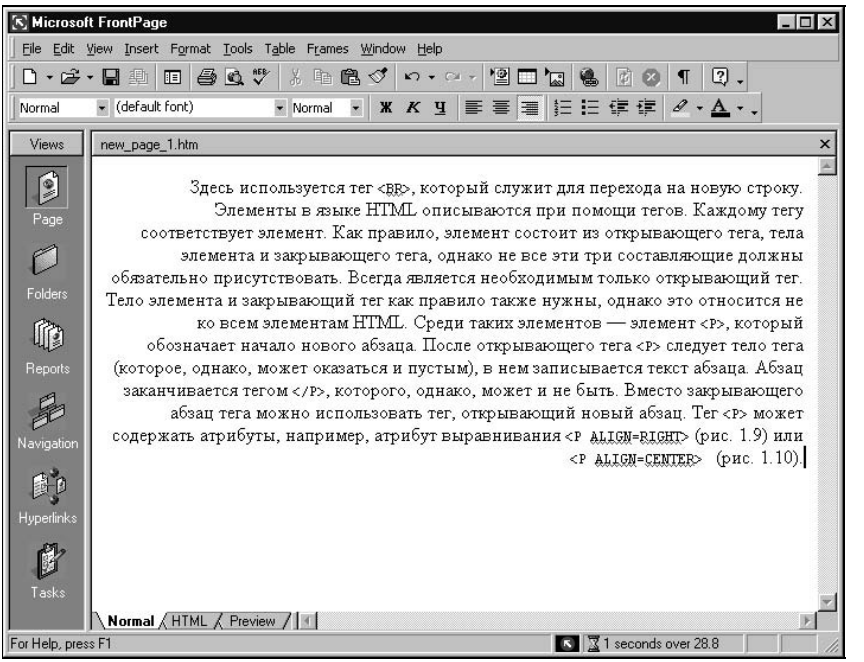


Рис. 1.9. Выравнивание по правой стороне

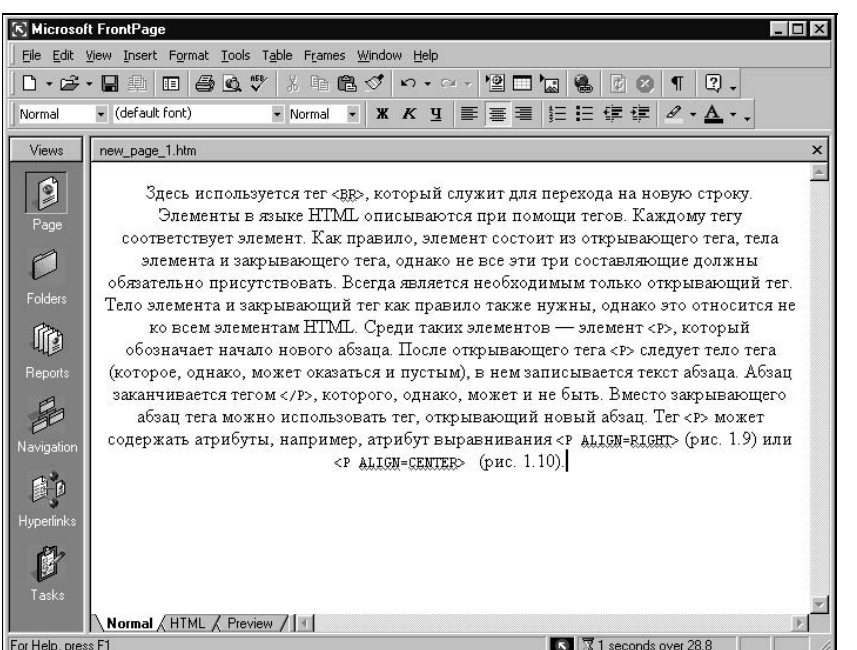


Рис. 1.10. Выравнивание по центру

Большинство тегов требует наличия парных им закрывающих тегов, которые выглядят так:

```
</имя_тега>
```

Однако не все теги таковы, некоторые элементы не требуют использования закрывающих тегов, например (рис. 1.8):

```
Vadim A. Budilov<BR>
OLYMP @ KED<BR>
Talstrasse 3<BR>
63517 Rodenbach
GERMANY<BR>
```

Здесь используется тег
, который служит для перехода на новую строку. Элементы в языке HTML описываются при помощи тегов. Каждому тегу соответствует элемент. Как правило, элемент состоит из открывающего тега, тела элемента и закрывающего тега, однако не все эти три составляющие должны обязательно присутствовать. Всегда является необходимым только открывающий тег. Тело элемента и закрывающий тег как правило также нужны, однако это относится не ко всем элементам HTML. Среди таких элементов — элемент <P>, который обозначает начало нового абзаца. После открывающего тега <P> следует тело тега (которое, однако, может оказаться и пустым), в нем записывается текст абзаца. Абзац заканчивается тегом </P>, которого, однако, может и не быть. Вместо закрывающего абзац тега можно использовать тег, открывающий новый абзац. Тег <P> может содержать атрибуты, например, атрибут выравнивания <P ALIGN=RIGHT> (рис. 1.9) или <P ALIGN=CENTER> (рис. 1.10).

Далее будут рассмотрены очень кратко основные элементы HTML.

1.3.1. Общая структура HTML-документа

HTML-документ имеет стандартную структуру. Документ состоит из элемента <HTML>, он открывается тегом <HTML> и заканчивается тегом </HTML>. Между этими тегами располагается заголовок документа, обозначаемый при помощи тегов <HEAD> и </HEAD>. Основное содержание HTML-документа располагается между тегами <BODY> и </BODY> (листинг 1.9).

Листинг 1.9. Общая структура HTML-документа

```
<HTML>
  <HEAD>
    <TITLE>
      Заголовок страницы
    </TITLE>
```

```
  </HEAD>
  <BODY>
    Содержимое страницы
  </BODY>
</HTML>
```

Тег <BODY> имеет множество атрибутов, например,

```
<BODY bgcolor=white>
```

Этот тег указывает, что цвет фона страницы будет белый. Помимо этого существуют и другие атрибуты, например, цвет текста (TEXT), цвет ссылок (LINK), цвет посещенных ссылок (VLINK):

```
<BODY bgcolor=black text=white link=blue alink=red vlink=gray>
```

Какие атрибуты связаны с тем или иным HTML-элементом? Ответ на этот вопрос можно найти в документации по HTML или в справках в специализированных редакторах, например, в редакторе SlickEdit.

Вот список всех атрибутов элемента <BODY>, определенных в HTML 4. Однако это не значит, что те или иные браузеры не могут определить и дополнительные атрибуты для элемента <BODY>, значительно расширив перечень используемых атрибутов. Еще раз повторим, что это лишь те атрибуты элемента <BODY>, которые описаны в стандарте HTML 4.

- background = url
- text = color
- link = color
- vlink = color
- alink = color
- id
- class
- lang
- dir
- title
- style
- bgcolor
- onload
- onunloadonclick
- ondblclick
- onmousedown
- onmouseup

- onmouseover
- onmousemove
- onmouseout
- onkeypress
- onkeydown
- onkeyup

Существует набор стандартных названий цветов.

- Black = #000000
- Green = #008000
- Silver = #C0C0C0
- Lime = #00FF00
- Gray = #808080
- Olive = #808000
- White = #FFFFFF
- Yellow = #FFFF00
- Maroon = #800000
- Navy = #000080
- Red = #FF0000
- Blue = #0000FF
- Purple = #800080
- Teal = #008080
- Fuchsia = #FF00FF
- Aqua = #00FFFF

Справа от названия указано соответствующее этому цвету значение в формате RRGGBB. Вместо названия цвета при задании цвета можно использовать этот формат.

1.3.2. Заголовки и шрифты

В HTML предусмотрено несколько элементов для оформления структуры текста. Для выделения заголовков используются теги <H1>, <H2>, ..., <H6>. Они соответствуют заголовкам соответствующих уровней. Эти теги всегда должны иметь парные им закрывающие теги, такие, как </H1>. Заголовки могут быть выровнены с использованием атрибута ALIGN со значениями LEFT, RIGHT, CENTER. Например:

```
<H1 align=center>Заголовок первого уровня</H1>
```

Стиль текста также создается с использованием тегов. Теги <B> и </B> используются для выделения фрагмента текста полужирным шрифтом, тег <i> служит для выделения курсивом, <U> — подчеркнутый текст, <TT> — моноширинный шрифт печатной машинки, <SUB> — нижний индекс, <SUP> — верхний индекс, например (рис. 1.11):

```
x<SUP>2</SUP> — это икс в квадрате
```

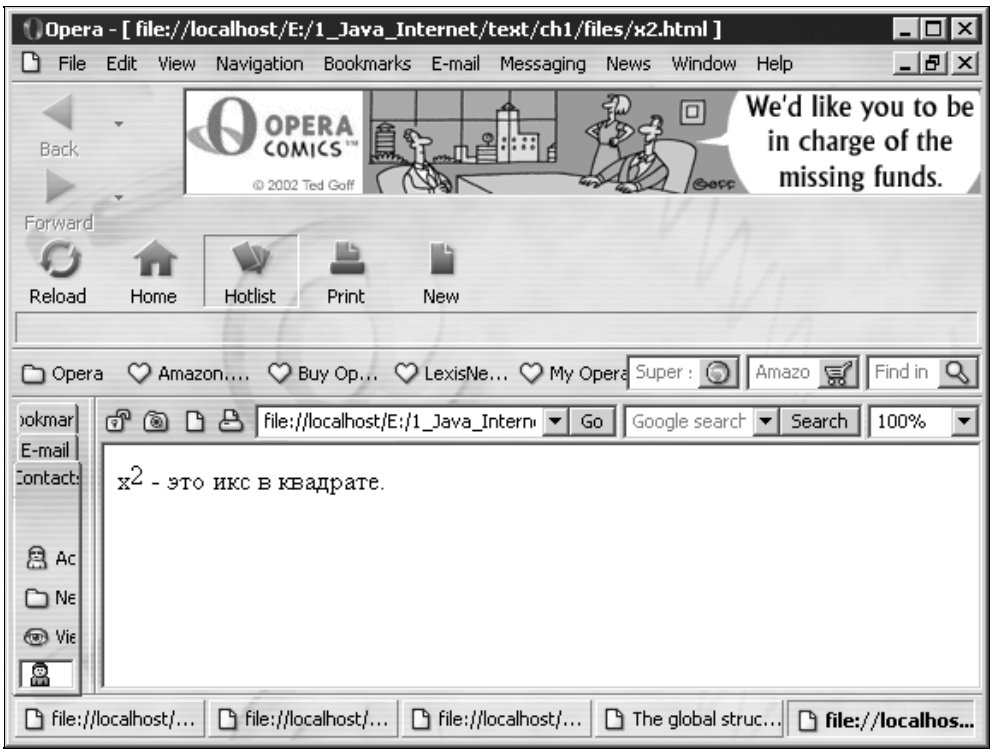


Рис. 1.11. Использование верхнего индекса в HTML

Более гибкое средство управления внешним видом шрифта — это теги <FONT>...</FONT>. Элемент <FONT> имеет свой набор атрибутов, определенных в стандарте HTML 4, например:

```
<FONT color=blue size="+3">Большой текст синего цвета</FONT>
```

1.3.3. Списки

Списки образуются с использованием тегов <UL> и <OL>. <OL> — это упорядоченный список, то есть нумерованный список, <UL> — это список неупо-

рядоченный, маркированный список. Каждый элемент списка начинается с тега `<LI>`. Список заканчивается тегом `</OL>` или `</UL>`, например:

```
<UL>
  <LI>Isaac Asimov
  <LI>Ursula Leguin
  <LI>Greg Bear
  <LI>C. J. Cherryh
</UL>
```

1.3.4. Ссылки

Практически все HTML-документы содержат ссылки на другие HTML-страницы и интернет-ресурсы. Ссылки создаются при помощи тега `<A>`, этот ярлык имеет атрибут `HREF`, значением которого указывается адрес URL (Uniform Resource Locator, универсальный адрес ресурсов), например:

```
<A HREF="http://www.e-olymp.com/">наша ссылка</A>
```

1.3.5. Рисунки

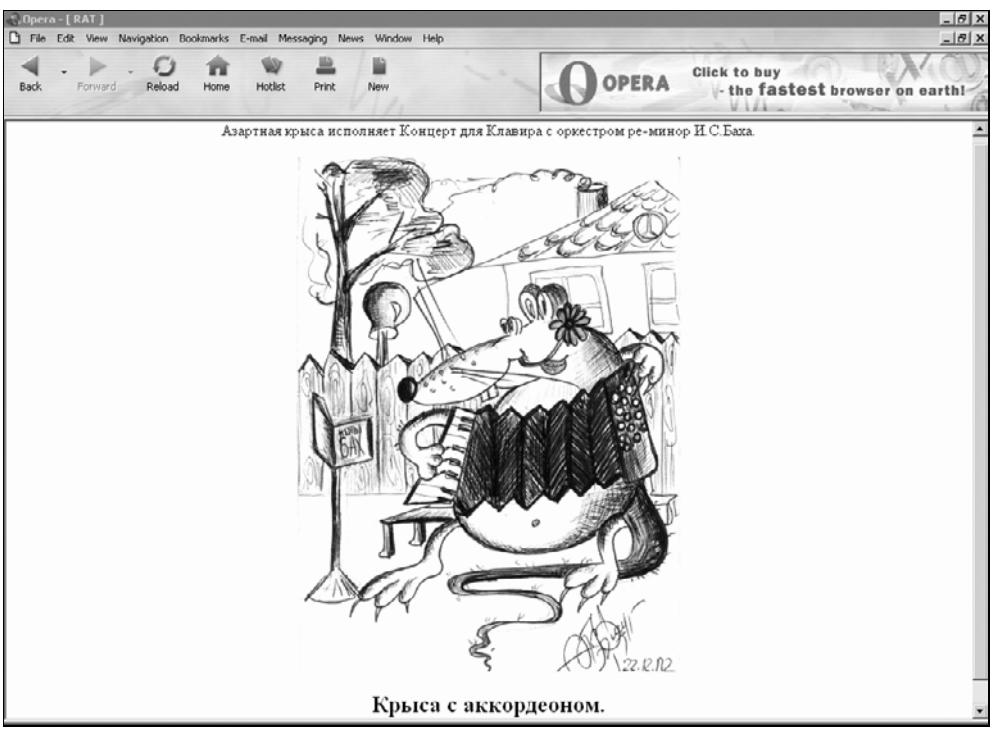


Рис. 1.12. Рисунок, отображенный при помощи HTML

Рисунки вставляются в HTML-страничку с помощью тега <IMG>. Этот тег не имеет парного закрывающего тега, а также нет и тела для соответствующего ему элемента. В качестве основного атрибута указывается URL картинки. Кроме того, можно задать размеры рисунка, при этом картинка будет сжата (или растянута) в соответствии с указанными размерами. Можно указать атрибуты выравнивания, альтернативный текст на тот случай, если картинка не сможет быть загружена в браузер (рис. 1.12):

```
<IMG SRC="figure1.gif" ALIGN=RIGHT HEIGHT=150 WIDTH=100 ALT="Figure 1">
```

1.3.6. Апплеты и параметры апплетов

Этот элемент наиболее важен для нас. Он описывает апплеты, вставляемые в HTML-страничку. Для вставки апплета используется тег <APPLET>. Элемент апплета обязан содержать парный закрывающий тег </APPLET>. Открывающий тег должен содержать атрибут CODE. Значением этого атрибута является файл с классом кода апплета. Помимо этого, открывающий тег содержит параметры размера апплета, то есть ширину и высоту поля, в которой будет показан апплет в окне браузера (параметры HEIGHT и WIDTH). Также можно указать способ выравнивания апплета. Например:

```
<P ALIGN=CENTER>
  <APPLET CODE="HelloWorldApplet.class" HEIGHT=50 WIDTH=150>
  </APPLET>
</P>
```

В этом примере неявно предполагается, что файл HelloWorldApplet.class расположен в том же директории, что и исходный HTML-файл. Если же это не так, то полезно использовать атрибут CODEBASE, в этом атрибуте указывается директория, в которой по умолчанию располагаются файлы классов апплетов. Значением атрибута CODE всегда является имя файла, но не URL.

Если в апплете используется несколько файлов class, то полезно объединить их в файл архива ZIP или JAR (Java ARchive). Это значительно снизит размер передаваемых файлов. Апплеты могут требовать задания значений тех или иных параметров. Значения параметров могут быть переданы апплету с помощью тегов <PARAM>, которые должны быть использованы внутри элемента <APPLET>, то есть должны быть составной частью тела элемента <APPLET>. Параметры должны включать в себя обязательные атрибуты NAME и VALUE, например:

```
<PARAM NAME="имя_параметра" VALUE="значение_параметра">
```

Во время выполнения апплета существует возможность воспользоваться методом getParameter() и с его помощью получить значение параметра:

```
String getParameter(String paramName);
```

Имена параметров чувствительны к регистру, то есть нельзя назвать параметр "parametr", а обратиться к нему с использованием метода getParameter(Parametr).

Если поместить что-либое отличное от элементов <PARAM> между тегами <APPLET> и </APPLET>, то все это будет игнорироваться браузером, который поддерживает Java. Например:

```
<APPLET code="ShowMessage.class" WIDTH=200 HEIGHT=50>
  <PARAM NAME="message" VALUE="Goodbye World!">
  <PARAM NAME="font" VALUE="Serif">
  <PARAM NAME="size" VALUE="36">
  <p align=center>Ваш браузер не поддерживает Java!</p>
</APPLET>
```

Чтение параметров апплета осуществляется при помощи следующих методов:

```
String display;
String fontName;
public void init() {
  String value;
  value = getParameter("message");
  if (value == null)
  display = "Nu net soobscheniya!";
  else
  display = value;
  value = getParameter("font");
    if (value == null)
    fontName = "SansSerif"
    else
    fontName = value;
}  ...
```

1.4. Апплеты и графика

Апплет сам по себе является графическим компонентом. Компонент — это визуальный элемент. К числу компонентов относятся кнопки, меню, текстовые поля, поля для ввода текста, поля с прокруткой, поля для отметки и т. п. Компоненты описываются в виде классов, основанных на классе java.awt.Component. Что касается апплетов Japplet, то большинство компонентов этих апплетов являются подклассами класса javax.swing.JComponent. Каждый компонент отвечает за отображение самого себя своими средствами.

Все, что требуется от программиста, — это вставить требуемый компонент в апплет. Нет необходимости специально заботиться о том, как прорисовать апплет с компонентами на экране, — это происходит автоматически. Если появляется необходимость нарисовать что-либо в составе того или иного компонента, причем нарисовать какой-либо нестандартный элемент, не входящий в набор заранее определенных элементов, то для этого необходимо определить собственный класс компонента, в котором будут заданы методы его прорисовки.

Как уже было сказано в начале этой главы, в простых апплетах (не `Swing`) рисование осуществляется при помощи метода `paint()`. Чтобы прорисовать элемент, необходимо определить подкласс класса `Applet`, вставив в него метод `paint()`, который будет осуществлять рисование. С апплетами `JApplet` ситуация несколько сложнее. Мы не можем рисовать в апплете `JApplet`, мы не можем рисовать ни в одном апплете верхнего уровня, ни в одном компоненте верхнего уровня пакета `Swing`. Для того чтобы иметь возможность рисовать компоненты, необходимо создать новый компонент, который будет использоваться в качестве своего рода холста, и этот компонент должен быть вставлен в апплет `JApplet`. Поэтому `JApplet` будет состоять по крайней мере из двух классов: класса апплета как такового `JApplet` и класса холста. Холст — это не очень правильное название. В данном случае холст — это не то, что определяется в классе `Canvas`, который более всего подходит для того, чтобы носить название холста, однако термин этот в нашей ситуации довольно понятно описывает происходящее. Чаще всего класс для рисования основывается на классе `javax.swing.JPanel`, который представляет собой пустую область на экране. Как и любой компонент `JComponent`, панель `JPanel` использует следующий метод для рисования самой себя:

```
public void paintComponent(Graphics g)
```

Чтобы создать "холст", мы создаем класс на основе `JPanel`, в котором задаем метод `paintComponent()`. Создаем объект, принадлежащий новому классу, вставляем класс в апплет `JApplet`. Когда приходит время прорисовать компонент, то вызывается метод `paintComponent()`. Все это вовсе не так и сложно, но, тем не менее, немного более запутанно, чем в апплетах `Applet`.

Отметим, что в качестве параметра методу `paintComponent()` передается объект типа `Graphics`. Чтобы иметь возможность что-либо рисовать, требуется графический контекст. Графический контекст — это объект класса `java.awt.Graphics`. Этот класс содержит методы рисования, с их помощью можно рисовать формы, выводить текст, создавать картинки. Каждый объект `Graphics` работает только с одним компонентом GUI (Graphic User Interface, графический интерфейс пользователя), принадлежащим подклассу `JComponent`. Класс `Graphics` — это абстрактный класс, то есть мы не можем создать экземпляр этого класса напрямую, используя конструктор. Существует два способа получения графического контекста. Первый способ — вызов функции `paintComponent()` и передача ей графического контекста,

второй способ — использование метода `getGraphics()`. Этот метод есть в каждом компоненте, он используется для получения контекста, который может быть затем использован для рисования вне метода `paintComponent()`.

Существует также метод `repaint()`, который удобно использовать тогда, когда в компоненте произошли изменения:

```
public void repaint();
```

Этот метод определен в классе `Component` и выполняется немедленно, однако при этом не происходит рисования как такового. Метод `paintComponent()` должен быть вызван после того, как будет выполнен метод `repaint()`. Метод `paintComponent()` вызывается в нескольких случаях: когда компонент впервые отображается на экране и когда данный компонент станет видимым вновь после того, как был в течение некоторого времени закрыт другим окном. Эти примеры говорят о том, что метод `paintComponent()` должен быть достаточно гибким.

Перейдем к рассмотрению некоторых моментов, знание которых необходимо для работы с компонентами.

1.4.1. Координаты

Экран компьютера состоит из маленьких элементов — пикселов, которые мы будем представлять в виде квадратов, так что весь экран представляет собой сетку, состоящую из квадратных пикселов одинакового размера. Цвет каждого пиксела может меняться и устанавливается независимо от цвета других пикселов. Рисование на экране сводится к заданию цвета каждого пиксела.

Графический контекст выводится в прямоугольнике, состоящем из пикселов. Положение пиксела в прямоугольнике определяется двумя координатами `(x, y)`. Верхний левый угол имеет координаты `(0, 0)`. Горизонтальная координата `x` увеличивается слева направо, вертикальная координата `y` растет снизу вверх. Координаты и пикселы не связаны раз и навсегда, но положение пикселов удобнее всего описывать при помощи координат. Для каждого компонента можно определить его размер с помощью метода `getSize()`. Этот метод возвращает объект типа `java.awt.Dimension`. Этот объект имеет две переменные (высоту и ширину: `getSize().width` — ширина, `getSize().height` — высота.

При создании апплета чаще всего мы не будем иметь представления о его размере. Размер апплета как правило задается в теге `<APPLET>`. Понятно, что не следует привязываться к фиксированному размеру апплета, размер апплета может меняться даже во время выполнения апплета. Об этом всегда следует помнить. Прочие компоненты апплета еще более "нестабильны" в отношении своих размеров. Поэтому полезно получать размер компонента

перед тем, как нарисовать его. Это можно сделать, например, при помощи следующего кода:

```
public void paintComponent(Graphics g) {
int width =  getSize().width;   // ширина компонента
int height = getSize().height;  // высота компонента
  ...   // рисование контекста компонента
}
```

Во время рисования необходимо знать размер компонента для вычисления значений координат `(x,y)`.

1.4.2. Цвет

В языке Java, как правило, работают с системой цветов RGB. Как известно, любой цвет можно разложить на три независимых составляющих. Каждый цвет можно представить в виде комбинации трех чистых цветов различной интенсивности. В качестве базовых цветов используются красный (обозначаемый буквой R — "red"), зеленый (обозначаемый буквой G — "green") и синий (обозначаемый буквой B — "blue"). Цвет — это объект класса `java.awt.Color`. Новый цвет можно задать с помощью конструктора, указав интенсивности каждого компонента цвета:

```
myColor = new Color(r,g,b);
```

Существует два конструктора цвета. В одном случае цвет задается в виде набора из трех целых значений в диапазоне от 0 до 255. В другом случае три компонента цветовой гаммы в системе RGB описываются вещественными числами в диапазоне от 0.0F до 1.0F. Часто оказывается излишним создание нового цвета, так как в классе `Color` определено несколько констант для часто используемых цветов (табл. 1.1).

Таблица 1.1. *Константы цветов*

Константа	Значение
`Black`	Черный
`Blue`	Синий
`Cyan`	Цианový
`DarkGray`	Темно-серый
`Gray`	Серый
`Green`	Зеленый
`LightGray`	Светло-серый
`Magenta`	Фуксиновый

Таблица 1.1 *(окончание)*

Константа	Значение
`Orange`	Оранжевый
`Pink`	Розовый
`Red`	Красный
`White`	Белый
`Yellow`	Желтый

Помимо системы RGB можно пользоваться системой HSB. В системе HSB цвет также определяется тремя параметрами, поэтому нет возможности создать другой конструктор класса `Color`. Для получения значений HSB для цвета используется метод `getHSBColor`. Чтобы создать цвет на основе трех значений HSB, вызовем этот метод:

```
myColor = Color.getHSBColor(h,s,b);
```

Значения H, S и B являются вещественными, они могут изменяться в пределах от 0.0F до 1.0F. Буквы H, S и B соответствуют словам hue (цвет), saturation (насыщенность) и brightness (яркость).

Вот пример создания случайного цвета:

```
myColor = Color.getHSBColor((float)Math.random(), 1.0F, 1.0F);
```

Обратите внимание на то, что необходимо использовать приведение типа, так как метод `Math.random` возвращает тип `double`.

Полезно поэкспериментировать с цветами, чтобы лучше их почувствовать.

Используйте файл ColorApplet.java (листинг 1.10) для вставки апплета в HTML-страницу ColorApplet.html (листинг 1.11).

Листинг 1.10. Файл ColorApplet.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ColorApplet extends Applet implements AdjustmentListener {
  private float[] hsb = new float[3];   // цвета HSB
  private int r = 0, g = 0, b = 0;      // цвета RGB
  // полосы прокрутки
  private Scrollbar hueScroll, brightnessScroll, saturationScroll,
     redScroll, greenScroll, blueScroll;
  // надписи полос прокрутки
```

```
  private Label hueLabel, brightnessLabel, saturationLabel,
     redLabel, greenLabel, blueLabel;
  private Canvas colorCanvas;  // холст для цветового поля
  public void init() {
   // получить HSB компоненты цвета для RGB = (0,0,0);
  Color.RGBtoHSB(0,0,0,hsb);
  // создание полос прокрутки со значениями от 0 до 255
  hueScroll = new Scrollbar(Scrollbar.HORIZONTAL, (int)(255*hsb[0]), 10, 0, 265);
  saturationScroll = new Scrollbar(Scrollbar.HORIZONTAL,
(int)(255*hsb[1]), 10, 0, 265);
  brightnessScroll = new Scrollbar(Scrollbar.HORIZONTAL,
(int)(255*hsb[2]), 10, 0, 265);
  redScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10, 0, 265);
  greenScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10, 0, 265);
  blueScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10, 0, 265);
  // надписи, показывающие текущие значения цветов
  hueLabel = new Label(" H = " + hsb[0]);
  saturationLabel = new Label(" S = " + hsb[1]);
  brightnessLabel = new Label(" B = " + hsb[2]);
  redLabel = new Label(" R = 0");
  greenLabel = new Label(" G = 0");
  blueLabel = new Label(" B = 0");
  // цвет фона полос прокрутки и надписей
  hueScroll.setBackground(Color.lightGray);
  saturationScroll.setBackground(Color.lightGray);
  brightnessScroll.setBackground(Color.lightGray);
  redScroll.setBackground(Color.lightGray);
  greenScroll.setBackground(Color.lightGray);
  blueScroll.setBackground(Color.lightGray);
  hueLabel.setBackground(Color.white);
  saturationLabel.setBackground(Color.white);
  brightnessLabel.setBackground(Color.white);
  redLabel.setBackground(Color.white);
  greenLabel.setBackground(Color.white);
  blueLabel.setBackground(Color.white);
  // прослушивание изменений значений в полосах прокрутки
  hueScroll.addAdjustmentListener(this);
  saturationScroll.addAdjustmentListener(this);
  brightnessScroll.addAdjustmentListener(this);
```

```
  redScroll.addAdjustmentListener(this);
  greenScroll.addAdjustmentListener(this);
  blueScroll.addAdjustmentListener(this);
  // создание холста с цветом фона, соответствующим текущим значениям
  colorCanvas = new Canvas();
  colorCanvas.setBackground(Color.black);
  // создание внешнего вида апплета из трех областей равного размера,
содержащих полосы прокрутки, метки-надписи и цветное поле */
  setLayout(new GridLayout(1,3,3,3));
  setBackground(Color.gray);
  Panel scrolls = new Panel();
  Panel labels = new Panel();
  add(scrolls);
  add(labels);
  add(colorCanvas);
  // вставка полей прокрутки и меток-надписей в соответствующие панели

  scrolls.setLayout(new GridLayout(6,1,2,2));
  scrolls.add(redScroll);
  scrolls.add(greenScroll);
  scrolls.add(blueScroll);
  scrolls.add(hueScroll);
  scrolls.add(saturationScroll);
  scrolls.add(brightnessScroll);
  labels.setLayout(new GridLayout(6,1,2,2));
  labels.add(redLabel);
  labels.add(greenLabel);
  labels.add(blueLabel);
  labels.add(hueLabel);
  labels.add(saturationLabel);
  labels.add(brightnessLabel);
  } // end init();
  public void adjustmentValueChanged(AdjustmentEvent evt) {
  // данный метод вызывается, когда пользователь меняет положение
  // одной из полос прокрутки
  // при этом все метки и значения цветов
  // изменяются на вновь установленные значения
  int r1, g1, b1;
  r1 = redScroll.getValue();
```

```
g1 = greenScroll.getValue();
b1 = blueScroll.getValue();
if (r != r1 || g != g1 || b != b1)
{
// один из цветов RGB изменен
r = r1;
g = g1;
b = b1;
Color.RGBtoHSB(r,g,b,hsb);
}
else
{
// один из цветов HSB изменен
hsb[0] = hueScroll.getValue()/255.0F;
hsb[1] = saturationScroll.getValue()/255.0F;
hsb[2] = brightnessScroll.getValue()/255.0F;
int rgb = Color.HSBtoRGB(hsb[0],hsb[1],hsb[2]);
r = (rgb >> 16) & 0xFF;
g = (rgb >> 8) & 0xFF;
b = rgb & 0xFF;
}
redLabel.setText(" R = " + r);
greenLabel.setText(" G = " + g);
blueLabel.setText(" B = " + b);
hueLabel.setText(" H = " + hsb[0]);
saturationLabel.setText(" S = " + hsb[1]);
brightnessLabel.setText(" B = " + hsb[2]);
redScroll.setValue(r);
greenScroll.setValue(g);
blueScroll.setValue(b);
hueScroll.setValue((int)(255*hsb[0]));
saturationScroll.setValue((int)(255*hsb[1]));
brightnessScroll.setValue((int)(255*hsb[2]));
colorCanvas.setBackground(new Color(r,g,b));
colorCanvas.repaint();  // перерисовка холста
} // end adjustmentValueChanged
public Insets getInsets() {
// определение пустого пространства
```

```
  // между границей апплета
  // и компонентами (3 пиксела)
  return new Insets(3,3,3,3);
  }
}  // end class ColorChooserApplet
```

Листинг 1.11. Файл ColorApplet.html

```
<p align=center>
  <applet code="ColorApplet.class" width=400 height=150
  alt="(Tut dolzhen byt applet 'ColorApplet'.)">
  <font color="#E70000">
  (Byla by Java, to byl by zeds applet "ColorChooserApplet" <br>.)
  </font>
  </applet>
</p>
```

Рис. 1.13. Изменение цвета

В апплете представлены три полосы прокрутки, соответствующие цветам RGB и HSB. В зависимости от изменения положения ползунков цвет поля будет изменяться. Играя с цветами, можно наблюдать, как зависят друг от друга положения ползунков для этих двух представлений цветов (рис. 1.13).

При изменении цвета (параметр H) с 0 до 1 начиная с красного, постепенно увеличивается интенсивность зеленого (до 1), затем интенсивность красного уменьшается (до 0), на следующем шаге растет интенсивность синего, после чего снижается интенсивность зеленого, затем вновь нарастает интенсивность красного, и, наконец, уменьшается интенсивность зеленого. При этом мы проходим всю палитру, видим все цвета радуги.

1.4.3. Шрифт

Шрифт имеет размер и стиль. Одни и те же символы могут быть показаны разным размером и разным стилем. Для отображения шрифта используются наборы символов. Таким образом, шрифт характеризуется названием, стилем и размером. Названия шрифтов могут меняться от системы к системе, но, как правило, существует стандартный набор шрифтов, которые присутствуют во всех системах, например, Serif, SansSerif, Monospaced, Dialog. В классе `Font` определены, в частности, следующие стили шрифтов:

- `Font.PLAIN`;
- `Font.ITALIC`;
- `Font.BOLD`.

Размер шрифта задается в виде целого числа, по умолчанию используется размер 12. Для представления шрифтов используется класс `java.awt.Font`. Можно создать новый фонт, используя конструктор, например:

```
Font nashFont = new Font("Serif", Font.PLAIN, 12);
Font bolshojFont = new Font("SansSerif", Font.BOLD, 32);
```

Каждый графический контекст обладает текущим шрифтом. Изменить текущий шрифт можно при помощи метода `setFont()`, например, для графического контекста `g` фонт можно поменять с помощью следующего метода:

```
g.setFont(bolshojFont);
```

Узнать, какой текущий шрифт установлен в данном контексте, можно при помощи следующего метода:

```
g.getFont();
```

Этот метод возвращает объект типа `Font`.

Каждый компонент имеет связанный с ним шрифт, который может быть задан при помощи метода `setFont(font)`. Этот метод определен в классе `Component`. При создании графического контекста для компонента, текущий шрифт контекста бывает равен текущему шрифту компонента.

1.4.4. Графические элементы

Класс `Graphics` содержит большое число методов, созданных для рисования различных фигур: линий, прямоугольников и овалов. При этом форма определяется координатами `(x, y)`. Фигуры рисуются текущим цветом рисования графического контекста. Текущий цвет рисования устанавливается равным цвету фона компонента при создании контекста, затем цвет этот может быть изменен при помощи метода `setColor()`.

Ниже приводится список некоторых наиболее часто используемых методов рисования. Если при рисовании часть фигуры оказывается за рамками компонента, то эта часть игнорируется. Все перечисленные ниже методы — это методы класса `Graphics`, их можно вызывать из объектов типа `Graphics`.

Метод *drawString*

Синтаксис:

```
drawString(String str, int x, int y);
```

Выводит текст строки `str`. При выводе текста используется текущий цвет и текущий шрифт графического контекста. Координаты `x` и `y` определяют положение текста, `x` — это положение левого конца строки, `y` — вертикальная координата базовой линии строки (линия, на которой расположены символы букв, некоторые части букв могут заходить ниже базовой линии, например, хвостик буквы `y`).

Метод *drawLine*

Синтаксис:

```
drawLine(int x1, int y1, int x2, int y2);
```

Рисует линию от точки (`x1`, `y1`) к точке (`x2`, `y2`).

Метод *drawRect*

Синтаксис:

```
drawRect(int x, int y, int width, int height);
```

Рисует прямоугольник с верхним левым углом в точке `(x, y)` с высотой `height` и шириной `width`. Для рисования прямоугольника вокруг компонента вдоль его границ используется функция следующего вида:

```
g.drawRect(0, 0, getSize().width-1, getSize().height- 1);
```

здесь `g` — графический контекст.

Метод *drawOval*

Синтаксис:

```
drawOval(int x, int y, int width, int height);
```

Рисует овал внутри прямоугольника с указанными параметрами *(см. "Метод `drawRect()`")*.

Метод *drawRoundRect*

Синтаксис:

```
drawRoundRect(int x, int y, int width, int height,
              int xdiam, int ydiam);
```

Рисует прямоугольник с закругленными углами. Основной прямоугольник задается первыми четырьмя параметрами `x`, `y`, `width`, `height`. Степень закругления определяется параметрами `xdiam` и `ydiam`. *Закругления* — это дуги эллипса с горизонтальным диаметром `xdiam` и вертикальным диаметром `ydiam`. Часто используются значения для диаметров, равные 16.

Метод *draw3DRect*

Синтаксис:

```
draw3DRect(int x, int y, int width, int height, boolean raised);
```

Рисует "трехмерный" (с оттенением) прямоугольник.

Метод *drawArc*

Синтаксис:

```
drawArc(int x, int y, int width, int height, int startAngle,
        int arcAngle);
```

Рисует дугу эллипса, вписанного в прямоугольник с указанными параметрами (координаты верхнего левого угла — `x` и `y`, высота — `height`, и ширина — `width`), параметры `arcAngle` и `startAngle` — это угол арки и начальный угол в градусах соответственно. При этом 0 градусам соответствует положение часовой стрелки на 3 часах, отсчет ведется против часовой стрелки. Для рисования дуги окружности следует установить высоту прямоугольника равной его ширине, эллипс, вписанный в квадрат — это окружность.

Метод *fillRect*

Синтаксис:

```
fillRect(int x, int y, int width, int height);
```

Рисует закрашенный прямоугольник (здесь `x` и `y` — координаты верхнего левого угла прямоугольника, `width` — его ширина, а `height` — высота). Для заполнения цветом всего компонента следует использовать метод

```
g.fillRect(0, 0, getSize().width, getSize().height);
```

Метод *fillOval*

Синтаксис:

```
FillOval(int x, int y, int width, int height);
```

Рисует закрашенный овал.

Метод *fillRoundRect*

Синтаксис:

```
fillRound(int x, int y, int width, int height, int xdiam, int ydiam);
```

Рисует закрашенный прямоугольник со скругленными углами (здесь `x` и `y` — координаты верхнего левого угла прямоугольника, `width` — его ширина, а `height` — высота, `xdiam` и `ydiam` — диаметры скругляющего эллипса по осям x и y соответственно.

Метод *fill3DRect*

Синтаксис:

```
fill3DRect(int x, int y, int width, int height, boolean raised);
```

Рисует закрашенный прямоугольник с трехмерным эффектом.

Метод *fillArc*

Синтаксис:

```
fillArc(int x, int y, int width, int height, int startAngle,
        int arcAngle);
```

Рисует закрашенный сегмент (здесь `x` и `y` — координаты верхнего левого угла прямоугольника, `width` — его ширина, а `height` — высота, `startAngle` — начальный угол сегмента, `arcAngle` — угловой размер сегмента).

Рассмотрим пример с рисованием. Напишем апплет `JApplet`. Используем компоненты `JComponent`, которые вставим в класс, созданный на основе класса `JPanel`. Этот класс будет вложен внутрь основного класса апплета. Рисование будет осуществлено при помощи метода `paintComponent()`.

Апплет (листинги 1.12 и 1.13) осуществляет рисование большого числа копий текста на темно-сером фоне. Каждая копия текста имеет случайный цвет, используется пять шрифтов разного размера и стиля. Строка содержит текст "Vsem Privet!". При загрузке апплета в окне браузера появляется апплет со случайным образом расположенными надписями разного цвета, размера и стиля (рис. 1.14). При повторной загрузке апплета расположение надписей меняется (рис. 1.15). Этот апплет работает нормально при любой величине, которая указывается в теге `<applet>`.

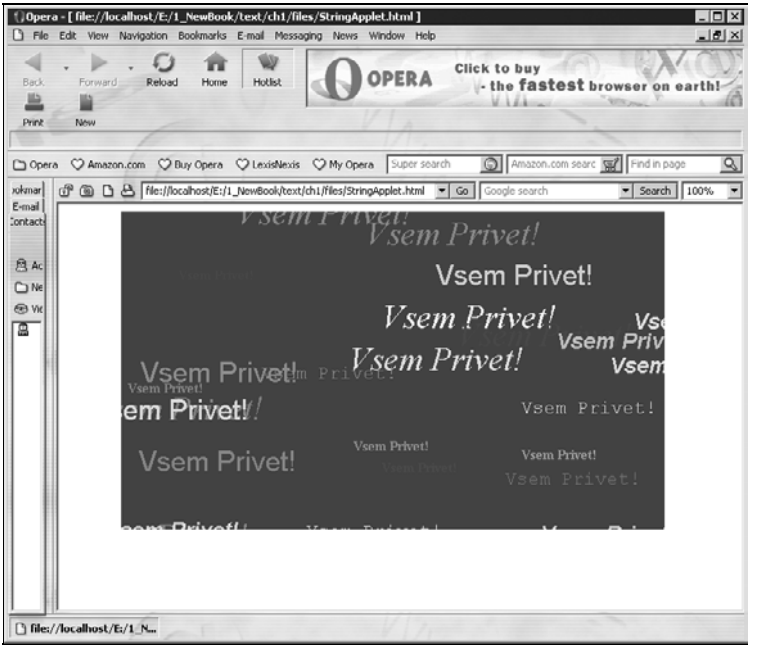


Рис. 1.14. Загрузка апплета "Vsem Privet!"

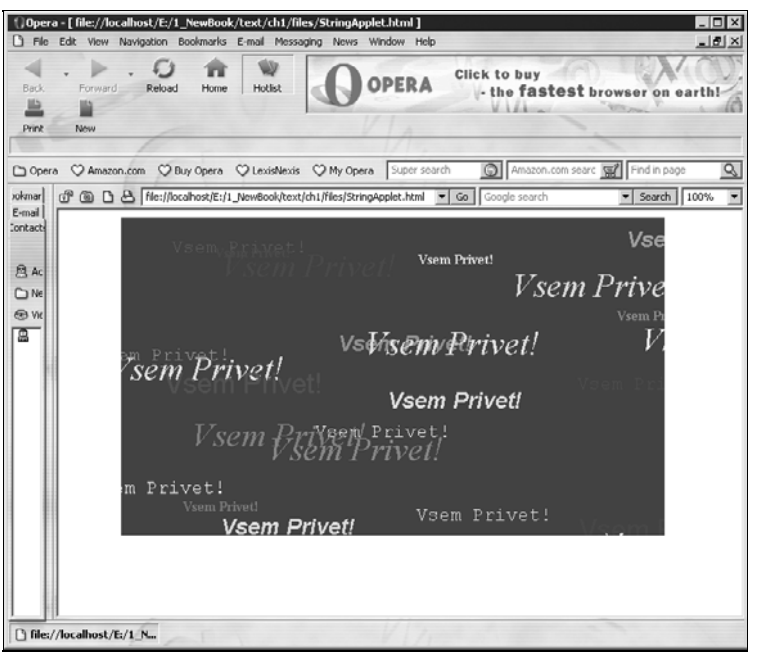


Рис. 1.15. Повторная загрузка того же апплета

Этот апплет воспроизводит 33 экземпляра строки, расположенных в разных местах, отображающихся разным цветом, стилем и пятью шрифтами на темно-сером фоне. При каждом обращении к методу `paintComponent()` генерируются новые случайные значения. При компиляции создается два класса `RandomStrings.class` и `RandomStrings$Display.class`. Оба требуются для работы апплета.

Листинг 1.12. Файл StringApplet.java

```
import java.awt.*;
import javax.swing.*;
public class StringApplet extends JApplet {
  String message;  // сообщения для отображения
  Font font1, font2, font3, font4, font5;  // пять шрифтов
  Display drawingSurface;  // объект для рисования — холст
  public void init() {
    // автоматически вызывается для инициализации апплета
    message = getParameter("message");
    if (message == null)
      message = "Vsem Privet!";
    font1 = new Font("Serif", Font.BOLD, 14);
    font2 = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
    font3 = new Font("Monospaced", Font.PLAIN, 20);
    font4 = new Font("Dialog", Font.PLAIN, 30);
    font5 = new Font("Serif", Font.ITALIC, 36);
    drawingSurface = new Display();
    drawingSurface.setBackground(Color.darkGray);
    setContentPane(drawingSurface);  // панель апплета заменяем холстом
                                     // drawingSurface
  }
  class Display extends JPanel {
  // панель для отображения компонентов
  public void paintComponent(Graphics g) {
  super.paintComponent(g);  // метод paintComponent вызывается из
                            // суперкласса JPanel
  int width = getSize().width;   // ширина компонента width
  int height = getSize().height; // высота компонента height
    for (int i = 0; i < 33; i++)
    {
    // шрифт выбирается случайным образом
```

```
    int fontNum = (int)(5*Math.random()) + 1;
      switch (fontNum)
      {
      case 1:
        g.setFont(font1);
        break;
      case 2:
        g.setFont(font2);
        break;
      case 3:
        g.setFont(font3);
        break;
      case 4:
        g.setFont(font4);
        break;
      case 5:
        g.setFont(font5);
        break;
       } // конец switch
   // выбор цвета
   float hue = (float)Math.random();
   g.setColor(Color.getHSBColor(hue, 1.0F, 1.0F));
   // выбор положения
   int x,y;
   x = -50 + (int)(Math.random()*(width+40));
   y = (int)(Math.random()*(height+20));
   // вывод строки
   g.drawString(message,x,y);
   }
   }
  }
}
```

Листинг 1.13. Файл StringApplet.html

```
<p align=center>
  <applet code="StringApplet.class" width=580 height=340>
  </applet>
</p>
```

1.5. События мыши

В предыдущем разделе были рассмотрены основные понятия и методы создания графического пользовательского интерфейса. Этот раздел посвящен основам работы с событиями, которые возникают при использовании мыши. В качестве примеров мы создадим четыре апплета. Первый апплет выводит текст, как это делалось в предыдущем разделе, с той разницей, что при щелчке мышью меняются параметры выводимого текста (размер, стиль, цвет, местоположение). Второй апплет выводит прямоугольник в месте щелчка мышью. Третий апплет показывает в окне текущие координаты мыши, и наконец, четвертый апплет будет использоваться для рисования.

Обработка событий в апплетах представляется довольно важной, поскольку события позволяют оживить апплет, делая его зависимым от действий пользователя. События предоставляют механизм обратной связи с пользователем. Наиболее важными являются события, наступающие при нажатии клавиши мыши или клавиатуры, а также при движении мыши. При возникновении таких событий компьютер будет соответствующим образом реагировать на них, производя те или иные действия.

В языке Java события представлены в виде объектов. При возникновении того или иного события компьютер собирает связанные с этим событием данные и создает подходящий объект, в котором эти данные представлены. Объектам разных классов соответствуют различные типы событий. Так, при нажатии кнопки мыши создается объект типа `MouseEvent`. Этот объект содержит всю необходимую информацию, в частности, информацию о том, в каком компоненте произошло это событие, какая именно кнопка мыши была нажата. При нажатии клавиши клавиатуры компьютера возникает событие типа `KeyEvent`. После того, как объект события будет создан, параметры, собранные и хранимые в объекте, передаются процедурам обработки события. Создание функций обработки событий — задача разработчика приложения. При создании таких функций мы будем рассматривать события в целом, без детализации. Java предоставляет высокоуровневый механизм программирования при обработке событий. Нас не будет интересовать весь арсенал действий, выполняемых между физическим событием (нажатием кнопки) и началом выполнения той или иной функции обработки этого события, полезно это лишь понимать и всегда иметь в виду. Где-то скрытно он нас работает механизм, который во время выполнения программы все время ожидает наступления событий, и как только какое-либо событие наступит, будет вызвана соответствующая функция обработки этого события. Данный процесс организован в виде цикла ожидания событий или цикла событий. Всякая программа, работающая с графическим интерфейсом пользователя, имеет цикл событий. Нам нет необходимости создавать цикл событий, он уже создан для нас. В этом разделе мы будем работать с событиями с использованием языка Java.

Чтобы обработать событие, программа должна обнаружить наступившее событие, а для этого программа должна прослушивать события. Это осуществляется при помощи прослушивателя событий. Прослушиватель событий — это объект, который содержит методы обработки событий, наступления которых он ждет. Если объект служит для прослушивания событий типа `MouseEvent`, то этот объект должен содержать в себе примерно такой метод:

```
public void mousePressed(MouseEvent evt) { ... }
```

Этот метод определяет способ обработки наступившего события в соответствии с параметрами события. Метод использует в качестве параметра объект `evt`, который содержит всю доступную информацию о событии. Эта информация и используется для того, чтобы прореагировать на событие соответствующим образом.

Методы, которые используются в прослушивателе событий мыши, описаны в интерфейсе `MouseListener`. Чтобы иметь возможность воспользоваться этими методами, необходимо имплементировать данный интерфейс. Каждое событие в языке Java связано с тем или иным компонентом графического интерфейса. Прежде чем прослушиватель сможет реагировать на наступающие события, он должен быть зарегистрирован в соответствующем ему компоненте. Допустим, что мы создаем объект `mListener` типа `MouseListener`, который будет прослушивать события мыши в компоненте `comp`, то вставка прослушивателя в этот компонент осуществляется с помощью такой строки кода:

```
comp.addMouseListener(mListener);
```

Метод `addMouseListener()` — это метод класса `Component`, его можно использовать со всеми компонентами GUI. В наших примерах мы будем использовать прослушиватели в компоненте `JPanel`, который будет представлять собой "холст" для отображения всех прочих компонентов в апплетах `JApplet`. Классы событий (например, класс `MouseEvent`), и интерфейсы прослушивателей (например, `MouseListener`) определены в пакете `java.awt.event`. При работе с событиями этот пакет следует вставить в файл с кодом:

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

Еще раз заострим наше внимание на основных моментах, которые необходимо соблюдать при работе с событиями.

1. Вставить пакет java.awt.event в начало файла с кодом
   ```
   import java.awt.event.*;
   ```
2. Объявить о том, что определенный класс является имплементацией интерфейса прослушивателя, например, интерфейса `MouseListener`.
3. Создать методы обработки событий в соответствии с методами, описанными в интерфейсе прослушивателя.
4. Зарегистрировать прослушиватель в компоненте, события в котором он будет прослушивать. Для этого используется метод объекта `Component`, например, метод `addMouseListener()` для прослушивания событий мыши.

В качестве прослушивателя событий может быть использован любой объект, для этого необходимо лишь указать, что соответствующий класс является имплементацией интерфейса прослушивателя событий данного типа. Компонент может являться прослушивателем событий, которые генерируются в нем самом. Апплет может быть прослушивателем событий, которые наступают в компонентах апплета. В качестве прослушивателя можно использовать специально созданный класс, также часто используются иерархически встроенные анонимные классы.

1.5.1. Класс *MouseEvent* и интерфейс *MouseListener*

Интерфейс `MouseListener` описывает пять методов:

```
public void mousePressed(MouseEvent evt);
public void mouseReleased(MouseEvent evt);
public void mouseClicked(MouseEvent evt);
public void mouseEntered(MouseEvent evt);
public void mouseExited(MouseEvent evt);
```

Метод `mousePressed` вызывается сразу после того, как пользователь нажмет одну из кнопок мыши. Метод `mouseReleased` выполняется после того, как кнопка мыши будет отпущена. Эти два метода нашли наиболее широкое применение. Следует иметь в виду, что в теле класса прослушивателя событий должны быть определены все методы соответствующего интерфейса прослушивателя. Если не требуется реагировать на те или иные события, то тело соответствующего метода следует оставить пустым.

Метод `mouseClicked` вызывается при щелчке мышью (двойной щелчок — нажать и быстро отпустить кнопку мыши). Следует, однако, иметь в виду, что при наступлении события `mouseClicked` будут вызваны также и методы обработки всех трех событий, из которых состоит событие `mouseClicked`, а именно, методы `mousePressed`, `mouseReleased` — в соответствующей последовательности. После выполнения этих методов будет выполнен метод `mouseClicked`. Метод `mouseEntered` вызывается тогда, когда курсор мыши входит в указанный компонент, а метод `mouseExited` — в том случае, когда мышь покидает этот компонент. Эти методы полезно использовать тогда, когда требуется произвести какие-либо действия, например, изменить внешний вид компонента, когда курсор находится над данным компонентом.

В качестве первого примера рассмотрим измененный вариант апплета, который мы уже изучали в предыдущем разделе. Этот апплет будет изменять свой вид при каждом щелчке мыши над апплетом (листинг 1.14).

Мы создадим новый класс апплета `ClickString`, используя созданный нами ранее класс апплета `AppletString` в качестве суперкласса.

Листинг 1.14. Файл ClickString.java

```
import java.awt.event.*;
public class ClickString extends StringApplet
              implements MouseListener {

  public void init()
  {
  /* После создания апплета инициализируется суперкласс
     RandomStrings.
     Апплет будет прослушивать события мыши на
     холсте drawingSurface.
     Переменная drawingSurface была определена
     в классе  RandomStrings —
     это прямоугольная область, заполняющая весь апплет. */
  super.init();
  drawingSurface.addMouseListener(this);
  }
  public void mousePressed(MouseEvent evt)
  {
  /* при нажатии кнопки мыши вызывается метод paintComponent()
     в компоненте drawingSurface */
    drawingSurface.repaint();
  }
 public void mouseEntered(MouseEvent evt) { }
  public void mouseExited(MouseEvent evt) { }
  public void mouseClicked(MouseEvent evt) { }
  public void mouseReleased(MouseEvent evt) { }
}
```

Апплет (листинг 1.15) загружается в браузер с помощью HTML-файла ClickString.html (рис. 1.16 и рис. 1.17).

Листинг 1.15. Файл ClickString.html

```
<p align=center>
<applet code="ClickString.class" width=650 height=350
   alt="(Applet 'ClickString' should be displayed here.)">
   <font color="#E70000">
   (Applet "ClickString" would be displayed here<br>
   if Java were available.)</font>
</applet>
</p>
```

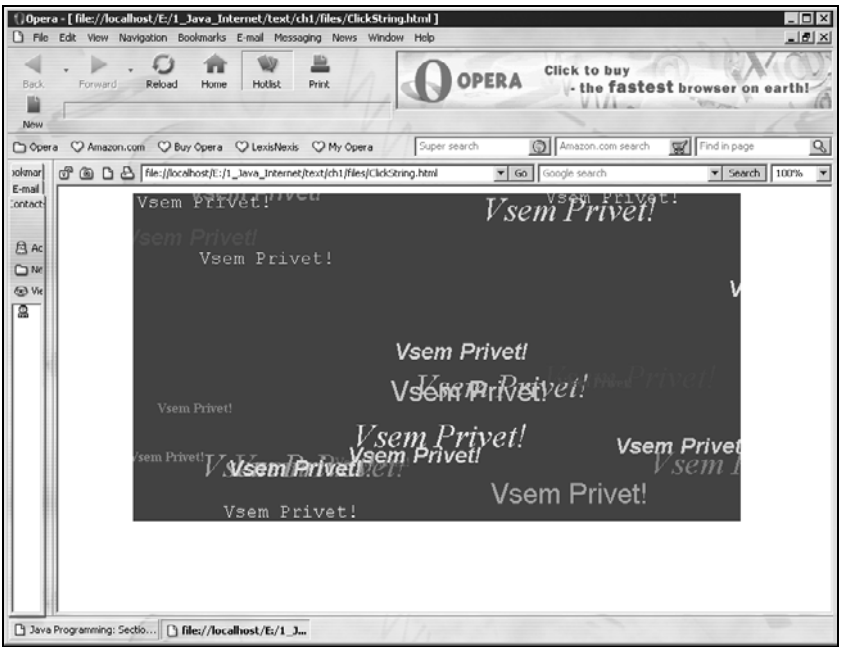


Рис. 1.16. Вид апплета сразу после загрузки

Рис. 1.17. Вид апплета после нажатия кнопки мыши

Класс апплета описан с помощью следующей строки:

```
public class ClickString extends StringApplet
             implements MouseListener { ... }
```

Это не единственный важный момент. При написании этого апплета мы используем пакет `event`:

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

Кроме того, необходимо имплементировать методы интерфейса прослушивателя событий мыши, то есть следующие функции:

```
public void mousePressed(MouseEvent evt)
  {
  drawingSurface.repaint();
  }
public void mouseEntered(MouseEvent evt) { }
public void mouseExited(MouseEvent evt) { }
public void mouseClicked(MouseEvent evt) { }
public void mouseReleased(MouseEvent evt) { }
```

Четыре из пяти функций мы фактически не используем, то есть оставляем их пустыми.

Наконец, прослушиватель должен быть вставлен в компонент. В нашем случае, это "холст" `drawingSurface`:

```
drawingSurface.addMouseListener(this);
```

Все эти изменения можно сделать внутри класса `AppletString`, но мы поступили иначе, создав отдельный самостоятельный класс.

В следующем примере мы рассмотрим часто встречающийся случай, когда требуется узнать местоположение курсора мыши в момент наступления того или иного события. Эти данные передаются на объект `evt`. Чтобы получить координаты, используются методы `evt.getX()` и `evt.getY()`. Эти методы возвращают целые значения, соответствующие координатам положения курсора относительно координат компонента. Верхний левый угол компонента имеет координаты `(0, 0)`. При наступлении события на клавиатуре компьютера могут быть нажаты клавиши-модификаторы, такие как <Shift> или <Ctrl>. Для обработки событий объект события содержит специальные методы проверки нажатия этих клавиш, а именно методы `evt.isShiftDown()`, `evt.isControlDown()`, `evt.isAltDown()` и `evt.isMetaDown()`, которые возвращают логические значения. Эти методы могут быть вызваны для того, чтобы проверить, была ли нажата та или иная клавиша-модификатор в момент наступления события.

Чтобы определить, какая именно клавиша мыши была нажата, используется метод `evt.isMetaDown()`. В качестве примера приведем такой апплет, где при нажатии левой кнопки мыши появляется красный прямоугольник

в месте щелчка, а при нажатии правой кнопки мыши будет нарисован синий овал в том месте, где произошел щелчок.

Код для класса `TwoClicks` приведен ниже (листинг 1.16).

Листинг 1.16. Файл TwoClicks.java

```
/*
Демонстрация работы с событиями мыши.
При нажатии левой кнопки мыши появляется красный прямоугольник
в том месте, где щелкнула мышь.
При нажатии правой кнопки мыши появляется синий овал.
Нарисованные фигуры будут стерты после того,
как апплет исчезнет из поля видимости.
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TwoClicks extends JApplet
  {
  public void init()
  {
    // инициализация апплета, дисплей
    // Display будет содержать панель для отображения
    // холста апплета, на котором рисуются фигуры
    Display display = new Display();
    setContentPanel(display);
  }
class Display extends JPanel implements MouseListener
  {
    // панель апплета, содержащая холст для рисования
    Display() {
      // конструктор задает цвет фона
      // панели и устанавливает прослушиватель
      setBackground(Color.yellow);
      addMouseListener(this);
    }
public void mousePressed(MouseEvent evt)
  {
  // Метод обработки события
  // нажатия кнопки.
  // Панель является прослушивателем
  // для самой себя.
```

```
  if (evt.isShiftDown())
  {
    // При удерджании клавиши <Shift>,
    // вызывается метод repaint.
    // Этот метод определен в суперклассе JPanel.
    // Он производит закраску фона
    // панели одним цветом.
    repaint();
    return;
    }
  int x = evt.getX();  // горизонтальная координата щелчка мыши
  int y = evt.getY();  // вертикальная координата щелчка мыши
  Graphics g = getGraphics();  // графический контекст рисования
                               // в панели JPanel
  if (evt.isMetaDown())
    {
    // Произведен щелчок правой кнопкой мыши в точке (x,y).
    // Прорисовка синего овала с центром в точке (x,y).
    // Черный контур делает овал более
    // отчетливым и заметным.
    g.setColor(Color.blue);
    g.fillOval(x — 55, y — 25, 110, 50);
    g.setColor(Color.black);
    g.drawOval(x — 55, y — 25, 110, 50);
    }
    else
    {
    // Произведен щелчок левой кнопкой мыши в точке (x,y).
    // Прорисовка красного прямоугольника с центром в (x,y).
    g.setColor(Color.red);
    g.fillRect(x — 55, y — 25, 110, 50);
    g.setColor(Color.black);
    g.drawRect(x — 55, y — 25, 110, 50);
    }
    g.dispose();  // работа с графическим контекстом завершена
  }
  // пустые функции в соответствии с интерфейсом MouseListener
  public void mouseEntered(MouseEvent evt) { }
  public void mouseExited(MouseEvent evt) { }
  public void mouseClicked(MouseEvent evt) { }
  public void mouseReleased(MouseEvent evt) { }
  }
}
```

Файл для загрузки апплета — файл TwoClicks.html (рис. 1.18).

Листинг 1.17. Файл TwoClicks.html

```
<p align=center>
  <applet code="TwoClicks.class" width=700 height=380
   alt="(Applet 'TwoClicks' should be displayed here.)">
   <font color="#E70000">
   (Applet "TwoClicks" would be displayed here<br>
    if Java were available.)</font>
  </applet>
</p>
```

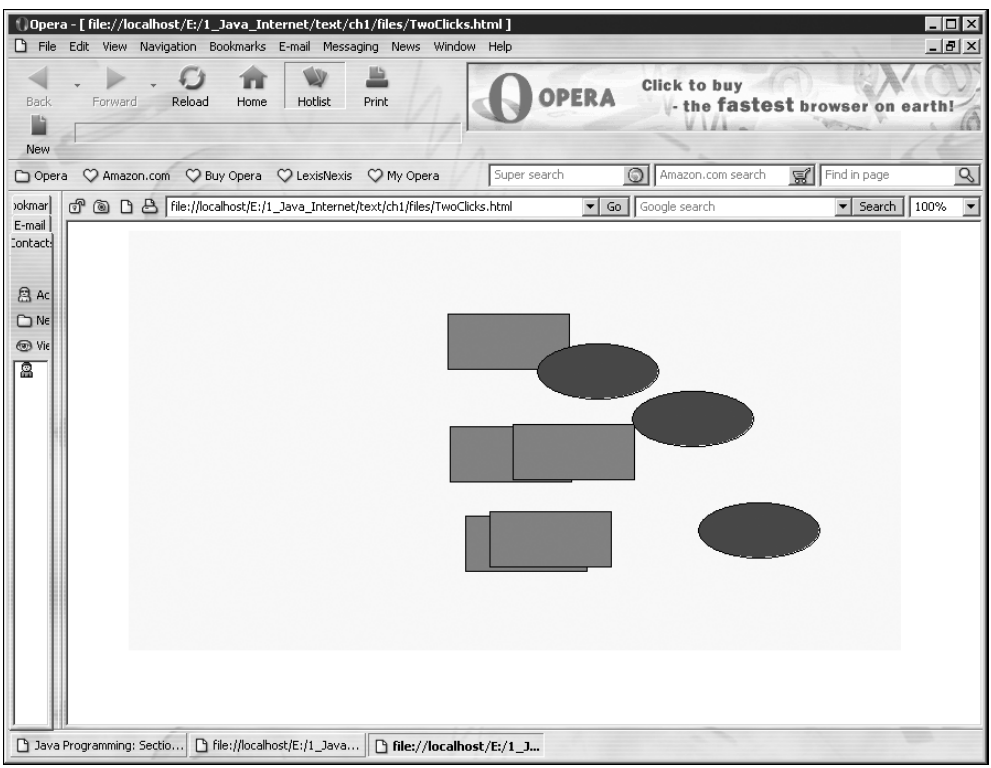


Рис. 1.18. Апплет реагирует на нажатия кнопок мыши и рисует фигуры

Здесь мы снова использовали `JApplet`, в который поместили созданный нами класс `Display`, внутри которого был определен холст (листинг 1.17). В классе `Display` содержится имплементация интерфейса `MouseListener`,

в класс вставлен прослушиватель addMouseListener(this), указатель this относится к классу Display, поскольку метод был вызван именно из этого класса. Иными словами, все события мыши, сгенерированные в объекте Display, будут направлены самому объекту Display.

Класс Display нарушает правило, согласно которому все рисование должно быть осуществлено при помощи метода paintComponent(). У нас же рисование прямоугольников и овалов осуществляется непосредственно с использованием mousePressed(). Это стало возможным потому, что мы предварительно получили графический контекст, указав g = getGraphics(). Затем для экономии ресурсов системы мы освобождаем графический контекст при помощи g.dispose().

1.5.2. Движение мыши. Перетаскивание

Во время своего движения мышь всегда создает события. Эти события можно использовать и реагировать на них. Важным моментом при работе с событиями движения мыши является перетаскивание. Перетаскивание — это перемещение элементов из одного места на экране компьютера в другое при удерживаемой в нажатом состоянии кнопки мыши.

Методы для работы с событиями движения мыши описаны в интерфейсе MouseMotionListener. Интерфейс определяет два метода:

```
public void mouseDragged(MouseEvent evt);
public void mouseMoved(MouseEvent evt);
```

Метод mouseDragged вызывается тогда, когда мышь передвигается при нажатой кнопке. Если при движении мыши кнопка не нажата, то вызывается метод mouseMoved. Параметр evt — это событие типа MouseEvent. Этот параметр содержит такую информацию, как, например, координаты курсора мыши. В процессе движения мыши один из этих методов будет вызываться снова и снова постоянно. Интерфейс определен отдельно, чтобы оставить возможность использовать события мыши, описанные в интерфейсе MouseListener без привлечения без необходимости методов обработки событий, связанных с перемещением мыши.

Для обработки событий перемещения мыши необходимо создать объект, имплементирующий интерфейс MouseMotionListener, зарегистрировать этот объект в качестве прослушивателя событий перемещения мыши. Для этого вызывается метод addMouseMotionListener. Этот объект будет прослушивать события mouseDragged и mouseMoved, связанные с тем компонентом, где зарегистрирован прослушиватель. В большинстве случаев объект прослушивателя также имплементирует и интерфейс MouseListener, тогда прослушиватель сможет реагировать и на события нажатия кнопок мыши. Для объекта drawingSurface определение класса апплета осуществимо, например, следующим образом:

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

```
import javax.swing.*;
  public class Mouser extends JApplet
    implements MouseListener, MouseMotionListener
  {
  public void init()
    {
    // инициализация апплета
    drasingSurface.addMouseListener(this);
    drawingSurface.addMouseMotionListener(this);
    ... // прочие фукнции инициализации
    }
   ...  // методы MouseListener
   ...  // методы MouseMotionListener
   ...  // прочие методы и переменные
  }
```

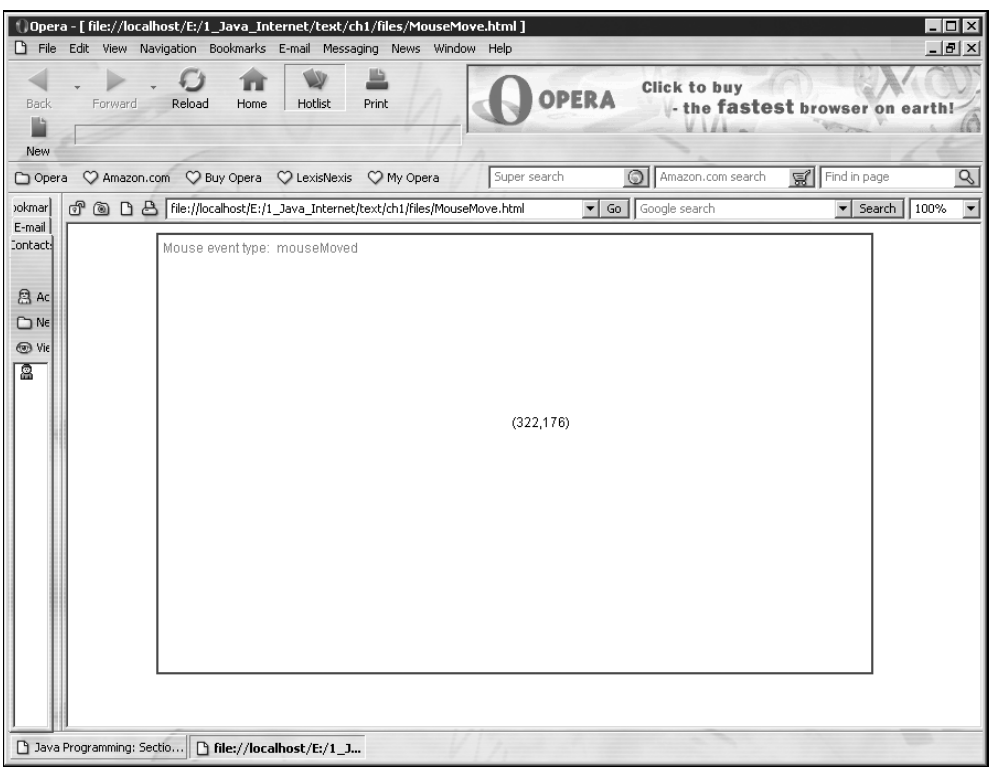


Рис. 1.19. Движение мыши

Код апплета содержится в файле MouseMove.java (листинг 1.18), апплет загружается в браузер с использованием HTML-страницы MouseMove.html (листинг 1.19). При перемещении указателя мыши по апплету, рядом с курсором возникает изображение, соответствующее текущим координатам курсора относительно компонента. Кроме того, апплет отображает и последнее наступившее событие, если были нажаты клавиши-модификаторы, нажатые при наступлении этого события (рис. 1.19).

Листинг 1.18. Файл MouseMove.java

```
/*
Простой апплет, в котором отображаются координаты мыши.
Двойная буферизация не использована, поэтому
экран слегка мерцает.
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MouseMove extends JApplet
    implements MouseListener, MouseMotionListener {
  Display display;
  int mouse_x, mouse_y;      // положение мыши
  String modifierKeys = "";  // если не null, то нажаты клавиши-
                             // модификаторы
  String eventType = null;   // если не null, то тип последнего
                             // события мыши
  public class Display extends JPanel {
  /*
  Класс, в котором задается холст для рисования апплета.
  Здесь также задаются переменные мыши:
  mouse_x, mouse_y, modifierKeys, eventType.
  */
  public void paintComponent(Graphics g) {
  // панель рисования и вывода информации о положении мыши
  super.paintComponent(g);
  g.setColor(Color.blue);
  g.drawRect(0,0, getSize().width - 1,getSize().height - 1);
  g.drawRect(1,1, getSize().width - 3,getSize().height - 3);
  g.setColor(Color.red);
  if (eventType == null) {
```

```
  // если null, то событий мыши не было
  // и информацию выводить не надо
  return;
  }
  g.drawString("Mouse event type:  " + eventType, 6, 18);
  if (modifierKeys.length() > 0)
  g.drawString("Modifier keys:   " + modifierKeys, 6, 34);
  g.setColor(Color.black);
  g.drawString("(" + mouse_x + "," + mouse_y + ")",
  mouse_x, mouse_y);
  }
}
  public void init()
  {
  // задание цвета фона, регистрация прослушивателей
  display = new Display();
  setContentPane(display);
  display.setBackground(Color.white);
  display.addMouseListener(this);
  display.addMouseMotionListener(this);
  }
   void setInfo(MouseEvent evt)
  {
  // задание информации о событиях для ее отображения
  mouse_x = evt.getX();
  mouse_y = evt.getY();
  modifierKeys = "";
  if (evt.isShiftDown())
    modifierKeys += "Shift  ";
  if (evt.isControlDown())
    modifierKeys += "Control  ";
  if (evt.isMetaDown())
    modifierKeys += "Meta  ";
  if (evt.isAltDown())
    modifierKeys += "Alt";
   display.repaint();
   }
   // Имплементация всех методов интерфейсов MouseListener
   // и MouseMotionListener.
```

```
    // Каждый метод задает значение для eventType
    // и вызывает setInfo для получения информации для отображения.
    public void mousePressed(MouseEvent evt) {
  eventType = "mousePressed";
  setInfo(evt);
    }
    public void mouseReleased(MouseEvent evt) {
  eventType = "mouseReleased";
  setInfo(evt);
    }
    public void mouseClicked(MouseEvent evt) {
  eventType = "mouseClicked";
  setInfo(evt);
    }
    public void mouseEntered(MouseEvent evt) {
  eventType = "mouseEntered";
  setInfo(evt);
    }
    public void mouseExited(MouseEvent evt) {
  eventType = "mouseExited";
  setInfo(evt);
    }
    public void mouseMoved(MouseEvent evt) {
  eventType = "mouseMoved";
  setInfo(evt);
    }
    public void mouseDragged(MouseEvent evt) {
  eventType = "mouseDragged";
  setInfo(evt);
    }
}
```

Листинг 1.19. Файл MouseMove.html

```
<p align=center>
<applet code="SimpleTrackMouse.class" width=650 height=400
    alt="(Applet 'MouseMove' should be displayed here.)">
    <font color="#E70000">
    (Applet "MouseMove" would be displayed here<br>
```

```
   if Java were available.)</font>
</applet>
```

В общем случае обработка событий перетаскивания осуществляется с использованием методов обработки событий `mousePressed()`, `mouseDragged()`, `mouseReleased()`. Код может иметь примерно следующий вид.

```
 private int prevX, prevY;  // координаты мыши
 private boolean dragging;  // равно true, если перетаскивание
                            // все еще происходит
   ... // прочие переменные
public void mousePressed(MouseEvent evt)
{
  if (мы_хотим_начать_перетаскивание)
  {
  dragging = true;
  prevX = evt.getX();  // запомнили исходное положение
  prevY = evt.getY();
  }
   ... // прочие функции
}
  public void mouseDragged(MouseEvent evt) {
  // проверка, происходит ли перетаскивание
  if (dragging == false)
    return;
  int x = evt.getX();    // текущее положение мыши
  int y = evt.getY();
   ...  // произвели перемещение от (prevX, prevY) к (x,y)
    prevX = x;  // запомнили положение для последующего вызова
    prevY = y;
  }
  public void mouseReleased(MouseEvent evt) {
  // проверка того, происходит ли перетаскивание
  if (dragging == false)
    return;
  dragging = false; // перетаскивание завершено
  ...  // прочие функции и высвобождение ресурсов
  }
```

Типичный пример использования событий перетаскивания — рисование. В данном примере (листинг 1.20) пользователь может рисовать цветные

кривые, а также выбирать цвет, щелкая на соответствующем цветовом квадрате, возможность очистить поле для рисования предусмотрена нажатием на кнопку **Ochistim**. Этот пример содержит и другие приемы, используемые при программировании графики. Этот апплет, тем не менее, не избавлен от того недостатка, что все нарисованное пропадает, если апплет исчезает из поля зрения, скрываясь за другим приложением (рис. 1.20).

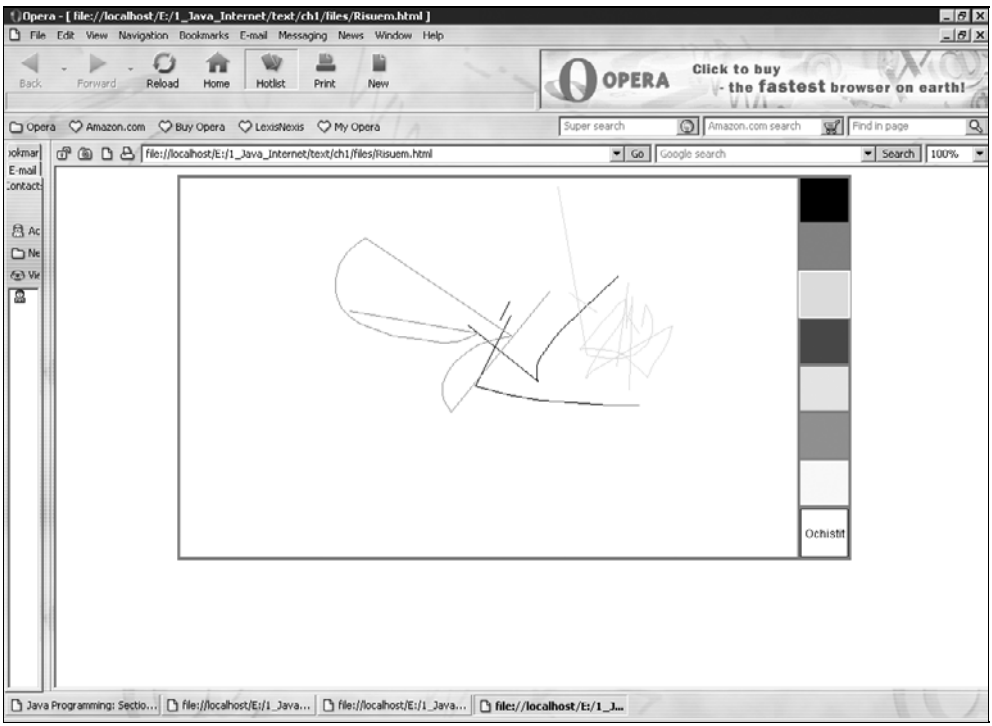


Рис. 1.20. Пример с апплетом для рисования

Код апплета приведен в файле Risuem.java. Этот апплет позволяет рисовать кривые разными цветами. Справа прилагается набор цветов. Выбор цвета осуществляется щелчком на соответствующей цветовой фигуре. Рисование осуществляется путем нажатия кнопки мыши и перемещения мыши с удерживаемой нажатой кнопкой. Рисунок исчезает при изменении размера апплета или после новой прорисовки ранее скрытого апплета.

Листинг 1.20. Файл Risuem.java

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

```
import java.applet.*;
public class Risuem extends Applet
   implements MouseListener, MouseMotionListener {
  private final static int
  BLACK = 0,
  RED = 1,  // константы соответствуют цветам
  GREEN = 2,
  BLUE = 3,
  CYAN = 4,
  MAGENTA = 5,
  YELLOW = 6;
  private int currentColor = BLACK;  // текущий цвет рисования
   /*
   переменные, определенные в этом фрагменте кода,
   используются при рисовании
   */
   private int prevX, prevY;   // предыдущее положение мыши
   private boolean dragging;   // true, если перетаскивание еще
                               // не завершилось
   private Graphics graphicsForDrawing;  // графический контекст,
                               // используемый для рисования кривых
 public void init() {
 // прослушиватель событий мыши
 addMouseListener(this);
 addMouseMotionListener(this);
   }
   public void update(Graphics g) {
   // переопределение метода update
   paint(g);
   }
   public void paint(Graphics g) {
   // рисование содержимого апплета
   int width = getSize().width;    // ширина
   int height = getSize().height;  // высота
   int colorSpacing = (height - 56) / 7 // вычисление размера квадратов
                                        // палитры
   /*
   Заполнение белым цветом области рисования.
   Оставляем по три пиксела для границ и 56 пикселов для палитры.
```

```
   */
   g.setColor(Color.white);
   g.fillRect(3, 3, width — 59, height — 6);
   // рисование границ
   g.setColor(Color.gray);
   g.drawRect(0, 0, width-1, height-1);
   g.drawRect(1, 1, width-3, height-3);
   g.drawRect(2, 2, width-5, height-5);
   // серый прямоугольник справа шириной в 56 пикселов
   g.fillRect(width — 56, 0, 56, height);
   // кнопка Ochistim
   g.setColor(Color.white);
   g.fillRect(width-53,  height-53, 50, 50);
   g.setColor(Color.black);
   g.drawRect(width-53, height-53, 49, 49);
   g.drawString("Ochistim", width-48, height-23);
   // 7 цветных прямоугольников палитры
   g.setColor(Color.black);
   g.fillRect(width-53, 3 + 0*colorSpacing, 50, colorSpacing-3);
   g.setColor(Color.red);
   g.fillRect(width-53, 3 + 1*colorSpacing, 50, colorSpacing-3);
   g.setColor(Color.green);
   g.fillRect(width-53, 3 + 2*colorSpacing, 50, colorSpacing-3);
   g.setColor(Color.blue);
   g.fillRect(width-53, 3 + 3*colorSpacing, 50, colorSpacing-3);
   g.setColor(Color.cyan);
   g.fillRect(width-53, 3 + 4*colorSpacing, 50, colorSpacing-3);
   g.setColor(Color.magenta);
   g.fillRect(width-53, 3 + 5*colorSpacing, 50, colorSpacing-3);
   g.setColor(Color.yellow);
   g.fillRect(width-53, 3 + 6*colorSpacing, 50, colorSpacing-3);
   // граница толщиной 2 пиксела вокруг цветных прямоугольников палитры
   g.setColor(Color.white);
   g.drawRect(width-55, 1 + currentColor*colorSpacing, 53, colorSpacing);
   g.drawRect(width-54, 2 + currentColor*colorSpacing, 51, colorSpacing-
2);
   }
   private void changeColor(int y) {
   // изменение цвета после щелчка на прямоугольнике палитры
```

```
int width = getSize().width;           // ширина
int height = getSize().height;         // высота
int colorSpacing = (height – 56) / 7;  // высота одного прямоугольника
                                       // палитры
int newColor = y / colorSpacing;       // номер выбранного цвета
if (newColor < 0 || newColor > 6)      // проверка номера цвета
 return;
/*
Удаление подсветки с прямоугольника старого цвета.
Изменение текущего цвета рисования.
Рисование подсветки вокруг прямоугольника нового цвета.
*/
Graphics g = getGraphics();
g.setColor(Color.gray);
g.drawRect(width-55, 1 + currentColor*colorSpacing, 53, colorSpacing);
g.drawRect(width-54, 2 + currentColor*colorSpacing, 51, colorSpacing-2);
currentColor = newColor;
g.setColor(Color.white);
g.drawRect(width-55, 1 + currentColor*colorSpacing, 53, colorSpacing);
g.drawRect(width-54, 2 + currentColor*colorSpacing, 51, colorSpacing-2);
g.dispose();
 } // end changeColor()
 private void setUpDrawingGraphics() {
// функция вызывается в mousePressed при щелчке на
// прямоугольнике цветовой палитры и
// задает графический контекст
graphicsForDrawing = getGraphics();
switch (currentColor) {
 case BLACK:
graphicsForDrawing.setColor(Color.black);
break;
 case RED:
graphicsForDrawing.setColor(Color.red);
break;
 case GREEN:
graphicsForDrawing.setColor(Color.green);
break;
```

```
 case BLUE:
graphicsForDrawing.setColor(Color.blue);
break;
 case CYAN:
graphicsForDrawing.setColor(Color.cyan);
break;
 case MAGENTA:
graphicsForDrawing.setColor(Color.magenta);
break;
 case YELLOW:
graphicsForDrawing.setColor(Color.yellow);
break;
}
 }
 public void mousePressed(MouseEvent evt) {
 // вызывается при нажатии кнопки мыши в апплете, а также
 // при рисовании, смене цвета и очистке области рисования
int x = evt.getX();              // горизонтальная координата щелчка
int y = evt.getY();              // вертикальная координата щелчка
int width = getSize().width;     // ширина апплета
int height = getSize().height;  // высота апплета
if (dragging == true) return;
if (x > width – 53) {
 // щелчок мыши правее области рисования
 if (y > height – 53)
repaint();       //  щелчок на кнопке очистки
 else
changeColor(y);  // поменять цвет рисования
}
else if (x > 3 && x < width – 56 && y > 3 && y < height – 3) {
// щелчок в области рисования
// начало рисования кривой от точки (x,y)
 prevX = x;
 prevY = y;
 dragging = true;
 setUpDrawingGraphics();
}
 }
 public void mouseReleased(MouseEvent evt) {
```

```
    if (dragging == false)
   return;
    dragging = false;
    graphicsForDrawing.dispose();
    graphicsForDrawing = null;
    }
    public void mouseDragged(MouseEvent evt) {
    // вызывается при перемещении мыши с удерживаемой нажатой кнопкой
    if (dragging == false)
   return;
    int x = evt.getX();   // горизонтальная координата мыши
    int y = evt.getY();   // вертикальная координата мыши
    if (x < 3)            // проверка координат
   x = 3;
    if (x > getSize().width - 57)
   x = getSize().width - 57;
    if (y < 3)
   y = 3;
    if (y > getSize().height - 4)
   y = getSize().height - 4;
    graphicsForDrawing.drawLine(prevX, prevY, x, y);  // прорисовка линии
    prevX = x;
    prevY = y;
    }
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseMoved(MouseEvent evt) { }
}
```

1.5.3. События клавиатуры

Все события Java связаны с компонентами графического интерфейса пользователя. При нажатии кнопки мыши, например, возникающие события связаны с тем компонентом, над которым располагается курсор мыши. А что происходит с событиями клавиатуры? С каким компонентом будет связано событие клавиатуры, например, событие нажатия клавиши?

В программировании графического интерфейса пользователя есть понятие фокус. В каждый момент времени только один элемент имеет на себе фокус. Фокус внимания может быть сконцентрирован только на одном элементе,

и если этот элемент является компонентом Java, то все события клавиатуры будут представлены в виде объектов Java с типом `KeyEvent` и переданы прослушивателю событий этого типа, связанных с тем компонентом, на котором установлен фокус. Если тот или иной компонент требует получения фокуса, то для этого используется метод `requestFocus()`, этот метод определен в классе `Component`. Вызов этого метода, однако, не дает гарантии того, что фокус будет помещен на компоненте, вызывающем фокус. В обычной ситуации фокус может быть передан компоненту путем щелчка мыши на этом компоненте или с помощью последовательного перемещения фокуса с одного элемента на другой при нажатии клавиши табуляции.

Некоторые компоненты все же не получают фокус даже в том случае, когда на них щелкают мышью. Чтобы решить эту проблему, можно вручную регистрировать события мыши. В ответ на щелчок мыши в методе `mousePressed()` следует вызвать `requestFocus()`. Такой метод работает, в частности, с компонентами, которые используются в качестве "холстов" для рисования (мы встречались с ними ранее). Например, создается "холст" — объект класса `JPanel` с именем `holst`. Чтобы прослушивать события мыши, "холст" должен зарегистрировать прослушиватель событий и в методе `mousePressed()` объекта прослушивателя вызвать `drawingSurface.requestFocus()`.

В качестве примера можно привести апплет, который прослушивает события мыши и переводит фокус на компонент, на котором щелкнула мышь. Затем компонент запрашивает фокус. Если фокус установлен на компоненте, то апплет позволяет перетаскивать цветной квадрат. Апплет реагирует на события клавиатуры, изменяя цвет этого квадрата на красный, зеленый, синий или черный в соответствии с тем, какая из четырех клавиш <R>, <G>, <B> или <K> была нажата на клавиатуре (листинг 1.21).

Листинг 1.21. Файл Focus.java

```
/*
Апплет для демонстрации событий клавиатуры и установки фокуса.
Нажимая клавиши клавиатуры со стрелками, можно перемещать квадрат
в соответствующие стороны.
Нажатие клавиш <R>, <G>, <B>, <K> приводит к изменению цвета квадрата
(красный, зеленый, синий, черный соответственно).
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Focus extends JApplet
```

```
 implements KeyListener, FocusListener, MouseListener {
  // прослушиватель  MouseListener используется только для того,
  // чтобы апплет мог по щелчку мыши установить на себе фокус
static final int SQUARE_SIZE = 40;  // сторона квадрата
Color squareColor;                  // цвет квадрата
int squareTop, squareLeft;     // координаты верхнего левого угла
                               // квадрата
 boolean focussed = false;     // true, если фокус на апплете
 DisplayPanel canvas;          // холст для рисования в апплете
 public void init()
 {
 // инициализация апплета
 squareTop = getSize().height / 2 - SQUARE_SIZE / 2;
 squareLeft = getSize().width / 2 - SQUARE_SIZE / 2;
 squareColor = Color.red;
 canvas = new DisplayPanel();        // создание холста
 setContentPane(canvas);
 canvas.setBackground(Color.white);  // цвет фона холста
 canvas.addFocusListener(this);      // регистрация прослушивателей
 canvas.addKeyListener(this);
 canvas.addMouseListener(this);
 class DisplayPanel extends JPanel {
// Описание холста
 public void paintComponent(Graphics g) {
 super.paintComponent(g);  // заполнение холста цветом фона(белый)
 /*
 Рисование границы, толщиной в три пиксела.  Цвет
 зависит от того, есть ли на апплете фокус.
 */
 if (focussed)
g.setColor(Color.cyan);
 else
g.setColor(Color.lightGray);
 int width = getSize().width;   // ширина апплета
 int height = getSize().height; //  высота апплета
 g.drawRect(0,0,width-1,height-1);
 g.drawRect(1,1,width-3,height-3);
 g.drawRect(2,2,width-5,height-5);
```

```
 // рисовать квадрат
 g.setColor(squareColor);
 g.fillRect(squareLeft, squareTop, SQUARE_SIZE, SQUARE_SIZE);
 // вывод сообщения о том, что фокуса нет
 if (!focussed) {
g.setColor(Color.magenta);
g.drawString("Navodim FOCUS",7,20);
 }
}
}
 // методы обработки событий
 public void focusGained(FocusEvent evt) {
 focussed = true;
canvas.repaint();
 }
 public void focusLost(FocusEvent evt) {
 // фокус потерян
focussed = false;
canvas.repaint();  // перерисовка границ апплета
 }
 public void keyTyped(KeyEvent evt) {
// Напечатан символ во время удержания фокуса.
// Если символ соответствует символу задания цвета,
// то апплет меняет цвет квадрата.
char ch = evt.getKeyChar();  // символ
if (ch == 'B' || ch == 'b') {
 squareColor = Color.blue;
 canvas.repaint();
}
else if (ch == 'G' || ch == 'g') {
 squareColor = Color.green;
 canvas.repaint();
}
else if (ch == 'R' || ch == 'r') {
 squareColor = Color.red;
 canvas.repaint();
}
else if (ch == 'K' || ch == 'k') {
 squareColor = Color.black;
```

```
 canvas.repaint();
}
 }
 public void keyPressed(KeyEvent evt) {
// передвижение квадрата при нажатии
// клавиш стрелок
int key = evt.getKeyCode();  // код нажатой клавиши
if (key == KeyEvent.VK_LEFT) {
 squareLeft -= 8;
 if (squareLeft < 3)
squareLeft = 3;
 canvas.repaint();
}
else if (key == KeyEvent.VK_RIGHT) {
 squareLeft += 8;
 if (squareLeft > getSize().width – 3 – SQUARE_SIZE)
squareLeft = getSize().width – 3 – SQUARE_SIZE;
 canvas.repaint();
}
else if (key == KeyEvent.VK_UP) {
 squareTop -= 8;
 if (squareTop < 3)
squareTop = 3;
 canvas.repaint();
}
else if (key == KeyEvent.VK_DOWN) {
 squareTop += 8;
 if (squareTop > getSize().height – 3 – SQUARE_SIZE)
squareTop = getSize().height – 3 – SQUARE_SIZE;
 canvas.repaint();
}
 }
 public void keyReleased(KeyEvent evt) {
 }
 public void mousePressed(MouseEvent evt) {
// при щелчке на апплете требуется установить
// фокус на компоненте холста
canvas.requestFocus();
 }
```

```
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
    public void mouseReleased(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
}
```

Если апплет не активирован, он обведен серой рамкой и на нем написана фраза "Navodim FOCUS". Далее мы разберем некоторые детали, с которыми мы имеем дело в этом апплете.

В языке Java события клавиатуры принадлежат классу `KeyEvent`. Чтобы объект был прослушивателем событий клавиатуры, он должен имплементировать интерфейс `KeyListener`. Затем объект прослушивателя должен быть вставлен в компонент, с которым связаны события клавиатуры, с помощью метода `addKeyListener()`:

```
component.addKeyListener(listener);
```

В интерфейсе `KeyListener` определены следующие методы:

```
public void keyPressed(KeyEvent evt);
public void keyReleased(KeyEvent evt);
public void keyTyped(KeyEvent evt);
```

Следует иметь в виду, что существует разница между событием нажатия клавиши и событием печати символа. На клавиатуре компьютера размещено большое количество клавиш. Это клавиши букв, клавиши цифр, клавиши модификаторов (например, <Shift>+<Ctrl>), клавиши стрелок, функциональные клавиши и пр. Во многих случаях нажатие клавиши не связано с печатью символов. С другой стороны, печать символа иногда требует нажатия нескольких клавиш. Например, чтобы напечатать символ `R`, необходимо нажать клавишу <Shift>, и до того, как эта клавиша будет отпущена, нажать клавишу <R>. Java располагает тремя типами событий клавиатуры, которые соответствуют нажатию клавиши, отпусканию клавиши и печати символа. Нажатию клавиши соответствует метод `keyPressed`, при отпускании клавиши вызывается метод `keyReleased`, а при печати символа вызывается метод `keyTyped`. Иногда одно действие приводит к появлению нескольких событий, например, событие печати символа `R` также генерирует два события `keyPressed`, два события `keyReleased` и одно событие `keyTyped`.

Удобнее всего мысленно разделить эти события на два независимых потока: первый поток — события `keyPressed` и `keyReleased`, второй поток — события печати символов `keyTyped`. В некоторых приложениях нужно будет работать только с событиями первого потока, в других приложениях более полезной окажется работа с событиями печати символов. Понятно, что из событий первого потока можно извлечь ту информацию, которая содержится в событиях второго потока, однако это не так уж просто. Иногда полезными оказываются события `keyPressed`. Нажатая клавиша может удержи-

ваться долгое время. При этом, однако, появляется небольшая проблема, состоящая в том, что при удержании клавиши в нажатом состоянии генерируется серия событий `keyPressed`, кроме того, может быть сгенерирована и серия событий `keyTyped`. Часто это оказывается не слишком важным. Следует, однако, помнить, что не каждое событие `keyPressed` приводит к появлению события `keyReleased` (клавиша отпущена).

Каждая клавиша на клавиатуре имеет связанное с ней числовое значение, соответсвующее коду символа. При вызове методов `keyPressed` или `keyReleased` передаваемый им аргумент типа события содержит соответствующее нажатой или отпущенной клавише числовое значение. Этот код может быть получен при помощи вызова функции `evt.getKeyCode()`. В Java вместо таблицы с ASCII-кодами используются именные константы, которые ставятся в соответствие каждой клавише на клавиатуре. Константы эти определены в классе `KeyEvent`. Так, например, для клавиши <Shift> эта константа будет `KeyEvent.VK_SHIFT`. Если необходимо произвести проверку того, была ли нажата клавиша <Shift>, то следует задать условие:

```
if (evt.getKeyCode() == KeyEvent.VK_SHIFT)
```

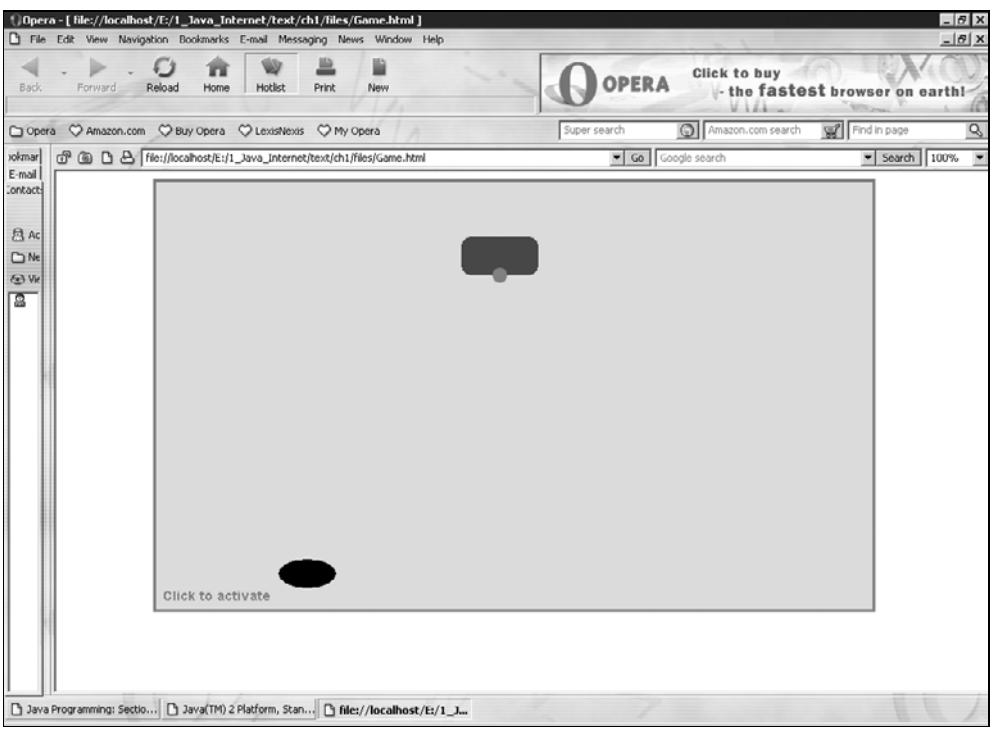


Рис. 1.21. Простая игра в виде апплета

Часто при работе с событиями `KeyTyped` необходимо узнать, какая именно клавиша была нажата. Это делается при помощи функции `evt.getKeyChar()`. В приведенном ранее примере обработка событий клавиатуры производится с использованием метода `keyPressed`, а изменение цветов квадрата посредством `keyTyped`.

Важным при работе с графическим интерфейсом пользователя является понятие состояния. Существует даже понятие "машина состояний". В нашем примере также использовались состояния. Например, записывалось состояние апплета, сохраняя при этом информацию о том, имеет ли апплет установленный на нем фокус, а также другие параметры, а именно переменные `focussed`, `squareLeft`, `squareTop`. Эти переменные состояния были использованы в методе `paintComponent()` при определении того, как будет выглядеть апплет.

Приведем еще один пример апплета (листинги 1.22 и 1.23), где состояние играет еще большую роль. Этот апплет предлагает аркадную игру (рис. 1.21).

Листинг 1.22. Файл Framework.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Framework extends JApplet
   implements ActionListener, KeyListener, FocusListener, MouseListener {
   protected void doInitialization(int width, int height) {
   }
   protected void drawFrame(Graphics g, int width, int height) {
  g.setColor(Color.lightGray);
  g.fillRect(0,0,width,height);
  g.setColor(Color.black);
  g.drawString("Elapsed Time:  " + (getElapsedTime()/1000),10,20);
  g.drawString("Frame Number:  " + (getFrameNumber()),10,35);
   }
   public void keyTyped(KeyEvent evt) {
   }
   public void keyPressed(KeyEvent evt) {
   }
   public void keyReleased(KeyEvent evt) {
   }
   public int getFrameNumber() {
  return frameNumber;
```

```
 }
 public void setFrameNumber(int frameNumber) {
if (frameNumber < 0)
 this.frameNumber = 0;
else
 this.frameNumber = frameNumber;
 }
 public long getElapsedTime() {
return elapsedTime;
 }
 public void setFrameCount(int max) {
if (max <= 0)
 this.frameCount = -1;
else
 frameCount = max;
 }
 public void setMillisecondsPerFrame(int time) {
millisecondsPerFrame = time;
if (timer != null)
 timer.setDelay(millisecondsPerFrame);
 }
 public void setFocusBorderColor(Color c) {
focusBorderColor = c;
 }
 private int frameNumber = 0;
 private int frameCount = -1;
 private int millisecondsPerFrame = 40;
 private long startTime;
 private long oldElapsedTime;
 private long elapsedTime;
 private Timer timer;
 private JPanel frame;
 private boolean focussed = false;
 Color focusBorderColor = Color.cyan;
 public Framework() {
frame = new JPanel() {
public void paintComponent(Graphics g) {
 int width = getSize().width;
 int height = getSize().height;
```

```
 drawFrame(g,width,height);
 if (focussed)
  g.setColor(focusBorderColor);
 else
  g.setColor(Framework.this.getBackground());
 g.drawRect(0,0,width-1,height-1);
 g.drawRect(1,1,width-3,height-3);
 g.drawRect(2,2,width-5,height-5);
 if (!focussed) {
  g.setColor(Framework.this.getForeground());
  g.drawString("Click to activate",10,height-12);
 }
}
 };
setContentPane(frame);
setBackground(Color.gray);
setForeground(Color.red);
frame.setFont(new Font("SanSerif",Font.BOLD,14));
frame.addFocusListener(this);
frame.addKeyListener(this);
addMouseListener(this);
 }
 public void init() {
doInitialization(getSize().width, getSize().height);
 }
 public void actionPerformed(ActionEvent evt) {
frameNumber++;
if (frameCount > 0 && frameNumber >= frameCount)
 frameNumber = 0;
elapsedTime = oldElapsedTime + (System.currentTimeMillis() - startTime);
frame.repaint();
 }
 private void startAnimation() {
if (focussed) {
if (timer == null) {
 timer = new Timer(millisecondsPerFrame, this);
 timer.start();
}
else
```

```
    timer.restart();
    startTime = System.currentTimeMillis();
   }
    }
    private void stopAnimation() {
   if (focussed) {
    oldElapsedTime += (System.currentTimeMillis() – startTime);
    elapsedTime = oldElapsedTime;
    frame.repaint();
    timer.stop();
   }
    }
    public void start() {
   startAnimation();
    }
    public void stop() {
   stopAnimation();
    }
    public void focusGained(FocusEvent evt) {
   focussed = true;
   startAnimation();
    }
   public void focusLost(FocusEvent evt) {
   stopAnimation();
   focussed = false;
    }
    public void mousePressed(MouseEvent evt) {
   frame.requestFocus();
    }
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
    public void mouseReleased(MouseEvent evt) { }
    public void mouseClicked(MouseEvent evt) { }
}
```

Листинг 1.23. Файл Game.java

```
import java.awt.*;
import java.awt.event.*;
public class Game extends Framework {
```

```
  protected void doInitialization(int width, int height) {
  initSubmarine(width, height);
  initBoat(width, height);
  initBomb(width,height);
   }
   synchronized public void drawFrame(Graphics g, int width, int height)
{
   g.setColor(Color.green);
   g.fillRect(0,0,width,height);
   doBoatFrame(g, width, height);
   doSubmarineFrame(g, width, height);
   doBombFrame(g, width, height);
   }
   synchronized public void keyPressed(KeyEvent evt) {
   int code = evt.getKeyCode();
   if (code == KeyEvent.VK_LEFT) {
  boatCenterX -= 15;
   }
   else if (code == KeyEvent.VK_RIGHT) {
  boatCenterX += 15;
   }
   else if (code == KeyEvent.VK_DOWN) {
  if (bombIsFalling == false)
   bombIsFalling = true;
   }
   }
   int bombCenterX;
   int bombCenterY;
   boolean bombIsFalling;
   void initBomb(int width, int height) {
   bombIsFalling = false;
   bombCenterY = 100;
   }
   void doBombFrame(Graphics g, int width, int height) {
   if (bombIsFalling) {
  if (bombCenterY > height) {
   initBomb(width, height);
   bombCenterX = boatCenterX;
  }
```

```
else if (Math.abs(bombCenterX – subCenterX) <= 36 &&
  Math.abs(bombCenterY – subCenterY) <= 21) {
 explosionFrameNumber = 1;
 initBomb(width, height);
 bombCenterX = boatCenterX;
}
else {
 bombCenterY += 10;
 g.setColor(Color.red);
 g.fillOval(bombCenterX – 8, bombCenterY – 8, 16, 16);
}
 }
 else {
bombCenterX = boatCenterX;
g.setColor(Color.red);
g.fillOval(bombCenterX – 8, bombCenterY – 8, 16, 16);
 }
 }
 int boatCenterX;
 int boatCenterY;
 void initBoat(int width, int height) {
boatCenterX = width/2;
boatCenterY = 80;
 }
 void doBoatFrame(Graphics g, int width, int height) {
if (boatCenterX < 0)
 boatCenterX = 0;
else if (boatCenterX > width)
 boatCenterX = width;
g.setColor(Color.blue);
g.fillRoundRect(boatCenterX – 40, boatCenterY – 20, 80, 40, 20, 20);
 }
 int subCenterX;
 int subCenterY;
 boolean subIsMovingLeft;
 int explosionFrameNumber;
 void initSubmarine(int width, int height) {
subCenterX = (int)(width * Math.random());
subCenterY = height – 40;
```

```
explosionFrameNumber = 0;
if (Math.random() < 0.5)
 subIsMovingLeft = true;
else
 subIsMovingLeft = false;
 }
 void doSubmarineFrame(Graphics g, int width, int height) {
if (explosionFrameNumber > 0) {
 if (explosionFrameNumber == 14) {
initSubmarine(width,height);
 }
 else if (explosionFrameNumber > 10) {
 explosionFrameNumber++;
 }
 else {
g.setColor(Color.black);
g.fillOval(subCenterX — 30, subCenterY — 15, 60, 30);
g.setColor(Color.yellow);
g.fillOval(subCenterX — 4*explosionFrameNumber,
   subCenterY — 2*explosionFrameNumber,
   8*explosionFrameNumber,
   4*explosionFrameNumber);
g.setColor(Color.red);
g.fillOval(subCenterX — 2*explosionFrameNumber,
   subCenterY — explosionFrameNumber/2,
   4*explosionFrameNumber,
   explosionFrameNumber);
explosionFrameNumber++;
 }
}
else {

 if (Math.random() < 0.04)
subIsMovingLeft = !subIsMovingLeft;
 if (subIsMovingLeft) {
subCenterX -= 5;
if (subCenterX <= 0) {
 subCenterX = 0;
 subIsMovingLeft = false;
```

```
   }
    }
    else {
   subCenterX += 5;
   if (subCenterX > width) {
    subCenterX = width;
    subIsMovingLeft = true;
   }
    }
    g.setColor(Color.black);
    g.fillOval(subCenterX – 30, subCenterY – 15, 60, 30);
   }
    }
}
```

Игра активизируется после щелчка мыши по апплету. Панель с шариком управляется при помощи клавиш стрелок. Шар-снаряд запускается клавишей "стрелка вниз". При попадании она взрывается.

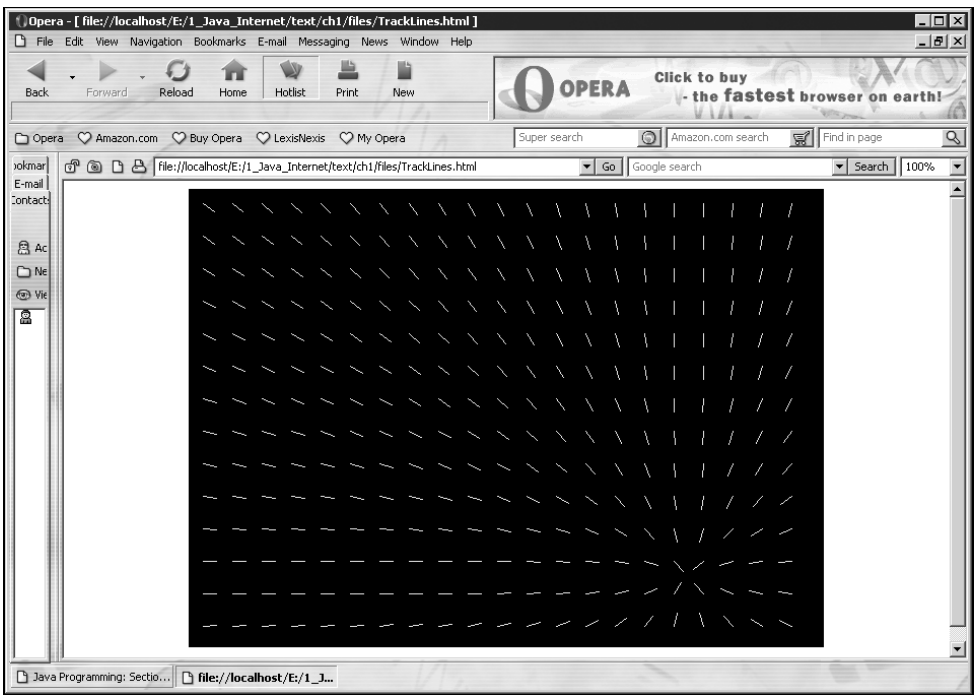


Рис. 1.22. Вращающиеся отрезки

В заключение этого раздела приведем еще один пример (рис. 1.22). В этом апплете показаны короткие отрезки, расположенные равномерно по рядам и столбцам. Один конец каждого отрезка зафиксирован, отрезок может вокруг него вращаться. Изначально все отрезки вращаются случайным образом. Если над апплетом появляется мышь, отрезки прекращают вращение,все отрезки поворачиваются в сторону курсора. Если курсор останавливается, то через некоторое время отрезки вновь начинают хаотическое вращение. Скорость вращения выбирается случайным образом. При нажатии кнопки мыши цвет отрезков меняется. Цвет зависит от положения курсора мыши.

Листинг 1.24. Файл LinesMove.java

```
import java.awt.*;
import java.applet.*;
public class LinesMove extends Applet implements Runnable {
  boolean used = false; // значение переменной used при первой ориентации
                        // на курсор мыши становится равным true
  int ROWS = 5;         // количество рядов
  int COLUMNS = 7;      // количество столбцов
  int[][] angle;        // угол
  int[][] driftSpeed;   // угловая скорость
  long lastTime = 0;    // время последней инициализации порядка
  int delayToDrift  = 1500;  // задержка при упорядочении
  int width = -1;       // ширина апплета
  int height = -1;      // высота апплета
  int hSpace;           // горизонтальное расстояние между линиями
  int vSpace;           // вертикальное расстояние между линиями
  int space;            // длина каждой линии
  int last_x = -1;      // координаты точки, на которую линии были
  int last_y = -1;      // ориентированы в прошлый раз
  int sleepTime = 100;  // задержка между фреймами
  Thread runner = null; // поток  анимации
  Image OSC = null;     // невидимый холст для двойной буферизации
  Graphics OSCGraphics;
  public void init() {
  setBackground(Color.black);
  setForeground(Color.white);
  String param;
  param = getParameter("rows");
```

```
if (param != null) {
 try {
ROWS = Integer.parseInt(param);
}
catch (NumberFormatException e) { }
}
param = getParameter("columns");
if (param != null) {
 try {
COLUMNS = Integer.parseInt(param);
}
catch (NumberFormatException e) { }
}
angle = new int[ROWS][COLUMNS];  // случайные скорости и углы
driftSpeed = new int[ROWS][COLUMNS];
for (int i = 0; i<ROWS; i++)
for (int j = 0; j<COLUMNS; j++) {
angle[i][j] = (int)(360*Math.random());
driftSpeed[i][j] = (int)(41*Math.random()) – 20;
 }
 }
 public void start() {
 if (runner == null) {
 int h = size().height;
 int w = size().width;
 if (h != height || w != width)
doResize(w,h);
 runner = new Thread(this);
 runner.start();
}
 }
 public void stop() {
// остановка анимации
if (runner != null) {
 runner.stop();
 runner = null;
}
 }
```

1.6. Работа с графикой

В компьютерном представлении рисунок — это набор чисел. Числа задают цвет каждого пиксела на экране компьютера и хранятся в буфере экрана. Видеокарта компьютера читает содержимое этого буфера с огромной скоростью (пробегая буфер несколько десятков и сотен раз в секунду) и расцвечивает пикселы экрана в соответствии с числами, хранимыми в буфере. Всякий раз, когда возникает необходимость изменить цвет того или иного пиксела, компьютер вставляет новые числа в буфер экрана, и долю секунды спустя новая информация отображается на мониторе.

Набор чисел не обязательно должен храниться в экранном буфере, его можно хранить в произвольном месте, например, в виде файла на жестком диске. В Java существуют стандартные классы и процедуры копирования изображений из одной части памяти в другую, получения изображений из файлов, отображения изображений на экране компьютера.

Для представления изображений в языке Java используется стандартный класс `java.awt.Image`. Каждый объект типа `Image` содержит информацию о конкретном изображении. Существует два типа объектов `Image`. Один тип — это изображения, хранимые в виде файлов. Другой тип — это изображения, хранимые в памяти компьютера.

Каждое изображение представлено с помощью набора чисел, но представление это может быть произведено не одним-единственным образом. Так, для файлов изображений существует два стандартных способа кодирования изображений, используемых в языке Java. Один способ используется для создания GIF изображений, другой — для создания JPEG-изображений. Соответсвующие им файлы имет расширения gif и jpg или jpeg. Тот и другой формат являются сжатыми, в них уменьшен объем памяти, требуемый для хранения изображения.

В классе `Applet` определен метод `getImage`, этот метод используется для загрузки изображения, хранимого в виде файла GIF или JPEG, например,

```
img = getImage(getCodeBase(), "ace.gif");
```

Эта инструкция приведет к появлению объекта изображения. Второй аргумент в методе — это имя файла с изображением. Первый аргумент — это директория, в которой расположен файл с рисунком, значение `getCodeBase()` соответствует тому, что файл с рисунком находится в том же каталоге, что и содержащий его апплет. После того как получен объект рисунка, его можно отобразить в любом графическом контексте. Наиболее часто при этом используется метод `paintComponent()` в компоненте `JPanel` (или в каком-либо другом компоненте):

```
g.drawImage(img, x, y, this);
```

Эта команда используется для отображения рисунка в компоненте. Параметры `x` и `y` — это положение верхнего левого угла рисунка, а размер

прямоугольника будет таким, чтобы весь рисунок был отображен в натуральную величину. Четвертый параметр — это компонент, где рисунок будет отображен. При обращении к `getImage()` запрашиваемый файл не будет скачиваться немедленно. Файл будет запрошен только тогда, когда он должен отобразиться в первый раз. Объект `Image` просто запоминает местонахождение файла. Этот метод начинает скачивание файла, но не дожидается его завершения. Четвертый параметр — это объект, который наблюдает за изображением. После того как рисунок будет полностью получен, система сообщит наблюдателю за изображением о том, что этот рисунок теперь стал доступным. В качестве наблюдателя может использоваться любой компонент `JComponent`. Если известно, что рисунок уже загружен, то четвертый параметр можно указать в виде `null`.

Существуют и другие методы вызова `drawImage()`. Например, задается размер изображения:

```
g.drawImage(img, x, y, width, height, this);
```

Можно нарисовать лишь часть изображения:

```
g.drawImage(img, dest_x1,   dest_y1,   dest_x2,   dest_y2,
            source_x1, source_y1, source_x2, source_y2, this);
```

Здесь координаты `source_x1`, `source_y1`, `source_x2` и `source_y2` задают верхний левый и нижний правый углы той части изображения, которая будет показана. Координаты `dest_x1`, `dest_y1`, `dest_x2`, `dest_y2` — это координаты графического контекста. При необходимости будет произведено соответствующее координатам сжатие или растяжение рисунка. Далее мы рассмотрим пример с картами. Этот пример будет использовать только один рисунок для отображения всех 52 карт (рис. 1.23).

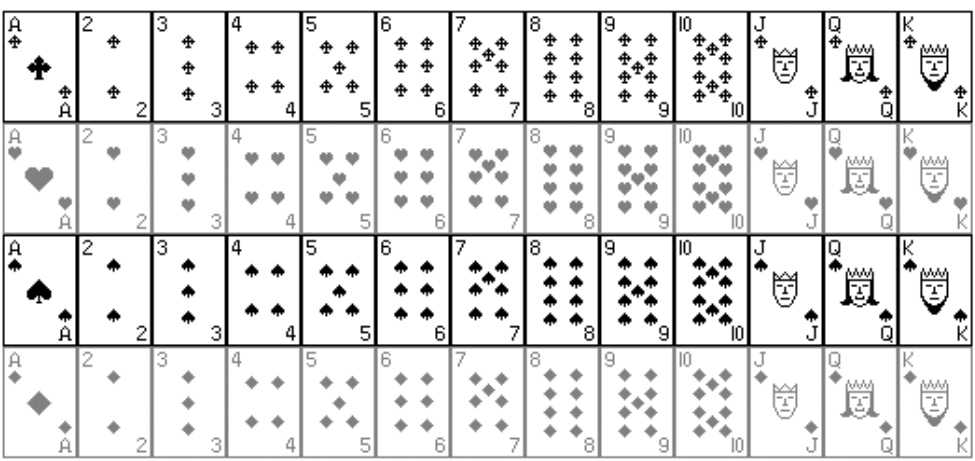


Рис. 1.23. Игральные карты на одном рисунке

Для отображения только одной карты следует указать координаты отображаемого фрагмента рисунка. Метод этот используется в классическом примере игры на угадывание (нужно угадать, какая придет следующая карта — более старшая или более младшая).

В этом апплете карты отображаются с использованием следующего метода. В переменной типа `Image` и именем `cardImages` хранится рисунок с изображениями 52 карт (рис. 1.23). Размер каждой карты составляет 40 × 60 пикселов. На основе этого создаются изображения для каждой карты в отдельности (листинг 1.25).

Листинг 1.25. Функция `drawCard()`

```
 void drawCard(Graphics g, Card card, int x, int y) {
if (card == null) {
 // описание карты рубашкой вверх
 g.setColor(Color.blue);
 g.fillRect(x,y,40,60);
 g.setColor(Color.white);
 g.drawRect(x+3,y+3,33,53);
 g.drawRect(x+4,y+4,31,51);
}
else {
 int row = 0;  // определение ряда, в котором нарисована текущая карта
 switch (card.getSuit()) {
case Card.CLUBS:  row = 0;  break;
case Card.HEARTS:   row = 1;  break;
case Card.SPADES:   row = 2;  break;
case Card.DIAMONDS: row = 3;  break;
 }
 int sx, sy;  // координаты верхнего левого угла карты
              // в заданном рисунке
 sx = 40*(card.getValue() - 1);
 sy = 60*row;
 g.drawImage(cardImages, x,  y,  x+40,  y+60,
   sx, sy, sx+40, sy+60, this);
}
 }
```

Далее приводится программный код, состоящий из четырех файлов. Основная программа — Cards.java. Кроме нее потребуется наличие файлов Card.java, Hand.java, Deck.java (листинги 1.26, 1.27 и 1.28).

Листинг 1.26. Файл Cards.java

```
/*
Простая карточная игра
«Угадай, что дальше»
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Cards extends JApplet {
   java.awt.Image cardImages;
   public void init() {
  cardImages = getImage(getCodeBase(), "smallcards.gif");
  setBackground(new Color(130,50,40));
  HighLowCanvas board = new HighLowCanvas();
  getContentPane().add(board, BorderLayout.CENTER);
  JPanel buttonPanel = new JPanel();
  buttonPanel.setBackground(new Color(220,200,180));
  getContentPane().add(buttonPanel, BorderLayout.SOUTH);
  JButton higher = new JButton("Starshe");
  higher.addActionListener(board);
  buttonPanel.add(higher);
  JButton lower = new JButton("Mladshe");
  lower.addActionListener(board);
  buttonPanel.add(lower);
  JButton newGame = new JButton("New Game");
  newGame.addActionListener(board);
  buttonPanel.add(newGame);
   }
   public Insets getInsets() {
  return new Insets(3,3,3,3);
   }
   class HighLowCanvas extends JPanel implements ActionListener {
  Deck deck;               // колода карт
  Hand hand;               // сбрасываемая карта
  String message;          // сообщение
  boolean gameInProgress;  // состояние игры
  Font bigFont;            // шрифт отображения сообщения
  Font smallFont;          // шрифт для отображения карт
```

```
  HighLowCanvas() {
   setBackground(new Color(0,120,0));
   setForeground(Color.green);
   smallFont = new Font("SansSerif", Font.PLAIN, 12);
   bigFont = new Font("Serif", Font.BOLD, 14);
   doNewGame();
  }
  public void actionPerformed(ActionEvent evt) {
   String command = evt.getActionCommand();
   if (command.equals("Starshe"))
  doHigher();
   else if (command.equals("Mladshe"))
  doLower();
   else if (command.equals("New Game"))
  doNewGame();
  }
  void doHigher() {
   if (gameInProgress == false) {
  message = "Click \"New Game\" first!";
  repaint();
  return;
   }
   hand.addCard(deck.dealCard());
   int cardCt = hand.getCardCount();
   Card thisCard = hand.getCard(cardCt - 1);
   // описание предыдущей карты
   Card prevCard = hand.getCard(cardCt - 2);
if (thisCard.getValue() < prevCard.getValue()) {
  gameInProgress = false;
  message = "Vy proigrali!";
   }
   else if (thisCard.getValue() == prevCard.getValue()) {
  gameInProgress = false;
  message = "Ploho poluchilos!";
   }
   else if (cardCt == 4) {
  gameInProgress = false;
  message = "OK!  Vy Vyuigrali!";
   }
```

```
 else {
message = "Pravilno!  Popytajtes " + cardCt + ".";
 }
 repaint();
}
void doLower() {
 if (gameInProgress == false) {
message = "Snachala nazhmite "New Game" !";
repaint();
return;
 }
 hand.addCard(deck.dealCard());
 int cardCt = hand.getCardCount();
 Card thisCard = hand.getCard(cardCt - 1);
 Card prevCard = hand.getCard(cardCt - 2);
 if (thisCard.getValue() > prevCard.getValue()) {
gameInProgress = false;
message = "Vy proigrali!";
 }
 else if (thisCard.getValue() == prevCard.getValue()) {
gameInProgress = false;
message = "Ne verno.";
 }
 else if (cardCt == 4) {
gameInProgress = false;
message = "Zamechatelno! Vy vyigrali!";
 }
 else {
message = "Pravilno!  Popytites " + cardCt + ".";
 }
 repaint();
}
void doNewGame() {
 if (gameInProgress) {
message = "Zavershite igru!";
repaint();
return;
 }
 deck = new Deck();
```

```
 hand = new Hand();
 deck.shuffle();
 hand.addCard(deck.dealCard());
 message = "Ugadaite, starshe ili mladshe sleduyuschaya karta?";
 gameInProgress = true;
 repaint();
}
public void paintComponent(Graphics g) {
 super.paintComponent(g);
 g.setFont(bigFont);
 g.drawString(message,10,getSize().height-10);
 g.setFont(smallFont);
 int cardCt = hand.getCardCount();
 for (int i = 0; i < cardCt; i++)
drawCard(g, hand.getCard(i), 30 + i * 70, 10);
 if (gameInProgress)
drawCard(g, null, 30 + cardCt * 70, 10);
}
 void drawCard(Graphics g, Card card, int x, int y) {
if (card == null) {
 g.setColor(Color.blue);
 g.fillRect(x,y,40,60);
 g.setColor(Color.white);
 g.drawRect(x+3,y+3,33,53);
 g.drawRect(x+4,y+4,31,51);
}
else {
 int row = 0;
 switch (card.getSuit()) {
case Card.CLUBS:  row = 0;  break;
case Card.HEARTS:   row = 1;  break;
case Card.SPADES:   row = 2;  break;
case Card.DIAMONDS: row = 3;  break;
 }
 int sx, sy;
 sx = 40*(card.getValue() - 1);
 sy = 60*row;
 g.drawImage(cardImages, x, y, x+40, y+60,
   sx, sy, sx+40, sy+60, this);
```

```
    }
     }
     }
}
```

Внешний вид апплета показан на рис. 1.24.

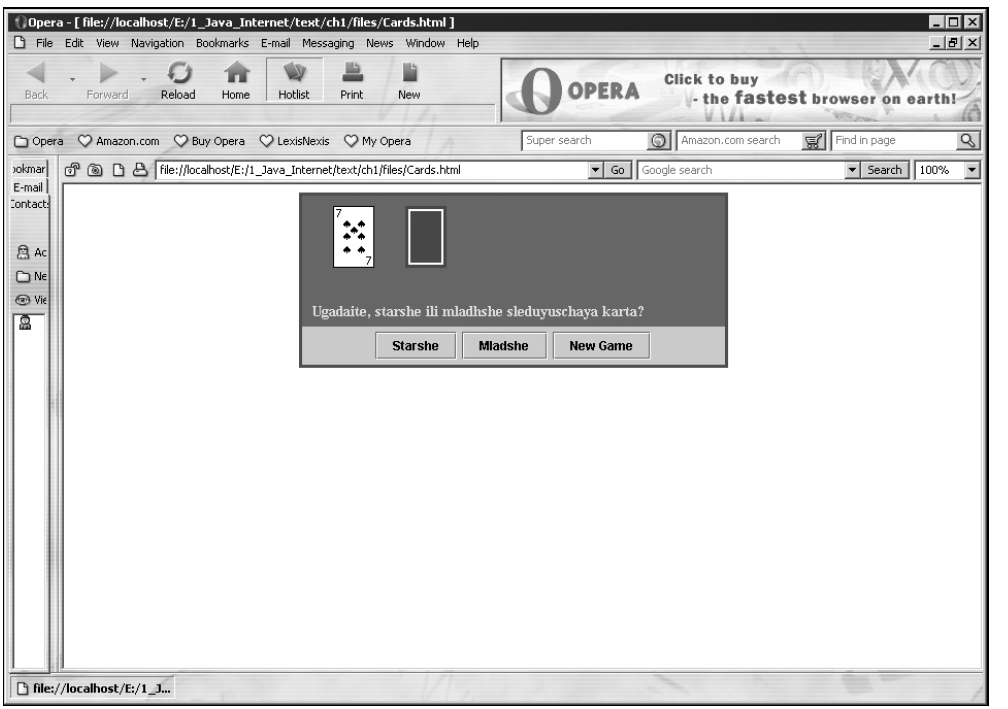


Рис. 1.24. Игра "Угадай, что дальше?"

Класс `Card` описывает отдельную карту.

Листинг 1.27. Код класса Card.java

```
/*
Отдельная карта
*/
public class Card {
  public final static int SPADES = 0,   // значения мастей
    HEARTS = 1,
    DIAMONDS = 2,
```

```
  CLUBS = 3;

public final static int ACE = 1,     // старшинство карт
  JACK = 11,
  QUEEN = 12,
  KING = 13;

private final int suit;   // один из вариантов:
                          //  SPADES, HEARTS, DIAMONDS, CLUBS
private final int value;

public Card(int theValue, int theSuit) {
value = theValue;
suit = theSuit;
}
public int getSuit() {
return suit;
}
public int getValue() {
return value;
}
public String getSuitAsString() {
  switch (suit) {
    case SPADES:   return "Spades";
    case HEARTS:   return "Hearts";
    case DIAMONDS: return "Diamonds";
    case CLUBS:    return "Clubs";
    default:       return "??";
  }
}
public String getValueAsString() {
  switch (value) {
   case 1:   return "Ace";
   case 2:   return "2";
   case 3:   return "3";
   case 4:   return "4";
   case 5:   return "5";
   case 6:   return "6";
   case 7:   return "7";
```

```
      case 8:   return "8";
      case 9:   return "9";
      case 10:  return "10";
      case 11:  return "Jack";
      case 12:  return "Queen";
      case 13:  return "King";
      default:  return "??";
     }
  }
  public String toString() {
    return getValueAsString() + " of " + getSuitAsString();
  }
}
```

Код класса `Hand` приводится далее (листинг 1.28). Он описывает сдаваемую карту или набор карт. Одновременно может быть сдано несколько карт, по умолчанию используется значение 5.

Листинг 1.28. Файл Hand.java

```
import java.util.Vector;
public class Hand {
   private Vector hand;   // сдаваемые карты
   public Hand() {
  hand = new Vector();
   }
   public void clear() {
  hand.removeAllElements();
   }
   public void addCard(Card c) {
  if (c != null)
   hand.addElement(c);
   }
   public void removeCard(Card c) {
  hand.removeElement(c);
   }
   public void removeCard(int position) {
  if (position >= 0 && position < hand.size())
   hand.removeElementAt(position);
   }
```

```
 public int getCardCount() {
return hand.size();
 }
 public Card getCard(int position) {
if (position >= 0 && position < hand.size())
 return (Card)hand.elementAt(position);
else
 return null;
 }
 public void sortBySuit() {
Vector newHand = new Vector();
while (hand.size() > 0) {
 int pos = 0;
 Card c = (Card)hand.elementAt(0);
 for (int i = 1; i < hand.size(); i++) {
Card c1 = (Card)hand.elementAt(i);
if (c1.getSuit() < c.getSuit() ||
 (c1.getSuit() == c.getSuit() && c1.getValue() < c.getValue())) {
pos = i;
c = c1;
}
 }
 hand.removeElementAt(pos);
 newHand.addElement(c);
}
hand = newHand;
 }

 public void sortByValue() {
Vector newHand = new Vector();
while (hand.size() > 0) {
 int pos = 0;
 Card c = (Card)hand.elementAt(0);
 for (int i = 1; i < hand.size(); i++) {
Card c1 = (Card)hand.elementAt(i);
if (c1.getValue() < c.getValue() ||
  (c1.getValue() == c.getValue() && c1.getSuit() < c.getSuit())) {
pos = i;
c = c1;
```

```
  }
   }
   hand.removeElementAt(pos);
   newHand.addElement(c);
  }
  hand = newHand;
   }
}
```

Кроме этого необходим класс, описывающий все 52 карты (листинг 1.29).

Листинг 1.29. Файл Deck.java

```
public class Deck {

  private Card[] deck;
  private int cardsUsed;
  public Deck() {
   deck = new Card[52];
   int cardCt = 0;
   for (int suit = 0; suit <= 3; suit++) {
  for (int value = 1; value <= 13; value++) {
   deck[cardCt] = new Card(value,suit);
   cardCt++;
  }
   }
   cardsUsed = 0;
  }
  public void shuffle() {
  for (int i = 51; i > 0; i--) {
  int rand = (int)(Math.random()*(i+1));
  Card temp = deck[i];
  deck[i] = deck[rand];
  deck[rand] = temp;
  }
  cardsUsed = 0;
  }
  public int cardsLeft() {
  return 52 – cardsUsed;
  }
```

```
  public Card dealCard() {
  if (cardsUsed == 52)
   shuffle();
  cardsUsed++;
  return deck[cardsUsed – 1];
  }
}
```

1.6.1. Двойная буферизация

Объекты `Image` могут храниться в памяти компьютера. Эти изображения, если они не записаны в буфер экрана, не видимы пользователю. Изображения, хранимые в памяти компьютера, могут быть очень быстро скопированы в буфер экрана и стать видимыми. Именно такая техника используется при рисовании компонентов `Swing`, которые предоставляют дополнительные возможности при работе с графикой. Картинка создается шаг за шагом, образуя изображение в памяти компьютера. Этот процесс может занять некоторое время. Если бы процесс рисования происходил в буфере экрана, пользователь смог бы заметить мерцание картинки. Удобнее заменить существующую картинку уже созданной новой картинкой и не рисовать ее в процессе отображения на экран. Замена одного изображения другим готовым изображением происходит мгновенно. Пользователь не заметит мерцания, вызванного тем, что картинка создается в процессе ее отображения. При этом изображение, особенно при использовани анимации, становится гладким.

Техника предварительного создания изображения в памяти компьютера и дальнейшее ее отображение на экран после того, как картинка уже создана, называется *двойной буферизацией*. Также используется термин кадровая буферизация. Видеокарта, переходя от буфера одного кадра к буферу другого кадра, отображает любой кадр практически мгновенно. Каждый кадровый буфер используется для отображения нового изображения на экране компьютера. Копирование памяти не требуется. Однако в Java при технике двойной буферизации копирование памяти необходимо. Конечно, можно отказаться от использования двойной буферизации, но при этом на экране будет заметно мерцание.

В качестве иллюстрации рассмотрим два примера. Файл DB.java (листинг 1.31) описывает перетаскиваемый красный квадрат с использованием техники двойной буферизации. Двойная буферизация не используется в файле NonDB.java (листинг 1.30). Второй апплет при перетаскивании квадрата будет мерцать. Чтобы сделать эффект мерцания более заметным, фон содержит множество серых горизонтальных линий, это увеличивает время прорисовывания, делает мерцание сильно заметным.

Листинг 1.30. Файл NonDB.java

```
/*
двойная буферизация отсутствует
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class NonDB extends JApplet {
   public void init() {
  setContentPane(new Display());
  getRootPane().setDoubleBuffered(false);
   }
   class Display extends JPanel
  implements MouseListener, MouseMotionListener {
  int x1, y1;
  boolean dragging;
  int offsetX, offsetY;
  Display() {
   x1 = 10;
   y1 = 10;
   setBackground(Color.white);
   addMouseListener(this);
   addMouseMotionListener(this);
   setDoubleBuffered(false);
  }

  public void paintComponent(Graphics g) {
   int width = getSize().width;   // ширина компонента
   int height = getSize().height; // высота компонента
   g.setColor(Color.white);
   g.fillRect(0,0,width,height);
   g.setColor(Color.gray);
   for (int i = 3; i < width; i += 2)   // вертикальные линии
  g.drawLine(i,0,i,height);
   for (int j = 3; j < height; j += 2)  // горизонтальные линии
  g.drawLine(0,j,width,j);
   g.setColor(Color.red);
   g.fillRect(x1, y1, 50, 50);  // красный квадрат
```

```
   g.setColor(Color.black);
   g.drawRect(0, 0, width – 1, height – 1);
   g.drawRect(1, 1, width – 3, height – 3);
  }
  public void mousePressed(MouseEvent evt) {
   if (dragging)
  return;
   int x = evt.getX();
   int y = evt.getY();
   if (x >= x1 && x < x1+50 && y >= y1 && y < y1+50) {
  dragging = true;
  offsetX = x – x1;
  offsetY = y – y1;
   }
  }
  public void mouseReleased(MouseEvent evt) {
  dragging = false;
  if (x1 + 50 < 3 || x1 > getSize().width – 3
    || y1 + 50 < 3 || y1 > getSize().height – 3) {
  x1 = 10;
  y1 = 10;
  repaint();
  }
  }
  public void mouseDragged(MouseEvent evt) {
  if (dragging == false)
  return;
  int x = evt.getX();   // положение мыши
  int y = evt.getY();
  x1 = x – offsetX;   // перемещение квадрата
  y1 = y – offsetY;
  repaint();
  }
  public void mouseMoved(MouseEvent evt) { }
  public void mouseClicked(MouseEvent evt) { }
  public void mouseEntered(MouseEvent evt) { }
  public void mouseExited(MouseEvent evt) { }
   }
}
```

Листинг 1.31. Файл DB.java

```
/*
используется двойная буферизация
*/
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DB extends JApplet {
   public void init() {
  setContentPane(new Display());
   }
   class Display extends JPanel
  implements MouseListener, MouseMotionListener {
  int x1, y1;
  boolean dragging;
  int offsetX, offsetY;
  Display() {
   x1 = 10;
   y1 = 10;
   setBackground(Color.white);
   addMouseListener(this);
   addMouseMotionListener(this);
  }
  public void paintComponent(Graphics g) {
   int width = getSize().width;
   int height = getSize().height;
   super.paintComponent(g);
   g.setColor(Color.gray);
   for (int i = 3; i < width; i += 2)
  g.drawLine(i,0,i,height);
   for (int j = 3; j < height; j += 2)
  g.drawLine(0,j,width,j);
   g.setColor(Color.red);
   g.fillRect(x1, y1, 50, 50);
   g.setColor(Color.black);
   g.drawRect(0, 0, width — 1, height — 1);
   g.drawRect(1, 1, width — 3, height — 3);
  }
```

```
  public void mousePressed(MouseEvent evt) {
   if (dragging)
  return;
   int x = evt.getX();   int y = evt.getY();
   if (x >= x1 && x < x1+50 && y >= y1 && y < y1+50) {
  dragging = true;
  offsetX = x - x1;
  offsetY = y - y1;
   }
  }
  public void mouseReleased(MouseEvent evt) {
  dragging = false;
  if (x1 + 50 < 3 || x1 > getSize().width - 3
     || y1 + 50 < 3 || y1 > getSize().height - 3) {
  x1 = 10;
  y1 = 10;
  repaint();
  }
  }
  public void mouseDragged(MouseEvent evt) {
  if (dragging == false)
  return;
  int x = evt.getX();
  int y = evt.getY();
  x1 = x - offsetX;
  y1 = y - offsetY;
  repaint();
  }
  public void mouseMoved(MouseEvent evt) { }
  public void mouseClicked(MouseEvent evt) { }
  public void mouseEntered(MouseEvent evt) { }
  public void mouseExited(MouseEvent evt) { }
   }
}
```

Бывает полезно создание собственного невидимого изображения. Это можно сделать при помощи метода `createImage()`. Этот метод определен в классе `Component`. Поэтому его можно использовать практически в любом месте апплета:

```
Image offScreenImage = createImage(width, height);
```

Класс `Image` содержит метод `getGraphics()`, этот метод возвращает объект `Graphics`:

```
Graphics offscreenGraphics = offScreenImage.getGraphics();
```

Теперь можно использовать рисование. Все рисование будет осуществлено на скрытом рисунке:

```
offscreenGraphics.drawRect(10,10,50,100);
```

После того как картинка нарисована, ее можно скопировать в другой графический контекст при помощи метода `drawImage()`:

```
g.drawImage(offScreenImage,0,0,null)
```

Скрытые изображения можно использовать для решения задачи восстановления изображений после того, как область апплета была невидимой.

1.6.2. Потоки. Анимация. Таймер

Java позволяет работать с потоками, при этом несколько независимых потоков могут работать одновременно друг с другом. Поток — базовая единица при работе программы. Поток выполняет последовательность инструкций, причем делает это последовательно, выбирая одну инструкцию за другой. Поток прекращается, если прекращается выполнение программы в целом. В каждый конкретный момент только один поток будет выполняться, поскольку центральный процессор может выполнять только одно задание в отдельный момент. Исключениями являются многопроцессорные системы, состоящие из нескольких процессоров. Компьютер использует разделение времени, что дает иллюзию одновременности выполнения нескольких потоков в одно и то же время. Процессор какое-то время выполняет задачу одного потока, затем переходит к другому и так далее и, наконец, возвращается снова к первому потоку, и цикл повторяется на новом "витке". Такой цикл может быть очень кратковременным, например, в одной секунде может уложиться около сотни таких циклов. Для разработчика и для пользователя создается полная иллюзия того, что потоки выполняются параллельно и одновременно. Многопоточность языка Java означает, что программа, созданная на этом языке, может создавать один или несколько потоков, которые затем выполняются параллельно вместе с основной программой. Многопоточность является фундаментальным понятием в Java. Использование потоков не всегда является простой задачей, поэтому потоки следует использовать только тогда, когда в них действительно есть необходимость.

Анимация и Swing

Потоки используются при создании анимации. Анимация создается в виде набора статических рисунков, которые появляются на экране последовательно один за другим. Если рисунки появляются друг за другом достаточно бытро, а изменения на рисунках невелики, то наблюдатель видит иллюзию

непрерывного движения. Чтобы создать анимацию, необходимо создать программу, отображающую последовательность изображений.

В GUI-программе существует по крайней мере один поток, задача которого — отслеживание и обработка событий, которые производит пользователь. Анимация никак не связана с этими событиями, она представляет собой программу, которая выполняется естественным путем без вмешательства извне. Наиболее естественным для этого будет создать независимый поток. До разработки и создания пакета `Swing` такой поток необходимо было создавать в явном виде. Как следствие этого работа с анимацией представляла гораздо более сложную задачу, чем следовало бы. С использованием `Swing` работа с простой анимацией значительно упростилась. Сейчас нет необходимости работать с потоками непосредственно, все это уже сделано в пакете `Swing`. Идея состояла в том, чтобы объединить в пакете `Swing` обработку событий так, чтобы работа с анимацией выглядела так же, как и задача по созданию других приложений на основе графического интерфейса пользователя.

Анимация может быть создана с использованем объектов, принадлежащих классу `javax.swing.Timer`. Объект `Timer` может генерировать последовательность событий без вмешательства пользователя. При работе с анимацией необходимо создать объект `Timer` и обрабатывать каждое сообщение таймера, отображая один кадр анимации за другим. При этом в фоне таймер работает в виде самостоятельного потока, но нам нет необходимости работать с этим поток в явном виде.

События, которые генерируются таймером, — это события типа `ActionEvent`. Конструктор таймера в качестве параметров получает два значения — промежуток времени между двумя событиями и прослушиватель событий `ActionListener`, который будет уведомляться о наступлении события:

```
Timer(int delayTime, ActionListener listener);
```

Прослушиватель событий должен отвечать на возникающие события, для этого используется метод `actionPerformed()`. Интервал между событиями указывается в миллисекундах (в одной секунде 1000 мс). Фактически этот интервал может оказаться несколько большим, что зависит от времени, которое может потребоваться для обработки события, и степени загруженности компьютера задачами. В целях воспроизведения анимации интервал устанавливается продолжительностью от 10 до 30 фреймов в секунду, то есть от 100 до 33 мс.

Таймер не начинает свою работу автоматически сразу после того, как он будет создан. Чтобы запустить таймер, необходимо воспользоваться методом `start()`. Для остановки таймера используется метод `stop()`. Этот метод вызывается для того, чтобы остановить процесс генерации событий таймера. Если таймер был остановлен, то для возобновления работы таймера используется метод `restart()`. Метод `start()` должен быть вызван только один раз. Все перечисленные методы не имеют параметров.

В следующем апплете (листинг 1.32) мы запускаем анимацию, нажимая кнопку **Start**. После нажатия кнопки надпись на ней изменяется, мы увидим новую надпись **Stop**. Нажатие этой кнопки в дальнейшем приведет к остановке анимации. Этот апплет "говорит": "Hello, World!". Анимация используется для постепенного изменения цвета выводимого текста (рис. 1.25).

Рис. 1.25. Апплет с анимацией

Листинг 1.32. Файл HelloWorldSpectrum.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class HelloWorldSpectrum extends JApplet {
   Display display;  // JPanel вложенного класса  "Display"
                     // для отображения "Hello, World!"
   JButton startStopButton; // кнопка, которая используется
                            // для запуска и остановки анимации
   Timer timer;             // таймер, управляющий анимацией
   // Таймер запускается, когда пользователь нажимает
   // кнопку. Каждый раз при получении события таймера
   // изменяется цвет выводимого текста.
   // Значение переменной равно null,
   // если анимация не работает.
```

```
 int colorIndex;  // Число в промежутке от 0 до 100,
                  // которое определяет цвет.
                  // При получении события это число
                  // увеличивается на 1.
 public void init() { // вызывается системой для инициализации апплета
 display = new Display(); // компонент, отображающий "Hello, World!"
 getContentPane().add(display, BorderLayout.CENTER);
   // вставляет панель отображения в центр
   // панели апплета JApplet
 JPanel buttonBar = new JPanel();
  // панель, содержащая кнопку
 buttonBar.setBackground(Color.gray);
 getContentPane().add(buttonBar, BorderLayout.SOUTH);
 startStopButton = new JButton("Start");
 buttonBar.add(startStopButton);
 startStopButton.addActionListener(new ActionListener() {
  // Прослушиватель событий, который реагирует
  // на нажатия кнопки, запускающей и
  // останавливающей анимацию
  // и проверяющей значение таймера.
  // Если таймер не null, то анимация запущена,
  // то есть ее надо остановить. Если таймер null, то
  // анимацию следует запустить.
 public void actionPerformed(ActionEvent evt) {
if (timer == null)
   startAnimation();
else
   stopAnimation();
 }
});
 }
 ActionListener timerListener = new ActionListener() {
 // Описывает прослушиватель событий, отвечающий
 // на события таймера.
 // При получении событий таймера индекс цвета изменяется на 1.
 public void actionPerformed(ActionEvent evt) {
 colorIndex++;  // число от 0 до 100
 if (colorIndex > 100)
colorIndex = 0;
```

```
 float hue = colorIndex / 100.0F;  // От  0.0F до 1.0F.
 display.setColor(Color.getHSBColor(hue,1,1));
 }
};
 void startAnimation() {
 // Запускается анимация, если она еще не запущена.
 // Состояние проверяется по значению таймера.
 // Если значение null, то анимация не запущена.
 if (timer == null) {
 // создается таймер, который будет
 // создавать событие каждые 50 мс, отправляя их
 // прослушивателю событий
timer = new Timer(50, timerListener);
timer.start();  // Запускаем таймер.
startStopButton.setText("Stop");
 }
 }
 void stopAnimation() {
 if (timer != null) {
  timer.stop();   // останавливаем таймер
  timer = null;   // переменной таймера присваиваем null
  startStopButton.setText("Start");
  }
 }
 public void stop() {
 // Метод апплета stop() вызывается системой перед тем,
 // как апплет должен прекратить работу временно или навсегда.
 // Работающий таймер не нужен тогда, когда апплет стоит,
 // останавливаем анимацию. Если анимация не работает,
 // то ее не надо останавливать.
 stopAnimation();
 }
 // вложенный класс для представления "холста" для рисования
 class Display extends JPanel {
 // Вложенный класс представляет панель для отображения
 // строки "Hello, World!". Цвет и шрифт запоминаются
 // в переменных colorNum и textFont.
```

```
  Color color;   // цвет, которым выводится текст (изначально красный)
  Font textFont; // шрифт, которым выводится текст
                 // объект шрифта содержит размер и стиль текста
  Display() {
  // Конструктор класса Display. Задается цвет фона и начальные
  // значения для переменных цвета и шрифта.
   setBackground(Color.black);
   color = Color.red; // начальный цвет текста
   textFont = new Font("Serif",Font.BOLD,36);
   // создается объект шрифта
  }
  void setColor(Color color) {
  // Метод вызывается из основного класса для установки цвета
  // текста и меняет цвет текста.
  // Цвет Color не должен быть null.
   this.color = color;
   repaint();
  }
  public void paintComponent(Graphics g){
  // Вызывается системой для прорисовки панели Jpanel.
  // Вызывается класс super.paintComponent()
  // для заполнения фона заданным цветом.
  // Затем выводится текст в соответствии с текущим цветом.
   super.paintComponent(g);
   g.setColor(color);
   g.setFont(textFont);   // задается шрифт
   g.drawString("Hello World!", 25,50);  // выводится текст
  }
   }
}
```

Апплет реагирует на события `ActionEvents`, которые генерируются либо кнопкой, либо таймером. Для каждого источника используется отдельный прослушиватель событий. В апплете заданы методы `startAnimation()` и `stopAnimation()`, эти методы вызываются при нажатии на кнопку. Каждый раз при нажатии на кнопку **Start** создается новый таймер:

```
  timer = new Timer(50,timerListener);
  timer.start();
```

Методы *start()* и *stop()* в апплете JApplet

Некоторые методы требуют дополнительных объяснений. Зачем используется метод `stop()`? Каждый апплет имеет набор методов, которые выполняются в тот или иной момент жизненного цикла апплета. Так, метод `init()` вызывается при создании апплета перед тем, как он будет отображен на экране. Метод `destroy()` непосредственно перед тем, как апплет будет уничтожен, чтобы дать шанс для высвобождения ресурсов. Методы `start()` и `stop()` вызываются между методами `init()` и `destroy()`. Метод `start()` вызывается сразу после метода `init()`, а метод `stop()` вызывается непосредственно перед методом `destroy()`. Методы `start()` и `stop()` можно вызывать и в других ситуациях. Причиной этого является тот факт, что апплет не всегда бывает активен. Предположим, что мы переходим на новую страницу, покидая ту, на которой расположен апплет. Апплет перестает быть видимым, в нем не выполняется метод прорисовки, апплет не получает никаких событий от пользователя. Система вызывает метод `stop()` тогда, когда мы покидаем страницу, на которой расположен апплет. При возвращении на страницу будет вызван метод `start()`, что делает апплет вновь активным. Это полезно потому, что апплет не будет использовать ресурсы системы, когда находится в неактивном состоянии.

В нашем случае вряд ли следует оставлять таймер работающим в течение того времени, когда апплет будет неактивен. В методе `stop()` происходит обращение к методу `stopAnimation()`, который выключает таймер (листинг 1.33).

Листинг 1.33. Файл ScrollingHelloWorld.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ScrollingHelloWorld extends JApplet
    implements ActionListener {
   Timer timer;  // Таймер, на основе которого создана анимация.
                 // Он создается и запускается в методе start()
                 // и останавливается в методе апплета stop().
   String message = "Hello World (for absolutely the last time)!";
                 // текст, который прокручивается на экране
   int messagePosition = -1;
   // Текущее положение текста (в пикселах) в
   // отступе от левого края апплета. Для каждого фрейма это
   // значение изменяется на ширину одной буквы до тех пор,
   // пока текст полностью не исчезнет из апплета.
   // После этого величина приравнивается к 0.
```

```
 // Значение -1 соответствует тому,
 // что прокрутка еще не началась.
 int messageHeight;  // данные о размере текстового сообщения
 int messageWidth;
 int charWidth;  // ширина одного символа
 public void init() {
 // Панель JPanel используется в качестве "холста"
 // для рисования апплета, здесь задаются цвета и шрифт.
 // Используется равноширинный шрифт "Monospaced",
 // все символы имеют одинаковую ширину.
JPanel display = new JPanel() {
public void paintComponent(Graphics g) {
 // отображение текста с позиции
 // messagePosition в пикселах
 super.paintComponent(g);
 g.drawString(message, getWidth() - messagePosition,
     getHeight()/2 + messageHeight/2);
}
 };
setContentPane(display);
display.setBackground(Color.white);
display.setForeground(Color.red);
Font messageFont = new Font("Monospaced", Font.BOLD, 30);
display.setFont(messageFont);
FontMetrics fm = getFontMetrics(messageFont);
messageWidth = fm.stringWidth(message);
messageHeight = fm.getAscent();
charWidth = fm.charWidth('H');
 }
 public void start() {
 // Вызывается при первом и повторном запусках апплета
 // Создает новый таймер или запускает существующий таймер.
if (timer == null) {
 timer = new Timer(300, this);
 timer.start();
}
else {
 timer.restart();
}
```

```
    }
    public void stop() {
    // Вызывается тогда, когда апплет должен быть остановлен.
    // Останавливает таймер.
   timer.stop();
    }
    public void actionPerformed(ActionEvent evt) {
   // Вычисляет и отображает следующий фрейм анимации.
   // Метод вызывается в ответ на событие таймера.
   // Передвигает текст на один символ влево.
   // Если текст полностью исчезает из апплета,
   // все возвращается в начальное состояние.
    messagePosition += charWidth;
    if (getSize().width – messagePosition + messageWidth < 0) {
   // текст перемещается
   messagePosition = 0;
    }
    repaint();
    }
}
```

Апплет показывает прокручивающийся текст (рис. 1.26, 1.27).

Рис. 1.26. Анимированный текст (начало движения)

Рис. 1.27. Анимированный текст (продолжение движения)

Прочие методы использования таймера

Помимо генерации последовательности равноудаленных во времени событий, таймер может быть использован для однократного создания события. Для этого необходимо после создания нового объекта таймера `setRepeats` передать ему в качестве параметра значение `false`. Например:

```
Timer alarm = new Timer(5000, listener);
alarm.setRepeats(false);
alarm.start();
```

Такой способ использования таймера создает своеобразные "звоночки". В нашем случае "звонок" прозвенит один раз по истечении 5000 миллисекунд (5 секунд). Звонок можно отменить, для этого необходимо использовать метод `stop()` объекта `Timer` в любое время до наступления события (до звонка).

После создания таймера мы имеем возможность изменить начальное значение времени, отделяющее события друг от друга в повторяющемся таймере, для этого необходимо вызвать метод `timer.setInitialDelay(delay)`.

1.6.3. Потоки

Таймер скрывает в себе потоки. Таким образом, работа с ними становится опосредованной. Однако могут возникнуть такие ситуации, когда необходимо обратиться к потокам и использовать их в явном виде. О потоках будет рассказано на примере. Этот пример представляет собой вполне реалистичную задачу, которая требует большого количества вычислений, связанных

с построением графического образа фрактального множества, в данном случае это будет множество Мандельброта. Строгое определение фрактального множества состоит в констатации того факта, что топологическая размерность этого множества должна быть строго меньше его хаусдорфовой размерности.

Вычисления производятся на основе следующего алгоритма. Построение начинается в точке (x, y), где x и y — это вещественные числа. Вычисляем новую точку $zx = x$, $zy = y$, повторяем процесс замены точки (zx, zy) новым значением $(z \times x \times zx - zy \times zy + x, 2zx \times zy + y)$.

Для построения картинки чуть иначе сформулируем задачу. Выберем точку *(x, y)* и зададимся вопросом, сколько шагов (не более заданного) необходимо для того, чтобы эта точка, при использовании описанной выше процедуры, удалилась от точки *(0, 0)* на определенное расстояние? Затем мы зададим цвет точки в соответствии с тем, какое количество шагов потребуется. Проделав эту процедуру для набора точек, мы получим некоторую картинку. Приводим текст апплета (листинг 1.34), который производит такие вычисления (рис. 1.28). Апплет начинает работать после нажатия кнопки **Start**. В апплете представлена область плоскости, ограниченная условиями $-1.25 \le x \le 1.0$ и $-1.25 \le y \le 1.25$.

Программа работает по следующему алгоритму:

```
Для квадратов размерами 64, 32, 16, 8, 4, 2, и 1:
  Для каждого квадрата в апплете:
   Пусть (a,b) — координаты центра квадрата.
   Пусть (x,y) — вещественные числа, соответствующие (a,b).
   Пусть (zx,zy) = (x,y).
   Пусть count = 0.
   Повтор до тех пор, пока count = 80 или (zx,zy) стало "большим":
   Пусть new_zx = zx*zx — zy*zy + x.
   Пусть zy = 2*zx*zy + y.
   Пусть zx = new_zx.
   Пусть count = count + 1.
   Пусть color = Color.getHSBColor(count/100.0F, 0.0F, 0.0F).
   Заполнить квадрат заданным цветом.
```

На практике оказывается, что это вычисление требует много времени.

Листинг 1.34. Файл Mandelbrot.java

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

```
public class Mandelbrot extends JApplet {
   Display canvas;
    // на "холсте" будет отображаться множество Мандельброта
   JButton stopButton, startButton;  // вычисления начинаются после
                                     // нажатия кнопки Start и
                                     // продолжаются до тех пор,
                                     // пока не нажата кнопка Stop
   public void init() {
     // инициализация апплета
  setBackground(Color.gray);
  canvas = new Display();
  getContentPane().add(canvas, BorderLayout.CENTER);
  JPanel bottom = new JPanel();
  bottom.setBackground(Color.gray);
  startButton = new JButton("Start");
  startButton.addActionListener(canvas);
  bottom.add(startButton);
  stopButton = new JButton("Stop");
  stopButton.addActionListener(canvas);
  bottom.add(stopButton);
  stopButton.setEnabled(false);
  getContentPane().add(bottom, BorderLayout.SOUTH);
   }
   public Insets getInsets() {
   // оставляем место вдоль границы апплета,
   // заполненное цветом фона (в данном случае – серым)
  return new Insets(2,2,2,2);
   }
   public void stop() {
   // вызывается системой перед тем, как апплет должен
   // быть остановлен
   canvas.stopRunning();
   }
   // следующий вложенный класс рисует
   // и производит необходимые вычисления
   class Display extends JPanel implements ActionListener, Runnable {
  Image OSI;  // Изображение (не на экране), в котором хранится
              // картинка, соответствующая множеству Мандельброта.
```

```
            // Изображение создается вычислительным потоком,
            // а затем копируется на холст.
Graphics OSG; // графический контекст
Thread runner;  // поток вычислений
boolean running;  // значение true, если поток работает
double xmin = -2.5;   // интервал координат
double xmax = 1;
double ymin = -1.25;
double ymax = 1.25;
public void paintComponent(Graphics g) {
  // Вызывается системой для рисования.
  // Копирует рисунок на экран
  // (если рисунок существует)
  // если рисунка нет, то заполняет экран черным цветом.
 if (OSI == null) {
g.setColor(Color.black);
g.fillRect(0,0,getWidth(),getHeight());
 }
 else {
g.drawImage(OSI,0,0,null);
 }
}
public void actionPerformed(ActionEvent evt) {
// Вызывается в том случае,
// если пользователь нажимает кнопку "Start"
// или кнопку "Stop".
// В ответ анимация либо запускается,
// либо останавливается.
 String command = evt.getActionCommand();
 if (command.equals("Start"))
startRunning();
 else if (command.equals("Stop"))
stopRunning();
}
void startRunning() {
 // простой метод, который запускает поток вычислений
 // (если он еще не запущен)
 if (running)
return;
```

```
 runner = new Thread(this);
 running = true;
 runner.start();
}
void stopRunning() {
 // метод, останавливающий поток вычислений
 running = false;
}
int countIterations(double x, double y) {
// Множество Мандельброта представлено с использованием
// различных цветов, которыми раскрашены квадраты
// в зависимости от того, сколько шагов требуется
// произвести до того, как точка достигнет заданного
// расстояния. Максимальное число
// шагов установлено равным 80. Прочие цвета
// относятся к точкам, не входящим в множество Мандельброта.
 int count = 0;
 double zx = x;
 double zy = y;
 while (count < 80 && Math.abs(x) < 100 && Math.abs(zy) < 100) {
double new_zx = zx*zx - zy*zy + x;
zy = 2*zx*zy + y;
zx = new_zx;
count++;
 }
 return count;
}
int i,j;   // центральный пиксел отображаемого квадрата
int size;  // размер отображаемого квадрата
int colorIndex;  // цвет квадрата — число в пределах от 1 до 80,
                 // соответствующее свойствам
                 // центральной точки квадрата
Runnable painter = new Runnable() {
  // объект Runnable, который рисует квадрат,
  // а затем копирует его на экран
public void run() {
 int left = i - size/2;
 int top = j - size/2;
 OSG.setColor(Color.getHSBColor(colorIndex/100.0F,1F,1F));
```

```
 OSG.fillRect(left,top,size,size);
 paintImmediately(left,top,size,size);
}
};
public void run() {
// Этот метод запускает поток вычислений. Он рисует множество
// Мандельброта, которое формируется в несколько проходов
// с увеличением разрешения картинки. С каждым разом
// сторона прорисовываемого квадрата уменьшается вдвое.
 startButton.setEnabled(false);  // выключить кнопку "Start"
 stopButton.setEnabled(true);  // включить кнопку "Stop"
                               // на время работы потока
 int width = getWidth();   // текущий размер холста
 int height = getHeight();
 OSI = createImage(getWidth(),getHeight());
   // Создать рисунок за экраном для хранения изображения.
   // Сначала картинка заполнена черным цветом.
 OSG = OSI.getGraphics();
 OSG.setColor(Color.black);
 OSG.fillRect(0,0,width,height);
 for (size = 64; size >= 1 && running; size = size/2) {
  double dx,dy;  // размер апплета
  dx = (xmax - xmin)/width * size;
  dy = (ymax - ymin)/height * size;
  double x = xmin + dx/2;  // x-координата центра квадрата
 for (i = size/2; i < width+size/2 && running; i += size) {
  // цикл рисования одного столбца квадратов
 double y = ymax - dy/2; // y-координата центра квадрата
 for (j = size/2; j < height+size/2 && running; j += size) {
  // самый нижний цикл рисования квадрата,
  // просчет числа итераций для определения цвета,
  // вызов "painter" для рисования
   colorIndex = countIterations(x,y);
   try {
  SwingUtilities.invokeAndWait(painter);
   }
   catch (Exception e) {
   }
   y -= dy;
```

```
    }
    x += dx;
    Thread.yield();  // дать шанс другому потоку
   }
    }
    running = false;  // поток завершен, поскольку либо
                      // вычисления завершены,
                      // либо выполнение потока где-либо
                      // прекращено
    startButton.setEnabled(true);  // восстановление состояний
                                   // кнопок
    stopButton.setEnabled(false);
   }
    }
}
```

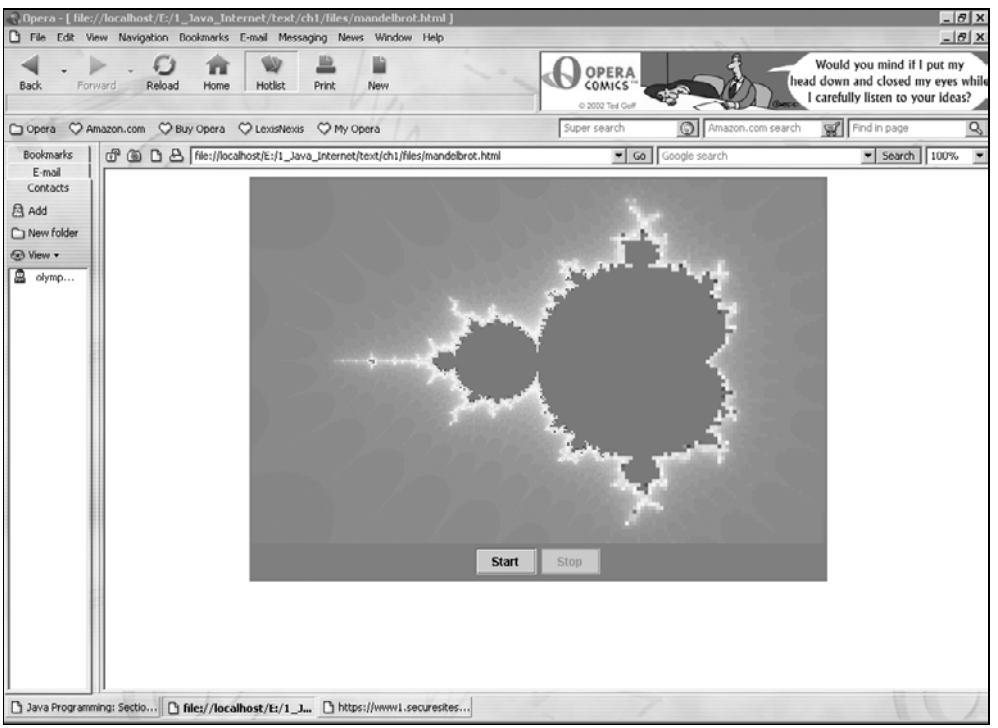


Рис. 1.28. Отображение множества Мандельброта

Потоки в Java принадлежат классу `java.lang.Thread`. Назначение потока — независимое выполнение самостоятельного фрагмента программы парал-

лельно с остальной программой. В случае с программой рисования множества Мандельброта, этот кусок программы реализует алгоритм, который был приведен выше. Как правило, в потоке используется метод

```
public void run();
```

Этот метод определяется для объекта, в котором имплементируется интерфейс `Runnable`. Интерфейс `Runnable` описывает единственный метод `run()`. Объект `Runnable` будет являться параметром конструктора объекта потока. Можно определить поток как подкласс класса `Thread` и определить метод `run()` в подклассе, но удобнее пользоваться объектом `Runnable`. Если этот объект является объектом, имплементирующим интерфейс `Runnable`, то поток можно создать с помощью команды:

```
Thread runner = new Thread(runnableObject);
```

Задача этого потока — выполнение метода `run()`. Как и в случае с таймером, создание потока — это не полное решение задачи. Необходимо запустить поток, для этого вызывается метод `runner.start()`. Вызов этого метода приводит к запуску потока, выполняется метод `run()`, выполнение его происходит параллельно с остальной частью программы. Для остановки потока не существует метода, аналогичного методу `stop()` в таймере, который бы приводил к остановке выполнения потока. Если мы хотим иметь возможность остановить поток, то необходимо позаботиться о создании такого механизма, который бы позволил сообщить потоку о том, что следует прекратить выполнение инструкций потока. Это можно сделать, например, задав переменную `running`, которая видима как в методе `run()`, так и в любой другой части программы. Когда появляется необходимость остановить поток, этой переменной присваивается значение `false`. В методе `run()` происходит регулярная проверка значения этой переменной. Если обнаруживается, что значение этой переменной становится равным `false`, то выполнение метода `run()` должно быть прекращено. Вот, например, как происходит остановка потока в апплете, отображающим множетсво Мандельброта:

```
void startRunning() {
 // Метод, запускающий поток вычислений.
 // Этот метод используется только один раз
 // нажатии кнопки "Start", после чего кнопка
 // выключается, а поток начинает работу.
 if (running)
return;
 runner = new Thread(this);
 // создает поток, который будет выполнять метод run()
 // класса, имплементирующего интерфейс Runnable
 running = true;
```

```
    runner.start();
  }
    void stopRunning() {
    // Простой метод, который останавливает выполняемый поток
    // с вычислениями.
    // Это происходит посредством установления значения
    // переменной running, которое становится равным false.
    // Поток регулярно проверяет значение этой переменной
    // и при необходимости прекращает работу.
    running = false;
  }
```

На некоторых платформах после начала своей работы поток забирает контроль над процессором и не позволяет другим потокам начать работать до тех пор, пока не отдаст его, для чего используется тактический метод `Thread.yield()`. Полезно использовать этот метод, чтобы не возникли трудности в дальнейшем. Существует и другой метод предоставить возможность другому потоку приступить к работе, при этом текущий поток "засыпает" на некоторое время, указываемое в качестве аргумента при обращении к методу `Thread.sleep(time)`. Здесь время `time`, отводимое для "сна", выражается в миллисекундах. В таймере, например, поток "спит" в перерывах от наступления одного события до другого. При работе с потоками совместно с компонентами `Swing` следует быть осторожными. Лучше не применять компоненты или используемые ими данные при работе с потоками, за исключением случаев работы с потоками обработки событий. Потоки можно использовать только в методах обработки событий или в методе `paintComponent()`. Если более одного потока будет работать с данными пакета `Swing`, то эти данные могут быть испорчены. Для решения этой проблемы существует специально разработанный механизм. `Swing` позволяет другому потоку осуществлять доступ к потоку обработки событий, запуская для этого функции. Функция должна представлять собой метод `run()` объекта `Runnable`. Когда поток вызывает метод

```
void SwingUtilities.invokeAndWait(Runnable runnableObject)
```

то метод `run()` будет выполнен в потоке обработки событий, и здесь он безопасно может делать все с компонентами `Swing` и соответствующими им данными. Метод `run()` в апплете Мандельброта использует метод `SwingUtilities.invokeAndWait()` для заполнения заданным цветом того или иного квадрата, изображаемого на холсте апплета (сначала за экраном, потом изображение переносится на экран). Весь метод `run()` выглядит следующим образом:

```
  public void run() {
    startButton.setEnabled(false);
```

```
 stopButton.setEnabled(true);
 int width = getWidth();
 int height = getHeight();
 OSI = createImage(getWidth(),getHeight());
 OSG = OSI.getGraphics();
 OSG.setColor(Color.black);
 OSG.fillRect(0,0,width,height);
 for (size = 64; size >= 1 && running; size = size/2) {
double dx,dy;
dx = (xmax – xmin)/width * size;
dy = (ymax – ymin)/height * size;
double x = xmin + dx/2;
for (i = size/2; i < width+size/2 && running; i += size) {
 double y = ymax – dy/2;
 for (j = size/2; j < height+size/2 && running; j += size) {
   colorIndex = countIterations(x,y);
   try {
  SwingUtilities.invokeAndWait(painter);
   }
   catch (Exception e) {
   }
   y -= dy;
 }
 x += dx;
 Thread.yield();
}
 }
 running = false;
 startButton.setEnabled(true);
 stopButton.setEnabled(false);
}
```

Глава 2

Работа с сетью

Сеть предоставляет возможность осуществлять ввод и вывод данных, взаимодействуя с удаленным компьютером. Работа с сетью несколько отличается от работы с файлами (работе с файлами уделено внимание ниже). В Java работа с сетью может быть осуществлена с использованием потоков ввода и вывода во многом таким же образом, как это происходит при работе с файлами. Открытие сетевого соединения между двумя компьютерами, тем не менее, усложняет задачу, так как эти компьютеры должны каким-то образом "договориться" между собой о том, как они будут общаться. Но прежде чем мы перейдем к изучению принципов работы с сетью, остановимся на рассмотрении основных моментов работы с файлами в Java.

2.1. Потоки ввода и вывода

Без возможности взаимодействия с окружающим миром программа не представляла бы никакого интереса, она попросту стала бы бесполезной. Взаимодействие программы с миром осуществляется посредством ввода/вывода. Компьютер включает в себя множество различных устройств. И если бы программирование каждого устройства требовало к себе индивидуального подхода, то создание программ стало бы слишком сложным делом. Для работы с вводом/выводом используются абстракции. В языке Java с вводом и выводом используется абстракция потоков ввода/вывода. Мы не станем рассматривать в деталях строение потоков. При работе с вводом/выводом всегда следует понимать, что данные могут быть представлены как в машинном виде, так и в таком виде, который удобен и легко читаем человеком. Сформатированные в машинном виде данные представлены также, как и данные, которые хранятся в памяти машины, то есть в виде последовательностей двоичных состояний (последовательностей нулей и единиц). Для чтения человеком данные представляются в виде символов. Мы читаем число π как 3.141592654, которое представлено в виде набора символов-цифр. В Java существует две категории представления данных. Это байтовые потоки, данные в которых представлены в машинном виде, и символьные потоки, которые более удобны для чтения человеком. Существует множество

заранее определенных классов, предназначенных для работы как с той, так и с другой категорией данных.

Каждый объект, который имеет возможность выводить данные в байтовый поток, принадлежит одному из подклассов абстрактного класса `OutputStream`. Объекты, которые читают данные байтового потока, относятся к подклассам класса `InputStream`. Для чтения и записи обычных символов, которые может читать человек, используются классы `Reader` и `Writer`. Все классы символьных потоков являются подклассами одного из этих классов. Если необходимо работать с данными, которые может читать человек, то используются символьные потоки, в противном случае используются байтовые потоки.

Стандартные классы для работы с потоками определены в пакете `java.io`, здесь же определен ряд вспомогательных классов. Перед работой с потоками следует импортировать этот пакет в программу при помощи инструкции

```
import java.io.*;
```

Эта инструкция должна быть помещена в самом начале файла, содержащего код программы. При работе с графическим интерфейсом пользователя потоки не используются, там работает свой собственный механизм ввода/вывода. Потоки необходимы при работе с файлами. Сетью потоки могут быть использованы для установления взаимодействия между параллельно выполняемыми подпроцессами (которые мы также называем потоками, но это не потоки ввода/вывода, а программные потоки). Достоинство потоков в том и состоит, что с их помощью можно легко записывать данные в файл или пересылать данные по сети, выводить данные на экран.

Базовые классы ввода/вывода, `Reader`, `Writer`, `InputStream`, `OutputStream` предоставляют только основные операции ввода/вывода. Так, класс `InputStream` объявляет метод

```
public int read() throws IOException
```

при помощи которого происходит чтение одного байта данных (0—255) потока ввода. Если достигнут конец потока ввода, то метод `read()` возвращает значение −1. Класс `InputStream` определяет методы, которые позволяют читать несколько байт в один прием, помещая прочитанные байты в байтовый массив. Вывод может быть осуществлен при помощи метода `write()`:

```
public void write(int b) throws IOException
```

Классы `Reader` и `Writer` предоставляют весьма схожие методы. Но здесь чтение и запись происходит с символьными данными. Так, подкласс `PrintWriter` класса `Writer` предоставляет удобные методы для вывода читаемых символов всех примитивных типов, используемых в Java. Если мы располагаем объектом, принадлежащим классу `Writer` или его подклассу, и хотим использовать метод `PrintWriter` для вывода данных, то для этого

необходимо объект типа `Writer` (в примере ниже это объект `charSink`) передать конструктору `PrintWriter`:

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

Перечислим методы, используемые в классе `PrintWriter` для осуществления вывода:

```
public void print(String s)   // вывод стандартных типов
public void print(char c)     // в читаемой форме
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(boolean b)

public void println()            // возврат каретки

public void println(String s)    // то же, что и выше,
public void println(char c)      // но после вывода символа
public void println(int i)       // выводится возврат каретки
public void println(long l)
public void println(float f)
public void println(double d
public void println(boolean b)
```

Эти методы не реагируют на ошибку `IOException`. Для определения ошибки используется метод

```
public boolean checkError()
```

Метод возвращает `true`, если произошла ошибка. В дальнейшем мы будем ссылаться на класс `TextReader`. Этот класс описан в файле `TextReader.java`. В этом классе содержатся полезные методы (листинг 2.1).

Листинг 2.1. Файл TextReader.java

```
/*
Класс TextReader — подкласс класса FilterReader.
Класс TextReader содержит методы чтения данных в удобочитаемом формате
в виде символов ASCII.
В файле определено три подкласса public от класса RuntimeException для
работы с ошибками, которые могут произойти во время ввода. Эти классы
являются статическими вложенными классами для класса TextReader.
*/
import java.io.*;
```

```
public class TextReader extends FilterReader {
// Задание исключений, с которыми работает класс, исключения — подклассы
// класса RuntimeException. Обработка исключений необязательна.
public static class Error extends RuntimeException {
   // любое исключение, возникающее в классе TextReader
   Error(String errorMessage) {
     super(errorMessage);
   }
}
public static class FormatError extends Error {
// неверные данные: неверное число или неверное логическое значение
FormatError(String errorMessage) {
      super(errorMessage);
   }
}
public static class EOFError extends Error {
   // попытка чтения
   EOFError(String errorMessage) {
      super(errorMessage);
   }
}
// конструкторы
public TextReader(BufferedReader s) {
  super(s);
}
public TextReader(Reader s) {
  super(new BufferedReader(s));
}
public TextReader(InputStream s) {
  super(new BufferedReader(new InputStreamReader(s)));
}
// проверка ошибок
public void IOCheck(boolean throwExceptions) {
  // вызов IOCheck(false)
  // для выключения ошибок
  throwExceptionOnError = throwExceptions;
 }
// и проверка с вызовом checkError()
public boolean error() {
```

```
  // возвращает true, если последняя операция ввода
  // в TextReader привела к ошибке
  return errorFlag;
}
 // сообщение об ошибке можно получить от getErrorMessage()
 public String getErrorMessage() {
   return errorFlag ? errorMessage : null;
 }
// методы ввода
// peek() - следующий символ для ввода из потока.
// Символ не убирается из потока, если следующий символ — конец строки,
// то возвращается '\n'.
// '\0'  соответствует концу файла.
// Не существует способа различить EOF и null.
// Для обнаружения конца файла используется eof().
// getAnyChar() — читает и возвращает следующий символ, в том числе
// символ пробела. Было бы ошибкой читать символы после конца файла.
// Отметим, что getChar() пропускает пробелы перед тем, как прочитает
// символы.
// getChar(), getByte(), и прочие — пропускают пробелы и читают значения
// соответствующего типа. Если данные указанного типа не будут найдены,
// то возникает ошибка. "word" — последовательность символов после
// пробела, "boolean" — одно из слов  "true", "false", "yes", "no",
// "t", "f", "y", "n", "0", "1".  "Alpha" — последовательность букв.
// getAlpha() — пропускает все небуквенные символы перед началом чтения.
// getln() — читает и выводит все символы до конца строки, записывая их
// в String. Символ конца строки читается, но игнорируется.
// getlnChar(), getlnByte()— работает так же, как getChar(),
// с тем исключением, что после чтения значений все символы в
// строке, включая символ конца строки, читаются, но затем игнорируются.
// eoln() — читает и игнорирует все пробелы и табуляции. Проверяет,
// является ли следующий символ символом конца строки. Конец файла также
// считается концом строки. Если необходимо проверить на наличие конца
// строки не пренебрегая пробелами и табуляциями, то следует использовать
// еще метод peek() == '\n' or '\0'.
// eof() — Читает и игнорирует все пробелы, концы строк и табуляции,
// затем проверяет, является ли следующий символ символом конца файла.
// Чтобы не игнорировать пробелы, концы строк, табуляции, используйте
// peek() == '\0'.
```

```
// skipWhiteSpace() — читает и игнорирует пробелы, табуляции и концы
// строк. Прекращает работу при нахождении другого символа или конца
// файла.
// skipNonLetters() — читает и игнорирует все небуквенные символы,
// прекращает работу по достижении буквенного символа или символа конца
// файла.
public char peek() {
  errorFlag = false; return lookChar();
}
public char getAnyChar() {
  errorFlag = false; return readChar();
}
public char getChar() {
  errorFlag = false; skipWhiteSpace();
  return readChar();
}
public byte getByte() {
  errorFlag = false;
  return (byte)readInteger(-128L,127L);
}
public short getShort() {
  errorFlag = false;
  return (short)readInteger(-32768L,32767L);
}
public int getInt() {
  errorFlag = false;
  return (int)readInteger((long)Integer.MIN_VALUE,
  (long)Integer.MAX_VALUE);
}
public long getLong() {
  errorFlag = false;
  return readInteger(Long.MIN_VALUE, Long.MAX_VALUE);
}
public float getFloat() {
  errorFlag = false; return readFloat();
}
public double getDouble() {
  errorFlag = false; return readDouble();
}
```

```
public boolean getBoolean() {
errorFlag = false; return readBoolean();
}
public String getWord() {
  errorFlag = false; return readWord();
}
public String getAlpha() {
  errorFlag = false; return readAlpha();
}
public String getln() {
  errorFlag = false; return readLine();
}
public char getlnChar() {
  char x=getChar();
  dropLine();
  return x;
}
public byte getlnByte() {
  byte x=getByte();
  dropLine();
  return x;
}
public short getlnShort() {
  short x=getShort();
  dropLine();
  return x;
}
public int getlnInt() {
  int x=getInt();
  dropLine();
  return x;
}
public long getlnLong() {
  long x=getLong();
  dropLine();
  return x;
}
public float getlnFloat() {
  float x=getFloat();
```

```
  dropLine();
  return x;
}
public double getlnDouble() {
  double x=getDouble();
  dropLine();
  return x;
}
public boolean getlnBoolean() {
  boolean x=getBoolean();
  dropLine();
  return x;
}
public String getlnWord() {
  String x=getWord();
  dropLine();
  return x;
}
public String getlnAlpha() {
  String x=getAlpha();
  dropLine();
  return x;
}
public boolean eoln() {
  char ch = lookChar();
  while (!EOF && !errorFlag && (ch == ' ' || ch == '\t')) {
    readChar();
    ch = lookChar();
  }
  return (ch == '\n' || EOF);
}
public boolean eof() {
  char ch = lookChar();
  while (!EOF && !errorFlag && (ch == ' ' ||
                                ch == '\n' || ch == '\t')) {
     readChar();
     ch = lookChar();
  }
  return EOF;
```

```
}
public void skipWhiteSpace() {
  char ch = lookChar();
  while (!errorFlag && (ch == ' ' || ch == '\n' || ch == '\t')) {
    readChar();
    ch = lookChar();
  }
}
public void skipNonLetters() {
 char ch = lookChar();
 while (!errorFlag && !EOF && !Character.isLetter(ch)) {
   readChar();
   ch = lookChar();
 }
}
public void close() {
  errorFlag = false;
  try {
    in.close();
  }
  catch (IOException e) {
    errorFlag = true;
    errorMessage = e.toString();
  }
}

private int lookAhead = -1;  // буфер для одного символа;
                             // -1 — буфер пуст
private boolean errorFlag = false;    // true — если последняя
                                      // операция была с ошибкой
private String errorMessage = "";     // сообщение об ошибке для
                                      // последней ошибки
private boolean EOF = false;          // обнаружен ли конец потока?
private boolean throwExceptionOnError = true;  // определяет тип
                                               // обработки ошибки
// функции обработки ошибок
private void doError(String message) {
  errorFlag = true;
  errorMessage = message;
```

```
  if (throwExceptionOnError)
    throw new Error(message);
}
private void doFormatError(String message) {
  errorFlag = true;
  errorMessage = message;
  if (throwExceptionOnError)
    throw new FormatError(message);
}
private void doEOFError(String message) {
  errorFlag = true;
  errorMessage = message;
  if (throwExceptionOnError)
    throw new FormatError(message);
}
// методы чтения основных типов данных из потока
private char readChar() {
  char ch = lookChar();
  if (EOF)
    doEOFError("Attempt to read past end-of-data in input stream.");
  lookAhead = -1;
  return ch;
}
private boolean possibleLineFeedPending = false;  // для работы
                                                  // с символами \r\n
private char lookChar() {
  if (lookAhead != -1) {
    if (lookAhead == '\r')
       return '\n';
    else
      return (char)lookAhead;
  }
  if (EOF)
    return '\0';
  try {
    int n = in.read();
    if (n == '\n' && possibleLineFeedPending) { // игнорировать
                                                // в сочетании \n of \r\n
      n = in.read();
```

```
    }
    possibleLineFeedPending = (n == '\r');
    lookAhead = n;
    if (lookAhead == -1) {
      EOF = true;
      return '\0';
    }
 }
 catch (IOException e) {
   doError(e.getMessage());
 }
 if (lookAhead == '\r')  // все eoln в виде \n
   lookAhead = '\n';
 return (char)lookAhead;
}
private void dropLine() {
  while (!errorFlag) {
    if (lookChar() == '\0')
      return;
    if (readChar() == '\n')
      return;
 }
}
private String readLine() {
  StringBuffer s = new StringBuffer(100);
  char ch = readChar();
  while (!errorFlag && ch != '\n') {
    s.append(ch);
    ch = lookChar();
    if (ch == '\0')
      break;
    ch = readChar();
   }
   return s.toString();
}
private String readWord() {
  skipWhiteSpace();
  if (errorFlag)
    return null;
```

```
  StringBuffer s = new StringBuffer(50);
  char ch = lookChar();
  if (EOF) {
    doEOFError("Attempt to read past end-of-data");
    return null;
  }
  while (!errorFlag && !EOF && ch != '\n' && ch != ' ' && ch != '\t') {
    s.append(readChar());
    ch = lookChar();
  }
  return s.toString();
}
private String readAlpha() {
  skipNonLetters();
  if (errorFlag)
    return null;
  StringBuffer s = new StringBuffer(50);
  char ch = lookChar();
  if (EOF) {
    doEOFError("Attempt to read past end-of-data");
    return null;
  }
  while (!errorFlag && Character.isLetter(ch)) {
    s.append(readChar());
    ch = lookChar();
  }
  return s.toString();
}
public float readFloat() {
  double d = readDouble();
  if (errorFlag)
    return Float.NaN;
  if (Math.abs(d) > Float.MAX_VALUE)
    doFormatError("Input number outside of legal range for values of type
    float");
  return (float)d;
}
public double readDouble() {
  double x = Double.NaN;
```

```
StringBuffer s=new StringBuffer(50);
skipWhiteSpace();
char ch = lookChar();
if (ch == '-' || ch == '+') {
  s.append(readChar());
  skipWhiteSpace();
  ch = lookChar();
}
if ((ch < '0' || ch > '9') && (ch != '.')) {
  if (EOF)
    doEOFError("Expecting a floating-point number and found end-of-
    data");
  else
    doFormatError("Expecting a floating-point number and found \""  +
    ch + "\"");
  return Double.NaN;
}
boolean digits = false;
while (ch >= '0' && ch <= '9') {
  s.append(readChar());
  ch = lookChar();
  digits = true;
}
if (ch == '.') {
  s.append(readChar());
  ch = lookChar();
  while (ch >= '0' && ch <= '9') {
  s.append(readChar());
  ch = lookChar();
  digits = true;
}
if (!digits) {
  doFormatError("No digits found in floating-point input.");
  return Double.NaN;
}
if (ch == 'E' || ch == 'e') {
  s.append(readChar());
  ch = lookChar();
  if (ch == '-' || ch == '+') {
```

```
      s.append(readChar());
      ch = lookChar();
  }
  if ((ch < '0' || ch > '9') && (ch != '.')) {
    if (EOF)
      doEOFError("Expecting exponent for a floating-point number and
      found end-of-data");
    else
      doFormatError("Expecting exponent for a floating-point number and
      found \""  + ch + "\"");
    return Double.NaN;
  }
  while (ch >= '0' && ch <= '9') {
    s.append(readChar());
    ch = lookChar();
  }  String str = s.toString();
    Double d;
    try {
      d = new Double(str);
      x = d.doubleValue();
    }
    catch (NumberFormatException e) {
      x = Double.NaN;
    }
    if (Double.isNaN(x) || Double.isInfinite(x)) {
      doFormatError("Illegal floating point number");
      return Double.NaN;
    }
    return x;
}
public boolean readBoolean() {
  boolean ans = false;
  String s = getWord();
  if (errorFlag)
    return false;
  if (s.equalsIgnoreCase("true") || s.equalsIgnoreCase("t") ||
      s.equalsIgnoreCase("yes") || s.equalsIgnoreCase("y") ||
      s.equals("1")) {
    ans = true;
```

```
  }
  else if (s.equalsIgnoreCase("false") || s.equalsIgnoreCase("f") ||
           s.equalsIgnoreCase("no") || s.equalsIgnoreCase("n") ||
           s.equals("0")) {
    ans = false;
  }
  else
    doFormatError("Illegal input for value of type boolean: \"" + s +
                  "\"");
  return ans;
}
private long readInteger(long min, long max) {
  // чтение большого целого, оговорен диапазон
  skipWhiteSpace();
  if (errorFlag)
    return 0;
  char sign = '+';
    if (lookChar() == '-' || lookChar() == '+') {
      sign = getChar();
      skipWhiteSpace();
    }
  long n = 0;
  char ch = lookChar();
  if (ch < '0' || ch > '9') {
    if (EOF)
      doEOFError("Expecting an integer and found end-of-data");
    else
      doFormatError("Expecting an integer and found \""  +
                    ch + "\"");
    return 0;
  }
  while (!errorFlag && ch >= '0' && ch <= '9') {
    readChar();
    n = 10*n + (int)ch - (int)'0';
    if (n < 0) {
      doFormatError("Integer value outside of legal range");
      return 0;
    }
```

```
    ch = lookChar();
  }
  if (sign == '-')
    n = -n;
  if (n < min || n > max) {
    doFormatError("Integer value outside of legal range");
    return 0;
  }
  return n;
}
// переопределение метода read() из FilterReader
public int read() throws IOException {
  if (lookAhead == -1)
    return super.read();
  else {
    int x = lookAhead;
    lookAhead = -1;
    return x;
  }
}
public int read (char[] buffer, int offset, int count)
       throws IOException {
  if (lookAhead == -1 || count <= 0)
    return super.read(buffer,offset,count);
  else {
    if (count == 1) {
      buffer[offset] = (char)lookAhead;
      lookAhead = -1;
      return 1;
    }
    else {
      buffer[offset] = (char)lookAhead;
      lookAhead = -1;
      int ct = super.read(buffer,offset+1,count-1);
      return ct + 1;
  }
}
}
```

Классы PrintWriter, TextReader, DataInputStream, DataOutputStream позволяют легко работать с примитивными типами. Но что делать с объектами? По традиции используются те или иные способы представления объектов в виде наборов обычных примитивных типов, которые могут быть выведены как последовательность символов или байтов. Такой подход называется *сериализацией объектов*. При чтении сериализованного объекта необходимо его восстановить. Вся работа выполняется с помощью классов ObjectInputStream и ObjectOutputStream. Эти классы являются подклассами классов InputStream и Outputstream, с их помощью осуществляется запись и чтение сериализованных объектов. Классы ObjectInputStream и ObjectOutputStream могут быть использованы с любыми потоками InputStreams и OutputStreams. Это позволяет записывать и читать объекты в любых байтовых потоках. Используемые при чтении и записи объектов методы — это методы readObject() (в классе ObjectInputStream) и writeObject(Object obj) (в класс ObjectOutputStream). Оба метода могут вызывать исключение IOExceptions. Метод readObject() возвращает значение типа Object, которое, как правило, должно быть приведено к более приемлемому для использования и работы типу.

Классы ObjectInputStream и ObjectOutputStream можно использовать только при работе с объектами, которые имплементируют интерфейс Serializable. Более того, все переменные экземпляра класса, к которому относится данный объект, также должны быть сериализуемы. Интерфейс Serializable не содержит других методов. Интерфейс как бы сообщает компилятору о том, что все члены класса, имплементирующего интерфейс Serializable, могут быть как записаны, так и прочитаны с применением потоков. Что требуется сделать разработчику? Всего лишь вставить слова implements Serializable в описание класса! Многие стандартные классы Java уже объявлены как Serializable, в том числе все компоненты Swing и AWT. Это, в частности, означает, что все компоненты графического интерфейса пользователя могут быть записаны в поток ObjectOutputStreams и прочитаны из потока ObjectInputStreams.

2.2. Файлы

Программы могут читать данные из существующего файла либо создавать новые файлы. В Java это происходит с использованием потоков. Данные, предназначенные для чтения человеком, читаются из файла с использованием объектов, принадлежащих классу FileReader, который является подклассом класса Reader. Данные, которые следует записать в файл и которые имеют формат, предназначенный для чтения человеком, записываются с использованием объектов класса FileWriter, который является подклассом класса Writer. Для данных, не предназначенных для чтения человеком,

используются классы FileInputStream и FileOutputStream. В этой части мы коснемся только ввода и вывода данных, содержащих символы, то есть данных, предназначенных для чтения человеком. Необходимо иметь в виду, что классы FileInputStream и FileOutputStream используются точно таким же способом, что и классы FileReader и FileWriter. Все эти классы определяются в пакете java.io.

В предыдущей главе мы имели дело с апплетами. Апплеты не могут работать с файлами. Таковы требования безопасности. Апплет может быть получен прямо в браузер без участия пользователя, причем программа апплета передается по интернету. Если бы полученный апплет мог иметь доступ к файловой системе пользовательского компьютера, то можно было бы с легкостью создать такой апплет, который мог бы уничтожить все данные, расположенные на компьютере пользователя. Чтобы предотвратить подобные неприятности, предусмотрен ряд мер, обеспечивающих безопасность. Среди них есть запрет на доступ к локальным файлам. Самостоятельные Java-программы могут обращаться к файлам в полном объеме. При создании таких программ мы имеем возможность работать с файлами так, как пожелаем, в том числе выполнять все те действия, о которых рассказано в настоящей главе.

Класс FileReader располагает конструктором, который в качестве своего параметра требует указания потока ввода. Конструктор создает поток ввода по имени, указанном в качестве параметра. Этот поток затем может использоваться для чтения данных. Если файл отсутствует, то возникает исключение FileNotFoundException. Этот тип исключений требует обязательной обработки возникающих исключений, поэтому вызов конструктора следует производить внутри инструкции try либо внутри функции, которая работает с этим исключением (throw FileNotFoundException). Например, если у нас есть файл, который назван data.txt, и мы хотим создать программу, которая будет читать данные из этого файла, то можно поступить следующим образом:

```
FileReader data;  // Объявляем переменную до блока try.
                  // Переменная внутри блока не будет доступна извне.
try {
  data = new FileReader("data.txt");  // создаем поток
}
catch (FileNotFoundException e) {
  ... // обработка ошибок
}
```

Класс ошибки FileNotFoundException — это подкласс класса IOException. После того как поток будет удачно создан (с использованием FileReader), можно приступить к чтению данных из этого потока. Класс FileReader содержит только простейшие методы чтения, которые он наследует из класса

`Reader`, поэтому наш объект типа `FileReader` удобно обработать при помощи других классов, например, при помощи класса `TextReader`. (Класс `TextReader` не является стандартным классом в Java, его описание приведено выше.) Для создания объекта `TextReader`, который будет производить чтение данных из файла data.dat, можно написать набор следующих инструкций:

```
TextReader data;
try {
  data = new TextReader(new FileReader("data.dat"));
}
catch (FileNotFoundException e) {
 ... // обработка ошибок
}
```

После того как создан объект `TextReader` с именем data, можно переходить к использованию методов, которые позволяют осуществлять чтение: `data.getInt()`, `data.getWord()`.

Работы с записью в файлы — несколько более сложная задача. Мы создаем объект класса `FileWriter`. Созданный поток можно использовать для работы с объектом класса `PrintWriter`. Предположим, мы хотим записать данные в файл result.dat. Конструктор `FileWriter` чувствителен к ошибкам типа `IOException`, поэтому используем блок `try`:

```
PrintWriter result;
try {
  result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
  ... // обработка ошибок
}
```

Если файл с именем result.dat не существует, то будет создан новый файл. Если файл с указанным именем уже существует, то содержимое этого файла будет стерто и заменено данными, которые будут получены от программы. При работе может возникнуть исключение `IOException`, если, например, будет произведена попытка записать на защищенный от записи диск.

После того как работа с файлом завершена, файл следует закрыть. Это сообщит операционной системе о том, что мы прекратили использование файла. Если мы забудем это сделать, то, как правило, файл будет автоматически закрыт после завершения выполнения программы или после того, как объект потока для данного файла будет удален в процессе сбора мусора. Однако лучше специально позаботиться и закрыть файл в явном виде. Закрыть файл можно с применением метода `close()` для указанного потока.

После того как файл был закрыт, чтение данных из него становится невозможным до тех пор, пока мы вновь не откроем его, создав новый поток. Для большинства классов потоков метод `close()` может приводить к появлению исключения `IOException`. Это исключение необходимо обработать, но в `PrintWriter` и `TextReader` этот метод переопределен и не вызывает указанного исключения.

В качестве примера приведем полную программу, которая читает числа из файла, названного data.dat, после чего программа записывает прочитанные числа в противоположном порядке в файл result.dat. Предполагается, что файл data.dat содержит лишь по одному числу в каждой строке и количество чисел в файле не превышает 1000. Для проверки возникающих ошибок используются соответствующие инструкции. Это приложение не слишком практично, но в нем демонстрируются основные методы работы с вводом и выводом данных, хранящихся в файлах.

Листинг 2.2. Пример программы ввода-вывода

```
import java.io.*;
public class ReverseFile {
public static void main(String[] args) {
  TextReader data;     // поток ввода символов
  PrintWriter result;  // поток вывода символов
  double[] number = new double[1000];  // массив для хранения всех
                                       // прочитанных из файла чисел
  int numberCt;  // количество чисел, хранимых в массиве
  // создание потока ввода
  try {
    data = new TextReader(new FileReader("data.dat"));
  }
  catch (FileNotFoundException e) {
    System.out.println("Can't find file data.dat!");
    return;
  }
  // создание потока вывода
  try {
    result = new PrintWriter(new FileWriter("result.dat"));
  }
  catch (IOException e) {
    System.out.println("Can't open file result.dat!");
    System.out.println(e.toString());
```

```
    data.close();  // закрытие файла ввода
    return;
  }
  // чтение данных из файла
  try {
    numberCt = 0;
    while (data.eof() == false) {  // читать до конца файла
    number[numberCt] = data.getlnDouble();
    numberCt++;
  }
  // вывод чисел в обратном порядке
  for (int i = numberCt-1; i >= 0; i--)
    result.println(number[i]);
    System.out.println("Done!");
  }
  catch (TextReader.Error e) {
    // ошибки при чтении данных из файла
    System.out.println("Input Error: " + e.getMessage());
  }
  catch (IndexOutOfBoundsException e) {
    // слишком много данных в файле
    System.out.println("Too many numbers in data file.");
    System.out.println("Processing has been aborted.");
  }
  finally {
    // завершение работы программы и закрытие файла
    data.close();
    result.close();
  }
}
}
```

2.3. Имена. Каталоги. Класс *File*

Полное имя файла состоит из имени файла как такового и имени каталога, в котором располагается этот файл. Простые имена файлов, например, такие как data.dat или result.dat, используются только для обращения к файлам, расположенным в текущем (или рабочем) каталоге. К другим файлам следует обращаться с указанием полного имени, включая имя каталога.

Существует возможность задания либо абсолютного имени файла, либо относительного. Абсолютное имя указывает на расположение файла среди прочих файлов, причем это имя независимо от других файлов или каталогов. В нем содержится информация о директории и простом имени файла. Относительный путь (относительное имя файла) задает путь к файлу из текущего каталога. Синтаксис имен файлов зависит от типа компьютера. Некоторые примеры имен файлов рассмотрены ниже:

- `data.dat` — на любом компьютере, это имя файла в текущем каталоге;
- `/home/eck/java/examples/data.dat` — абсолютный путь на системе UNIX;
- `C:\eck\java\examples\data.dat` — абсолютный путь для DOS и Windows;
- `Hard Drive:java:examples:data.dat` — для системы Macintosh OS 9 (`Hard Drive` — имя устройства);
- `examples/data.dat` — относительный путь на UNIX;
- `examples\data.dat` — относительный путь для DOS и Windows;
- `examples:data.dat` — относительный путь для Macintosh.

Чтобы хотя бы в этом избежать проблем, вызванных различиями в платформах, в языке Java используется класс `java.io.File`. Объект этого класса представляет файл, точнее не сам файл, а имя файла. Сам файл может либо существовать, либо не существовать. Каталоги воспринимаются точно так же, как и файлы. Объект типа `File` с равным успехом может представлять как файл, так и каталог.

Объект `File` создается при помощи конструктора `File(String)`, которому в качестве параметра передается путь к файлу с именем файла. Имя может быть указано как в абсолютной, так и в относительной форме. Например, `new File("data.dat")` создает объект типа `File`, который ссылается на файл data.dat, расположенный в текущем каталоге. Существует другой тип конструктора для создания объекта типа `File`, `new File(File, String)`, в этом конструкторе указывается два параметра. Первый — объект типа `File`, который ссылается на каталог, в котором расположен файл, имя файла (простое) указывается во втором аргументе.

Объект типа `File` обладает набором полезных методов. Пусть `file` — это объект типа `File`, перечислим некоторые полезные методы при работе с объектом типа `File`.

- `file.exists()` — логическое значение. Возвращает `true`, если файл с установленным именем файла существует. Этот метод полезно использовать в том случае, если мы желаем избежать возможной потери данных, записанных в уже существующем файле перед использованием `FileWriter`.
- `file.isDirectory()` — логическое значение. Возвращает `true`, если объект указывает на каталог. Если объект указывает на обычный файл или если каталог с заданным именем не существует, возвращает `false`.

- `file.delete()` — удаляет файл, если файл существует.
- `file.list()` — если объект указывает на каталог, то функция возвращает массив типа `String[]`, в котором содержатся имена файлов в данном каталоге. В противном случае возвращает `null`.

Ниже приведен пример программы (листинг 2.3), которая возвращает имена всех файлов, расположенных в указанном пользователем каталоге (рис. 2.1):

Листинг 2.3. Программа, выводящая на экран список файлов в каталоге

```
import java.io.File;
public class DirectoryList {
  public static void main(String[] args) {
    String directoryName;  // имя каталога, вводимое пользователем
    File directory;        // объект типа File, указывающий на каталог
    String[] files;        // массив имен файлов в каталоге
    TextIO.put("vvedite imya kataloga: ");
    directoryName = TextIO.getln().trim();
    directory = new File(directoryName);
    if (directory.isDirectory() == false) {
      if (directory.exists() == false)
        TextIO.putln("Takogo kataloga net!");
      else
        TextIO.putln("Eto ne katalog.");
    }
    else {
      files = directory.list();
      TextIO.putln("Spisok failov v kataloge \"" + directory + "\":");
      for (int i = 0; i < files.length; i++)
        TextIO.putln("   " + files[i]);
    }
  }
}
```

Все классы, которые читают и пишут данные из файла или в файл, получают в качестве параметра объект типа `File`. Например, если `file` — это переменная типа `File`, то мы можем создать поток `FileReader`, указав `new FileReader (file)`. Если используется `TextReader` для чтения из файла, то можно применить следующую конструкцию:

```
TextReader data;
try {
```

```
  data = new TextReader(new FileReader(file));
}
catch (FileNotFoundException e) {
  ... // обработка ошибок
}
```

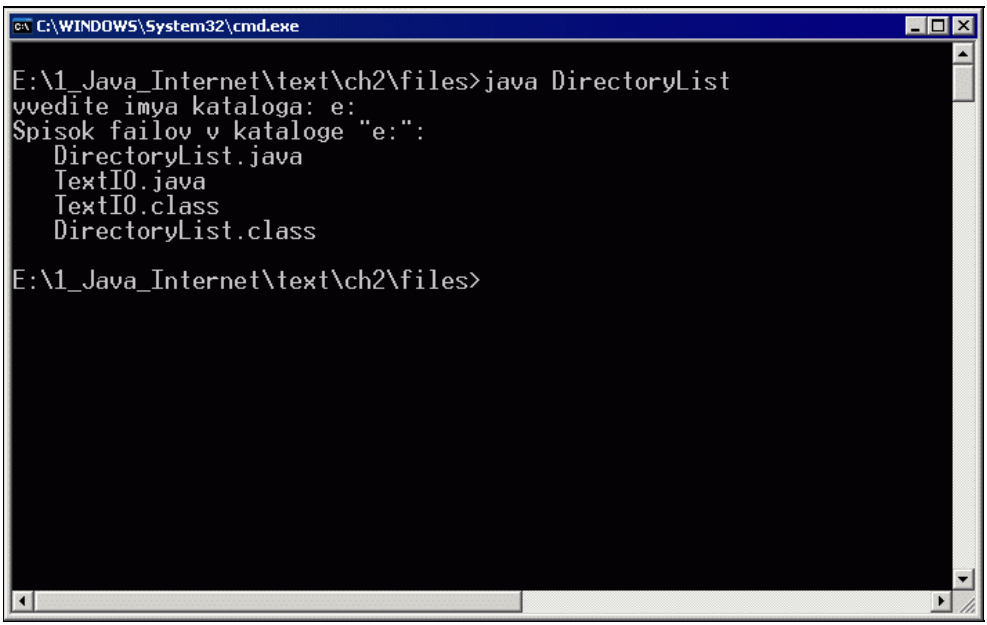


Рис. 2.1. Программа выводит список файлов в каталоге

Перед тем как перейти к рассмотрению способов работы с сетью, которые во многом схожи с тем, как организована работа с файлами, рассмотрим несколько примеров, посвященных работе с файлами.

Следующий пример создает список всех слов, встречающихся в заданном файле. Под словом понимается последовательность букв. Пользователь должен ввести имя файла. Список слов выводится в другой файл, который также задается пользователем. Список слов будет выведен в алфавитном порядке, причем слова не будут повторяться. Все буквы в словах будут приведены к нижнему регистру. Например, слова "The" и "the" будут восприниматься как одно и то же слово.

Поскольку мы хотим вывести слова в алфавитном порядке, то мы не имеем возможности записывать каждое слово сразу после того, как оно будет прочитано. Мы можем сохранять слова в массиве, но, поскольку мы не знаем, сколько у нас будет слов в читаемом файле, то нам необходим "динамический массив", который может увеличиваться настолько, насколько

нам будет необходимо. Программа содержится в файле WordList.java (листинг 2.4).
В программе использован класс `TextIO`, полностью файл TextIO.java приводится в приложении 4.

Листинг 2.4. Файл WordList.java

```
import java.io.*;
public class WordList {
   static String[] words;  // массив, хранящий слова
   static int wordCount;   // текущее число слов, хранимых в массиве
   public static void main(String[] args) {
      TextReader in;    // поток для чтения
      PrintWriter out;  // поток для записи
      String inputFileName;   // имя исходного файла
      String outputFileName;  // имя файла для записи
      words = new String[10];
      wordCount = 0;
      /*
      Получаем имя исходного файла от пользователя и создаем
      поток ввода. Если возникает FileNotFoundException, то печатаем
      сообщение и прекращаем выполнение программы.
      */
      TextIO.put("Input file name?  ");
      inputFileName = TextIO.getln().trim();
      try {
        in = new TextReader(new FileReader(inputFileName));
      }
      catch (FileNotFoundException e) {
        TextIO.putln("Fail ne suschestvuet: \"" + inputFileName + "\".");
        return;
      }
      /*
      Получаем от пользователя имя файла для записи слов. При
      возникновении IOException печатаем сообщение и прекращаем
      выполнение программы.
      */
      TextIO.put("Output file name? ");
      outputFileName = TextIO.getln().trim();
      try {
        out = new PrintWriter(new FileWriter(outputFileName));
```

```
}
catch (IOException e) {
  TextIO.putln("Can't open file \"" + outputFileName +
               "\" for output.");
  TextIO.putln(e.toString());
  return;
}
/*
Чтение слов из потока ввода и вставка в массив слов. Чтение
может привести к возникновению ошибки TextReader.Error. Тогда
печатаем сообщение и прекращаем выполнение программы.
*/
try {
  while (true) {
    // Пропускаем небуквенные символы. Читаем слово и вставляем
    // его в массив.
    while (! in.eof() && ! Character.isLetter(in.peek()))
      in.getAnyChar();
    if (in.eof())
      break;
    insertWord(in.getAlpha());
  }
}
catch (TextReader.Error e) {
  TextIO.putln("Oshibka pri chetenii faila.");
  TextIO.putln(e.toString());
  return;
}
// пишем все слова в поток вывода
for (int i = 0; i < wordCount; i++)
  out.println(words[i]);
/*
Завершаем. Проверяем наличие ошибок при записи.
Выводим на экран либо предупреждение, либо сообщение о количестве
записанных слов.
*/
if (out.checkError() == true) {
  TextIO.putln("Some error occured while writing output.");
  TextIO.putln("Output might be incomplete or invalid.");
}
else {
```

```
        TextIO.putln(wordCount + " words from \"" + inputFileName +
                     "\" output to \"" + outputFileName + "\".");
      }
     }
static void insertWord(String w) {
  // Вставляем слово w в массив слов, если его там еще нет. Все
  // слова записаны строчными буквами. Слова в массиве хранятся
  // в алфавитном порядке. Если массива по длине становится
  // недостаточно, то его размер удваивается.
  int pos = 0;  // место в массиве для хранения слова w
  w = w.toLowerCase();
  // определение места в массиве для хранения слова w
  while (pos < wordCount && words[pos].compareTo(w) < 0)
    pos++;
  if (pos < wordCount && words[pos].equals(w))
    return;
  /*
  Если массив полон, то создаем новый массив, который в 2 раза
  больше старого. Копируем все слова в новый массив.
  */
  if (wordCount == words.length) {
    String[] newWords = new String[words.length*2];
    System.arraycopy(words,0,newWords,0,wordCount);
    words = newWords;
  }
  /*
  Располагаем слово w в нужное место в массиве. Все слова после
  него сдвигаем на одну позицию далее.
  */
  for (int i = wordCount; i > pos; i--)
    words[i] = words[i-1];
    words[pos] = w;
    wordCount++;
   }
}
```

Предположим, следующий текст сохранен в файле input.txt:

```
Виктория, поздравляю тебя с днем рождения!
Разреши пожелать тебе всего самого хорошего.
```

Запускаем программу, вводим запрашиваемые имена (рис. 2.2). В результирующем файле output.txt получаем список слов:

```
виктория
```

```
всего
днем
пожелать
поздравляю
разреши
рождения
с
самого
тебе
тебя
хорошего
```

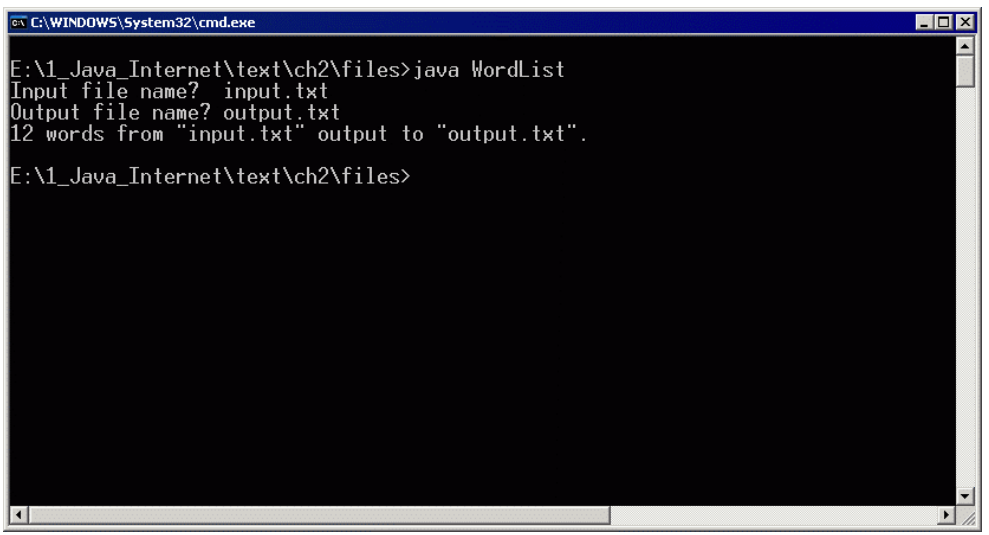


Рис. 2.2. Пример организации работы с файлами

В качестве следующего примера рассмотрим программу, которая производит копирование файла. Это полезно еще и потому, что многие операции с файлами похожи на копирование файлов, с той разницей, что файл не просто копируется, но производятся некоторые действия с данными, хранящимися в исходном файле. Поскольку программа должна уметь копировать любой файл, мы не можем заведомо предположить, что файл содержит данные, предназначенные для чтения человеком. Поэтому мы будем использовать потоки `InputStream` и `OutputStream`, а не потоки `Reader` и `Writer`. Программа попросту копирует все данные, получаемые из потока `InputStream`, записывая их в поток `OutputStream` байт за байтом. Если переменная `source` ссылается на поток `InputStream`, то функция `source.read()` читает из потока байт за байтом. Функция возвращает `-1`, когда все байты прочитаны. Аналогично, если `copy` — это поток типа

OutputStream, то метод copy.write(b) пишет один байт в соответствующий файл вывода. Поэтому основу программы составляет простой цикл while.

```
while(true) {
   int data = source.read();
   if (data < 0)
     break;
   copy.write(data);
}
```

В системах DOS и UNIX для копирования файлов используется команда, записываемая в командной строке как "copy original.dat backup.dat", при помощи этой команды можно скопировать существующий файл original.dat в новый файл с именем backup.dat. Аргументы, используемые в командной строке, можно применять и в программе, написанной на языке Java. Аргументы командной строки хранятся в строковом массиве args (листинг 2.5), который является аргументом функции main(). Если, например, программа с именем CopyFile использует аргументы, то команда "java CopyFile work.dat oldwork.dat" приведет к тому, что элемент args[0] будет содержать строку "work.dat", args[1] будет содержать "oldwork.dat". Значение args.length сообщает длину массива аргументов.

Программа CopyFile получает имена файлов из командной строки. Программа выводит сообщение об ошибке в том случае, если имена файлов не заданы. Чтобы сделать программу несколько более интересной, она построена таким образом, что может быть вызвана либо с двумя, либо с тремя аргументами. В первом случае программа в качестве аргументов получает имя файла-источника и имя файла, куда будет скопирован исходный файл. Если последний файл уже существует, то программа выдаст сообщение об ошибке и не будет копировать файл, чтобы не потерять данные, хранящиеся во втором файле. В случае с тремя аргументами в качестве первого аргумента следует задать модификатор -f. Его появление изменяет поведение программы, он сообщает программе о том, что если файл, куда будет производиться копирование, уже существует, то в него разрешается запись.

Листинг 2.5. Файл CopyFile.java

```
import java.io.*;
  public class CopyFile {
    public static void main(String[] args) {
      String sourceName;   // имя исходного файла
                           // указывается в командной строке
      String copyName;     // имя файла для копирования
                           //  указывается в командной строке
      InputStream source;  // поток чтения файла
      OutputStream copy;   // поток записи
```

```
boolean force;       // если задана опция -f, то значением
                     // будет true
int byteCount;  // количество байт, скопированных из файла
/*
Получаем имя файла из командной строки и проверяем наличие
модификатора -f.  Если командная строка имеет вид, отличающийся
от двух заданных типов, то выводим сообщение об ошибке и прекращаем
выполнение программы. */
if (args.length == 3 && args[0].equalsIgnoreCase("-f")) {
  sourceName = args[1];
  copyName = args[2];
  force = true;
}
else
  if (args.length == 2) {
    sourceName = args[0];
    copyName = args[1];
    force = false;
  }
  else {
    System.out.println(
    "Pravilno pisat tak: java CopyFile <source-file> <copy-name>");
    System.out.println(
    " ili  java CopyFile -f <source-file> <copy-name>");
    return;
   }
   /*
   создаем поток ввода, если появляется ошибка, останавливаем
   выполнение программы
   */
   try {
     source = new FileInputStream(sourceName);
   }
   catch (FileNotFoundException e) {
     System.out.println("Can't find file \"" + sourceName+ "\".");
     return;
   }
   /*
   если файл вывода уже существует и опция -f не задана, то
   выводим сообщение об ошибке и прекращаем выполнение программы
   */
   File file = new File(copyName);
   if (file.exists() && force == false) {
```

```
          System.out.println("Fail uzhe suschestvuet. Ispolzujte -f dlya
perezapisi fajla.");
          return;
        }
        /*
        Создаем поток вывода. При возникновении ошибки пишем
        сообщение и прекращаем выполнение программы. */
        try {
          copy = new FileOutputStream(copyName);
        }
        catch (IOException e) {
          System.out.println("Can't open output file \""
                             + copyName + "\".");
          return;
        }
        /*
        Копируем по одному байту раз за разом до тех пор, пока метод
        read() не возвратит -1 (достигнут конец потока ввода). При
        возникновении ошибки выводим сообщение об ошибке. При удачном
        копировании выводим сообщение о том, что копирование прошло
        успешно. */
        byteCount = 0;
        try {
          while (true) {
            int data = source.read();
            if (data < 0)
              break;
            copy.write(data);
            byteCount++;
          }
          source.close();
          copy.close();
          System.out.println("Uspeshno skopirovano "
                             + byteCount + " bait.");
        }
        catch (Exception e) {
          System.out.println("Vo vremya kopirovaniya proizoshla oshibka.
                          Vsego skopirovano  "  + byteCount + " bait.");
          System.out.println(e.toString());
        }
      }
    }
```

В качестве заключительного примера работы с файлами приведем программу, которая использует графический интерфейс пользователя для чтения и записи файла (листинг 2.6). Программа, как это обычно бывает, содержит пункт меню **Open**. При переходе по нему пользователю будет предложено выбрать открываемый файл. Сохранение файла происходит при обращении к пункту меню **Save**. Окно программы содержит в себе в качестве основного компонента текстовую область `JTextArea`, которая может содержать некоторый текст и использоваться для редактирования текста. В верхней части располагается панель меню, в которой находится меню **File**, содержащее команды **Open** и **Save**. При нажатии кнопки **Save** в меню **File** появляется диалоговое окно, где пользователю предлагается выбрать имя файла, куда будет записан текст, расположенный в окне редактирования текста. Имя файла можно ввести вручную (рис. 2.3).

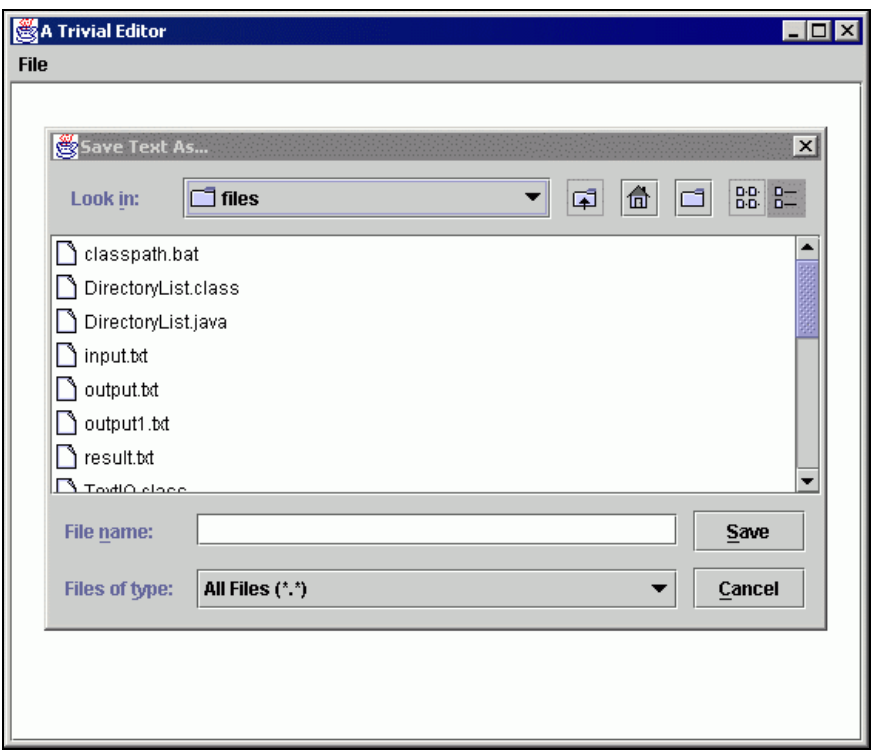


Рис. 2.3. Диалоговое окно выбора файла

Для открытия файла также предлагается диалоговое окно, вызываемое командой **Open** меню **File** (рис. 2.4).

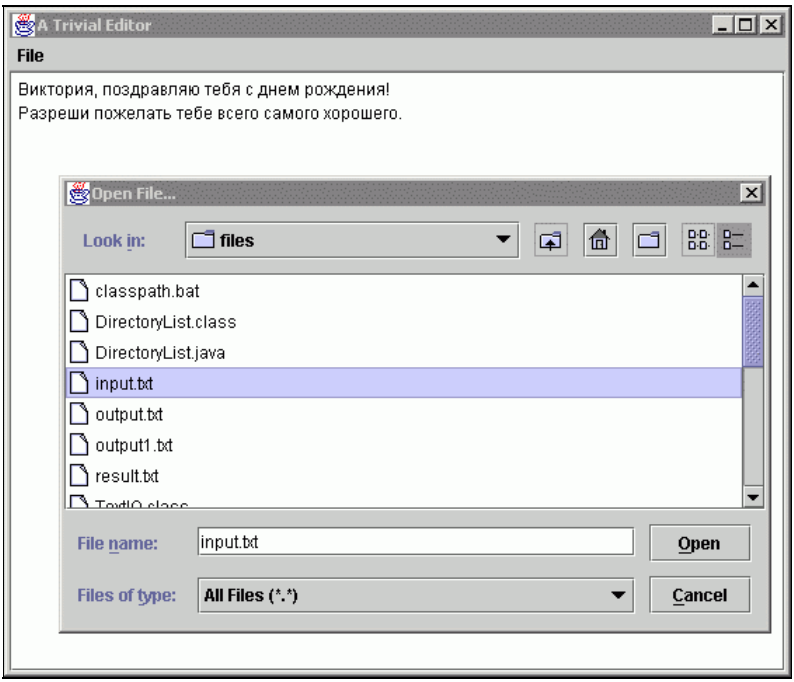


Рис. 2.4. Диалоговое окно открытия файла

Листинг 2.6. Файл TrivialEdit.java

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TrivialEdit extends JFrame {
  public static void main(String[] args) {
  // основная программа, открывающая окно
    new TrivialEdit();
  }
  private JTextArea text;   // окно редактирования текста
  public TrivialEdit() {
    // Вставляет панель меню. Первая строка вызывает конструктор
    // суперкласса для создания заголовка окна. Команда pack()
    // задает размер окна.
    super("A Trivial Editor");
    setJMenuBar(makeMenus());
    text = new JTextArea(25,50);
    text.setBackground(Color.white);
    text.setMargin(new Insets(3,5,0,0));
```

```
    JScrollPane scroller = new JScrollPane(text);
    setContentPane(scroller);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    pack();
    setLocation(50,50);
    show();
}
private JMenuBar makeMenus() {
  // Создает панель меню с одним разделом меню File. Этот раздел содержит
  // четыре команды. Каждой команде соответствует эквивалент,
  // вызываемый при помощи клавиатуры ("горячие клавиши" - hot keys).
  ActionListener listener = new ActionListener() {
    // Объект, который служит прослушивателем для элементов меню.
    public void actionPerformed(ActionEvent evt) {
      // Метод вызывается, когда пользователь производит
      // выбор в меню File. Функция проверяет, какой выбор
      // сделан, и вызывает другую функцию для выполнения
      // выбранной команды.
      String cmd = evt.getActionCommand();
      if (cmd.equals("New"))
        doNew();
      else
        if (cmd.equals("Open..."))
          doOpen();
        else
          if (cmd.equals("Save..."))
            doSave();
          else
            if (cmd.equals("Quit"))
              doQuit();
    }
  };
  JMenu fileMenu = new JMenu("File");
  JMenuItem newCmd = new JMenuItem("New");
  newCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl N"));
  newCmd.addActionListener(listener);
  fileMenu.add(newCmd);
  JMenuItem openCmd = new JMenuItem("Open...");
  openCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
  openCmd.addActionListener(listener);
  fileMenu.add(openCmd);
  JMenuItem saveCmd = new JMenuItem("Save...");
  saveCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));
  saveCmd.addActionListener(listener);
  fileMenu.add(saveCmd);
  JMenuItem quitCmd = new JMenuItem("Quit");
```

```
  quitCmd.setAccelerator(KeyStroke.getKeyStroke("ctrl Q"));
  quitCmd.addActionListener(listener);
  fileMenu.add(quitCmd);
  JMenuBar bar = new JMenuBar();
  bar.add(fileMenu);
  return bar;
}
private void doNew() {
  // Создание нового файла.
  // Очистка текстовой области JTextArea.
  text.setText("");
}
private void doSave() {
  // Выполнение команды Save. Выбор файла по указанию
  // пользователя. Запись текста из текстовой
  // области JTextArea в указанный файл.
  File file;  // файл, в который будет производиться запись
  JFileChooser fd; // диалоговое окно выбора файла
  fd = new JFileChooser(new File("."));
  fd.setDialogTitle("Save Text As...");
  int action = fd.showSaveDialog(this);
  if (action != JFileChooser.APPROVE_OPTION) {
    // отмена или появилась ошибка
    return;
  }
  file = fd.getSelectedFile();
  if (file.exists()) {
    // если файл уже существует, то перед перезаписью выводим вопрос
    action = JOptionPane.showConfirmDialog(this,
             "Perepisat suschestvuyuschij fail?");
    if (action != JOptionPane.YES_OPTION)
      return;
  }
  try {
    // создаем PrintWriter для записи в указанный файл и пишем
    // текст из окна в созданный поток
    PrintWriter out = new PrintWriter(new FileWriter(file));
    String contents = text.getText();
    out.print(contents);
    if (out.checkError())
      throw new IOException("Oshibka zapisi v fail.");
    out.close();
  }
  catch (IOException e) {
    // Voznikla oshibka pri zapisi f fail.
    // Vyvodim soobscheinie ob oshibke.
```

```
    JOptionPane.showMessageDialog(this,
    "Proizoshla oshibka:\n" + e.getMessage());
  }
}
private void doOpen() {
  // Открытие файла по указанию пользователя.
  // Выбор производится в диалоговом окне. Содержимое файла
  // заменяет текст, расположенный в текстовом окне JTextArea.
  File file;  // файл, который пользователь выбрал для открывания
  JFileChooser fd; // диалоговое окно выбора файла
  fd = new JFileChooser(new File("."));
  fd.setDialogTitle("Open F7ile...");
  int action = fd.showOpenDialog(this);
  if (action != JFileChooser.APPROVE_OPTION) {
    // отмена или ошибка
    return;
  }
  file = fd.getSelectedFile();
  try {
    // Чтение строк файла до достижения конца файла. Строки
    // вставляются в текстовую область. После каждой строки
    // вставляется символ перевода строки.
    TextReader in = new TextReader(new FileReader(file));
    String line;
    text.setText("");
    int lineCt = 0;
    while (lineCt < 100 && in.peek() != '\0') {
    line = in.getln();
    text.append(line + '\n');
    lineCt++;
  }
  if (in.eof() == false)
    text.append("\n\n******* Text sokraschen do 100 strok! *******\n");
    in.close();
  }
  catch (Exception e) {
    // ошибка при чтении файла
    JOptionPane.showMessageDialog(this,
    "Sorry, some error occurred:\n" + e.getMessage());
  }}
private void doQuit() {
  // выполняется команда выхода Quit
  System.exit(0);
}
}
```

2.4. Работа с сетью

Работа с сетевыми потоками несколько отличается от работы с потоками чтения и записи данных в файлы. Однако общие принципы остаются прежними. Сложности возникают с тем, что при работе с сетью происходит взаимодействие двух компьютеров.

При работе с сетью используется стандартный пакет Java, который называется java.net. Он включает в себя несколько классов, которые используются для работы с сетью. При этом можно воспользоваться как высокоуровневым способом взаимодействия с сетью, так и низкоуровневыми приемами создания и работы с сокетами. Сокеты предоставляют программный интерфейс, с помощью которого оказывается возможным сетевое взаимодействие между самостоятельными программами. Высокоуровневое программирование предполагает взаимодействие с ресурсами всемирной паутины WWW, при этом создаваемое приложение приобретает черты Web-браузера, основная задача которого — получение HTML-документа. Этот подход основан на использовании классов `java.net.URL` и `java.net.URLConnection`. Объект URL — это абстрактное представление универсального локатора ресурса (Universal Resource Locator), который представляет собой адрес HTML-документа или другого ресурса в Интернете. Объект `URLConnection` представляет собой подключение к этому ресурсу.

Кроме высокоуровневого программирования при работе с сетью разработчик имеет возможность работать с сокетами, которые общаются друг с другом, передавая информацию по протоколу TCP. Для этого в языке Java существует класс `java.net.Socket`. Сокет используется в программе для установления связи с другой программой. Связь с использованием сокетов предполагает наличие по крайней мере двух сокетов. Сокет должен присутствовать на двух (или более) программах, которые осуществляют взаимодействие друг с другом.

2.4.1. Классы *URL* и *URLConnection*

Любой ресурс Интернета имеет адрес, который сокращенно называется URL. Объект класса `URL` представляет собой такой адрес. Если мы имеем объект `URL`, то мы можем установить связь с ресурсом по указанному адресу. Для установления связи используется объект `URLConnection`. URL представляет собой строку, например, **`http://e-olymp.com/url.php`**. Существуют также относительные адреса. Относительный адрес указывается по отношению к каталогу, который задан в качестве базового (base). Объект типа `URL` — это не просто строка, он может быть создан на основе строкового представления URL. Объект `URL` может также быть создан на основе другого

объекта типа URL, который определяет базовый URL. Тогда нужно задать только простой URL. Конструкторы выглядят следующим образом:

```
public URL(String urlName) throws MalformedURLException
public URL(URL context, String relativeName) throws MalformedURLException
```

При задании URL может возникнуть ошибка MalformedURLException. Это означает, что адрес указан в неверной форме. Класс MalformedURLException является подклассом класса IOException, для него требуется обязательная обработка ошибок, поэтому конструктор следует вызывать в блоке try...catch. Второй конструктор особенно удобен при создании апплетов. При работе с апплетами полезен метод getDocumentBase(), который определен в классе Applet. Этот метод возвращает объект типа URL, представляющий адрес базы, то есть адрес HTML-страницы, из которой был загружен апплет. Зная этот адрес, мы можем загрузить другие страницы, расположенные по соседству с исходной страницей, например

```
URL url = new URL(getDocumentBase(), "data.txt");
```

Этот конструктор создает объект URL, ссылающийся на файл data.txt, расположенный на том же компьютере и в том же каталоге, что и исходный файл (файл страницы, на которой работает апплет). Другой полезный метод — это метод getCodeBase(). С его помощью мы получаем местоположение файла класса апплета (он может располагаться в другом месте, которое не совпадает с положением страницы, использующей апплет).

После того как мы располагаем объектом типа URL, мы имеем возможность вызвать его метод openConnection() и установить с его помощью связь. Метод возвращает ссылку на объект URLConnection. Этот объект может быть в свою очередь использован для создания потока InputStream, который осуществляет чтение данных из указанного URL. Это делается при помощи метода getInputStream(), например:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

Методы openConnection() и getInputStream() могут приводить к возникновению ошибок типа IOException. После создания потока InputStream, чтение из него производится обычным способом. В том числе можно читать с использованием других объектов, например, объекта TextReader. Конечно, чтение из потока также может приводить к появлению ошибок.

Еще один очень полезный метод класса URLConnection — это метод getContentType(). Он возвращает строку, которая содержит информацию о типе данных, находящихся по указанному адресу. Возвращаемое значение может быть null, если тип либо неизвестен, либо тип не удалось определить. Тип данных не может быть определен до того, как будет создан поток, поэтому метод getContentType() следуеть использовать после метода

`getInputStream()`. Строка, которую возвратит `getContentType()` — это формальное название mime-типа данных, например `text/plain`, `text/html`, `image/jpeg`, `image/gif` и т. п. Mime-тип состоит из двух частей. Общая часть, например, `text` или `image` и более детализированная часть, например, `html` или `gif`. Если нас интересуют только текстовые данные, то мы можем проверить, вернет ли метод `getContentType()` строку, начинающуюся словом `text`. Mime-типы впервые были использованы в целях идентификации частей e-mail-сообщений. Аббревиатура mime соответствует словам Multipurpose Internet Mail Extensions (многофункциональные расширения почтовых приложений). Сейчас типы mime используются довольно широко, с их помощью можно идентифицировать практически любое содержимое файла.

Рассмотрим простой пример, в котором происходит чтение данных по указанному URL. Функция открывает соединение с заданным URL, проверяет тип данных и копирует данные на экран:

```
static void readTextFromURL(String urlString) throws Exception {
  // попытка копировать текст на экран с использованием заданного
  // URL
  URL url = new URL(urlString);
  URLConnection connection = url.openConnection();
  InputStream urlData = connection.getInputStream();
  // проверка типа
  String contentType = connection.getContentType();
  if (contentType == null || contentType.startsWith("text") == false)
    throw new Exception("URL does not refer to a text file.");
  /*
  Копирование символов потока на экран до достижения конца потока или
  ошибки.
  */
  while (true) {
    int data = urlData.read();
    if (data < 0)
      break;
    System.out.print((char)data);
  }
}
```

Файл ReadURL.java (листинг 2.7) содержит программу, которая осуществляет чтение текста по заданному адресу (рис. 2.5).

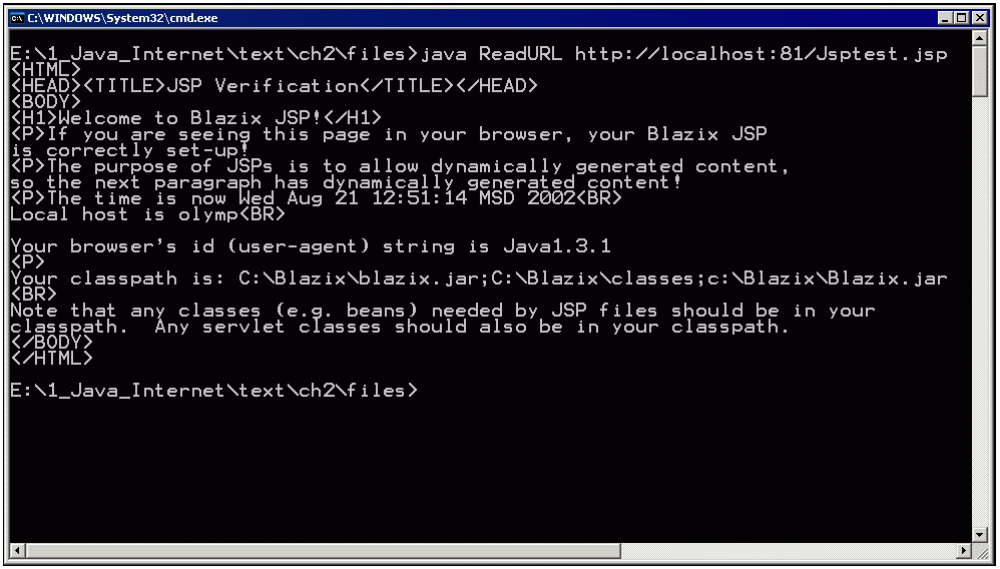


Рис. 2.5. Чтение по указанному в командной строке URL

Листинг 2.7. Файл ReadURL.java

```
import java.net.*;
import java.io.*;
public class ReadURL {
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("Please, specify a URL on the command line.");
      return;
    }
    try {
      readTextFromURL(args[0]);
    }
    catch (Exception e) {
      System.out.println("\n*** Sorry, an error has occurred ***\n");
      System.out.println(e);
    }
  }
  static void readTextFromURL(String urlString)
  throws Exception {
    URL url = new URL(urlString);
```

```
    URLConnection connection = url.openConnection();
    InputStream urlData = connection.getInputStream();
    String contentType = connection.getContentType();
    if (contentType == null || contentType.startsWith("text") == false)
      throw new Exception("URL does not refer to a text file.");
      while (true) {
        int data = urlData.read();
        if (data < 0)
          break;
        System.out.print((char)data);
      }
  }
}
```

2.4.2. Сокеты. Клиенты. Серверы

Связь в сети Интернет осуществляется на основе двух протоколов — TCP (Transmission Control Protocol) и IP (Internet Protocol), оба протокола объединяются в стек-протокол — TCP/IP. Помимо этого часто используется протокол UDP (User Datagram Protocol — протокол пользовательских датаграмм). Если две программы должны общаться друг с другом посредством протокола TCP/IP, то в каждой программе должен быть создан сокет. Затем сокеты соединяются друг с другом, устанавливая связь между программами. После того как связь будет создана, общение происходит на основе потоков ввода и вывода. Данные, записанные программой в поток вывода, передаются другой программе, которая может быть расположена на другом компьютере. При чтении программы данных из потока, эти данные получаются потоком из другой программы.

Создание связи на основе сокетов происходит по следующей схеме. Одна программа создает сокет, который пассивно слушает, ожидая запросы от другого сокета. Другая программа создает сокет, который посылает запрос на соединение прослушивающему сокету. По получении запроса прослушивающий порт отвечает и устанавливается соединение. После этого программы могут обмениваться данными, то есть записывать в поток и читать из потока. Это может происходить до тех пор, пока одна из программ не закроет связь.

Программа, которая создает прослушивающий сокет, часто называется *сервером*, а соответвующий сокет называется *серверным сокетом*. Программа, которая подключается к серверу, создавая запросы, называется *клиентом*, она работает с *клиентским сокетом*. Сервер, как правило, предлагает тот или иной сервис, а клиент обращается к серверу за услугами, предоставляемыми этим сервисом. Такая модель носит название *клиент-серверной модели связи*. Когда клиент подсоединяется к серверу, прослушивающий сокет не прекращает ожидание запросов от клиентов. Сервер продолжает прослуши-

вать возможные запросы от других клиентов. Чтобы была возможность осуществлять одновременную работу с несколькими клиентами, необходимо использовать потоки.

Класс `URL` использует сокеты, но сокеты при этом скрыты. При работе с классом `URL` на другой стороне взаимодействия сокетов располагается серверная программа, как правило, Web-сервер, который обрабатывает запросы и посылает документ, расположенный по указанному URL (если документ существует). По окончании пересылки данных соединение закрывается.

Для работы с TCP/IP-соединениями в Java существуют классы `ServerSocket` и `Socket`. Класс `ServerSocket` — это серверный сокет, ожидающий запросы на соединение от клиентов. Класс `Socket` используется для создания клиентских сокетов, запрашивающих сервер. Класс `Socket` может быть использован также и сервером для обработки запросов на соединение от клиента. Это позволяет серверу работать с несколькими сокетами, создавая несколько соединений одновременно. Объект типа `ServerSocket` сам по себе не участвует в соединении, он только прослушивает, ожидая запрос на соединение, затем он создает `Socket`, который используется для работы соединения как такового.

Тот и другой вид сокетов требует задания адресов в сети Интернет. Каждый компьютер имеет IP-адрес. Многие компьютеры могут быть опознаны по доменному имени, например, **www.yahoo.com**. На каждом компьютере может быть установлено несколько программ. Чтобы иметь возможность выбрать конкретную программу, используются дополнительные адреса — порты. Номер порта — это 16-битное целое число. Сервер не просто прослушивает в ожидании запроса на установление связи, он прослушивает конкретный порт. При обращении к серверу клиент должен знать Интернет-адрес (или доменное имя) и номер порта. Стандартные сервера работают по стандартным протоколам. С ними, как правило, связаны конкретные порты. Так, обычные Web-сервера прослушивают порт 80. Все стандартные номера портов являются числами, меньшими 1024. Для детального знакомства с протоколами TCP и UDP можно обратиться к документам RFC768, RFC793. В таблицах 2.1 и 2.2 приведен список наиболее употребительных стандартных номеров портов для протоколов TCP и UDP и связанных с ними протоколов, которые работают поверх протоколов TCP и UDP.

Таблица 2.1. *Основные группы портов для сокетов*

Номер порта десятичный	Номер порта восьмеричный	Описание
0—63	0—77	Общесетевой стандарт
64—131	100—203	Специальные функции отдельных хостов
132—223	204—337	Зарезервированы
224—255	340—377	Для экспериментального использования

Таблица 2.2. *Наиболее употребительные порты TCP и UDP (по классификации 1980 года)*

Десятичный порт	Восьмеричный порт	Описание
3	3	Для передачи файлов
13	15	Дата и время
15	17	Whois
17	21	Короткие текстовые сообщения
19	23	Генератор символов
21	25	Новая передача файлов ftp
23	27	Новый Telnet
25	31	SMTP
27	33	NSW User System w/COMPASS
29	35	MSG-3 ICP
31	37	MSG-3 Authentication
37	45	Time Server
41	51	Graphics
42	52	Name Server
43	53	Whols

В таблице мы не увидим многих привычных сегодня протоколов. Это и неудивительно, в начале 80-х годов XX века многие из них еще только зарождались. Более подробный и полный список протоколов и стандартных номеров портов приведен в таблице 2.3. Номер порта указан в начале каждой строки, за номером порта следует протокол, который используется для работы на этом порте (TCP и/или UDP), затем идет назначение порта.

Таблица 2.3. *Некоторые порты для протоколов tcp и udp и связанные с ними протоколы и службы*

Порт/протоколы/описание	Порт/протоколы/описание
❒ 0/tcp/udp Reserved	❒ 11/tcp/udp Active Users
❒ 1/tcp TCP Port Service Multiplexer	❒ 13/tcp/udp Daytime
❒ 2/tcp Management Utility	❒ 17/tcp/udp Quote of the Day
❒ 3/tcp Compression Process	❒ 18/tcp/udp RWP rwrite
❒ 5/tcp Remote Job Entry	❒ 18/tcp/udp Message Send Protocol
❒ 7/tcp/udp Echo	❒ 19/tcp/udp Character Generator

Таблица 2.3 *(окончание)*

Порт/протоколы/описание	Порт/протоколы/описание
❒ 20/tcp File Transfer [Default Data]	❒ 433/tcp/udp NNSP
❒ 21/tcp File Transfer [Control]	❒ 433/udp NNSP
❒ 23/tcp Telnet	❒ 434/tcp MobileIP-Agent
❒ 24/tcp/udp any private mail system	❒ 435/tcp MobillP-MN
❒ 25/tcp Simple Mail Transfer	❒ 439/tcp/udp dasp Thomas Obermair
❒ 37/tcp/udp Time	❒ 453/tcp/udp CreativeServer
❒ 38/tcp/udp Route Access Protocol	❒ 454/tcp/udp ContentServer
❒ 39/udp Resource Location Protocol	❒ 455/tcp/udp CreativePartnr
❒ 41/tcp/udp Graphics	❒ 512/tcp/udp remote process execution
❒ 42/udp Host Name Server	❒ 513/tcp/udp remote login a la telnet
❒ 43/tcp Who Is	❒ 514/tcp/udp like exec, but automatic
❒ 53/tcp/udp Domain Name Server	❒ 515/tcp spooler
❒ 70/tcp Gopher	❒ 517/udp talk
❒ 79/tcp Finger	❒ 518/tcp ntalk
❒ 80/tcp World Wide Web HTTP	❒ 519/tcp/udp unixtime
❒ 106/tcp Password Server	❒ 520/tcp/udp extended file name server
❒ 107/tcp Remote Telnet Service	❒ 525/tcp/udp timeserver
❒ 117/tcp UUCP Path Service	❒ 526/tcp/udp newdate
❒ 118/tcp/udp SQL Services	❒ 530/tcp/udp rpc
❒ 150/tcp/udp SQL-NET	❒ 531/tcp/udp chat
❒ 156/tcp SQL Service	❒ 532/tcp/udp readnews
❒ 396/tcp/udp Novell Netware over IP	❒ 532/udp readnews
❒ 397/tcp/udp Multi Protocol Trans. Net	❒ 533/tcp/udp for emergency broadcasts
❒ 414/tcp/udp InfoSeek	❒ 540/tcp uucpd
❒ 420/udp SMPTE	❒ 541/tcp/udp uucp-rlogin
❒ 425/tcp ICAD	❒ 565/tcp/udp whoami
❒ 426/tcp/udp smartsdp	❒ 751/tcp/udp pump
❒ 427/tcp/udp Server Location	

Во время создания сокета следует задать номер порта, который будет прослушивать сервер. Конструктор выглядит следующим образом:

```
public ServerSocket(int port) throws IOException
```

После того как создан сокет `ServerSocket`, он приступает к прослушиванию и ожиданию запросов на соединение. Метод `accept()` класса `ServerSocket` принимает запрос на соединение, образует соединение с клиентом и возвращает сокет, который будет использоваться для коммуникации с клиентом. Метод `accept()` выглядит следующим образом:

```
public Socket accept() throws IOException
```

После вызова метода `accept()` от него не будет никакого отклика до тех пор, пока не будет получен запрос на соединение или пока не возникнет какая-либо ошибка. В то время, пока метод блокирован, поток, вызвавший метод, не может продолжать работу. Другие же потоки продолжают выполняться независимо. Объект `ServerSocket` будет прослушивать до тех пор, пока не будет вызван метод `close()` (или если произойдет ошибка). В качестве небольшого примера запишем фрагмент, в котором сервер прослушивает порт 1728, а метод `provideService (Socket)` служит для осуществления коммуникации с одним клиентом. Базовый вариант серверной программы выглядит следующим образом:

```
try {
  ServerSocket server = new ServerSocket(1728);
  while (true) {
  Socket connection = server.accept();
  provideService(connection);
  }
}
catch (IOException e) {
  System.out.println("Server shut down with error: " + e);
}
```

На клиентской части сокет создается с использованием конструктора класса `Socket`, при этом необходимо знать адрес компьютера и номер порта:

```
public Socket(String computer, int port) throws IOException
```

Первый параметр — это либо `IP`-адрес, либо доменное имя. Этот конструктор блокируется до тех пор, пока не будет установлена связь или пока не возникнет ошибка. После создания связи вызываются методы `getInputStream()` и `getOutputStream()` класса `Socket`, эти методы используются для работы с потоками. Например:

```
void doClientConnection(String computerName, int listeningPort) {
  // computerName — доменное имя или IP-адрес; port — номер порта
  Socket connection;
```

```
  InputStream in;
  OutputStream out;
  try {
     connection = new Socket(computerName,listeningPort);
     in = connection.getInputStream();
     out = connection.getOutputStream();
   }
   catch (IOException e) {
      System.out.println(
        "Attempt to create connection failed with error: " + e);
      return;
  }
    .
    .
    .
  try {
    connection.close();
  }
  catch (IOException e) {
  }
}
```

На практике ситуация несколько осложняется тем, что возникают ошибки в сетях и ошибки, обусловленные человеческим фактором.

Здесь мы рассмотрели базовые идеи, которые дают представление о методах программирования при работе с сетью. Далее мы рассмотрим некоторые работающие примеры, использующие технологию клиент/серверного программирования.

В качестве примера рассмотрим программы, описывающие клиент и сервер. Клиент образует связь с сервером, читает строку текста, присылаемую с сервера, и показывает ее на экране. Посылаемый сервером текст состоит из текущей даты и времени в соответствии с системным временем компьютера, на котором работает сервер. Сервер будет слушать на порте 32007. Программа приведена в файле DateClient.java (листинг 2.8).

Листинг 2.8. Файл DateClient.java

```
import java.net.*;
import java.io.*;
public class DateClient {
   static final int LISTENING_PORT = 32007;
```

```
  public static void main(String[] args) {
    String computer;     // имя компьютера, к которому
                         // устанавливается соединение
    Socket connection;   // сокет
    Reader incoming;     // поток
    // получаем имя компьютера из командной строки
    if (args.length > 0)
      computer = args[0];
    else {
      // имя компьютера не задано, печатаем сообщение и
      // останавливаемся
      System.out.println("Usage:  java DateClient <server>");
      return;
    }
    // устанавливаем связь и читаем строку текста, затем показываем ее
    try {
      connection = new Socket(computer, LISTENING_PORT);
      incoming = new InputStreamReader(connection.getInputStream());
      while (true) {
        int ch = incoming.read();
        if (ch == -1 || ch == '\n' || ch == '\r')
          break;
        System.out.print((char)ch);
      }
      System.out.println();
      incoming.close();
    }
    catch (IOException e) {
      TextIO.putln("Error:  " + e);
    }
}
```

Весь процесс установки связи происходит в блоке `try...catch`. Конец строк соответствует символам `'\n'` или `'\r'`. Серверная программа для работы с этим клиентом (листинг 2.9) названа `DateServer` (рис. 2.6 и 2.7).

Листинг 2.9. Файл DateServer.java

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

```
import java.util.Date;
public class DateServer {
   static final int LISTENING_PORT = 32007;
   public static void main(String[] args) {
     ServerSocket listener;  // прослушивает запросы на соединение
     Socket connection;      // для взаимодействия с другими
                             // программами
     try {
       listener = new ServerSocket(LISTENING_PORT);
       TextIO.putln("Listening on port " + LISTENING_PORT);
       while (true) {
         connection = listener.accept();
         sendDate(connection);
       }
     }
     catch (Exception e) {
       TextIO.putln("Sorry, the server has shut down.");
       TextIO.putln("Error:  " + e);
       return;
     }
   }
   static void sendDate(Socket client) {
     try {
       System.out.println("Connection from " +
                          client.getInetAddress().toString());
       Date now = new Date();  // дата и время
       PrintWriter outgoing;   // поток вывода
       outgoing = new PrintWriter(client.getOutputStream());
       outgoing.println(now.toString());
       outgoing.flush();  // проверка посылки данных
       client.close();
     }
     catch (Exception e){
       System.out.println("Error: " + e);
     }
   }
}
```

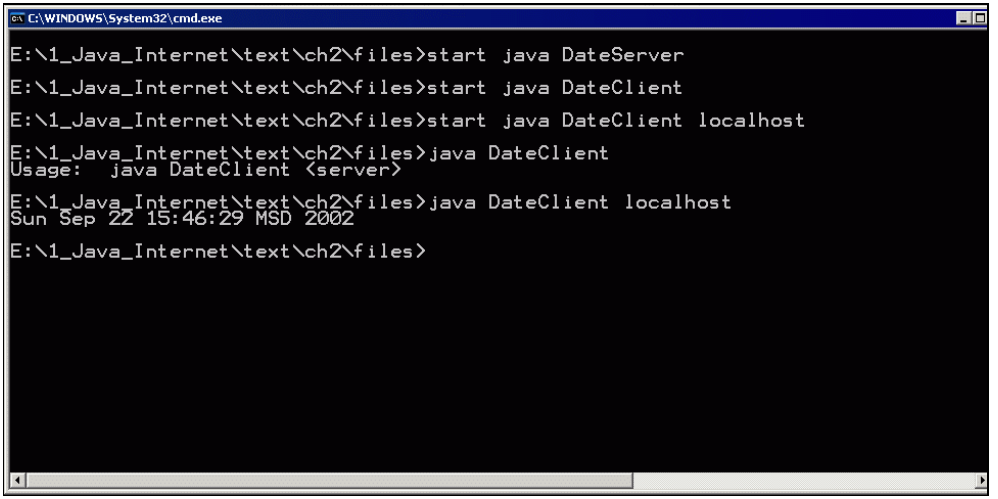


Рис. 2.6. Клиент

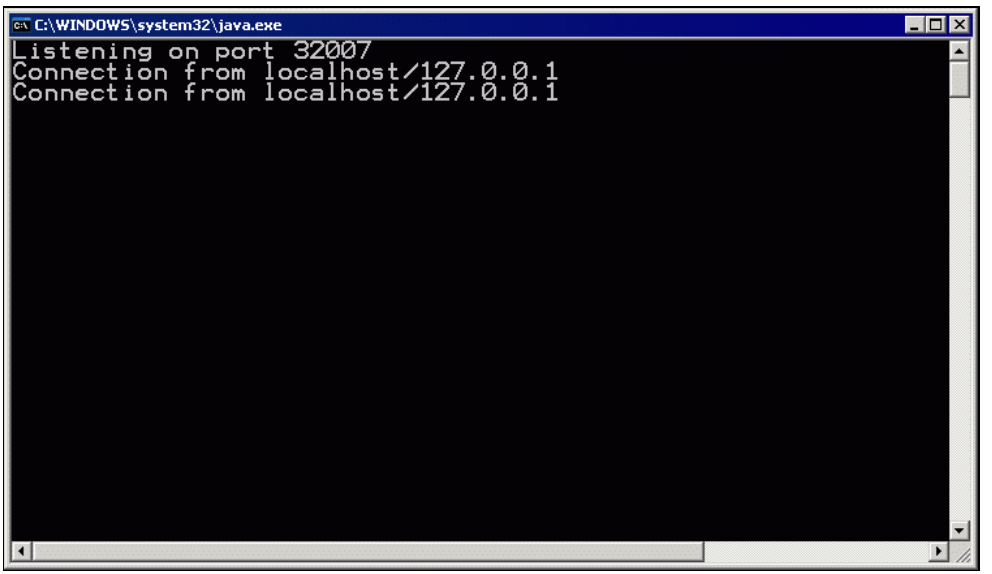


Рис. 2.7. Сервер

После вызова метода `out.println()` мы используем метод `out.flush()`. Метод `flush()` доступен во всех потоках вывода. С его помощью данные посылаются по сетевому соединению. Если мы не используем этот метод, то после записи данных в поток вывода реальные данные могут быть не посланы, поток будет дожидаться поступления достаточного количества данных, и лишь затем они будут посланы. Избежать задержки помогает метод `flush()`.

В предыдущем примере сервер посылал строку клиенту — этим все заканчивалось. Можно создать такие клиент и сервер, которые могут обмениваться сообщениями, например, из командной строки (листинг 2.10 и 2.11). Сервер закрывает прослушиватель порта по получении первого запроса на соединение. Сервер не сможет установить более одного соединения. После установления соединения клиент и сервер ведут себя схожим образом, позволяя друг другу обмениваться сообщениями (рис. 2.8).

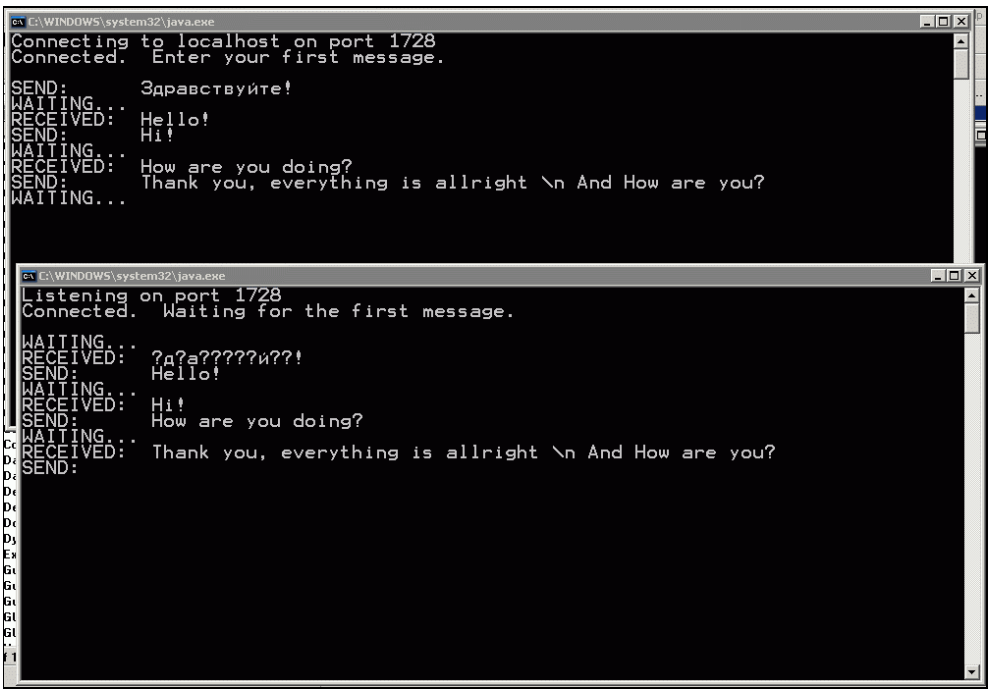


Рис. 2.8. Общение между клиентом и сервером

Листинг 2.10. Файл ChatClient.java

```
import java.net.*;
import java.io.*;
  public class ChatServer {
    static final int DEFAULT_PORT = 1728;  // номер порта;
                                           // используется, если порт
                                           // не задан в командной строке
    static final String HANDSHAKE = "CLChat";  // строка приветствия
                    // каждый участник связи посылает эту строку после
```

```
                // установления связи для того, чтобы быть
                // уверенным, что "собеседником" является "своя"
                // программа
static final char MESSAGE = '0';  // этот символ посылается в
                                  // начале каждого сообщения
static final char CLOSE = '1';    // этот символ посылается при
                                  // выходе пользователя
public static void main(String[] args) {
  int port;
  ServerSocket listener;
  Socket connection;    // сокет для связи
  TextReader incoming;  // поток от клиента
  PrintWriter outgoing; // поток к клиенту
  String messageOut;    // сообщение клиенту
  String messageIn;     // сообщение от клиента
  /*
  Получить номер порта из командной строки, если его нет, то
  использовать порт по умолчанию.
  */
  if (args.length == 0)
    port = DEFAULT_PORT;
  else {
    try {
      port = Integer.parseInt(args[0]);
      if (port < 0 || port > 65535)
        throw new NumberFormatException();
    }
    catch (NumberFormatException e) {
      TextIO.putln("Illegal port number, " + args[0]);
      return;
    }
  }
  /*
  Ожидание запроса на соединение. После получения запроса
  закрываем прослушиватель запросов, создаем потоки и обмениваемся
  строками приветствия.
  */
  try {
    listener = new ServerSocket(port);
```

```
  TextIO.putln("Listening on port " + listener.getLocalPort());
  connection = listener.accept();
  listener.close();
  incoming = new TextReader(connection.getInputStream());
  outgoing = new PrintWriter(connection.getOutputStream());
  outgoing.println(HANDSHAKE);
  outgoing.flush();
  messageIn = incoming.getln();
  if (! messageIn.equals(HANDSHAKE)) {
    throw new IOException("Connected program is not Chat!");
   }
  TextIO.putln("Connected.  Waiting for the first message.\n");
}
catch (Exception e) {
  TextIO.putln("An error occurred while opening connection.");
  TextIO.putln(e.toString());
  return;
}
// обмен сообщениями
try {
  while (true) {
  TextIO.putln("WAITING...");
  messageIn = incoming.getln();
  if (messageIn.length() > 0) {
    // Первый символ сообщения — это команда. Если
    // приходит команда CLOSE, то связь закрывается.
    // В противном случае первый
    // символ просто удаляется.
    if (messageIn.charAt(0) == CLOSE) {
      TextIO.putln("Connection closed at other end.");
      connection.close();
      break;
     }
    messageIn = messageIn.substring(1);
  }
  TextIO.putln("RECEIVED:  " + messageIn);
  TextIO.put("SEND:      ");
  messageOut = TextIO.getln();
  if (messageOut.equalsIgnoreCase("quit"))  {
```

```
            // пользователь хочет выйти
            outgoing.println(CLOSE);
            outgoing.flush();
            connection.close();
            TextIO.putln("Connection closed.");
            break;
          }
          outgoing.println(MESSAGE + messageOut);
          outgoing.flush();
          if (outgoing.checkError()) {
            throw new IOException(
            "Error occurred while transmitting message.");
          }
        }
        catch (Exception e) {
          TextIO.putln("Sorry, an error has occurred.
                       Connection lost.");
          TextIO.putln(e.toString());
          System.exit(1);
        }
    }
}
```

Листинг 2.11. Файл ChatServer.java

```
import java.net.*;
import java.io.*;
public class ChatServer {
   // если порт не указан
   // в командной строке, то используется порт, указанный здесь
   static final int DEFAULT_PORT = 1728;
   static final String HANDSHAKE = "CLChat";  // строка приветствия
                  // Каждая программа посылает строку приветствия,
                  // тем самым подтверждая, что она является
                  // приложением Chat. Это своеобразный "пароль".
   static final char MESSAGE = '0';  // приставка к каждому сообщению
   static final char CLOSE = '1';    // этот символ посылается в качестве
                                     // приставки, когда пользователь
                                     // выходит из программы
```

```
public static void main(String[] args) {
  int port;   // Порт.
  ServerSocket listener;  // слушает запросы на соединение
  Socket connection;      // для связи с клиентом
  TextReader incoming;    // поток данных от клиента
  PrintWriter outgoing;   // поток данных к клиенту
  String messageOut;      // сообщение клиенту
  String messageIn;       // сообщение от клиента
  /*
  Получаем номер порта из командной строки, если он не указан,
  то используем порт по умолчанию.
  */
  if (args.length == 0)
    port = DEFAULT_PORT;
  else {
    try {
      port = Integer.parseInt(args[0]);
      if (port < 0 || port > 65535)
        throw new NumberFormatException();
      }
      catch (NumberFormatException e) {
        TextIO.putln("Illegal port number, " + args[0]);
        return;
      }
  }
  /*
  Ожидаем запрос на соединение.
  После получения запроса закрываем слушающий сокет,
  создаем связь и посылаем строку приветствия.
  */
  try {
    listener = new ServerSocket(port);
    TextIO.putln("Listening on port " + listener.getLocalPort());
    connection = listener.accept();
    listener.close();
    incoming = new TextReader(connection.getInputStream());
    outgoing = new PrintWriter(connection.getOutputStream());
    outgoing.println(HANDSHAKE);
    outgoing.flush();
```

```
    messageIn = incoming.getln();
    if (! messageIn.equals(HANDSHAKE)) {
      throw new IOException("Connected program is not CLChat!");
    }
    TextIO.putln("Connected.  Waiting for the first message.\n");
  }
  catch (Exception e) {
    TextIO.putln("An error occurred while opening connection.");
    TextIO.putln(e.toString());
    return;
  }
  // обмен сообщениями
  try {
    while (true) {
      TextIO.putln("WAITING...");
      messageIn = incoming.getln();
      if (messageIn.length() > 0) {
        // Первый символ сообщения — команда.
        // Если получен смивол CLOSE, то соединение
        // закрывается. В противном случае
        // первый символ просто удаляется.
        if (messageIn.charAt(0) == CLOSE) {
          TextIO.putln("Connection closed at other end.");
          connection.close();
          break;
        }
        messageIn = messageIn.substring(1);
      }
      TextIO.putln("RECEIVED:  " + messageIn);
      TextIO.put("SEND:      ");
      messageOut = TextIO.getln();
      if (messageOut.equalsIgnoreCase("quit"))  {
        // Пользователь выходит.
        // Информируем об этом собеседника.
        outgoing.println(CLOSE);
        outgoing.flush();
        connection.close();
        TextIO.putln("Connection closed.");
        break;
```

```
        }
        outgoing.println(MESSAGE + messageOut);
        outgoing.flush();
        if (outgoing.checkError()) {
          throw new IOException("Error occurred while transmitting
                                 message.");
        }
      }
    }
    catch (Exception e) {
      TextIO.putln("Sorry, an error has occurred.  Connection lost.");
      TextIO.putln(e.toString());
      System.exit(1);
    }
  }
}
```

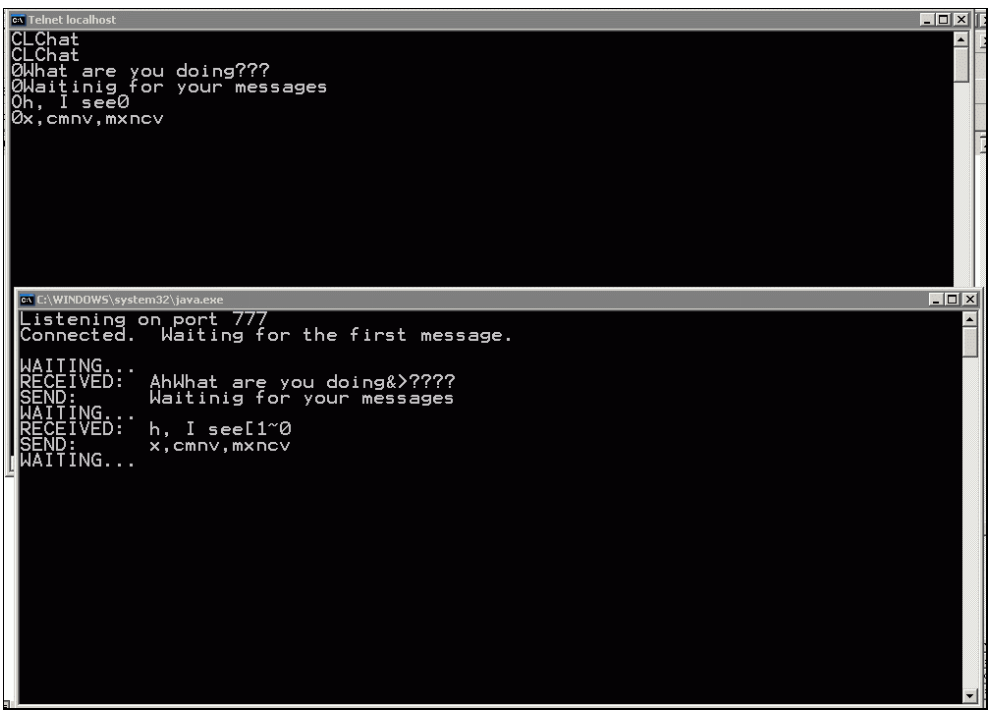


Рис. 2.9. Сервер (листинг 2.11) и клиент telnet на порте 777

Этот пример несколько более приближен к действительности. В нем мы также используем строку приветствия, которая в нашем случае является частью протокола. Можно поэкспериментировать с сервером, подключившись к работающему серверу, например, с помощью клиента Telnet (рис. 2.9). Можно попытаться соединиться с сервером из браузера. Можно также запустить программу-клиент и попытаться установить соединение с каким-либо другим сервером. Проблема лишь в том, что протоколы общения клиентов с серверами будут отличаться друг от друга. Для эффективной коммуникации необходимо учитывать эти протоколы, то есть общение должно происходить в рамках установленного протокола.

2.4.3. Потоки и работа с сетью

Программирование и работа с сетью невозможны без использования одновременного выполнения нескольких потоков. Потоки полезны в клиентских программах. Но еще более необходимы они в серверных приложениях. Для осуществления связи с новым клиентом сервер создает новый поток. При этом второму клиенту не придется ждать, когда первый клиент завершит работу с сервером. Если рассмотреть предыдущий пример, то здесь пользователь (с любой стороны) будет вынужден ждать до тех пор, пока он не получит ответное сообщение. Хорошо было бы иметь возможность, не дожидаясь ответа с противоположной стороны, работать с текстом, создавая следующее сообщение. Такой подход неприемлем в командном интерфейсе, но графический интерфейс пользователя вполне приспособлен для решения такого рода задач. Тут как раз и могут пригодиться потоки. Но есть и еще одна проблема: если наш сервер занят, то мы не сможем подсоединить к нему еще один клиент. Хорошо было бы знать, какие клиенты ждут очереди на соединение. Такая задача также может быть решена с применением потоков.

Рассмотрим программу общения, использующую графический интерфейс пользователя. Когда один пользователь соединяется с другим, открывается окно с областью для ввода текста. В это окно пользователь может ввести текст, который будет передан другому пользователю. Пользователь должен иметь возможность отправить созданное сообщение в любое время. Программа также должна быть готова получить сообщение в любой момент, а полученные сообщения должны быть отображены и видимы пользователю сразу после их поступления. Сервер может общаться с несколькими клиентами одновременно. Такой сервер можно поместить в общедоступное место, где он будет поддерживать список участников чата (сетевого общения в реальном времени). Клиент может связаться с сервером из любого места Интернета, зарегистрироваться и участвовать в чате. Сервер представлен в файле ConnectionBroker.java (листинг 2.12).

Листинг 2.12. Файл ConnectionBroker.java

```
/*
ConnectionBroker — это сетевой сервер, устанавливающий связь с клиентами.
Клиенты регистрируют себя на сервере, тем самым делая себя доступными для
участия в чате. Каждый, кто соединяется с сервером, должен выполнить одну
из следующих команд:
        REGISTER: Пользователь добавляется в список доступных клиентов,
                  все символы после команды REGISTER сохраняются как
                  дополнительная информация о пользователе. Пользователю
                  присваивается идентификатор ID. Клиент может иметь
                  только одного партнера. Клиент не может посылать данные
                  до тех пор, пока он не имеет партнера. Исключение —
                  команда close. Когда пользователь получает партнера,
                  к нему направляется сообщение, которое начинается со
                  слова CONNECTED.
         CONNECT: Данные в строке после команды CONNECT должны быть
                  идентификатором ID. Соединение устанавливается между
                  клиентом, пославшим команду CONNECT, и клиентом, чей
                  идентификатор был послан в строке после команды
                  CONNECT. После этого сервер устанавливает связь между
                  клиентами, которая существует до тех пор, пока один из
                  них не закроет соединение. Сообщение, начинающееся
                  с символа закрытия связи, служит для передачи информации
                  о закрытии соединения. Если соединение не может быть
                  установлено, то сервер отвечает строкой, которая
                  в качестве своего первого символа содержит символ
                  NOT_AVAILABLE. Если соединение установлено, то обоим
                  клиентам посылается сообщение, начинающееся символом
                  CONNECTED.
SEND_CLIENT_LIST: Сервер отвечает, посылая информацию о всех
                  клиентах, ожидающих партнера. Каждая строка
                  начинается с символа CLIENT_INFO, за которой
                  следует идентификатор клиента ID и информация
                  о клиенте. Строка, начинающаяся символом
                  END_CLIENT_INFO служит признаком конца списка
                  клиентов, находящихся в состоянии ожидания
                  партнера.
```

```
Порт, который прослушивается сервером, может быть задан в качестве пара-
метра в командной строке. Если номер порта в командной строке
не указан, то используется порт, установленный по умолчанию в качестве
значения для DEFAULT_PORT. В качестве клиентов могут выступать апплеты.
*/
import java.io.*;
import java.net.*;
import java.util.Vector;
public class ConnectionBroker {
   // первые символы сообщений
   static final char REGISTER = '[';    // символы команд
   static final char CONNECT = '=';     // от клиента серверу
   static final char SEND_CLIENT_LIST = ':';
   static final char CLOSE = ']';              // закрыть связь
   static final char NOT_AVAILABLE = '!';      // символы команд
   static final char CONNECTED = '.';          // от сервера клиенту
   static final char CLIENT_INFO = '>';
   static final char END_CLIENT_INFO = '<';
   /*
   если порт не был указан в командной строке, то сервер прослушивает
   порт DEFAULT_PORT
   */
   static final int DEFAULT_PORT = 3030;
   private static Vector clientList = new Vector();  // доступные
                                                     // клиенты
   private static int nextClientID = 1;  // следующий идентификатор
                                         // клиента
   public static void main(String[] args) {
           // Функция main() создает прослушиваемый порт.
           // После получения запроса на соединение
           // создается новый сокет в отдельном потоке.
      int port;  // порт для прослушивания
      ServerSocket listener;
      Socket client;
      if (args.length == 0)
        port = DEFAULT_PORT;
      else {
        try {
          port = Integer.parseInt(args[0]);
```

```
      }
      catch (NumberFormatException e) {
        System.out.println(args[0] + " is not a legal port number.");
        return;
      }
    }
    try {
      listener = new ServerSocket(port);
    }
    catch (IOException e) {
      System.out.println("Can't start server.");
      System.out.println(e.toString());
      return;
    }
    System.out.println("Listening on port " + listener.getLocalPort());
    try {
      while (true) {
        client = listener.accept();
        new ClientThread(client);
      }
    }
    catch (Exception e) {
      System.out.println("Server shut down unexpectedly.");
      System.out.println(e.toString());
      System.exit(1);
    }
  }
  /*
  Четыре функции для работы со списком пользователей.
  Используется синхронизация.
  */
  static void addClient(Client client) {
    // вставка нового клиента
    synchronized(clientList) {
      client.ID = nextClientID++;
      if (client.info.length() == 0)
        client.info = "Anonymous" + client.ID;
        clientList.addElement(client);
      }
```

```
    System.out.println("Added client " + client.ID + " " +
                       client.info);
  }
  static void removeClient(Client client) {
    // удаление клиента
    synchronized(clientList) {
      clientList.removeElement(client);
    }
    System.out.println("Removed client " + client.ID);
  }
  static Client getClient(int ID) {
    // удаление клиента по идентификатору
    synchronized(clientList) {
      for (int i = 0; i < clientList.size(); i++) {
        Client c = (Client)clientList.elementAt(i);
        if (c.ID == ID) {
          clientList.removeElementAt(i);
          System.out.println("Removed client " + c.ID);
          c.ID = 0;
          return c;
        }
      }
      return null;
    }
  }
  static Client[] getClients() {
    synchronized(clientList) {
      if (clientList.size() == 0)
        return null;
      Client[] clients = new Client[ clientList.size() ];
        for (int i = 0; i < clientList.size(); i++)
          clients[i] = (Client)clientList.elementAt(i);
        return clients;
    }
  }
  private static class Client {
    // Данные клиента. Для клиентов, ожидающих соединение, ID  > 0.
    // Для клиентов, запросивших соединение, ID равно 0.
    int ID;  // идентификатор клиента
```

```
  String info;  // информация клиента
  Socket connection;  // связь между сервером и клиентом
  Reader incoming;    // чтение данных
  PrintWriter outgoing;     // отправка данных
  volatile Client partner;  // если не null, то данные могут
                            // передаваться от клиента другому
                            // клиенту и наоборот
  private StringBuffer line = new StringBuffer();  // чтение данных
  private boolean checkLineFeed;  // обработка символов конца строки
  String getln() throws IOException {
    // чтение одной строки
    if (incoming == null)
      throw new IOException("Not connected.");
    int ch;
    ch = incoming.read();
    if (ch == -1)
      throw new IOException("Attempt to read past end-of-stream.");
    if (ch == '\n' && checkLineFeed)
      ch = incoming.read();
    line.setLength(0);
    while (ch != -1 && ch != '\n' && ch != '\r') {
      line.append((char)ch);
      ch = incoming.read();
    }
    checkLineFeed = (ch == '\r');
    return line.toString();
  }
  void send(String message) throws IOException {
    // отправка одной строки
    if (outgoing == null)
      throw new IOException("Not connected.");
    outgoing.println(message);
    outgoing.flush();
    if (outgoing.checkError())
      throw new IOException("Error while sending data.");
  }
}
```

```
private static class ClientThread extends Thread {
  // Поток создания соединения с сервером.
  // Поток открывается конструктором.
  Client client;  // клиент
  ClientThread(Socket connection) {
    // Конструктор. Создается объект Client и
    // запускается поток создания соединения.
    client = new Client();
    client.connection = connection;
    start();
  }
  public void run() {
    try {
      client.connection.setSoTimeout(30000);
      processClient();
    }
    catch (Exception e) {
    }
    finally {
      try {
        client.connection.close();
      }
      catch (Exception e) {
      }
      if (client.ID > 0)
        removeClient(client);
      }
    }
    void processClient() throws IOException {
      client.incoming =
      new InputStreamReader(client.connection.getInputStream());
      client.outgoing =
      new PrintWriter(client.connection.getOutputStream());
      String message = client.getln();
      if (message.length() == 0) {
        throw new IOException(
          "Unexpected input to 'ConnectionBroker' program.");
```

```
    }
    client.connection.setSoTimeout(3600000); // 60 минут
    char cmd = message.charAt(0);
    message = message.substring(1); // здесь могут быть данные
                                    // для команды
    if (cmd == REGISTER) {
      client.info = message;
      addClient(client);
      int ID = client.ID;
      while (true) {
        message = client.getln();
        if (client.partner != null)
          client.partner.send(message);
        if (message.length() > 0 && message.charAt(0) == CLOSE) {
          System.out.println("Closing down relay for client " +
                             ID);
          break;
        }
      }
    }
    else if (cmd == CONNECT) {
      // соединение с клиентом
      int partnerID;
      try {
        partnerID = Integer.parseInt(message);
      }
      catch (NumberFormatException e) {
        client.send(NOT_AVAILABLE + "Client ID is not an
                                                integer.");
        throw new IOException("Illegal connect request.");
      }
      Client partner = getClient(partnerID);
      if (partner == null) {
        client.send(NOT_AVAILABLE + "Unknown client ID.");
        throw new IOException("Requested connection not found.");
      }
      client.partner = partner;      // установление партнера
```

```
            partner.partner = client;
            System.out.println("Setting up relay to client " +
                                                          partnerID);
            partner.send("" + CONNECTED);  // сообщение партнерам
            client.send("" + CONNECTED);
            while (true) {
              message = client.getln();
              partner.send(message);
              if (message.length() > 0 && message.charAt(0) == CLOSE) {
                System.out.println("Closing down relay for client " +
                                                          partnerID);
                break;
              }
            }
          }
          else
            if (cmd == SEND_CLIENT_LIST) {
              // Запрашивается список ждущих соединение клиентов.
              Client[] clients = getClients();
              if (clients != null) {
                for (int i =0; i < clients.length; i++)
                  client.send("" + CLIENT_INFO + clients[i].ID + " " +
                                                    clients[i].info);
              }
              client.send("" + END_CLIENT_INFO);
            }
          else {
            throw new IOException("Unexpected input to 'ConnectionBroker'
                                                          program.");
          }
        }
      }
}
```

Этот сервер работает совместно с клиентскими программами. Клиент представлен в файле BrokeredChat.java (листинг 2.13). После запуска сервера (на компьютере с именем `localhost`) с указанием номера порта запускаем `BrokeredChat` с указанием компьютера и номера порта (пишем в командной строке `java BrokeredChat localhost`). Затем регистрируемся и ожидаем

соединения с другим клиентом. После регистрации появляется окно ChatWindow, которое описано в файле ChatWindow.java (рис. 2.10).

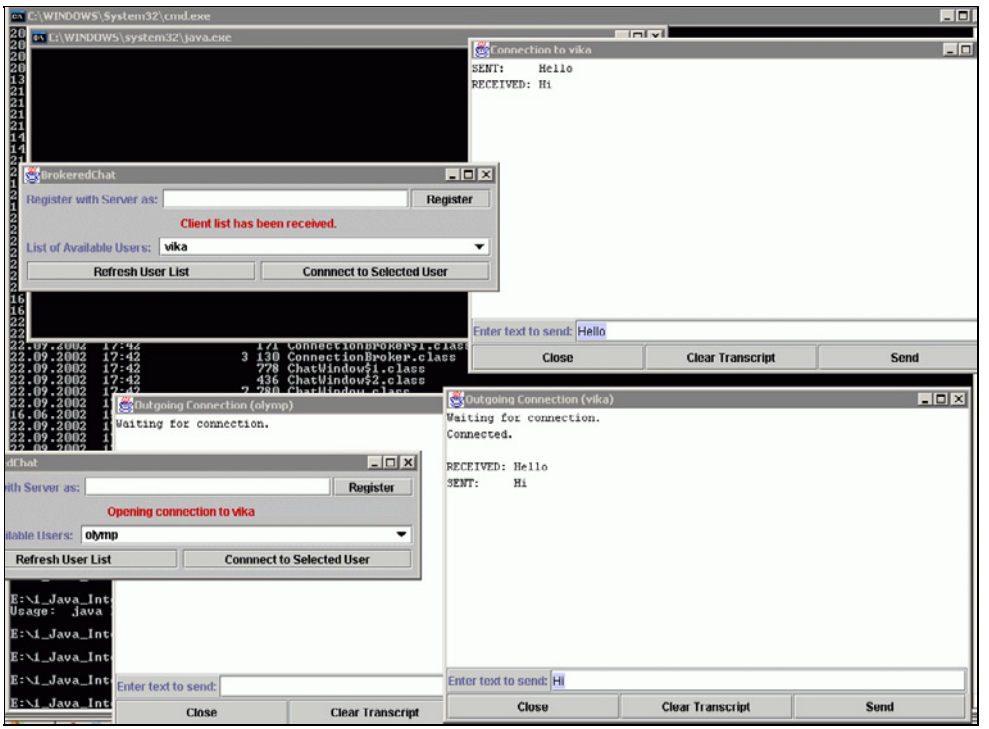


Рис. 2.10. Регистрация клиента и участие в чате

Листинг 2.13. Файл BrokeredChat.java

```
/*
Апплет предназначен для чата между двумя пользователями.
Апплет представляет собой клиент для сервера ConnectionBroker.
Сервер устанавливает связь между двумя такими клиентами.
Клиент решает две задачи. Первая состоит в том, что
клиент может зарегистрироваться на сервере
для ожидания соединения с другим клиентом.
Вторая состоит в том, что клиент устанавливает соединение
с другим клиентом, который ожидает соединение.
Апплет может запросить и получить список клиентов,
ожидающих соединение.
```

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

public class BrokeredChat extends JApplet implements ActionListener {
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("Usage:  java BrokeredChat <server> [<port>]");
      return;
    }
    BrokeredChat applet = new BrokeredChat();
    applet.computer = args[0];
    applet.port = DEFAULT_PORT;
    if (args.length > 1) {
      try {
        applet.port = Integer.parseInt(args[1]);
      }
      catch (NumberFormatException e) {
      }
    }
    JFrame window = new JFrame("BrokeredChat");
    window.setContentPane(applet.makeInterfacePanel());
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    window.setSize(500,135);
    window.setLocation(20,50);
    window.show();
    applet.doCommand(SEND_CLIENT_LIST);
  }
  // команды для общения с сервером
  static final char REGISTER = '[';
  static final char CONNECT = '=';
  static final char SEND_CLIENT_LIST = ':';
  static final char NOT_AVAILABLE = '!';
  static final char CONNECTED = '.';
  static final char CLIENT_INFO = '>';
```

```
static final char END_CLIENT_INFO = '<';
static final int DEFAULT_PORT = 3030;

JLabel display;
JComboBox clients;
JButton connectButton;
JButton refreshButton;
JButton registerButton;
JTextField infoInput;
ArrayList clientStrings;
String computer;
int port;

public void init() {
  setBackground(Color.lightGray);
  setContentPane(makeInterfacePanel());
  doCommand(SEND_CLIENT_LIST);
}

JPanel makeInterfacePanel() {
  JPanel panel = new JPanel();

  panel.setBackground(Color.lightGray);
  panel.setLayout(new GridLayout(4,1,5,5));
  panel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
  JPanel top = new JPanel();
  top.setBackground(Color.lightGray);
  top.setLayout(new BorderLayout(2,2));
  JPanel middle = new JPanel();
  middle.setBackground(Color.lightGray);
  middle.setLayout(new BorderLayout(5,5));
  JPanel bottom = new JPanel();
  bottom.setBackground(Color.lightGray);
  bottom.setLayout(new GridLayout(1,2,5,5));

  display = new JLabel("",JLabel.CENTER);
  display.setForeground(new Color(200,0,0));

  panel.add(top);
```

```
panel.add(display);
panel.add(middle);
panel.add(bottom);

clients = new JComboBox();
clients.setBackground(Color.white);

connectButton = new JButton("Connnect to Selected User");
connectButton.addActionListener(this);

refreshButton = new JButton("Refresh User List");
refreshButton.addActionListener(this);

registerButton = new JButton("Register");
registerButton.addActionListener(this);

infoInput = new JTextField();
infoInput.setBackground(Color.white);
infoInput.addActionListener(this);

top.add(infoInput,BorderLayout.CENTER);
top.add(registerButton,BorderLayout.EAST);
top.add(new JLabel("Register with Server as: "),
        BorderLayout.WEST);
middle.add(clients,BorderLayout.CENTER);
middle.add(new JLabel("List of Available Users: "),
                      BorderLayout.WEST);
bottom.add(refreshButton);
bottom.add(connectButton);

if (computer == null) {
  String portStr = getParameter("port");
  if (portStr == null)
    port = DEFAULT_PORT;
  else {
    try {
      port = Integer.parseInt(portStr);
    }
    catch (NumberFormatException e) {
```

```
        port = DEFAULT_PORT;
      }
    }
    computer = getParameter("server");
    if (computer == null) {
      // используем компьютер, с которого был загружен апплет
      computer = getCodeBase().getHost();
    }
  }
  return panel;

}

public void actionPerformed(ActionEvent evt) {
  // ответ на нажатие кнопки пользователем
  Object source = evt.getSource();
  if (source == connectButton) {
    doCommand(CONNECT);
  }
  else if (source == registerButton || source == infoInput) {
    doCommand(REGISTER);
  }
  else {
    doCommand(SEND_CLIENT_LIST);
  }
}

void doCommand(char command) {
  String message;  // команда серверу
  String name;
  if (command == CONNECT) {
    int partner = clients.getSelectedIndex();
    if (clientStrings == null || partner < 0 ||
        partner >= clientStrings.size()) {
      display.setText("Can't connect -- no user!");
      return;
    }
    String info = (String)clientStrings.get(partner);
    int pos = info.indexOf(" ");
```

```
    if (pos > 0) {
      message = "" + CONNECT + info.substring(0,pos);
      name = info.substring(pos+1);
    }
    else {
      message = "" + CONNECT + info;
      name = "unknown user";
    }
  }
  else
    if (command == REGISTER) {
      // регистрация на сервере
      name = infoInput.getText().trim();
      if (name.length() == 0) {
        display.setText("To register, you must enter a name.");
        return;
      }
      infoInput.setText("");
      message = REGISTER + name;
      display.setText("Contacting server to register you...");
    }
    else {
      // команда получения клиентского списка с сервера
      clients.removeAllItems();
      clients.addItem("(Checking)");
      message = "" + SEND_CLIENT_LIST;
      display.setText("Contacting server to get user list...");
      name = null;
    }
    registerButton.setEnabled(false);  //кнопка не работает до
                                       // завершения операции
    connectButton.setEnabled(false);
    refreshButton.setEnabled(false);
    new ConnectionHandler(message,name);
  }
  class ConnectionHandler extends Thread {
    // поток установки соединения
    String message;
    String name;  // имя для окна ChatWindow
```

```
  ConnectionHandler(String message, String name) {
     this.message = message;
     this.name = name;
     start();
}
public void run() {
  Socket connection = null;
  ArrayList clients = null;
  String displayText = "Client list has been received.";
  try {
    if (message.charAt(0) == REGISTER) {
      connection = new Socket(computer,port);
      PrintWriter out = new
                        PrintWriter(connection.getOutputStream());
      out.println(message);
      out.flush();
      if (out.checkError())
        throw new IOException("Can't send command.");
        new ChatWindow("Outgoing Connection (" + name + ")",
        connection,true);
        displayText = "Opening window to wait for connection";
      }
      else
        if (message.charAt(0) == CONNECT) {
          connection = new Socket(computer,port);
          connection.setSoTimeout(30000);  // ожидание 30 секунд
          TextReader in = new
                          TextReader(connection.getInputStream());
          PrintWriter out = new
                           PrintWriter(connection.getOutputStream());
          out.println(message);  // запрос соединения
          out.flush();
          if (out.checkError())
            throw new IOException("Can't send command.");
            String answer = in.getln();  // чтение ответа сервера
            if (answer.length() == 0 || answer.charAt(0) !=
                                                    CONNECTED)
            {
               displayText = "Can't connect to " + name;
```

```
        }
        else {
          connection.setSoTimeout(0);
          new ChatWindow("Connection to " + name, connection);
          displayText = "Opening connection to " + name;
        }
      }

      connection = null;
      connection = new Socket(computer,port);
      connection.setSoTimeout(30000);
      TextReader in = new
                      TextReader(connection.getInputStream());
      PrintWriter out = new
                       PrintWriter(connection.getOutputStream());
      out.println("" + SEND_CLIENT_LIST);  //запрос списка
      out.flush();
      if (out.checkError())
        throw new IOException("Can't send command.");
      ArrayList clientInfo = new ArrayList();
      while (true) {
        String line = in.getln();
        if (line.length() > 0) {
          if (line.charAt(0) == END_CLIENT_INFO)
             break;
          else if (line.charAt(0) == CLIENT_INFO)
            clientInfo.add(line.substring(1));
         else
             throw new IOException("Bad data received.");
       }
    }
    clients = clientInfo;
}
catch (Exception e) {
  displayText = "Error: " + e.getMessage();
}
finally{
  finishConnection(clients,displayText);
  if (connection != null) {
```

```
          try {
            connection.close();
          }
          catch (IOException e) {
          }
        }
      }
    }
}

void finishConnection(final ArrayList clientList, final String
                                                    displayText)
{
  SwingUtilities.invokeLater (new Runnable() {
    public void run() {
    if (clientList == null) {
      clients.removeAllItems();
      clients.addItem("(Can't connect)");
    }
    else
      if (clientList.size() == 0) {
        clients.removeAllItems();
        clients.addItem("(None available)");
      }
      else {
        makeClientList(clientList);
        connectButton.setEnabled(true);
      }
      display.setText(displayText);
      refreshButton.setEnabled(true);  // включить кнопки
      registerButton.setEnabled(true);
      clientStrings = clientList;
    }
 });
}

void makeClientList(ArrayList clientInfo) {
  clientStrings = clientInfo;
  clients.removeAllItems();
```

```
    for (int i = 0; i < clientInfo.size(); i++) {
      String info = (String)clientInfo.get(i);
      int spacePos = info.indexOf(" ");
      if (spacePos > 0)
        info = info.substring(spacePos+1);
        clients.addItem(info);
      }
    }
}
```

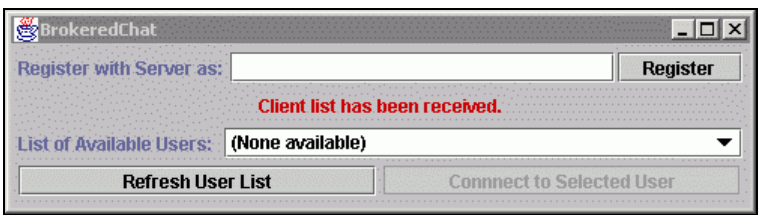


Рис. 2.11. Сервер найден

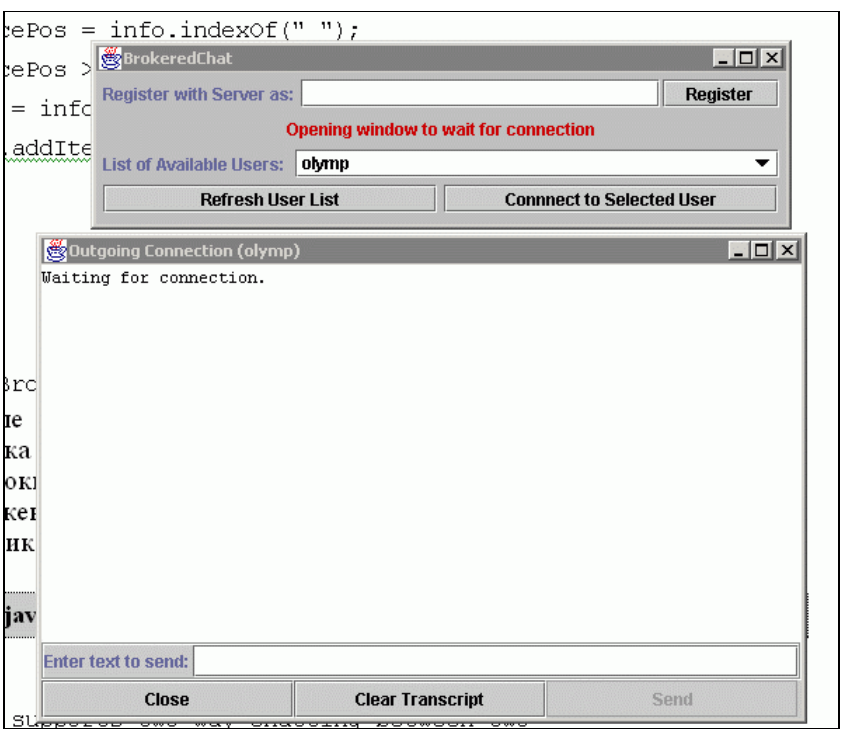


Рис. 2.12. Регистрация пользователя

Приведенный файл является клиентской программой. При его запуске происходит попытка установки связи с сервером `ConnectionBroker`. Если связь установлена, то в окне появится надпись **Client list has been received.** (рис. 2.11). Пользователь должен зарегистрироваться, для этого вводится какое-либо имя пользователя (рис. 2.12). После того как пользователь зарегистрирован, он получает диалоговое окно для ввода текста для чата и ожидает соединения. Если соединение устанавливается, пользователи могут переписываться друг с другом.

Листинг 2.14. Файл ChatWindow.java

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

public class ChatWindow extends JFrame
       implements ActionListener, WindowListener, Runnable {

   static final String HANDSHAKE = "<Independent Connection Window>";
   static final int DEFAULT_PORT = 17171;
   public static final void main(String[] args) {
     int port;
     Socket connection;
     TextReader in;
     PrintWriter out;
     String message;
     if (args.length == 0) {
       System.out.println("Usage:  java ChatWindow <server-computer>
         [<port>]");
       System.out.println("    or  java ChatWindow -s [<port>]");
       return;
     }
     if (args.length == 1)
       port = DEFAULT_PORT;
     else {
       try {
         port = Integer.parseInt(args[1]);
       }
```

```
    catch (NumberFormatException e) {
      System.out.println(args[1] + " is not a legal port number.");
      return;
    }
  }
  try {
    if (args[0].equalsIgnoreCase("-s")) {
      ServerSocket listener = new ServerSocket(port);
      System.out.println("Listening on port " +
                         listener.getLocalPort());
      connection = listener.accept();
      listener.close();
    }
    else {
      connection = new Socket(args[0],port);
    }
    out = new PrintWriter(connection.getOutputStream());
    out.println(HANDSHAKE);
    out.flush();
    in = new TextReader(connection.getInputStream());
    message = in.getln();
    if (! message.equals(HANDSHAKE))
      throw new IOException("Connected program is not ChatWindow");
      System.out.println("Connected.");
    }
    catch (Exception e) {
      System.out.println("Error opening connection.");
      System.out.println(e.toString());
      return;
    }
    ChatWindow w = new ChatWindow("ChatWindow", connection);
    w.standalone = true;
  }
  static final char MESSAGE = '0';
  static final char CLOSE = ']';
  JTextArea transcript;
  JTextField messageInput;
  Socket connection;
  TextReader incoming;
```

```
  PrintWriter outgoing;
  Thread reader;
  JButton clearButton;
  JButton sendButton;
  JButton closeButton;
  boolean  waitFor;
  boolean standalone = false;
  public ChatWindow(String title, Socket connection) {
  this(title,connection,false);
}

public ChatWindow(String title, Socket connection, boolean waitFor) {
  super(title);
  this.connection = connection;
  addWindowListener(this);
  transcript = new JTextArea();
  transcript.setBackground(Color.white);
  transcript.setEditable(false);
  transcript.setFont(new Font("Monospaced", Font.PLAIN, 12));
  messageInput = new JTextField();
  messageInput.addActionListener(this);
  messageInput.setBackground(Color.white);
  closeButton = new JButton("Close");
  closeButton.addActionListener(this);
  clearButton = new JButton("Clear Transcript");
  clearButton.addActionListener(this);
  sendButton = new JButton("Send");
  sendButton.addActionListener(this);
  // компоненты
  JPanel bottom = new JPanel();
  bottom.setLayout(new GridLayout(2,1));
  JPanel inputBar = new JPanel();
  inputBar.setLayout(new BorderLayout());
  inputBar.add(messageInput, BorderLayout.CENTER);
  inputBar.add(new JLabel("Enter text to send: "), BorderLayout.WEST);
  inputBar.setBorder(BorderFactory.createEtchedBorder());
  JPanel buttonBar = new JPanel();
  buttonBar.setLayout(new GridLayout(1,3));
  buttonBar.add(closeButton);
```

```
  buttonBar.add(clearButton);
  buttonBar.add(sendButton);
  bottom.add(inputBar);
  bottom.add(buttonBar);
  getContentPane().add(bottom, BorderLayout.SOUTH);
  getContentPane().add(transcript, BorderLayout.CENTER);

  try {
    incoming = new TextReader(connection.getInputStream());
    outgoing = new PrintWriter(connection.getOutputStream());
  }
  catch (IOException e) {
    transcript.setText("Error opening I/O streams!\n"
                       + "Connection can't be used.\n"
                       + "You can close the window now.\n");
    closeButton.setText("Close Window");
    sendButton.setEnabled(false);
    clearButton.setEnabled(false);
    this.connection = null;
  }

  if (this.connection != null) {
    reader = new Thread(this);
    reader.start();
    if (waitFor) {
      sendButton.setEnabled(false);
      transcript.setText("Waiting for connection.\n");
      this.waitFor = true;
    }
  }

  // размер окна
  setSize(550,350);
  setLocation(50,80);
  show();
  setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
}

public void actionPerformed(ActionEvent evt) {
```

```
  Object source = evt.getSource();
  if (source == clearButton)
    transcript.setText("");
  else
    if (source == sendButton || source == messageInput)
      doSend();
  else
    if (source == closeButton)
      doClose();
}

synchronized void doSend() {
  if (connection == null) {
    return;
  }
  if (waitFor) {
    return;
  }
  String message = messageInput.getText();
  outgoing.println(MESSAGE + message);
  outgoing.flush();
  if (outgoing.checkError()) {
    transcript.append("ERROR:    An error occured while sending
                                                        data.\n");
    doForceClose();
  }
  else
    transcript.append("SENT:     " + message + '\n');
  messageInput.selectAll();
  messageInput.requestFocus();
}

void doClose() {
  if (connection != null) {
    outgoing.println(CLOSE);
    outgoing.flush();
    try {
      connection.close();  // прервать соединение
    }
```

```
    catch (IOException e) {
    }
    connection = null;  //соединение разорвано
  }
  dispose();  // окно закрыто
}

void doForceClose() {
   if (connection == null) {
     return;
   }
   closeButton.setText("Close Window");
   sendButton.setEnabled(false);
   clearButton.setEnabled(false);
   try {
     connection.close();
   }
   catch (IOException e) {
   }
   transcript.append("\nCLOSED:   Connection has been closed.\n");
   transcript.append("          You can now close the window.\n");
   connection = null;
}

void addToTranscript(final String message) {
  Runnable postit = new Runnable() {
    public void run() {
      transcript.append(message + "\n");
    }
  };
  SwingUtilities.invokeLater(postit);
}

void closeFromThread() {
  Runnable closeit = new Runnable() {
    public void run() {
      doForceClose();
    }
  };
```

```
  SwingUtilities.invokeLater(closeit);
}

public void run() {
  try {
    while (true) {
      String message = incoming.getln();
      if (connection == null) {
        break;
      }
      if (waitFor) {
        addToTranscript("Connected.\n");
        sendButton.setEnabled(true);
        waitFor = false;
      }
      else
        if (message.length() > 0) {
          if (message.charAt(0) == CLOSE) {
            addToTranscript("\nConnection closed from other side.");
            closeFromThread();
            break;
          }
          message = message.substring(1);
          addToTranscript("RECEIVED: " + message);
        }
      }
   }
   catch (Exception e) {
     synchronized (this) {
       if (connection != null) {
         addToTranscript("ERROR:    An error occurred while
                                                  receiving data.");
         closeFromThread();
       }
     }
   }
}

public void windowClosing(WindowEvent evt) {
```

```
  doClose();
}

public void windowClosed(WindowEvent evt) {
   if (standalone)
     System.exit(0);
}

public void windowOpened(WindowEvent evt) { }
public void windowIconified(WindowEvent evt) { }
public void windowDeiconified(WindowEvent evt) { }
public void windowActivated(WindowEvent evt) { }
public void windowDeactivated(WindowEvent evt) { }
}
```

Этот пример (листинг 2.14) вполне работоспособен. Конечно, он не лишен недостатков и вовсе не безупречен, но в нем продемонстрированы основные методы работы с сокетами, используя которые, можно создавать вполне совершенные программы.

После запуска потока обработки соединения, поток может принимать команды и обрабатывать их соответствующим образом. Поток понимает три типа команд. Команда `REGISTER` добавляет клиента в список доступных участников чата. Соединение остается открытым до тех пор, пока кто-либо из участников не обратится к клиенту для установления связи. После установления связи между клиентами участники удаляются из списка доступных клиентов. Команда `SEND_CLIENT_LIST` запрашивает копию доступных участников чата. Сервер в ответ посылает список участников чата. Команда `CONNECT` используется для установления соединения с одним из доступных участников чата. Сервер устанавливает связь и, если не возникли ошибки, посылает сообщение обоим участникам о том, что связь установлена. После этого участники могут обмениваться сообщениями друг с другом. Сообщения пересылаются через сервер, прямая связь остается между сервером и двумя общающимися через сервер клиентами. Сервер передает сообщения от одного клиента другому и наоборот. При этом сервер `ConnectionBroker` работает с апплетами, которые являются для него клиентами. Апплеты загружаются с компьютера, на котором работает сервер. При таком подходе прямое общение между двумя клиентами невозможно, так как апплеты не могут устанавливать соединения с ресурсами, расположенными на другом компьютере, отличающемся от того, с которого был загружен апплет.

Чтобы иметь возможность работать с сервером `ConnectionBroker`, создан апплет, который является для него клиентом. Апплет пытается установить

связь с ConnectionBroker, который располагается на том же компьютере, с которого загружается апплет. Если требуемый сервер не работает, апплет выводит сообщение об ошибке.

2.4.4. Синхронизация

В сервере ConnectionBroker существует момент, обойти вниманием который мы не сможем. Что произойдет, если два или более потоков обратятся к одним и тем же данным? В этом случае данные могут быть неверно использованы. Чтобы такого не произошло, используется синхронизация. Предположим, что количество денег на счету в банке представлено в виде следующего класса:

```
public class BankAccount {
  private double balance;  // сумма на счету
  public double getBalance() {
    return balance;
  }
  public void withdraw(double amount) {
    // условие:  баланс balance должен быть больше или равен сумме amount
    balance = balance — amount;
  }

.  // прочие методы
.
}
```

Предположим, что account — это объект типа BankAccount, и эта переменная используется сразу в нескольких потоках. Предположим, что один из потоков желает снять со счета $100:

```
if (account.getBalance() >= 100)
  account.withdraw();
```

Теперь представим, что не только этот поток стремится снять суммы со счета, а сразу два потока пытаются снять указанную сумму. Если на счету есть 150 долларов и оба потока действуют в одно и то же время, то в принципе каждый из них может получить по 100 долларов, и на счету еще останется $50. Работа потоков будет происходить примерно в такой последовательности:

1. Первый поток читает баланс и получает значение $150.
2. Второй поток читает баланс и получает значение $150.
3. Второй поток снимает $100, всего остается $50.
4. Второй поток сохраняет новое значение баланса, то есть $50.

5. Первый поток вычитает $100 из $150, остается $50.
6. Первый поток сохраняет новое значение баланса счета $50.

Вряд ли банку "понравится" такое программирование.

Можно подумать, что такая ситуация маловероятна. И пусть довольно редко, но она может возникнуть. В конце концов она обязательно произойдет. Поэтому синхронизация просто необходима, и с ее помощью решается задача доступа к общим данным.

Библиотека `Swing` работает с синхронизацией весьма прямолинейно. В пакете `Swing` только один поток может работать с данными, используемыми компонентом `Swing`. Если другой поток попробует сделать что-либо с данными, то ему это просто не будет позволено. Но `Swing` обладает методами `SwingUtilities.invokeLater()` и `SwingUtilities.invokeAndWait()`, с их помощью доступ может быть отложен на некоторое время и осуществлен позже. Именно такой тип синхронизации используется в классах `ChatSimulation`, `ChatWindow`, `BrokeredChat`.

Во многих случаях метод синхронизации, используемый в `Swing`, оказывается неприемлемым. Существует более общий способ синхронизации в целях контроля за доступом к общим данным. Для этого используется инструкция `synchronized`. Например:

```
synchronized (<object-reference>) {
  <statements>
}
```

В нашем случае можно было написать следующее:

```
synchronized(account) {
  if (account.getBalance() >= amount)
    balance = balance – amount;
}
```

Идея состоит в том, что `<object-reference>` становится как бы запертым на время выполнения инструкций, помещенных в блок `synchronized`. В нашем случае все методы объекта `account` будут блокированы до тех пор, пока весь блок `synchronized` не будет выполнен. Это значит, что два потока не смогут одновременно обратиться к банковскому счету.

Один и тот же объект может быть использован в нескольких блоках `synchronized`, но только один из блоков инструкций будет выполняться в каждый момент времени. Программа `ConnectionBroker` работает с потоками, которые имеют доступ к одним и тем же данным, например, к списку участников чата, который представлен в виде вектора с именем `clientList`. Этот вектор используется многими потоками. С помощью нижеприведенного кода (листинг 2.15) происходит обращение к списку пользователей `clientList` (синхронизация выделена полужирным шрифтом).

Листинг 2.15. Пример использования синхронизации

```
/*
Синхронизация доступа к данным типа Vector с именем clientList.
Список содержит участников чата. Синхронизация также защищает
переменную nextClientInfo.
*/
static void addClient(Client client) {
// добавляем нового клиента в clientList
synchronized(clientList) {
  client.ID = nextClientID++;
  if (client.info.length() == 0)
    client.info = "Anonymous" + client.ID;
    clientList.addElement(client);
  }
  System.out.println("Added client " + client.ID + " " + client.info);
}
static void removeClient(Client client) {
  // удаляем клиента из списка clientList
  synchronized(clientList) {
    clientList.removeElement(client);
  }
  System.out.println("Removed client " + client.ID);
}
static Client getClient(int ID) {
  // Удаляем клиента из списка. Если клиент обладает указанным
  // ID, возвращает удаляемого клиента. В противном случае возвращается
  // значение null.
  synchronized(clientList) {
    for (int i = 0; i < clientList.size(); i++) {
      Client c = (Client)clientList.elementAt(i);
      if (c.ID == ID) {
        clientList.removeElementAt(i);
        System.out.println("Removed client " + c.ID);
        c.ID = 0;  // Since this client is no longer waiting!
        return c;
      }
      return null;
```

```
    }
  }
  static Client[] getClients() {
    // Возвращает массив клиентов из списка
    // clientList.  Если никого нет — возвращает null.
    synchronized(clientList) {
      if (clientList.size() == 0)
        return null;
      Client[] clients = new Client[ clientList.size() ];
      for (int i = 0; i < clientList.size(); i++)
        clients[i] = (Client)clientList.elementAt(i);
      return clients;
    }
  }
```

Таким образом, синхронизация часто бывает весьма полезной. С ее помощью мы имеем возможность защитить ресурсы от вероятного некорректного использования.

Глава 3

Серверные страницы Java

Ауффенберга я не читал. Полагаю, что он напоминает Арленкура, которого я тоже не читал.

Г. Гейне. Мысли, заметки, афоризмы

Серверные страницы Java (JSP, Java Server Pages) — это технология создания серверных приложений на языке Java, предназначенная для работы с Web-сервером. Чтобы иметь возможность работать с серверными страницами Java, необходимо установить Web-сервер, поддерживающий серверные страницы Java. Существует богатый выбор таких серверов. Можно остановиться, например, на сервере Blazix. Его можно найти по адресу **http://blazix.com**.

Установка сервера не представляет трудностей. Однако если на вашем компьютере уже установлен какой-либо Web-сервер, то необходимо позаботиться о том, чтобы новый сервер работал на другом порте. По умолчанию сервер прослушивает порт 80. Конфигурация сервера описывается в файле Web.ini, расположенном в основном каталоге сервера. Пусть сервер прослушивает порт 81. В этом случае файл Web.ini должен содержать строчку

```
server.port: 81
```

Запускаем сервер (**Start** | **Programs** | **Blazix** | **Blazix Web Server**), во время старта сервер читает файл Web.ini. В консольной панели сервера выводится сообщение о состоянии сервера (рис. 3.1).

Сейчас мы можем обратиться к серверу из браузера, указав адрес **http://localhost:81** (рис. 3.2).

Основным каталогом документов, доступных серверу, по умолчанию является каталог %HOME%\Blazix\Webfiles (где %HOME% — место, куда был установлен Blazix). Чтобы протестировать сервер, создадим простой файл test.html, который поместим в этот каталог (листинг 3.1).

Листинг 3.1. Файл test.html

```
<HTML>
  <HEAD>
  <title>Simple test.</title>
```

```
  </HEAD>
  <BODY>
  <h1>
    Это простой тест
  </h1>
  </BODY>
</HTML>
```

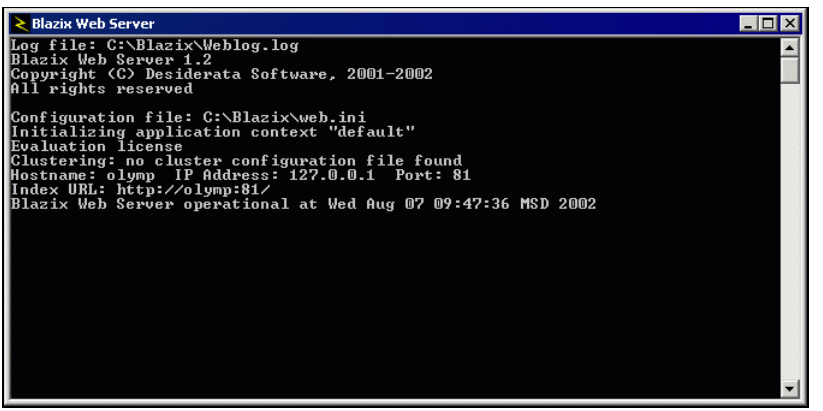


Рис. 3.1. Состояние сервера Blazix

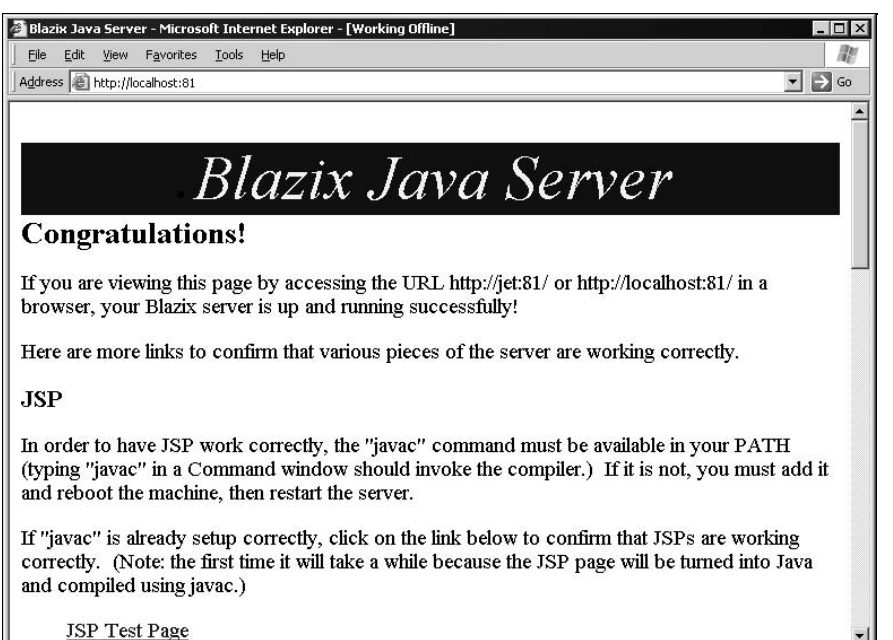


Рис. 3.2. Обращение к работающему Web-серверу

Обратимся к файлу по адресу **http://localhost:81/test.html**, в окне браузера появится сообщение (рис. 3.3).

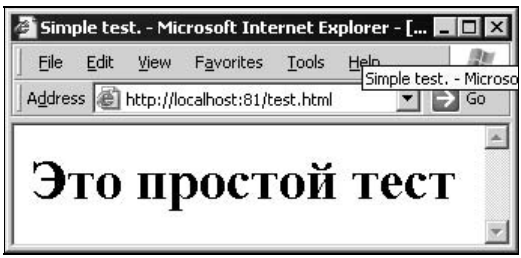


Рис. 3.3. Обращение к тестовой HTML-странице

3.1. Создаем первую серверную страничку

Поскольку мы будем работать с Java-приложениями, необходимо убедиться в том, что все необходимые переменные окружения заданы, в частности, переменная `CLASSPATH`.

Создадим наш первый простой JSP-файл. Возьмем файл test.html и немного изменим его. Результат сохраним в файле с расширением jsp (листинг 3.2).

Листинг 3.2. Файл test.jsp

```
<HTML>
<head>
<title>Simple test.</title>
</head>
<body>
<h1>
Just simplest jsp test
</h1>
Current time is
<font size=+2>
<%= new java.util.Date() %>
</font>
</BODY>
</HTML>
```

Обратимся к файлу, указав адрес **http://localhost:81/test.jsp** (рис. 3.4), при первом обращении происходит компиляция файла компилятором javac. Это может занять некоторое время.

Рис. 3.4. Обращение к JSP-странице

Скомпилированный Java-класс хранится в каталоге %HOME%\Blazix\jspfiles, при повторном обращении к JSP-файлу компиляция не потребуется, при условии, что не были внесены изменения в исходный текст JSP-файла.

3.2. Скриптлеты

Внешний признак скриптлета — обрамляющие скриптлет символы <%= и %>. Мы уже встречались с JSP-вставками в HTML-файл, когда при помощи символов <%= и %> осуществляет вставку выражения в поток HTML-кода. Не всегда оказывается удобно вставлять Java-выражения в HTML-текст. Для создания блоков программ Java в HTML-странице удобно пользоваться скриптлетами. Скриптлет сам по себе не должен генерировать HTML-код, хотя, конечно, он может это делать. *Скриптлет* — это кусок Java-программы, встроенной в HTML-страницу. Изменим предыдущий файл test1.jsp и вставим в него скриптлет, который будет выводить некоторый текст (рис. 3.5).

Листинг 3.3. Файл test1.jsp

```
<HTML>
<head>
<title>Simple test.</title>
</head>
<body>
<h1>
Just simplest jsp test
</h1>
Current time is
<font size=+2>
```

```
<%= new java.util.Date() %>
</font>

<HTML> <BODY>
<%
// переменная "date", описанная в этом скриптлете,
// доступна в выражении, которое приводится ниже
System.out.println("Evaluating date now");
java.util.Date date = new java.util.Date();
%>
Hello! The time is now <%= date %>
</BODY>
 </HTML>
</BODY>
</HTML>
```

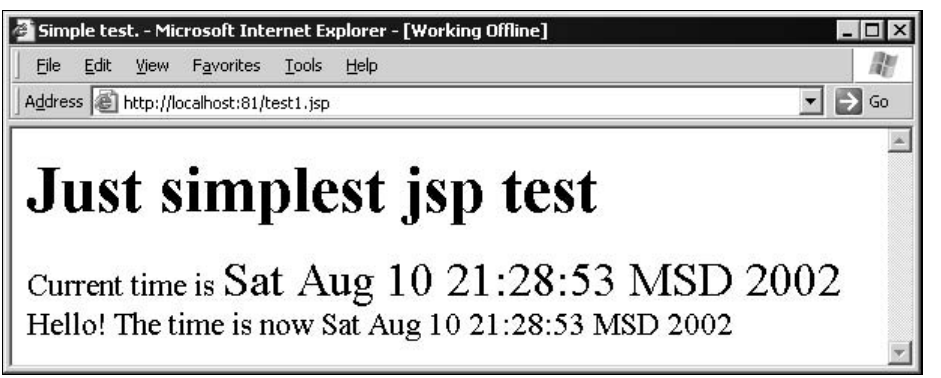


Рис. 3.5. Работа со скриптлетами

Сам по себе скриптлет не создает никакого HTML-текста. Чтобы вывести HTML-сообщение, нужно специально об этом позаботиться, например, так, как это сделано в файле jsp4.jsp (рис. 3.6).

Листинг 3.4. Файл jsp4.jsp

```
<HTML>
<BODY>
<%
// создаем переменную "date"
System.out.println("Evaluating date now");
```

```
java.util.Date date = new java.util.Date();
%>
Hello! The time is now
<%
// генерируем текст HTML
out.println(String.valueOf(date));
%>
</BODY>
</HTML>
```

Рис. 3.6. Выведенное HTML-сообщение

Заметьте, что в окне сервера будут выведены соответствующие сообщения о компиляции файла jsp4.jsp (рис. 3.7).

```
System.out.println("Evaluating date now");
```

Рис. 3.7. Консоль сервера отражает текущее состояние сервера

Скриптлеты и HTML

В предыдущем примере мы использовали метод `out` для того, чтобы выводить HTML-текст. При программировании JSP вряд ли всегда будет удобно выводить HTML-текст средствами потока вывода `out`. Допустим, нам нужно вывести результат поиска по базе данных. Это удобно сделать в виде HTML-таблицы. Для этого вовсе не обязательно использовать `out`. Более удобное решение показано в файле jsp1.jsp (листинг 3.5).

Листинг 3.5. Файл jsp1.jsp

```
<html>
<Head>
<Title>HTML and TABLE</title>
<h1>Generating table on the fly.</h1>
<body>
<TABLE BORDER=2 width=70%>
<%
 for (int i = 0; i <20; i++)
{
%>
 <TR>
<TD>Number</TD>
<TD><%= i+1 %></TD>
 </TR>
 <%
}
 %>
</TABLE>
</body>
</html>
```

Как будет выглядеть такая таблица, видно на рис. 3.8.

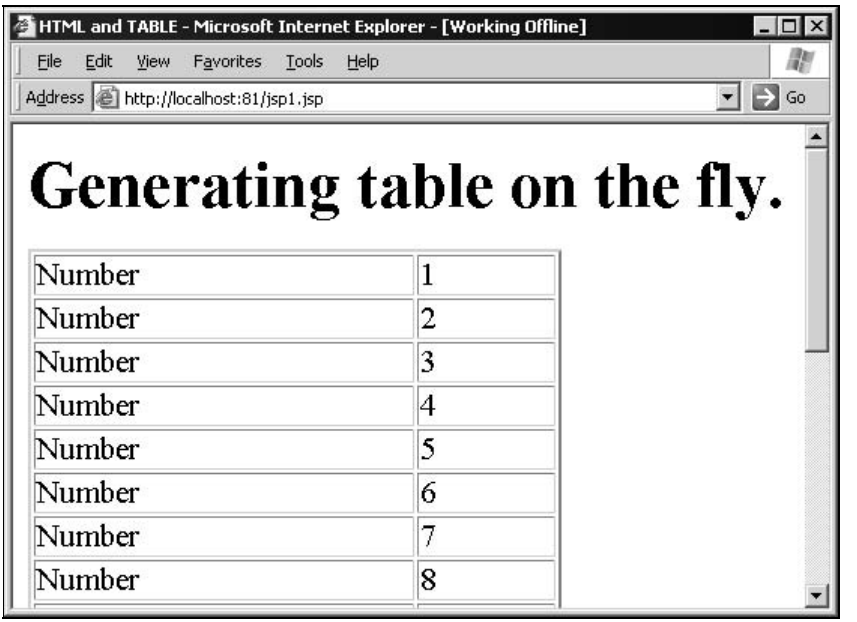


Рис. 3.8. Скриптлет и HTML при создании таблицы

3.3. Директивы JSP

Директивы JSP используются для вставки существующих фрагментов кода в текущий текст, они напоминают директивы препроцессора. Директивы помещаются между знаками `<%@` и `%>`. Пример приведен в файле jsp2.jsp (листинг 3.6).

Листинг 3.6. Файл jsp2.jsp

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%
  System.out.println("Evaluating date now");
  Date date = new Date();
%>
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

Кроме вставки уже существующих фрагментов кода директивы могут быть использованы для импортирования файлов (листинг 3.7). Результат обращения к этой странице показан на рис. 3.9.

Листинг 3.7. Файл jsp3.jsp

```
<HTML>
<BODY>
Going to include jsp1.jsp...
<BR>
<%@ include file="jsp2.jsp" %>
</BODY>
</HTML>
```

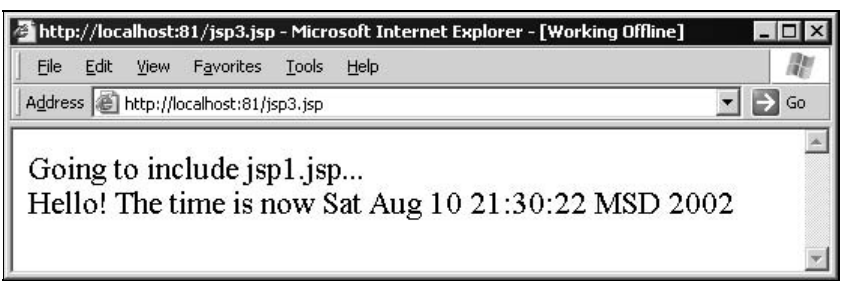


Рис. 3.9. Применение директив JSP

Декларации в JSP

В JSP существует специальный ярлык, внутри которого описываются новые типы данных. Объявления типов и описания методов производятся между знаками `<%!` и `%>`. Например, описание методов может быть следующим (листинг 3.8):

Листинг 3.8. Файл jsp4.jsp

```
<BODY>
<%!
Date theDate = new Date();
Date getDate()
{
System.out.println("In getDate() method");
return theDate;
```

```
}
%>
Hello! The time is now <%= getDate() %>
</BODY>
</HTML>
```

Эта страница приводит к отображению привычной картинки в браузере, как это видно из рис. 3.10.

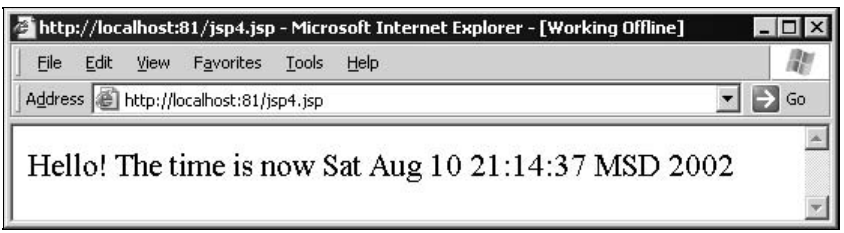


Рис. 3.10. Пример объявления типа данных в декларации JSP

3.4. Ярлыки JSP

В JSP существует возможность создания пользовательских ярлыков, т. е. таких ярлыков, которые определяются разработчиком самостоятельно. Однако существуют и заранее определенные ярлыки. Ярлыки JSP очень похожи на ярлыки HTML или XML. Ярлык бывает открывающим. Он начинается с символа открывающей угловой скобки, за которым следует набор букв с разделяющим двоеточием и закрывающая угловая скобка. Закрывающий ярлык после открывающей угловой скобки содержит знак косой черты. Между открывающим ярлыком и закрывающим ярлыком находится тело элемента, описываемого этим ярлыком. Если элемент не имеет тела, то открывающий и закрывающий ярлыки могут быть совмещены, например `<nechto/>`. Общий вид элемента с ярлыками такой

```
<some:tag>
body
</some:tag>
```

Заранее определенные ярлыки начинаются с последовательности символов `<jsp:`, например:

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<jsp:include page="hello.jsp"/>
</BODY>
</HTML>
```

3.5. JSP и работа с сессиями

При переходе от одной страницы к другой часто оказывается полезно сохранять информацию о пользователе. Как это можно сделать? Здесь полезно работать с сессией. Можно автоматически сгенерировать идентификатор сессии и передавать его наподобие cookies (фрагменты информации, которая передается в составе HTTP-запросов), также можно получить имя от пользователя через HTML-форму. Создадим простую форму, сохранив ее в файле ses.html (листинг 3.9).

Листинг 3.9. Файл ses.html

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="Name.jsp">
Input your name
<INPUT TYPE=TEXT NAME=username SIZE=20>
<BR> Input your e-mail address
<INPUT TYPE=TEXT NAME=email SIZE=20>
<BR> Type your age
<INPUT TYPE=TEXT NAME=age SIZE=4>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Вводим значения, соблюдая типы, как показано на рис. 3.11.

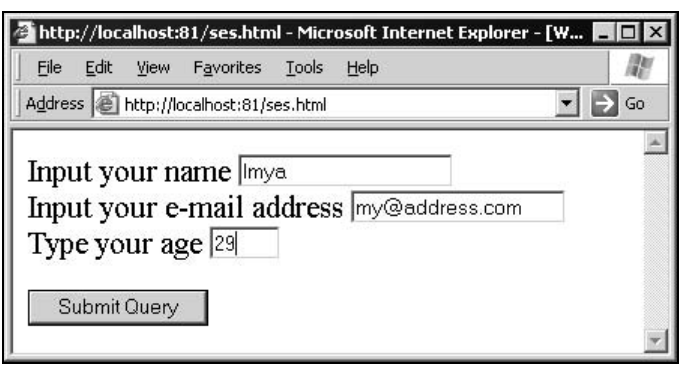


Рис. 3.11. Заполнение формы

Этот файл будет обрабатываться скриптом Name.jsp (листинг 3.10).

Листинг 3.10. Файл Name.jsp

```
<jsp:useBean id="user" class="UserData" scope="session"/>
<jsp:setProperty name="user" property="*"/>
<HTML>
<BODY>
<A HREF="Next.jsp">Sleduyuschaya stranitsa</A>
</BODY>
</HTML>
```

Данные, занесенные в форму, передадутся в файл Name.jsp. Если файл Name.jsp существует, то после нажатия кнопки **Submit Query** должна появиться страница со ссылкой на следующую страницу Name.jsp.

Листинг 3.11. Файл Name.jsp

```
<jsp:useBean id="user" class="UserData" scope="session"/>
<HTML>
<BODY> My vvodili<BR>
Imya:
<%= user.getUsername() %>
<BR> Email:
<%= user.getEmail() %>
<BR> Vozrast:
<%= user.getAge() %>
<BR>
</BODY>
</HTML>
```

На рис. 3.12 показан ответ, полученный из скрипта Next.jsp (листинг 3.11). Обратите внимание на то, что мы видим те же значения, что вводили ранее.

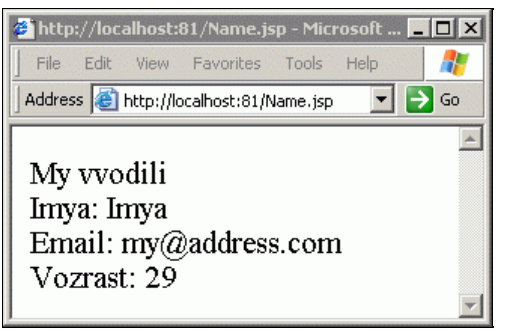


Рис. 3.12. Пример вывода на экран переменных, сохраненных как переменные сессии

3.6. Обработка форм с использованием компонентов Beans

JSP предоставляет удобный механизм получения значений для имен HTML-форм. В предыдущем примере мы столкнулись с необходимостью обработки формы. Для этого удобно применять компоненты Beans. Рассмотрим эту технику на примере. Создадим HTML-форму и запишем файл form.html (листинг 3.12).

Листинг 3.12. Файл form.html

```
<HTML>
  <BODY>
   <FORM METHOD=POST ACTION="NameBean1.jsp">
     Vvedite imya
     <INPUT TYPE=TEXT NAME=username SIZE=20>
     <BR>
     Vvedite e-mail
     <INPUT TYPE=TEXT NAME=email SIZE=20>
     <BR>
     vvedite vozrast
     <INPUT TYPE=TEXT NAME=age SIZE=4>
     <P><INPUT TYPE=SUBMIT>
    </FORM>
  </BODY>
</HTML>
```

Введем в поля формы требуемую информацию, как на рис. 3.13.

Рис. 3.13. Заполнение формы

В форме мы задали три имени для различных элементов: `username`, `email`, `age`. Создадим класс `Data` (листинг 3.13).

Листинг 3.13. Файл Data.java

```
public class Data {
  String username;
  String email;
  int age;

  public void setUsername(String value)
  {
    username = value;
  }
  public void setEmail(String value)
  {
    email = value;
  }
  public void setAge(int value)
  {
    age = value;
  }
  public String getUsername()
  {
    return username;
  }
  public String getEmail()
  {
    return email;
  }
  public int getAge()
  {
    return age;
  }
}
```

В этом классе описаны методы `get` и `set` для каждого элемента формы на основе имени этого элемента. Необходимо скомпилировать созданный класс и убедиться в том, что папка, в которой он находится, указана в переменной окружения `CLASSPATH`.

После этого перейдем к созданию файла NameBean.jsp (листинг 3.14), в нем будет указан класс `Data`.

Листинг 3.14. Файл NameBean.jsp

```
<jsp:useBean id="user" class="Data" scope="session"/>
<jsp:setProperty name="user" property="*"/>
<HTML>
  <BODY>
  <A HREF="Next1.jsp">Continue</A>
  </BODY>
</HTML>
```

Наконец, внесем необходимые изменения в файл Next1.jsp (листинг 3.15).

Листинг 3.15. Файл Next1.jsp

```
<jsp:useBean id="user" class="Data" scope="session"/>
<HTML>
  <BODY>
    You entered<BR>
    Name: <%= user.getUsername() %><BR>
    Email: <%= user.getEmail() %><BR>
    Age: <%= user.getAge() %><BR>
  </BODY>
</HTML>
```

При переходе по ссылке открывается файл Next1.jsp. На рис. 3.14 отображены заданные значения полей. Таким образом, данная форма была обработана компонентом Bean.

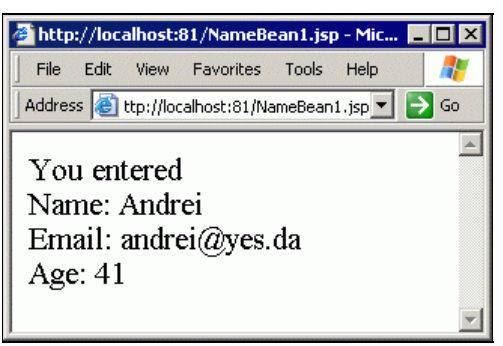


Рис. 3.14. Отображение значений

3.7. Библиотеки ярлыков

Далее в этой книге мы научимся создавать пользовательские ярлыки и библиотеки ярлыков. Этот раздел посвящен рассмотрению простого примера использования ярлыков из библиотеки ярлыков сервера Blazix. Для того чтобы включить эту библиотеку в JSP-страницу, используется следующая директива:

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
```

Префикс `blx` обозначает, что все ярлыки этой библиотеки будут начинаться с `blx`. Создадим файл NameBlx.jsp (листинг 3.16).

Листинг 3.16. Файл NameBlx.jsp

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
<jsp:useBean id="user" class="UserData" scope="session"/>
<HTML>
  <BODY>
    <blx:getProperty name="user" property="*">
      <FORM METHOD=POST ACTION="NameBean.jsp">
        What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20><BR>
        What's your e-mail address? <INPUT TYPE=TEXT NAME=email
                                          SIZE=20><BR>
        What's your age? <INPUT TYPE=TEXT NAME=age SIZE=4>
        <P><INPUT TYPE=SUBMIT>
      </FORM>
    </blx:getProperty>
  </BODY>
</HTML>
```

В библиотеке `blx` описан ярлык `blx:getProperty`, который мы использовали в этом файле. Полезно добавить средства обработки ошибок в страницу Next1.jsp:

```
 <%@ taglib prefix="blx" uri="/blx.tld" %>
<%!
  boolean haveError;
  StringBuffer errors;

  public void errorHandler(String field,
                String value,
                Exception ex)
```

```
  {
    haveError = true;
    if (errors == null)
      errors = new StringBuffer();
    else
      errors.append("<P>");
    errors.append("<P>Value for field \"" +
           field + "\" is invalid.");
    if (ex instanceof java.lang.NumberFormatException)
      errors.append(" The value must be a number.");
  }
%>
<%
  // переменные инициализируются вне фрагмента объявления
  haveError = false;
  errors = null;
%>
<HTML>
  <BODY>
    <jsp:useBean id="user" class="Data" scope="session"/>
      <blx:setProperty name="user"
         property="*"
         onError="errorHandler"/>
<%
  if (haveError) {
    out.println(errors.toString());
    pageContext.include("GetName.jsp");
  } else
    pageContext.forward("NextPage.jsp");
%>
  </BODY>
</HTML>
```

Существует множество библиотек ярлыков. Но пользователи также имеют возможность создавать собственные ярлыки.

3.8. Отправка почты средствами JSP

Для того чтобы отправить почту средствами JSP, необходимо указать адрес почтового сервера. Предположим, что он нам известен и называется **mail.server.com**. Создадим HTML-страничку mail.html (листинг 3.17).

Листинг 3.17. Файл mail.html

```
<HTML>
  <BODY>
    <FORM METHOD=POST ACTION="Mail.jsp">
      Vvedite imya: <INPUT TYPE=TEXT NAME=username SIZE=20><BR>
      Vvedite adres e-mail: <INPUT TYPE=TEXT NAME=email SIZE=20><BR>
      <P><INPUT TYPE=SUBMIT>
    </FORM>
  </BODY>
</HTML>
```

Далее необходимо создать файл Mail.jsp (листинг 3.18), а также файл c:\mail.txt. Этот файл будет послан в виде приложения.

Листинг 3.18. Файл Mail.jsp.

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
<HTML>
  <BODY>
  <%
  // имя пользователя
  String email = request.getParameter("email");
  %>
  <% if (email == null || email.equals("")) { %>
  Please enter an email address.
  <% } else { %>
     <blx:email host="mail.server.com" from="ya@moiserver.com">
       <blx:emailTo><%= email %></blx:emailTo>
       Vashe imya: <%= request.getParameter("username") %>
       Va zaregistrirovany <%= new java.util.Date() %>
       Attached, please find a contents file.
       <blx:emailAttach file="C:\\mail.txt"
        contentType="text/plain" name="mail.txt/>
     </blx:email>
    <!--  прочий HTML-текст -->
    Thank you.  A confirmation email has been sent to <%= email %>
    <% } %>
  </BODY>
</HTML>
```

3.9. Создание JSP-ярлыков

Для того чтобы создать свой собственный новый JSP-ярлык, необходимо создать обработчик для этого ярлыка, то есть Java-классы, в которых будут описаны методы обработки содержимого, приводимого в ярлыке на JSP-странице. Полезно также создать подсказку для данного ярлыка. После того как классы обработки ярлыка созданы, нужно объявить ярлык, то есть сделать его "видимым" в библиотеке ярлыков.

3.9.1. Классы обработки ярлыка

Обработчик ярлыка — это объект, при помощи которого происходит обработка содержимого элемента, описанного при помощи JSP-ярлыка при анализе JSP-страницы сервером, которая осуществляется в соответствии со ссылками, заданными в библиотеке ярлыков. Обработчик ярлыка должен имплементировать интерфейс `Tag` (пакет `javax.servlet.jsp.tagext`) либо `BodyTag` (из того же пакета). Эти пакеты позволяют использовать уже существующие объекты Java и делать их обработчиками JSP-ярлыков. Для создания новых (еще не существующих) обработчиков ярлыков следует использовать классы `TagSupport` и `BodyTagSupport` пакета `javax.servlet.jsp.tagext`.

Когда на JSP-странице встречается созданный нами ярлык, сервлет, обрабатывающий эту страницу, вызывает методы, инициализирующие соответсвующий ярлыку обработчик, а затем вызывает метод `doStartTag` обработчика ярлыка. Как только появляется закрывающий ярлык, сервлет вызывает метод `doEndTag`. В ходе выполнения программы могут быть вызваны другие методы, в зависимости от того, что содержится в элементе, описанном ярлыком, и как построен обработчик данного ярлыка. В процессе создания обработчика ярлыка необходимо имплементировать методы, которые приведены в табл. 3.1. Эти методы часто оказываются полезными при работе с ярлыками.

Обработчик ярлыка содержит возможности, которые позволяют ему осуществлять связь с JSP-страницей. Для этого существует класс `PageContext` пакета `javax.servlet.jsp`. Этот класс помогает получить все объекты, доступные из JSP-страницы. Объекты могут иметь атрибуты, которые обладают именами. Имена можно задать и получить при помощи методов `setAttributes` и `getAttributes`. Если элемент, описанный при помощи родительского ярлыка, содержит дочерний элемент, то обработчик дочернего ярлыка имеет доступ к обработчику родительского ярлыка.

Таблица 3.1. *Методы обработчиков ярлыка*

Тип обработчика ярлыка	Методы
Simple — простой	`doStartTag`, `doEndTag`, `release`
Attributes — атрибуты	`doStartTag`, `doEndTag`, `set/getAttribute1...`*N*, `release`

Таблица 3.1 *(окончание)*

Тип обработчика ярлыка	Методы
Body, evaluation and no interaction — тело ярлыка, не взаимодействующий с телом	`doStartTag`, `doEndTag`, `release`
Body, iterative evaluation — тело ярлыка с итерацией	`doStartTag`, `doAfterBody`, `doEndTag`, `release`
Body, interaction — тело, взаимодействующий с телом	`doStartTag`, `doEndTag`, `release`, `doInitBody`, `doAfterBody`, `release`

Далее в этой главе будут подробно рассмотрены приведенные в таблице типы обработчиков.

3.9.2. Описатель ярлыка

Описатель ярлыка (TLD — Tag Library Descriptor) представляет собой XML-документ, в котором приведено описание связанной с данным ярлыком библиотеки. Точнее, описатель содержит не описание ярлыка как такового, а описание всей библиотеки в целом, а также описания отдельных ярлыков, содержащихся в библиотеке. Описатель ярлыка в узком смысле — это описание данного конкретного ярлыка внутри XML-документа, внутри описателя библиотеки, к которой относится данный ярлык. Имена файлов описателей библиотек имеют расширение tld. В серверах фирмы Sun файлы tld хранятся в каталоге WEB-INF и в его подкаталогах, TLD-файл может быть упакован в архив WAR (Web ARchive). Файл TLD должен начинаться со стандартной строки описания версии XML и ссылки на файл DTD описания типа документа:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

Основным элементом, который содержится в файле TLD, является элемент `taglib`. Этот элемент содержит дочерние элементы, список которых приведен в табл. 3.2.

Таблица 3.2. *Элементы, дочерние для элемента* `taglib`

Элемент	Описание
`tlib-version`	номер версии библиотеки
`jsp-version`	номер версии спецификации JSP

Таблица 3.2 (окончание)

Элемент	Описание
Short-name	необязательное имя
Uri	адрес библиотеки URI
Display-name	необязательное имя (для отображения)
small-icon	необязательная иконка (маленькая)
Large-icon	необязательная иконка (большая)
Description	необязательное описание ярлыка
Listener	элемент listener
Tag	элемент tag

В качестве примера приведем фрагмент из файла описания библиотеки ярлыков сервера Blazix (листинг 3.19).

Листинг 3.19. Файл описания библиотеки ярлыков сервера Blazix

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "web-jsptaglib_1_1.dtd">

<!-- built-in tag library descriptor for the Blazix blx library -->
<!-- Copyright (C) Desiderata Software, 2001 -->
<taglib>
  <tlibversion>1.1</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>blx</shortname>
  <info>
     Desiderata Softare Blazix Tag Library
  </info>
  <tag>
  <name>setProperty</name>
  <tagclass>desisoft.jsp.tagext.SetProp</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
Replacement for jsp:setProperty *, with better handling
of empty strings and exceptions. A function may be named via  onError
to handle errors and continue processing.  Empty values entered
```

```
by users reset the bean elements.  (Checkboxes must be specifed
in the checkbox property.)  Set stringNull to true if empty values
should reset string objects to null, false if empty values should
reset string objects to the empty string.
  </info>
  <attribute>
  <name>name</name>
  <required>true</required>
  </attribute>
  <attribute>
  <name>property</name>
  <required>true</required>
  </attribute>
  <attribute>
  <name>onError</name>
  <required>false</required>
  </attribute>
  <attribute>
  <name>stringNull</name>
  <required>false</required>
  </attribute>
  <attribute>
  <name>checkbox</name>
  <required>false</required>
  </attribute>
  <attribute>
  <name>emptyInt</name>
  <required>false</required>
  </attribute>
  </tag>
  <tag>
  <name>getProperty</name>
  <tagclass>desisoft.jsp.tagext.GetProp</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
Inverse of jsp:setProperty *, retrieves data from
bean and places in outgoing HTML.  Modifies
the INPUT, TEXTAREA and SELECT tags to insert
values from the bean.
```

```
  </info>
  <attribute>
  <name>name</name>
  <required>true</required>
  </attribute>
  <attribute>
  <name>property</name>
  <required>false</required>
  </attribute>
  <attribute>
  <name>dateFormat</name>
  <required>false</required>
  </attribute>
  <attribute>
  <name>timeFormat</name>
  <required>false</required>
  </attribute>
  <attribute>
  <name>emptyInt</name>
  <required>false</required>
  </attribute>
  </tag>
  . . .
</taglib>
```

Полностью файл описания библиотеки ярлыков сервера Blazix входит в комплект поставки, его можно получить с дистрибутивом сервера.

3.9.3. Элемент *listener*

Описание библиотеки может содержать ссылки на классы, которые рассматриваются в качестве обработчиков событий. Эти классы ожидают наступления тех или иных событий. В TLD-файле такие классы быть описаны внутри элемента `listener`. Единственный элемент, который содержится внутри элемента `listener`, является элемент `listener-class`. Элемент `listener-class` содержит имя класса обработчика события.

3.9.4. Элемент *tag*

Каждый ярлык (или тег, tag) в библиотеке ярлыков, описывается путем задания самого ярлыка (имени ярлыка) и указания класса обработчика этого

ярлыка. Кроме этого библиотека содержит информацию об атрибутах ярлыка и переменные сценариев, которые описываются либо внутри TLD, либо в дополнительном классе описания ярлыка. Каждый атрибут содержит информацию о том, является ли данный атрибут обязательным, информацию о типе атрибута, может ли атрибут задаваться в процессе обработки запроса. В табл. 3.3 приведен список элементов, которые могут быть вложены в элемент `tag`.

Таблица 3.3. *Список элементов, вкладываемых в элемент* `tag`

Элемент	Описание
`Name`	Уникальное имя ярлыка
`tag-class`	Полное имя класса обработчика ярлыка
`tei-class`	Необязателен, подкласс от javax.servlet.jsp.tagext.TagExtraInfo
`body-content`	Тип содержимого ярлыка
`Display-name`	Необязателен. Отображаемое имя
`small-icon`	Необязателен. Маленькая иконка для отображения
`large-icon`	Необязателен. Большая иконка для отображения
`Description`	Необязателен. Описание ярлыка
`Variable`	Необязателен. Информация о переменных сценариев
`Attribute`	Информация об атрибутах ярлыка

Сейчас мы перейдем к рассмотрению элементов, используемых в ярлыках различных типов.

3.9.5. Простые ярлыки — Simple Tags

Обработчики ярлыков

Обработчик простых ярлыков должен имплементировать методы `doStartTag` и `doEndTag` интерфейса `Tag`. Метод `doStartTag` вызывается тогда, когда на JSP-странице встречается соответствующий ярлык. Данный метод возвращает значение `SKIP_BODY`, так как простой ярлык не содержит внутри элемента, описываемого при помощи этого ярлыка, а значит, тело элемента может быть проигнорировано. Метод `doEndTag` вызывается тогда, когда появляется закрывающий элемент ярлык. Этот метод должен возвращать значение `EVAL_PAGE`, при этом оставшаяся JSP-страница будет обработана в обычном порядке. Если то, что осталось на странице после текущего элемента,

не требуется обрабатывать, то метод должен возвратить значение SKIP_PAGE. Например, ярлык

```
<yarlyk:simple />
```

может быть связан со следующим обработчиком ярлыка:

```
public SimpleTag extends TagSupport {
   public int doStartTag() throws JspException {
    try {
     pageContext.getOut().print("Privet vsem ot hozyaina.");
    } catch (Exception ex) {
     throw new JspTagException("SimpleTag: " +
      ex.getMessage());
    }
    return SKIP_BODY;
   }
   public int doEndTag() {
    return EVAL_PAGE;
   }
}
```

Здесь класс SimpleTag содержит два необходимых метода.

Элемент body-content

Элементы, которые не содержат тела элемента, должны быть описаны при помощи ярлыка body-content следующим образом:

```
<body-content>empty</body-content>
```

3.9.6. Ярлыки с атрибутами

Описание атрибутов в обработчиках элементов

В соответствии с архитектурой JavaBeans, каждому атрибуту ярлыка должны быть сопоставлены соответствующие методы get и set в обработчике ярлыка, например

```
<yarlyk:svojstvo parameter="Clear">
```

может быть сопоставлен следующему обработчику ярлыка:

```
protected String parameter = null;
public String getParameter() {
   return (this.parameter);
}
public void setParameter(String parameter) {
```

```
    this.parameter = parameter;
}
```

Если же атрибут имеет имя `id`, а обработчик ярлыка основан на классе `TagSupport`, то нет необходимости описывать свойства и задавать методы, поскольку они уже описаны в `TagSupport`.

Элемент *attribute*

Для каждого атрибута ярлыка необходимо указывать, является ли данный атрибут обязательным. Определяется, может ли значение атрибута быть задано в виде выражения, а также (необязательно) задается тип атрибута. Все эти описания вкладываются внутрь элемента `attribute`. Для статических элементов тип ярлыка всегда указывается в виде `java.lang.String`. Если внутри элемента `rtexprvalue` записано `true` или `yes`, то типом, возвращаемым данным элементом, может быть любой доступный тип. Общая структура элемента `attribute` имеет следующий вид:

```
<attribute>
    <name>attr1</name>
    <required>true|false|yes|no</required>
    <rtexprvalue>true|false|yes|no</rtexprvalue>
    <type>полное название типа</type>
</attribute>
```

Если тот или иной атрибут не является обязательным, то обработчик ярлыка должен предоставлять значение этого атрибута, задаваемое по умолчанию. Например, если для ярлыка `yarlyk:nechto` описано, что атрибут `parameter` не является обязательным, то можно использовать следующий код:

```
<tag>
    <name>present</name>
    <tag-class>classname</tag-class>
    <body-content>JSP</body-content>
    ...
    <attribute>
     <name>parameter</name>
     <required>false</required>
     <rtexprvalue>true</rtexprvalue>
    </attribute>
    ...
</tag>
```

При описании библиотеки для каждого атрибута ярлыка должны быть указаны разрешенные типы. При обработке страницы JSP будут наложены соответствующие ограничения согласно указанным в описании библиотеки. Атрибуты, передаваемые ярлыку, должны быть также проверены с использованием метода `isValid` в классе, производном от `TagExtraInfo`.

Например, пусть ярлык `<yarlyk:whatever attr1="value1"/>` содержит в себе атрибут со следующим описанием:

```
<attribute>
   <name>attr1</name>
   <required>true</required>
   <rtexprvalue>true</a>
</attribute>
```

Это описание сообщает, что значение атрибута `attr1` может быть задано во время выполнения программы. Метод `isValid` проверяет, является ли значение атрибута `attr1` логическим. В приведенном ниже коде производится также проверка того, указал ли пользователь, что будет использовано значение, получаемое в ходе выполнения программы.

```
public class CLASSNAME extends TagExtraInfo {
   public boolean isValid(Tagdata data) {
    Object o = data.getAttribute("attr1");
    if (o != null && o != TagData.REQUEST_TIME_VALUE) {
     if (o.toLowerCase().equals("true") ||
      o.toLowerCase().equals("false"))
      return true;
     else
      return false;
    }
    else
     return true;
   }
}
```

3.9.7. Элементы, содержащие тело элемента

Обработчики ярлыков

Несколько иначе должен выглядеть обработчик ярлыка для элемента, который содержит его тело. Конструкция обработчика в значительной степени зависит от того, будет ли обработчик ярлыка взаимодействовать с телом ярлыка, то есть будет ли обработчик ярлыка читать и изменять тело ярлыка.

Обработчики ярлыков без взаимодействия с телом

Если обработчик ярлыка не взаимодействует с телом элемента, описываемого при помощи этого ярлыка, то обработчик создается на основе интерфейса `Tag` или как класс, производный от `TagSupport`. Когда необходимо взаимодействие с телом элемента, то метод `doStartTag` должен возвращать значение `EVAL_BODY_INCLUDE`. Если обработка тела не требуется, этот метод возвращает значение `SKIP_BODY`.

При необходимости итеративной обработки содержимого элемента, обработчик имплементирует интерфейс `IterationTag` или же создается на основе `TagSupport`. Если в процессе обработки тела элемента будет установлено, что требуется повторная обработка тела, то методы `doStartTag` и `doAfterBody` должны возвратить значение `EVAL_BODY_AGAIN`.

Обработчики ярлыков со взаимодействием с телом

Если существует необходимость осуществлять взаимодействие между обработчиком ярлыка и телом элемента, то обработчик ярлыка имплементирует `BodyTag` или создается на основе `BodyTagSupport`. При этом имплементируются методы `doInitBody` и `doAfterBody`. Эти методы осуществляют взаимодействие с телом элемента. Для работы с телом элемента используется несколько методов, например, методы `getString` и `getReader`. С помощью этих методов можно извлечь информацию из тела элемента, а метод `writeOut` (`out`) будет полезен тогда, когда необходимо вывести полученную информацию в поток вывода. Вместе с методом `writeOut` удобно использовать метод `getPreviousOut` (для определения аргумента, передаваемого методу `writeOut`). Важно иметь в виду, что тогда, когда необходимо производить обработку тела элемента, метод `doStartTag` должен возвратить значение `EVAL_BODY_BUFFERED`. (Если обработка тела не требуется, то этот метод возвращает `SKIP_BODY`.)

Рассмотрим прочие методы, используемые в обработчике ярлыка, взаимодействующем с телом элемента, описываемого данным ярлыком.

- Метод `doInitBody`.

  Метод `doInitBody` вызывается до того, как тело элемента будет проанализировано. Метод используется для инициализации всех необходимых параметров.

- Метод `doAfterBody`.

  Метод `doAfterBody` вызывается после того, как будет проанализировано тело элемента. Этот метод возвращает значение, которое информирует о том, требуется ли последующая обработка. При необходимости повторной обработки тела элемента (например, если мы используем ярлык-итератор), этот метод возвращает значение `EVAL_BODY_BUFFERED`. Если же повторная обработка не нужна, то метод должен возвращать значение `SKIP_BODY`.

- Метод `release`.

  Обработчик ярлыка должен после работы высвободить все ресурсы. Это делается с использованием метода `release`.

Ниже приводится пример обработчика ярлыка (листинг 3.20), описывающего элемент, содержащий тело. Обработчик читает содержимое тела элемента (в нашем случае тело содержит SQL-запрос) и передает то, что было прочитано, объекту (этот объект может, например, выполнить данный запрос). После того как запрос прочитан, повторное чтение тела не потребуется, поэтому метод `doAfterBody` возвращает значение `SKIP_BODY`.

Листинг 3.20. Пример обработчика ярлыка со взаимодействием с телом элемента

```
public class QueryTag extends BodyTagSupport {
   public int doAfterBody() throws JspTagException {
    BodyContent telo = getBodyContent();
    // получаем тело telo в виде строки
    String strokazaprosa = telo.getString();
    // очистка
    telo.clearBody();
    try {
     Statement zapros = connection.createStatement();
     result = zapros.executeQuery(strokazaprosa);
    } catch (SQLException e) {
     throw new JspTagException("QueryTag: " +
       e.getMessage());
    }
    return SKIP_BODY;
   }
}
```

Элемент *body-content*

Если элемент содержит тело элемента, то в файле описания библиотеки необходимо указать тип содержимого этого элемента с использованием ярлыка `body-content`:

```
<body-content>JSP|tagdependent</body-content>
```

Если содержимое элемента состоит из пользовательских ярлыков, заранее определенных ярлыков и HTML-текста, то такое содержимое считается содержимым JSP. Прочее содержимое, в том числе SQL-запросы, представляет собой содержимое, описание которого зависит от конкретного вида ярлыка,

то есть является tagdependent. Этот ярлык не влияет на то, как будет обработано содержимое. Он служит для отображения типа содержимого элемента.

3.9.8. Ярлыки описания переменных сценариев

Обработчики ярлыков

Обработчик ярлыков используются также и для того, чтобы создавать объекты, доступные из JSP-страницы, к которым можно передавать переменные сценариев. При этом часто используется метод pageContext.setAttribute (name, value, scope) или метод pageContext.setAttribute(name, value). Атрибут, передаваемый ярлыку, содержит имя переменной объекта. Это имя может быть получено при помощи метода get. Значение переменной объекта может зависеть от самого содержимого, которое можно получить при помощи метода pageContext.getAttribute(name, scope). Иными словами, обработчик ярлыка получает значение переменной и обрабатывает ее определенным образом. А затем задает значение переменной с использованием метода pageContext.setAttribute(name, object). В табл. 3.4 приведены поля доступности переменных для различных объектов.

Таблица 3.4. *Области видимости объектов*

Имя	Откуда доступно	Срок жизни
Page	текущая страница	пока не отправлен ответ пользователю или не отправлен запрос на новую страницу
Request	текущая страница, а также страницы, на которые есть ссылки, как на вставки, или страницы, на которые производится переадресация	пока не отправлен ответ пользователю
Session	текущий запрос или подзапрос из того же браузера	пока не истек срок действия пользовательской сессии
Application	текущий и последующий запрос из того же самого Web-приложения	в течение срока жизни приложения

Информация о переменных сценария

Приведем пример, в котором используется переменная сценария peremennaya:

```
<yarlyk:whatever id="peremannaya" name="peremennayaName"
property="peremennayaProperty"
   type="database.PeremannayaProchee"/>
<font color="red" size="+2">
```

```
   <%=messages.getString("Imya")%>
   <strong><jsp:getProperty name="peremennaya"
     property="title"/></strong>
<br> <br>
</font>
```

Если во время трансляции JSP-страницы встречается созданный выше ярлык, то генерируется код, в котором синхронизируется переменная сценария с объектом, на который она ссылается. Для этого необходима информация о переменной, а именно:

- имя переменной;
- класс переменной;
- ссылается переменная на существующий объект или на новый объект;
- доступность переменной.

Эта информация может быть задана двумя способами. Можно создать элемент `variable` в файле TLD либо создать дополнительный класс для описания ярлыка, указав его в ярлыке `tei-class` в файле TLD. Использование элемента `variable` гораздо проще, но этот способ предоставляет значительно менее гибкие возможности.

Элемент *variable*

Элемент `variable` содержит дочерние элементы:

- `name-given`: имя переменной (константа);
- `name-from-attribute`: имя атрибута, значение этого атрибута, которое будет получено во время трансляции, станет именем переменной.

Необходимо задать один из этих элементов. Прочие элементы являются необязательными:

- `variable-class`: полное имя класса переменной, по умолчанию `java.lang.String`;
- `declare`: ссылается ли переменная на новый объект, по умолчанию `true`;
- `scope`: область доступности переменной, по умолчанию `NESTED`. Области доступности для переменных приведены в табл. 3.5.

Таблица 3.5. *Переменные сценариев, области доступности*

Значение	Доступность	Методы
`NESTED`	от открывающего до закрывающего ярлыка	в методах `doInitBody` и `doAfterBody` при имплементации `BodyTag`; или в `doStartTag`

Таблица 3.5 *(окончание)*

Значение	Доступность	Методы
AT_BEGIN	от открывающего ярлыка до конца страницы	в методах doInitBody, doAfterBody, doEndTag при имплементации BodyTag; или в doStartTag и doEndTag
AT_END	после закрывающего ярлыка до конца страницы	в методе doEndTag

Еще один пример с ярлыком:

```
<tag>
   <variable>
    <name-from-attribute>id</name-from-attribute>
    <variable-class>database.peremennayaProchee</variable-class>
    <declare>true</declare>
    <scope>AT_BEGIN</scope>
   </variable>
</tag>
```

Класс *TagExtraInfo*

Класс дополнительной информации о ярлыке создается на основе класса javax.servlet.jsp.TagExtraInfo. В нем должна содержаться имплементация метода getVariableInfo, который возвращает массив объектов VariableInfo со следующей информацией:

- переменная класса;
- ссылается ли переменная на новый объект;
- доступность переменной.

В приведенном ниже примере (листинг 3.21) мы создали класс описания ярлыка, а затем вставили ссылку на него в файле TLD.

Листинг 3.21. Класс дополнительного описания ярлыка

```
public class ZadaemTei extends TagExtraInfo {
   public VariableInfo[] getVariableInfo(TagData data) {
   String tip = data.getAttributeString("type");
    if (tip == null)
     tip = "java.lang.Object";
    return new VariableInfo[] {
```

```
     new VariableInfo(data.getAttributeString("id"),
      tip,
      true,
      VariableInfo.AT_BEGIN)
    };
   }
}
```

Следующий фрагмент описывает ярлык с учетом того, что существует класс дополнительного описания ярлыка, класс `ZadaemTei`:

```
<tei-class>put'_k_klassu.ZadaemTei
</tei-class>
```

3.9.9. Взаимодействие ярлыков

Взаимодействие ярлыков может быть осуществлено путем создания объектов, доступных разным ярлыкам. Объекты, доступные для разных ярлыков, могут быть созданы двумя способами. Первый способ подразумевает создание объектов, обладающих именами, причем эти объекты должны храниться в контексте страницы. Эти объекты доступны для JSP-страниц и для обработчиков ярлыков. Для обращения к объекту, созданному одним из ярлыков, из другого ярлыка используется метод `pageContext.getAttribute(name, scope)`. Второй метод состоит в том, что объект создается в обработчике родительского ярлыка, в котором имеются другие ярлыки, для обработчиков которых созданный объект автоматически доступен. Такой способ разделения объектов уменьшает риск появления конфликтов имен объектов. Чтобы обратиться к объекту, созданному родительским ярлыком, обработчик ярлыка должен определить сам родительский ярлык при помощи метода `TagSupport.findAncestorWithClass(from, class)` или метода `TagSupport.getParent`. После того как родительский ярлык определен, обработчик ярлыка имеет возможность получить любой статический или динамический объект. В примере рассмотрены оба метода разделения объектов:

```
public class QueryTag extends BodyTagSupport {
   private String connectionId;
   public int doStartTag() throws JspException {
    String cid = getConnection();
    if (cid != null) {
    // есть идентификатор id, используем его
     connection =(Connection)pageContext.
      getAttribute(cid);
    } else {
```

```
      ConnectionTag ancestorTag =
       (ConnectionTag)findAncestorWithClass(this,
          ConnectionTag.class);
      if (ancestorTag == null) {
       throw new JspTagException("Запрос без атрибута whatever должен быть
                                  помещен внутрь ярлыка yarlyk:whatever.");
      }
      connection = ancestorTag.getConnection();
     }
    }
}
```

Фрагмент описания ярлыка для этого примера:

```
<yarlyk:whatever id="con01" ....> ... </yarlyk:whatever>
<yarlyk:zapros id="balances" connection="con01">
   SELECT account, balance FROM acct_table
    where customer_number = <%= request.getCustno()%>
</yarlyk:zapros>

<yarlyk:wahtever ...>
   <sss:zapros id="balances">
    SELECT account, balance FROM acct_table
     where customer_number = <%= request.getCustno()%>
   </sss:zapros>
</yarlyk:wahtever>
```

В файле TLD следует указать, что атрибут `whatever` не является обязательным:

```
<tag>
   ...
   <attribute>
    <name>whatever</name>
    <required>false</required>
   </attribute>
</tag>
```

3.10. Сервер Blazix

Сервер Blazix представляет собой полнофункциональный Web-сервер с поддержкой J2EE. В приложении 3 содержится справочный материал о сервере Blazix.

3.10.1. Архитектура сервера

Сервер Blazix состоит из Web-сервера, сервера компонентов EJB и различных утилит, включая проводника для создания серверных компонентов EJB, который называется Blazix, отладчика JSP, упаковщика Web-приложений, удаленного администратора и некоторые другие утилиты. Общая архитектура сервера показана на рис. 3.15.

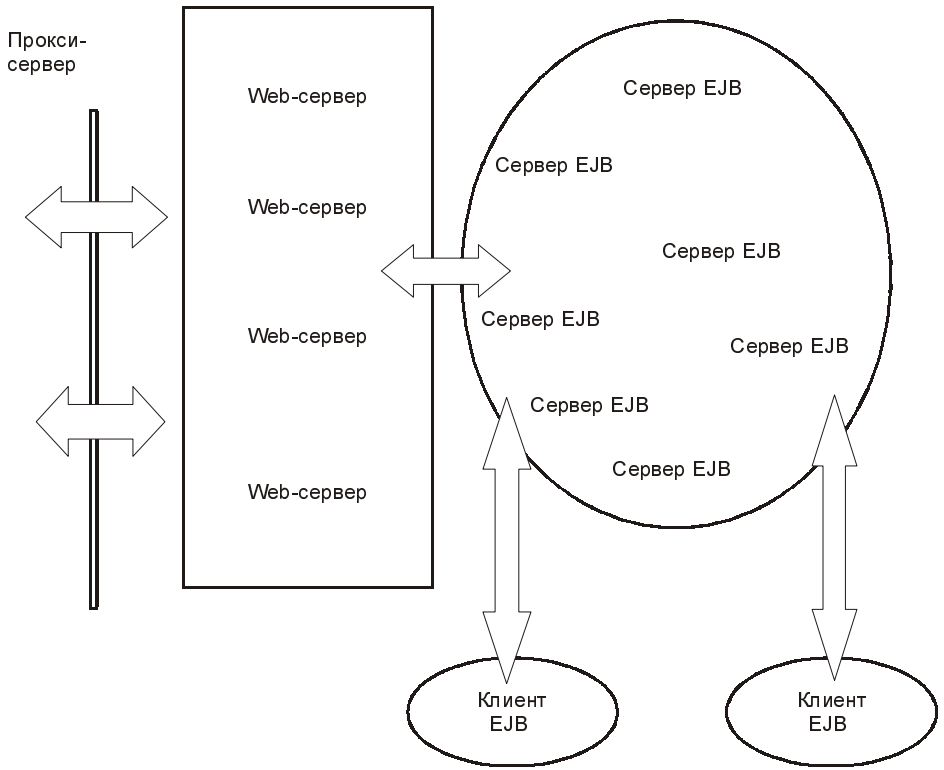


Рис. 3.15. Архитектура сервера Blazix

Несколько серверов Blazix могут работать совместно, образуя кластеры, эти серверы могут распределять между собой нагрузку, используя возможности серверов имен EJB. Разделение Web-сервера и сервера EJB друг от друга повышает гибкость системы, увеличивает инструментарий, обогащает средства, используемый при разработке архитектуры приложения. Так, например, Web-сервер можно расположить за proxy-сервером, при этом сервер EJB оказывается полностью скрыт от внешней сети Интернет и доступен только для обращений с внутренних Web-серверов. На рисунке показан именно такой вариант архитектуры. Здесь Web-серверы образуют один кластер серверов. Серверы EJB образуют отдельный кластер. По мере роста нагрузки

как один, так и другой кластеры могут быть легко увеличены, а выход из строя одного сервера не приводит к выходу из строя системы в целом.

3.10.2. Web-сервер Blazix

Рассмотрим параметры конфигурации Web-сервера Blazix.

Расположение файла конфигурации сервера

Положение файла конфигурации сервера может быть задано при помощи следующей команды с переключателем `-c`):

```
blxweb -c c:\path\myweb.ini
```

или

```
java desisoft.server.ServerMain -c c:\path\myweb.ini
```

Если местоположение файла конфигурации Web-сервера не указано, то сервер будет искать этот файл, придерживаясь следующего порядка:

- если задано свойство `desisoft.web.config`, то используются указанные в нем параметры;
- производится поиск в папке c:\BlazixWebServer.ini;
- производится поиск файла web.ini в текущем каталоге.

Как работает сервер

Приложения

Приложение состоит из одного или нескольких файлов. Таким образом файлы, расположенные на сервере, группируются в приложения. По умолчанию используется приложение с именем `default`. Файлы можно разместить и в другом приложении. Имя приложения указывается в качестве значения свойства `application.name`. Что такое приложение? Приложение соответствует интерфейсу `javax.servlet.ServletContext`. Файлы архивов JAR (Java ARchive) могут быть загружены с указанием параметра `application.<appname>.jarFile`. Можно задать иное имя для приложения, используемого по умолчанию. Для этого следует задать значение свойства `application.default.<property-name>`. Здесь в угловых скобках показаны имена, которые могут быть изменены в зависимости от желания разработчика или администратора.

Базы данных

Сервлеты (небольшие программы, расширяющие и дополняющие функциональность Web-сервера) и серверные страницы JSP имеют возможность осуществлять доступ к одной или нескольким базам данных JDBC (Java DataBase Connectivity — связь с данными на языке Java). Имена источников

данных можно указать в библиотеке ярлыков JSP в качестве значений свойств `connection`. Можно использовать источники данных со средствами JNDI (Java Naming and Directory Interface — интерфейс имен и каталогов), задав соответствующие значения в виде `java:comp/env/jdbc/<data-source-name>`. При этом мы будем иметь возможность пользоваться объектами типа `javax.sql.DataSource`.

Конфигурирование источников данных производится посредством задания класса драйвера `driverClass` и положения базы данных (указания `URL` базы данных). Можно также имплементировать `desisoft.deploy.JdbcResourceFactory` или указать имя ODBC (Open Database Connectivity — связь с открытыми базами данных).

Роли

Сервер работает со средствами безопасности: пользовательские роли (при помощи ролей определяется уровень доступа пользователя к ресурсам системы) определены и защищены парами *name/password*. Любой ресурс сервера может быть защищен, включая сервлеты, страницы HTML, страницы JSP и тому подобное, причем любой из этих ресурсов может быть сделан доступным только для определенного набора ролей.

Интерфейс ION

Интерфейс ION (Interfaces Over the Net) — технология, реализованная в сервере Blazix, которая позволяет клиентам использовать Java-интерфейс сервера для работы с сетью.

ION — это технология работы с распределенными системами. ION основан на интерфейсах Java. Имплементация того или иного интерфейса располагается на сервере, клиент ищет интерфейс и использует его. Сервер имеет возможность осуществлять обратные запросы к клиенту при помощи интерфейса Callback Interface. Отличие интерфейса ION от таких технологий, как RMI (Remote Method Invocation — вызов удаленных методов) или EJB, состоит в том, что он предельно прост и позволяет легко осуществлять обратные запросы к клиенту. При создании интерфейсов первая задача — это описать интерфейс. При этом следует иметь в виду, что все аргументы, используемые в интерфейсе, а также все переменные, возвращаемые интерфейсом, должны быть Java-примитивами или сериализуемыми. После этого интерфейс компилируется и местоположение классов указывается в переменных `CLASSPATTH` таким образом, чтобы интерфейс был доступен как серверу, так и клиенту. Таким образом, следует выполнить следующие действия.

1. Зарегистрировать интерфейс на Web-сервере.
2. Создать класс, имплементирующий интерфейс.
3. Зарегистрировать класс в файле web.ini.

Для работы с интерфейсом клиент производит поиск интерфейса на сервере, затем использует его.

Сервер имеет возможность обращаться к клиенту. Это может происходить, например, в таких ситуациях, когда клиент требует от сервера обращения к методам клиента при изменении тех или иных значений или при требовании сообщать о состоянии через каждую минуту.

Для обратных запросов используется другой интерфейс. Этот интерфейс также должен быть зарегистрирован на Web-сервере. Имплементация интерфейса осуществляется в клиенте.

При поиске экземпляра сервера могут использоваться несколько потоков, однако внутри каждого потока выполнение методов происходит последовательно. При вызове сервера клиентом этот клиент блокируется на время выполнения запроса (до завершения выполнения запроса). Если сервер выполняет обратный запрос, то на время этого запроса клиент также будет оставаться блокированным. При этом другие потоки клиента по-прежнему имеют возможность осуществлять запросы к серверу и производить другую работу.

Обратные запросы ION никогда не инициируются сервером. Обратные запросы могут быть осуществлены только тогда, когда сервер находится в состоянии обработки клиентского запроса.

Ниже приводится пример интерфейса для работы с ION.

```
package compute;
public interface Factorial {
   public int computeFactorial(int n);
}
```

На серверной стороне регистрируется объект, который осуществляет вычисление факториала. Если клиент ищет интерфейс `Factorial`, то он может обнаружить его на сервере и вызвать метод `computeFactorial()`.

Клиент ION осуществляет поиск интерфейса на удаленной машине. Для этого он вызывает метод `lookup` класса `Ion`, указывая имя удаленного хоста и искомый интерфейс (листинг 3.22).

Листинг 3.22. Пример реализации клиента

```
import desisoft.dsap.*;

public class IonClient {
 public static void main(String[] args)
 {
   try {
```

```
    Ion ion = new Ion();
    Factorial fact = (Factorial)
      ion.lookup("www.mopjhost.com", Factorial.class);
    System.out.println("Factorial 3 = " +
            fact.computeFactorial(3));
    System.out.println("Factorial 4 = " +
            fact.computeFactorial(4));
    ion.close();
    } catch (Exception ex) {
    ex.printStackTrace();
    }
  }
}
```

Этот клиент ищет имплементацию метода `compute.Factorial`, описанного в интерфейсе `Factorial` на хосте **www.mojhost.com**. Найденный метод будет выполнен на этом хосте. После завершения работы связь должна быть закрыта при помощи метода `close()`.

Методы, используемые при работе с ION, описаны в классе `ION` (листинг 3.23). При работе с SSL (Secure Socket Layer — безопасное соединение через сокеты) вместо этого класса следует использовать класс `IonSsl`.

Листинг 3.23. Класс ION

```
public class Ion
{
  public Object lookup(String host, Class intf) throws IOException;
  public Object lookup(String host, String url) throws IOException;
  public Object lookup(String host, String url, Class intf)
                                                    throws IOException;
  public Object lookup(String host, int port, Class intf)
                                                    throws IOException;
  public Object lookup(String host, int port, String url)
                                                    throws IOException;
  public Object lookup(String host, int port, String url, Class intf)
                                                    throws IOException;
  public Object lookup(InetAddress host, int port, Class intf)
                                                    throws IOException;
  public Object lookup(InetAddress host, int port, String url)
                                                    throws IOException;
```

```
  public Object lookup(InetAddress host, int port, String url,
                                        Class intf) throws IOException;

  public void close();

  public void setCallback(Object c, Class intf);

  public void setAuthorization(String user, String pass);
}
```

Имплементация интерфейса факториала может быть такой, как показано в листинге 3.24.

Листинг 3.24. Имплементация класса сервера

```
package mojpaket;
import compute.Factorial;

public class MojServer implements Factorial {
   public int computeFactorial(int n)
   {
     int result = 1;
     for (int i = 2; i < n; i++)
       result = result * i;
     return result;
   }
}
```

Все интерфейсы ION должны быть зарегистрированы на сервере. Это справедливо как для серверных интерфейсов, так и для интерфейсов обратных вызовов. Для того чтобы зарегистрировать интерфейс, используется утилита blxionreg в качестве аргумента, которой указывает имя интерфейса. В данном случае регистрация осуществляется при помощи команды

```
blxionreg compute.Factor
```

После каждого изменения, внесенного в интерфейс, его следует заново зарегистрировать. После того, как интерфейс зарегистрирован на сервер, он может быть использован в качестве серверного интерфейса или интерфейса обратного вызова. На клиенте регистрировать ничего не нужно.

Кроме такой регистрации, серверные классы должны быть зарегистрированы в файле конфигурации Web.ini:

```
ion.name: MojIonServer
```

```
ion.MojIonServer.className: mojpackage.MojServer
ion.MojIonServer.interface: compute.Factorial
```

Эти три строки производят регистрацию класса `mojpackage.MojServer`, который имплементирует интерфейс `compute.Factorial`.

Если сервер производит обратный вызов, то операции сервера при этом прекращаются, работа передается клиентской стороне. При этом клиент возвращает ответ серверу сразу. Но может возникнуть такая ситуация, когда, прежде чем клиент пошлет ответ серверу, он будет вынужден произвести дополнительные обращения к серверу. При этом сервер также может сделать еще несколько обратных запросов к клиенту. Клиент сначала возвращает результаты по "вторичным" запросам, а затем возвращается окончательный результат.

Чтобы протестировать вышеописанное, снова запустим Web-сервер Blazix. Из отдельного консольного окна запустим клиент. Клиент может быть запущен и с другого компьютера. На компьютере клиента должен присутствовать файл интерфейса и файл клиентского класса. Серверный класс `MojIonServer` там не понадобится. На клиентском компьютере также будет нужен файл Blazix.jar или файл IonClient.jar. Эти файлы должны быть переданы с клиентом, и путь к ним указан в переменной `CLASSPATH`. Для того чтобы сервер Blazix имел возможность работать с протоколом DSAP (Dynamic Stream Access Protocol — протокол доступа с использованием динамических потоков), необходимо его правильно сконфигурировать, иначе мы получим сообщение об ошибке, в котором будет указано, что DSAP не доступен. О том, как сконфигурировать сервер для работы с протоколом DSAP, будет объяснено немного позже.

Чтобы использовать обратные запросы, необходимо определить интерфейс обратного запроса:

```
public interface MojObrZapros {
   void intermediateStep(String message);
}
```

Файл `MojObrZapros.java` компилируется и интерфейс регистрируется при помощи утилиты `blxionreg`.

Клиент должен содержать объект, имплементирующий этот интерфейс. Чтобы создать имплементацию интерфейса обратного запроса, изменим файл IonClient.java.

```
public class IonUser implements MyCallback {
  // вставим метод
  public void intermediateStep(String message)
  {
    System.out.println("Server says: " + message);
  }
```

Вставим также строку:

```
ion.setCallback(new IonUser(), MyCallback.class);
```

Эта строка располагается после инициализации в методе `main`, но перед обращением к `lookup`. Необходимо также внести изменения в класс сервера. Определим класс сервера при помощи следующей строки:

```
class MojIonServer implements Factorial, desisoft.dsap.IonServer
```

Затем вставим объект:

```
MojObrZapros obrzaprosObj = null;
```

И добавим в сервер методы:

```
public void onConnect() {}
public void onClose() { }
public void setCallback(Object obj)
  {
    obrzaprosObj = (MojObrZapros) obj;
  }
```

Изменим метод вычисления факториала:

```
public int computeFactorial(int n)
  {
   int result = 1;
   if (mojzaprosObj != null)
     mojzaprosObj.intermediateStep("Vychislenie nachato");
   for (int i = 2; i <= n; i++) {
     result = result * i;
     if (mojzaprosObj != null)
      mojzaprosObj.intermediateStep(
          "Promezhutochnyj rezultat = " + result);
   }
   System.out.println("Polucheno !" + n + " = " + result);
   return result;
  }
```

Следует помнить о том, что необходимо изменить файл конфигурации Web.ini, вставив в него:

```
ion.MojServer.callback: MojObrZapros
```

Разработчики сервера особо подчеркивают, что все интерфейсы ION и соответствующие им классы имплементации должны быть доступны как клиенту, так и серверу, то есть пути к ним указаны в качестве значений соответствующей переменной окружения `CLASSPATH`. Доступным должен быть также файл классов `blazix.jar`.

Протокол HTTP основан на обмене запросами и ответами, он не поддерживает непрерывную связь. Имплементация ION произведена на основе протокола DSAP. Протокол DSAP может работать только в том случае, если Blazix используется в качестве Web-сервера непосредственно, но не как Java-сервер в сочетании с каким-либо другим сервером, обеспечивающим фронт-энд (внешний вид), потому что такой сервер не будет поддерживать DSAP. Разработчики сервера утверждают, что Blazix является весьма эффективным сервером, и дополнительные серверы, обеспечивающие фронт-энд, вряд ли будут нужны.

Для того чтобы иметь возможность работать с DSAP, а значит, и с ION, необходимо учесть некоторые особенности сервера. Чтобы сервер Blazix имел возможность работать с протоколом DSAP, имплементируем интерфейс `desisoft.dsap.Server`. Этот интерфейс имеет два метода:

```
public interface desisoft.dsap.Server {
  public boolean authCheck(String username, String pwd);
  public void onConnect(String arg,
             ServletInputStream input,
             ServletOutputStream output)
    throws IOException;
}
```

Метод `authCheck`() в качестве аргументов принимает имя и пароль, если они будут указаны. Если в доступе будет отказано, то метод возвратит `false`.

Метод `onConnect`() вызывается для обработки удаленных запросов. Потоки ввода и вывода `input` и `output` осуществляют обмен данными между клиентом и сервером.

Метод `onConnect`() вызывается из независимого потока. Метод не возвращает никакого значения до тех пор, пока сессия не будет завершена. Для повторного вызова этого метода создается отдельный поток. После возвращения значения этим методом закрывается соответствующий ему сокет.

После того как класс, имплементирующий этот интерфейс, создан, его необходимо зарегистрировать в файле конфигурации Web.ini. Регистрация производится в следующем виде:

```
dsap.name: <имя для DSAP>
dsap.<имя для DSAP>.className: <полное имя класса имплементации>
dsap.<имя для DSAP>.url: <адрес>
```

URL — это не полноценный URL, а некоторая строка, которая известна и серверу, и клиенту, например:

```
dsap.name: MojDsapServer
dsap.MojDsapServer.className: mojdsapserver
dsap.MojDsapServer.url: /moj/dsapserver
```

Класс `dsap` должен быть открытым (`public`), конструктор тоже должен быть открытым и не иметь аргументов. Метод `doConnect` должен поддерживать потоки. Строка соединения `arg` может иметь в своем составе знак "?" в соответствии с правилами синтаксиса протокола HTTP.

Библиотека ярлыков

В настоящей главе мы достаточно подробно рассмотрели вопрос о создании пользовательских ярлыков JSP. Описатель библиотеки может быть задан в виде обычного URL либо помещен в папку с иерархией Web-документов. При этом следует иметь в виду, что конфигурация сервера зависит от параметров, указанных в файле конфигурации Web-сервера. Описание самой библиотеки ярлыков приводится в приложении 3.

Параметры конфигурации Web-сервера

В Web-сервере Blazix предусмотрены следующие параметры конфигурации, задаваемые в файле Web.ini.

- `server.port` — порт сервера, целое число. По умолчанию — 80.
- `server.address` — адрес сервера, полезен для машин, использующих несколько IP-адресов. Здесь необходимо указать действительное имя хоста или IP.
- `admin.port` — порт администрирования. Если значение `admin.port` или `admin.password` не указано, то сервер не будет доступен для удаленного администрирования. При использовании proxy-сервера необходимо позаботиться о том, чтобы данный порт не был доступен извне.
- `admin.password` — пароль администратора.
- `connection.timeout` — время жизни неиспользуемого постоянного соединения (по протоколу HTTP/1.1).
- `session.timeout` — время "жизни" неиспользуемых данных сессии.
- `MaxUsers` — максимальное количество пользователей, по умолчанию для Windows это значение равно 512, для других систем — 256. Реальный предел можно найти путем проведения экспериментов по загрузке сервера. При использовании ION это число полезно увеличить, так как ION-соединения требуют сравнительно б*о*льшего времени.
- `PrintAllAccess` — задает необходимость (или отмену) печати всех строк URL-запросов в файл логов. По умолчанию используется значение `true`.
- `TempDir` — временная папка.
- `access.logDir` — указанный в качестве значения этого параметра каталог будет использоваться для хранения файлов дневных логов. Если задан этот параметр, то будут создаваться ежедневные файлы логов, если параметр не задан, то ежедневные логи не будут сохранены.

- `server.classDir` — эта папка и все содержащиеся в ней файлы JAR будут добавлены в качестве значений переменной `classapth` для работы сервлетов и компонентов EJB. Полезно иметь в виду, что JAR-файлы могут быть помещены в эту папку с удаленного компьютера администратором сервера.
- `WarDir` — все файлы архивов Web-приложений (файлы WAR), помещенных в эту папку, будут автоматически размещены на сервере. Для таких файлов предусмотрен доступ через указание в URL имени WAR-файла. Имя default.war используется для доступа по умолчанию, то есть когда запрашивает основной документ и в URL указывается "/". Утилита администрирования сервера предоставляет возможность размещать новые WAR-файлы в указанной папке, а также удалять ненужные WAR-файлы из этой папки.
- `IonDir` — папка, которая будет использоваться ION для поддержки своего регистра.
- `IonLogExceptions` — будут или нет выводиться сообщения об ошибках ION в поток вывода.
- `mimeType.*` — типы Mime-кодировок, поддерживаемых сервером, например:

  ```
  mimeType.yesyes: application/yesyes-files
  ```

- `ejb.nameServer1`, `ejb.nameServer2`, , `ejb.nameServer9` — серверы имен EJB. Если используется более одного сервера, то они могут подменять друг друга. Так, если сервер ejb.nameServer1 окажется недоступен, то его может подменить сервер ejb.nameServer2.
- `jms.nameServer1`, `jms.nameServer2`, ..., `jms.nameServer9` — JMS-серверы. Если указано более одного сервера, то серверы могут подменять друг друга. Для функционирования всего кластера JMS-серверов достаточно лишь одного сервера.
- `ejb.protocol` — протокол, используемый при работе с серверными компонентами EJB, может быть либо JRMP, либо IIOP. Протокол должен совпадать с протоколом, указанным в качестве значения параметра ejb.protocol в файле конфигурации EJB (ejb.ini). По умолчанию задается протокол JRMP.
- `license.file` — местоположение файла с лицензией.
- `backup.readonly` — сервер кластера используется только для чтения, и никакие записи сохраняемых данных не производятся. Используется для разрешения ошибочных ситуаций, требующих устранения ошибок.
- `jsp.keepGenerated` — файлы Java для создания файлов JAR будут расположены в этой папке.
- `ssl.port` — порт для сервера с поддержкой SSL.

- `ssl.share` — разделение ресурсов при работе с SSL.
- `ssl.keystore` — путь к файлу keystore.
- `ssl.keystorePassword` — пароль для keystore.
- `ssl.showPasswordDialog` — запускать диалоговое окно установки пароля при старте сервера или нет?
- `startup.asynchronous` — имя класса, который имплементирует интерфейс `Runnable`. Этот класс будет инициализирован и запущен в отдельном потоке при запуске сервера.
- `startup.synchronous` — имя класса, который имплементирует интерфейс `Runnable`. Этот класс будет запущен (вызван метод `run` этого класса) при запуске сервера. Запуск сервера не будет продолжен до тех пор, пока метод `run` не будет завершен (не возвратит значение).
- `UseSystemLoader` — используется для задания папок и файлов JAR в качестве параметров переменной `classpath`.
- `application.name` — имя приложения. Имя `default` определено заранее.
- `application.<name>.authType` — используется с ресурсами, для которых задан доступ с указанием пароля. Может принимать значения `Basic` или `Form`. Тип `Basic` заставляет браузер вывести стандартное окно запроса пароля, тип `From` приводит к тому, что браузер перенаправляется на страницу аутентификации, если аутентификация проходит успешно, то после этого браузер перенаправляется обратно к запрошенному ресурсу.
- `application.<name>.authLoginForm` — название HTML-страницы, на которую происходит перенаправление в случае использования метода аутентификации `Form`. HTML-страница должна содержать форму с полями `j_username` и `j_password`, атрибут `action` должен иметь значение `j_security_check`.
- `application.<name>.authErrorPage` — имя HTML-страницы, показываемой в случае неправильно указанной пары имя/пароль.
- `application.<name>.authRealm` — используется при аутентификации с типом `Basic`, `authRealm` посылается браузеру и отображается в виде имени диалогового окна для введения имени и пароля.
- `application.<name>.authClass` — указывается класс, имплементирующий `desisoft.deploy.AuthCheck`. Этот класс, если он указан, будет производить аутентификацию.
- `application.<name>.authParam.*` — используется совместно с предыдущим параметром, задает параметры, передаваемые классу аутентификации.
- `application.<name>.url` — адрес URL, где располагается контекст приложения с указанным именем.

- `application.<name>.jarFile` — файл архива WAR или JAR, содержащий приложение с указанным именем, в соответствии со спецификацией сервлетов (версия 2.2).
- `application.<name>.dir` — папка, в которой расположены файлы приложения, включая файлы HTML, картинки и т. п.
- `application.<name>.jspDir` — папка для хранения классов JSP.
- `application.<name>.param.*` — параметры инициализации контекста, доступные при помощи интерфейса `ServletContext`.
- `application.<name>.virtualHost` — если приложение содержит страницы виртуального хостинга, то здесь необходимо указать виртуальный хост.
- `application.<name>.reloadUrl` — здесь указывается URL, с помощью которого производится разгрузка сервера, то есть удаление всех ранее загруженных классов.
- `application.<name>.filter` — вставляет в приложение фильтр в формате `name=class`, например:

  ```
  application.default.filter: myfilter=mypackage.MyFilterClass
  ```

- `role.name` — имя роли.
- `role.<name>.url` — адрес URL ресурса, доступного для данной роли.
- `role.<name>.application` — имя приложения для роли.
- `servlet.name` — имя сервлета.
- `servlet.<name>.className` — имя класса, иплементирующего сервлет.
- `servlet.<name>.url` — URL-адрес сервлета.
- `servlet.<name>.param.*` — параметры инициализации сервлета, доступные посредством интерфейса `Servlet`.
- `servlet.<name>.application` — если сервлет является частью приложения (не приложения `default`), то имя этого приложения можно указать в этом параметре.
- `dataSource.name` — имя источника данных dataSource.
- `dataSource.<name>.jndiName` — имя JNDI для источника данных, если оно не совпадает с именем самого источника данных.
- `dataSource.<name>.driverClass` — имя класса драйвера JDBC.
- `dataSource.<name>.providerClass` — класс имплементации `desisoft.deploy.JdbcResourceFactory`.
- `dataSource.<name>.param.*` — параметры инициализации, передаваемые классу `desisoft.deploy.JdbcResourceFactory`.
- `dataSource.<name>.odbc` — имя источника данных ODBC.

- `dataSource.<name>.username` — имя пользователя при установлении соединений.
- `dataSource.<name>.password` — пароль при установлении соединений.
- `ion.name` — имя для ION.
- `ion.<name>.url` — адрес URL для интерфейса ION. Это не URL, используемый в Интернете, но условная строка, которая должна быть известна клиенту. Этот параметр не является обязательным, поиск может быть осуществлен и без него либо по имени интерфейса, а не только по URL.
- `ion.<name>.interface` — полное имя класса интерфейса ION. Регистрация осуществляется при помощи утилиты blxionreg.
- `ion.<name>.className` — полное имя класса имплементации интерфейса ION.
- `ion.<name>.callback` — полное имя интерфейса обратных вызовов, регистрируется при помощи blxionreg.
- `ion.<name>.callbackNullable` — может ли клиент возвращать `null` и не имплементировать методы интерфейса обратного вызова.
- `ion.<name>.authClass` — стандартный механизм аутентификации сервера Blazix.
- `ion.<name>.authParam.*` — параметры аутентификации.
- `virtualHost.name` — имя виртуального хоста.
- `virtualHost.<name>.addr` — домен виртуального хоста. Это имя может быть задано для нескольких виртуальных имен доменов, например, **www.mojadres.com** и **mojadres.com**.
- `virtualHost.<name>.accessLogDir` — папка файлов логов для виртуального хоста.

В заключение раздела приведем пример файла конфигурации Web-сервера Blazix, который используется по умолчанию (листинг 3.25).

Листинг 3.25. Файл web.ini

```
server.port: 81
admin.port: 3010
admin.password: vad
ejb.nameServer1: olymp:2050
license.file: C:\Blazix\license.dat
tempDir: C:\Blazix\temp
ionDir: C:\Blazix\iondir
application.default.dir: C:\Blazix\webfiles
```

```
application.default.jspDir: C:\Blazix\jspdir
# Простая идентификация
# Для того чтобы удалить идентификацию, уберите строки
# Для базовой идентификации "Basic" используется другой пример
application.default.authType: Form
#application.default.authRealm: Sample
application.default.authLoginForm: /login.html
application.default.authErrorPage: /loginFailed.html
role.name: user
role.user.url: /AuthenticationTest.html

# Используется во время разработки
application.default.reloadUrl: /_reload

# Для использования файлов .java в страницах JSP, удалите комментарий в
# следующей строке
# jsp.keepGenerated: true

# Примерный файл для аутентификации.
application.default.authParam.file: C:\Blazix\sampleAuth.txt
```

Этот файл должен быть изменен в соответствии с возникающими требованиями.

3.10.3. Конфигурирование сервера EJB

В этом разделе будут рассмотрены параметры конфигурации сервера компонентов EJB.

Расположение файла конфигурации сервера EJB

Местоположение файла конфигурации может быть задано в качестве параметра в командной строке с переключателем `-c`:

```
blxejbs -c c:\path\myejb.ini
```

или

```
java desisoft.ejb.server.EjbServer -c c:\path\myejb.ini
```

Если местоположение файла конфигурации сервера EJB не указано, то сервер производит поиск этого файла в следующем порядке:

- проверяется свойство `desisoft.ejb.server.config`, если находится его значение, то используется это значение;

- проверяется, существует ли файл c:\BlazixEjbServer.ini;
- проверяется, нет ли файла ejb.ini в текущем каталоге.

Работа сервера

Рассмотрим отдельные службы сервера, а также дополнительные сторонние службы и их связь с сервером.

Сервис имен

Каждый сервер EJB может быть использован в качестве сервера имен. Сам сервер имен может иметь компоненты EJB, размещенные на нем, или не иметь их. Сервер имен служит своего рода посредником между клиентами и серверами компонентов EJB, предоставляя сервисы, которыми обладают другие распределенные серверы.

Прочие EJB-серверы должны быть зарегистрированы в одном или нескольких серверах имен. В простейшем случае существует один сервер имен, который обслуживает все домашние объекты EJB.

Источники данных

Для сервера имен может быть задан один или несколько источников данных JDBC. Все серверные компоненты EJB-сущностей должны иметь связь с источником данных. Кроме того, компоненты EJB-сущности и компоненты EJB-сессий могут иметь доступ к дополнительным источникам данных. Компоненты EJB-сессий могут иметь доступ только к тем источникам данных, которые зарегистрированы в файле инициализации ejb.ini. Источники данных могут быть также доступны посредством JNDI, если указывается следующее местоположение ресурса: `java:comp/env/jdbc/<data-source-name>`, при этом возвращается объект типа `javax.sql.DataSource`.

Источники данных задаются с указанием класса драйвера `driverClass` и адреса URL базы данных, или при помощи имплементации `desisoft.deploy.JdbcResourceFactory`, или же путем задания имени ODBC.

Параметры конфигурации сервера

Можно задать следующие параметры конфигурации для сервера компонентов EJB.

- `server.port` — порт сервера, например (по умолчанию) 2050.
- `server.address` — адрес сервера.
- `admin.port` — порт для администрирования.
- `admin.password` — пароль администратора.
- `EjbDir` — путь к папке с файлами компонентов EJB, эти файлы автоматически будут загружены в момент запуска сервера.

- `EjbJar` — путь к файлу архива компонентов EJB. Файл автоматически загружается во время старта сервера.
- `ejb.protocol` — протокол: либо JRMP, либо IIOP. Должен соответствовать аналогичному параметру в файле web.ini.
- `ejb.loadCache` — поведение при кэшировании (`off` или `on`).
- `ejb.storeCache` — значения кэширования данных `off`, `on`, `nochange`.
- `iiop.iorFile` — доступно только в том случае, если используется протокол IIOP. Указывает на файл, содержащий строку IOR.
- `DefaultDataSource` — источник данных, используемый по умолчанию, если источник данных не указан в качестве значения свойства `ejb.<name>.dataSource`.
- `unuse.timeout` — интервал времени, отводимый для ожидания неиспользуемого объекта EJB.
- `ususe.interval` — как часто производится проверка неиспользуемых ресурсов. Параметр должен быть достаточно маленьким, чтобы скорость работы сервера не снизилась из-за чрезмерно большого значения этого параметра.
- `IsNameServer` — принимает значение `true`, если сервер используется в качестве сервера имен.
- `ejb.nameServer` — сервер имен, в котором должны быть зарегистрированы все домашние объекты EJB-сервера. Параметр может повторяться несколько раз, задавая несколько различных серверов имен.
- `license.file` — путь к файлу с лицензией.
- `startup.asynchronous` — имплементация интерфейса `Runnable`, запускается вместе с сервером в отдельном потоке.
- `startup.synchronous` — имплементация интерфейса `Runnable`. Запускается вместе с сервером (вызывается метод `run`), запуск сервера будет продолжен только после того, как метод `run` возвратит то или иное значение.
- `ejb.<name>.jndiName` — имя JNDI, под которым известен EJB, если в свойство не задано, то JNDI-имя совпадает с именем EJB.
- `ejb.<name>.dataSource` — имя источника данных для компонентов EJB-сущности. Все компоненты EJB-сущностей должны быть связаны с источником данных.
- `ejb.<name>.table` — имя таблицы в базе данных, в которой хранятся данные при работе с компонентом EJB-сущности. Все компоненты EJB-сущности связаны с таблицей в базе данных.
- `ejb.<name>.field.*` — имена столбцов в таблице базы данных, они должны совпадать с именами полей в компоненте EJB-сущности. Если

же названия не совпадают, то должно быть задано отображение в следующем формате:

```
ejb.myEjb.field.beanField: tableField
```

- `ejb.<name>.upcaseColumnNames` — принимает значение `true` или `false`. Если установлено `false`, то имена столбцов не будут переводиться в верхний регистр перед передачей запроса к базе данных. По умолчанию используется `true`. Значение этого параметра игнорируется в том случае, если задан параметр отображения столбцов в файле ejb-jar.xml.
- `ejb.<name>.loadCache` — поведение кэша загрузки для отдельного компонента EJB.
- `ejb.<name>.storeCache` — поведение кэша данных для отдельного компонента EJB.
- `AuthClass` — класс имплементации интерфейса `desisoft.deploy.AuthCheck`. Класс проверки паролей.
- `authParam.*` — параметры аутентификации, передаваемые классу `application.<name>.authClass`.
- `dataSource.name` — имя источника данных.
- `dataSource.<name>.jndiName` — имя JNDI источника данных, если оно отличается от имени самого источника данных.
- `dataSource.<name>.driverClass` — имя класса драйвера источника данных JDBC.
- `dataSource.<name>.url` — адрес URL, передаваемый драйверу JDBC.
- `dataSource.<name>.providerClass` — класс, имплементирующий интерфейс `desisoft.deploy.JdbcResourceFactory`.
- `dataSource.<name>.param.*` — параметры инициализации, передаваемые классу `desisoft.deploy.JdbcResourceFactory`.
- `dataSource.<name>.odbc` — источник данных ODBC.
- `dataSource.<name>.username` — имя пользователя при установлении соединения с источником данных.
- `dataSource.<name>.password` — пароль, используемый при установлении соединения с источником данных.

Пример файла инициализации сервера компонентов EJB, используемого по умолчанию, приведен в листинге 3.26.

Листинг 3.26. Исходный файл ejb.ini

```
server.port: 2050
isNameServer: yes
```

```
ejb.protocol: JRMP

admin.port: 2051
admin.password: vad

license.file: C:\Blazix\license.dat
tempDir: C:\Blazix\ejbtemp

ejbDir: C:\Blazix\ejbdir
```

3.11. Защита Web-страниц. Пароли

Защита информации на основе задания имен пользователей и паролей может быть установлена для любых Web-страниц или компонентов EJB, расположенных на сервере Blazix. Для установки защиты необходимо указать, какой именно ресурс будет защищен, а также установить имена пользователей и соответствующие им пароли. При этом аутентификация будет осуществляться с использованием двух стандартных методов: `Basic` и `Form`. От пользователя требуется лишь знание имени пользователя и соответствующего ему пароля.

Методы аутентификации

При базовом методе аутентификации `Basic` браузер выводит диалоговое окно, в котором пользователю предлагается указать имя и пароль. Приложение не может повлиять на внешний вид выводимого окна. Если используется метод `From`, то тогда создается HTML-страница, содержащая форму, в которую следует ввести имя пользователя и пароль. Форма содержит поля `j_username` и `j_password`. Атрибут `ACTION` этой формы должен быть равен `j_security_check`. Помимо этой формы создается страничка, которая выводится в том случае, если пароль будет указан неверно.

Метод аутентификации выбирается путем задания значения свойства `application.<appname>.authType`, которое может быть либо `Basic`, либо `Form`. Если используется базовая аутентификация, то дополнительно указывается свойство `application.<appname>.authRealm`. Браузер использует это свойство, отображая в выводимом диалоговом окне. Для аутентификации по типу `Form` задается значение свойств `application.<appname>.authLoginForm` (форма для аутентификации) и `application.<appname>.authErrorPage` (страница ошибки аутентификации):

```
application.default.authType: Form
application.default.authLoginForm: /login.html
```

```
application.default.authErrorPage: /loginFailed.html

application.default.authType: Basic
application.default.authRealm: TestSite
```

Чтобы удалить сессию пользователя при использовании метода `Form`, следует удалить атрибут объекта сессии `desisoft.form.authorization`.

Выбор защищаемых ресурсов

Какие именно ресурсы будут защищены паролями, указывается в файле конфигурации Web-сервера или в файлах Web-архива или архива компонентов EJB. В обоих случаях защита осуществляется с использованием ролей. С каждой ролью может быть связано несколько паролей. Для Web-сервера следует указать URL защищаемого ресурса в виде значения параметра `role.<role-name>.url`. URL может быть указан полностью, либо с использованием звездочек: `<path>/*` и `*.<extension>`, например:

```
role.name: user
role.user.url: /safearea/*
role.user.url: *.jsp
```

Приведенная в примере конфигурация сообщает, что область `/safearea/`, а также все URL, завершающиеся `.jsp`, будут доступны для роли с именем `user`.

Для файлов Web-архивов имена ролей и адреса защищенных URL указываются в файле WEB-INF/Web.xml. В файлах архивов серверных компонентов EJB роли и защищаемые ресурсы обозначены в файле ejb-jar.xml.

Задание имен и паролей

Пары пользовательских имен и паролей указываются при создании имплементации интерфейса `desisoft.deploy.AuthCheck`. Сервер Blazix содержит встроенную по умолчанию имплементацию этого интерфейса, основанную на использовании файла паролей. Чтобы воспользоваться этой возможностью как простейшим вариантом, следует задать пары имен и паролей, указав их в этом файле. Местоположение файла имен и паролей задается в файле конфигурации Web-сервера или в файле конфигурации EJB-сервера. Для Web-сервера защита осуществляется для того или иного приложения. Если существует только одно приложение, то для него используется имя `default`.

Если создается имплементация интерфейса `desisoft.deploy.AuthCheck`, тогда имя имплементирующего класса следует указать в качестве значения

application.<appname>.authClass (для Web-сервера) или authClass (для сервера EJB). Этот класс может требовать указания тех или иных параметров, которые передаются классу имплементации в виде значений application.<appname>.authParam.* или authParam.*. При использовании аутентификации с заданием файла паролей следует указать значение параметра application.<appname>.authParam.file или authParam.file, например:

```
application.default.authParam.file: htaccess.ini
```

Для Web-сервера класс аутентификации можно задать следующим образом:

```
application.default.authClass: MyAuth.MyAuthClass
application.default.authParam.myParam: myParamValue
```

Работа с файлом конфигурации доступа

Файл конфигурации доступа состоит из строк, каждая из которых содержит одну запись. Запись состоит из полей, разделенных запятыми. Строка, начинающаяся со знаков # или ;, считается комментарием.

Первое поле в строке представляет собой URL защищаемого ресурса или EJB. Здесь можно указать *, что обозначает защиту для всех URL и EJB. Второе поле — имя пользователя, третье поле — пароль. За паролем следуют поля (одно или несколько), в которых указаны названия ролей:

```
*,user1,pwd1,admin,employee
*,user2,pwd2,employee
```

Этот файл сообщает, что имя и пароль user1/pwd1 действуют для всех URL и EJB и используются для осуществления доступа для ролей admin и employee. Имя и пароль user2/pwd2 также используются для всех URL и EJB и открывают доступ для роли employee.

Файл паролей используется для защиты всех URL, указанных в файле конфигурации Web-сервера, и для всех EJB. Чтобы выборочно защитить те или иные компоненты EJB, необходимо указать их имена в первом поле. Можно использовать отдельные файлы паролей для Web-сервера и для сервера EJB.

Пример задания класса аутентификации для Web-сервера (класс аутентификации), а также дополнительные параметры конфигурации сервера Blazix описаны в приложении 3.

3.12. Защита передаваемых данных

Сервер Blazix позволяет защищать передаваемые данные, используя SSL (безопасное соединение через сокеты). Для того чтобы иметь возможность использовать SSL, необходимо установить JSSE (Java Secure Socket Extension — расширение для работы с SSL на Java), который входит в состав пакета Java 1.4. Если вы пользуетесь более ранней версией Java, то необходимо скачать и установить JSSE. Скачать пакет можно с сайта производителя **http://java.sun.com/products/jsse**. После того как файл будет скачан, следует распаковать полученный файл и указать путь к архивам jcert.jar, jnet.jar, jsse.jar в качестве значения переменной окружения `CLASSPATH`.

Следующим шагом будет конфигурирование файла web.ini.

- `ssl.port` — номер порта, на котором будет работать сервер HTTPS, по умолчанию используется значение 443.
- `ssl.share` — процент ресурсов, отводимых серверу HTTPS (от 1 до 100) от обычного HTTP-сервера. Значение 100 выключает обычный HTTP-сервер, все ресурсы отдаются серверу HTTPS. Значение 50 делит ресурсы поровну.
- `ssl.keystore` — местоположение ключей шифрования SSL.
- `ssl.keystorePassword` — пароль keystore. Пароль можно опустить, тогда сервер Blazix запросит ручной ввод пароля при старте. В случае работы сервера в качестве сервиса NT ручной ввод невозможен, пароль необходимо задать заранее.
- `ssl.showPasswordDialog` — если пароль не задан, то сервер Blazix выведет диалоговое окно для задания пароля вручную, при том пароль будет отображен в фоне. Чтобы выключить отображения вводимого пароля, устанавливаем значение свойства `ssl.showPAsswordDialog` равным `true`. При этом необходимо помнить, что работа должна осуществляться в таких условиях, когда доступен пакет AWT.

Ключи и keystore

Для работы с SSL необходимо иметь ключи для шифрования. Для тестирования можно использовать прилагаемые к JSSE ключи. JDK1.3 содержит утилиту создания keystores. Процедура, используемая при работе с ключами, состоит из нескольких шагов.

- Создание пары ключей при помощи команды `keytool -genkey`:

```
keytool -genkey -keyalg RSA -keystore keyfile.key -keypass
yourPassword
```

  В процессе работы утилита будет задавать вопросы, на последнем этапе она запросит имя *"your first and last name"*. Укажем имя Web-сайта, например, **www.mojsait.com**. В результате секретный ключ и открытый ключ

будут расположены в виде пары и записаны в созданном файле с именем `keyfile.key`.

- Создание запроса сертификата с помощью команды `keytool -certreq`:
  `keytool -certreq -keystore keyfile.key -file certreq.txt`

  Файл запроса сертификата будет сохранен под именем `certreq.txt`.
- Передача запроса сертификата агенту, подтверждающему сертификаты.
- Получение подтверждения сертификата.
- Импортирование ответа с помощью команды `keytool -import`:
  `keytool -import -keystore keyfile.key -file certreply.txt`
- keystore доступен и может быть использован в ssl.keystore.

В целях тестирования можно использовать сертификаты, которые генерируются собственными средствами, для этого выполняется команда `keytool -selfcert`.

3.13. Работа с базами данных

3.13.1. Источники данных и Blazix

Источники данных JDBC могут быть сконфигурированы либо для Web-серверов, либо для серверов EJB. Источники данных рекомендуется использовать непосредственно в Java-коде приложения. При этом создается пул соединения, что полезно в тех случаях, когда источник данных не предоставляет такого сервиса.

Кроме того, EJB-сервер будет поддерживать транзакции (механизм обеспечения безопасности выполнения серверных функций, когда доступ к данным блокируется до завершения выполнения всех функций, составляющих транзакцию). Доступ осуществим при помощи библиотеки ярлыков JSP-сервера Blazix. Источники данных могут быть доступны с применением механизма имен JNDI. Конфигурирование источников данных легко проводится с использованием файлов конфигурации Web-сервера или сервера EJB (листинг 3.27). Источники данных доступны при использовании следующего местоположения JNDI: `java:comp/env/jdbc/<jndi-name>`.

Листинг 3.27. Пример работы с источниками данных (для сервлетов, JSP, EJB)

```
import javax.sql.*;
import javax.naming.*;
import java.sql.*;

Context ctx = new InitialContext();
```

```
DataSource ds = (DataSource)
           ctx.lookup("java:comp/env/jdbc/myDataSource1");
Connection con = ds.getConnection();
// или Connection con = ds.getConnection(username, pwd);
// далее используется созданное соединение
...
con.close();
```

При использовании библиотеки ярлыков JSP-сервера Blazix имена источников данных могут быть указаны непосредственно, без префикса `java:comp/env/jdbc/`.

Чтобы сконфигурировать источник данных JDBC, необходимо указать имя класса драйвера и URL в свойствах `dataSource.<name>.driverClass` и `dataSource.<name>.url` соответственно, например:

```
dataSource.name: myJdbc1
dataSource.myJdbc1.driverClass: com.myprovider.jdbc.classname
dataSource.myJdbc1.url: myhost:4500/mydatabase
```

Если применяются базы данных ODBC, то указывается имя базы данных для коннектора ODBC/JDBC, который используется по умолчанию, например

```
dataSource.name: myOdbcDatabase
dataSource.myOdbcDataBase.odbc:  myOdbcName
```

Некоторые базы данных могут использовать имплементацию интерфейса `XAResource`. В такой ситуации необходимо иметь класс, имплементирующий интерфейс `desisoft.deploy.JdbcResourceFactory`. Имя класса имплементации указывается в качестве значения свойства `dataSource.<name>.providerClass`. Параметры инициализации передаются классу в виде значений свойства `dataSource.<name>.param.*`, например:

```
dataSource.name: myDbms1
dataSource.myDbms1.providerClass: Supplier.Wrapper.MyClass
dataSource.myDbms1.param.dbmsName: mydbms1/myschema1
```

Глава 4

Сервлеты

4.1. Понятие сервлета

Сервлет — это Java-программа, которая в качестве своего суперкласса использует класс `HttpServlet`. Сервлет используется для того, чтобы расширить возможности существующего сервера, в частности, Web-сервера. Как правило, сервлет работает на специальном сервере. Такие серверы носят название серверов Java-приложений (Java Application Server). В состав сервера Java-приложений в качестве составного блока входит Web-сервер (иногда не один, а несколько), а также серверы, работающие с серверными компонентами, серверы вспомогательных служб и т. п. Сервлет работает в окружении, которое предоставляет ему сервер. Часть сервера, предназначенная для работы с сервлетами, называется *контейнером сервлетов*. Спецификация сервлетов предполагает наличие в классе сервлета стандартных методов, выполнение которых происходит на том или ином этапе жизненного цикла сервлета. Вызов этих методов осуществляется контейнером сервлетов. Имплементация спецификации сервлетов входит в набор стандартных пакетов языка Java.

В данной книге рассматривается сервер Java-приложений под названием Blazix. Сервер Blazix предоставляет полный набор возможностей для работы с сервлетами. Помимо создания класса (или классов) сервлета, а также для конфигурирования созданной программы-сервлета, установления ее на сервер, необходимо изменить файл конфигурации Web-сервера. Основные значения указываются в файле конфигурации в следующем виде:

```
servlet.name: myservlet
servlet.myservlet.className: mypackage.MyServletClass
servlet.myservlet.url: /mysrvlet
```

У каждого сервлета должно быть задано имя (`servlet.name`), по которому он идентифицируется на сервере. Это имя используется для задания свойств сервлета, в частности, для указания имени класса, в котором хранится программа сервлета (следующая строка), а также адреса, по которому происходит обращение к этому сервлету (третья строка).

Клиент запрашивает у Web-сервера адрес, по которому расположен сервлет (адрес должен быть указан в качестве значения `servlet.myservlet.url`

в файле конфигурации Web-сервера). Сервер передает запрос и данные (если они есть) сервлету, получает ответ от сервлета и направляет его клиенту.

На этом объяснения о том, что такое сервлет, можно было бы закончить. Однако существует множество интересных и полезных деталей, на которых следует задержать внимание и изучить их подробнее.

Особенно важно иметь в виду, что путь к классу сервлета должен быть указан в переменной `classpath` или же его можно поместить в каталог C:\Blazix\classes или в каталог, который указан в файле конфигурации Web-сервера в качестве значения `server.classDir`. После того как файл конфигурации был изменен и в него вставлена информация о новом сервлете, сервер должен быть остановлен и запущен вновь. Сервлет может быть размещен и на работающем сервере при помощи утилиты администрирования, но для этого сервлет должен быть упакован в файл Web-архива WAR. Если файл класса сервлета был изменен, то останавливать сервер и запускать его вновь необязательно. По умолчанию сервер сконфигурирован так, что вызов сервлета по адресу **http://localhost:81/_reload** приводит к тому, что все классы будут перезагружены и измененный класс сервлета станет доступен для клиентских запросов (рис. 4.1). Посетите эту страницу после того, как будет изменен файл класса сервлета. Остановка сервера попросту не нужна.

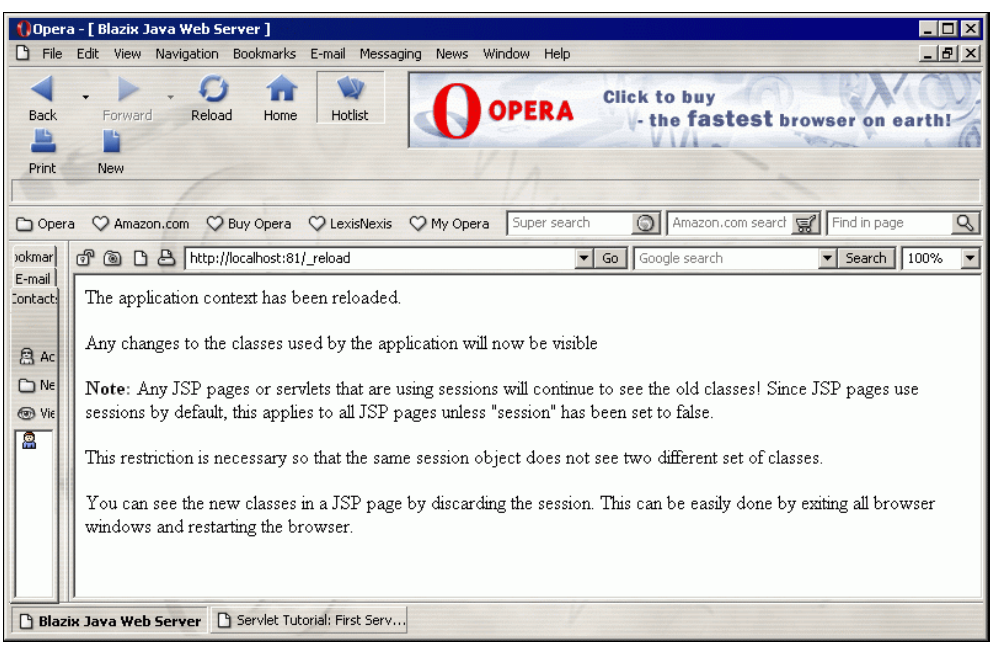


Рис. 4.1. Перезагрузка всех сконфигурированных сервлетов, загрузка классов

Если читатель уже имел некоторый опыт работы с сервлетами, то он оценит ту простоту, которая свойственна серверу Blazix в сравнении с другими Java-серверами, например, с сервером Tomcat.

Чтобы сразу приступить к делу, рассмотрим простой пример, сервлет `SomeServlet` (листинг 4.1).

Листинг 4.1. Сервлет SomeServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SomeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("Hello World");
  }
}
```

В файле конфигурации Web-сервера запишем следующее:

```
servlet.name: first
servlet.first.className: SomeServlet
servlet.first.url: /dofirst
```

Тогда обращение к сервлету из браузера примет вид **http://localhost:81/dofirst** (рис. 4.2).

Сервлеты Java предоставляют платформонезависимый метод создания Web-приложений (это не бесспорно, однако мы не будем рассматривать этот вопрос глубже, поскольку тогда нам потребуется определиться с тем, что понимать под Web-приложением, что не входит в перечень рассматриваемых тем), причем многие приложения отличаются быстротой в работе и лишены тех ограничений, которые имеются у CGI-приложений. Сервлет работает под управлением Web-сервера и, в отличие от апплета, не требует графического интерфейса пользователя. Сервлет взаимодействует с сервером, обмениваясь с ним запросами и ответами. Клиентская программа обращается к серверу с запросами. Запрос обрабатывается сервером, передается сервлету, сервлет посылает ответ через сервер клиенту. На сегодняшний день сервлеты весьма популярны при создании интерактивных Web-приложений. Существует множество Web-серверов, способных работать с сервлетами, среди них такие серверы, как Tomcat, iPlanet Web Server (ранее Netscape

Enterprise Server), Blazix. JBuilder Enterprise использует сервер Borland Enterprise Server (BES), входящий в состав пакета, а также поддерживает работу с серверами WebLogic, WebSphere и iPlanet. JBuilder Enterprise Server включает в себя сервер Tomcat, который используется "по умолчанию".

Рис. 4.2. Обращение к сервлету

Одним важным преимуществом сервлетов является скорость их работы. В отличие от приложений CGI сервлеты загружаются в память лишь один раз и затем выполняются непосредственно из памяти. Сервлеты по существу являются многопоточными приложениями. К тому же они платформонезависимы, поскольку написаны на языке Java.

Технология JSP, которой посвящена *глава 3*, является расширением технологии сервлетов, в JSP особое внимание уделяется работе с HTML- и XML-документами. В составе HTML- и XML-кода JSP-фрагменты находят свое наиболее частое применение. Как сделать выбор и решить, что использовать: сервлеты или серверные страницы? Сервлеты более подходят для решения задач низкоуровневого программирования и менее приспособлены для решения задач создания логики представления приложения. Серверные страницы JSP, наоборот, в основном сконцентрированы на том, как представить результат пользователю в наиболее удобном виде. Серверные стра-

ницы создаются встроенными в HTML-код, используя стиль создания HTML-документов. Технология JSP предоставляет гораздо более богатые возможности, чем простой HTML. Страницы JSP могут предоставлять возможности реализации логики приложений с использованием простых компонентов Java, а также серверных компонентов EJB, путем создания пользовательских библиотек ярлыков. Сами по себе серверные страницы Java могут являться модульными, многократно используемыми компонентами, работающими с логикой представления, которые можно использовать совместно с различными шаблонами и фильтрами. JSP-страницы преобразуются в сервлеты, поэтому теоретически можно пользоваться исключительно сервлетами. Однако технология JSP создана для того, чтобы упростить процесс создания Web-документов, отделив логику представления приложения от содержания документа. В большинстве случаев отправляемый клиенту ответ состоит как из шаблонов представления документа, так и из данных, которые генерируются автоматически, заполняя шаблон. В таких ситуациях оказывается много легче работать с JSP, чем с сервлетами.

4.1.1. Архитектура сервлетов

Основа API для работы с сервлетами — это интерфейс `Servlet`. Все сервлеты имплементируют этот интерфейс напрямую, либо косвенно, являясь имплементацией дочернего интерфейса `HttpServlet`. Интерфейс `Servlet` содержит методы для работы с сервлетами, в том числе методы, позволяющие сервлету взаимодействовать с клиентами. Разработчик сервлета использует все или некоторые из этих методов. При получении запроса от клиента, сервлет принимает два объекта: объект `ServletRequest` и объект `ServletResponse`. Интерфейс `ServletRequest` предназначен для осуществления связи от клиента к сервлету, в то время как интерфейс `ServletResponse` создан для установления связи от сервлета к клиенту. Интерфейс `ServletRequest` позволяет сервлету получать информацию о параметрах, передаваемых клиенту: протокол и имя удаленного хоста, с которого направлен запрос. Сервлет получает данные, которые передаются по протоколу HTTP. Подклассы интерфейса `ServletRequest` позволяют получать детальную информацию о передаваемых значениях, в том числе информацию, передаваемую в HTTP-заголовках (`HttpServletRequest`). Интерфейс `ServletResponse` предоставляет средства для отправки ответа сервлета клиенту посредством потока вывода `ServletOutputStream` (возможно также использование потока `Writer`). Информация может быть передана и в HTTP-заголовках при помощи методов, содержащихся в `HttpServletResponse`. Помимо базовых функций взаимодействия с клиентом, сервлет может также осуществлять поддержку работы с сессиями.

4.1.2. Жизненный цикл сервлета

Сервлет загружается сервером и выполняется в контексте сервера. Сервер может удалить сервлет. При загрузке сервлета сервер вызывает метод `init()`. Во время загрузки сервлета не используется возможность многопоточного выполнения задач, решаемых сервлетом, даже в обычных условиях многопоточного серверного окружения. Метод `init()` вызывается только один раз и может быть вызван вновь только при перезагрузке сервлета. Сервер не может перезагрузить сервлет до тех пор, пока сервлет не будет удален с сервера при помощи метода `destroy()`. Инициализация сервлета должна завершиться до того, как будет вызван метод `service()`. После инициализации сервлет способен принимать и обрабатывать клиентские запросы. Запросы обрабатываются методом `service()`. Каждый клиентский запрос приводит к вызову метода `service()`, который выполняется в отдельном для каждого запроса потоке. Метод получает клиентский запрос и посылает ответ. Сервлет может выполнять несколько методов `service()` одновременно. При этом важно иметь в виду, что метод `service()` должен быть создан так, чтобы его выполнение в нескольких потоках одновременно не приводило к ошибкам, в частности здесь будет полезным применение синхронизации. Если сервлет не должен производить многопоточные вычисления, то сервлет может имплементировать интерфейс `SingleThreadModel`. При этом сервер не сможет вызвать несколько потоков выполнения метода `service()`.

Сервлет работает на сервере до тех пор, пока он не будет удален с сервера. Это может произойти по требованию администратора. Перед удалением сервлет выполняет метод `destroy()`. Методы выполняется только один раз. Во время выполнения этого метода, метод `service()` в отдельном потоке может продолжать работу. При этом, если метод `destroy()` производит зачистку, касающуюся совместных ресурсов (например, требуется закрыть сетевую связь), доступ к которым может быть осуществлен из выполняющегося параллельно метода `service()`, то следует синхронизировать такие обращения к разделенным ресурсам.

По сути сервлет — это экземпляр класса, имплементирующего интерфейс `javax.servlet.Servlet`. Многие сервлеты используют стандартную имплементацию этого интерфейса, а именно классы `javax.servlet.GenericServlet` и `javax.servlet.http.HttpServlet`. Во время инициализации сервлета сервер загружает класс сервлета (при необходимости также и другие классы), для чего используется конструктор, который не имеет аргументов. Затем вызывается метод `init(ServletConfig config)`. Объект типа `ServerConfig` может быть использован в дальнейшем, для этого вызывается метод `getServletConfig()`. Если сервлет создается на основе класса `GenericServlet` (или его подкласса `HttpServlet`), то следует вызвать метод `super.init(config)`, расположив его в самом начале метода `init()` сервлета. Объект `ServletConfig` содержит параметры сервера и ссылку на `ServletContext`. После инициализации вызывается метод `service(ServletRequest req, ServletResponse res)`. Этот метод вызыва-

ется всякий раз при поступлении клиентского запроса. Схема жизненного цикла сервлета представлена на рис. 4.3.

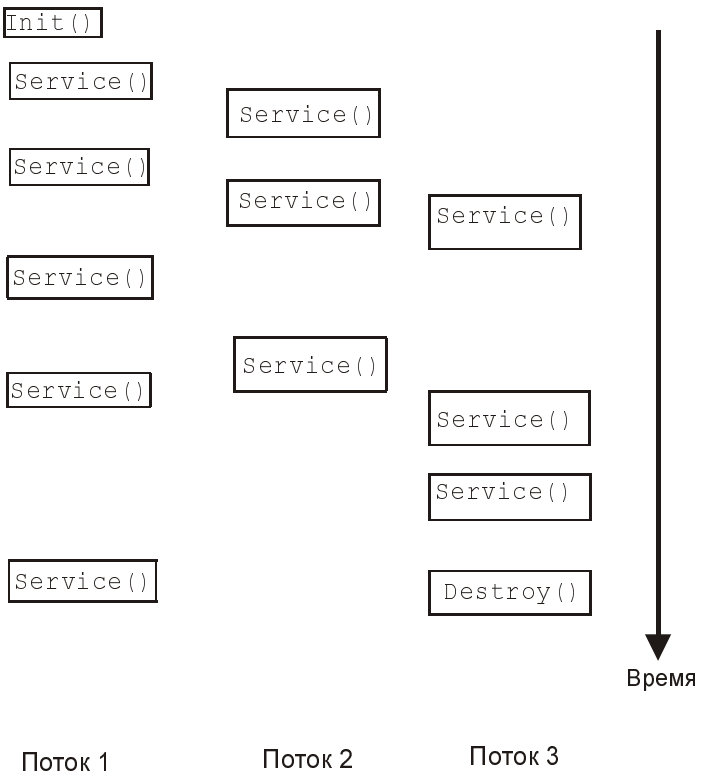


Рис. 4.3. Жизненный цикл сервлета

4.2. API для работы с сервлетами

Программный интерфейс для работы с сервлетами представлен в пакете `javax.servlet`. Все сервлеты прямо или опосредованно должны имплементировать интерфейс `javax.servlet.Servlet`. Наиболее часто используемые классы и интерфейсы отражены в табл. 4.1.

Таблица 4.1. *Наиболее часто используемые классы и интерфейсы при работе с сервлетами*

Название	Класс или интерфейс	Описание
`GenericServlet`	Класс	Независимый от протокола сервлет

Таблица 4.1 *(окончание)*

Название	Класс или интерфейс	Описание
`RequestDispatcher`	Интерфейс	Объект, получающий клиентские запросы и посылающий их ресурсам, например, сервлету, HTML-файлу, JSP-файлу на сервере
`Servlet`	Интерфейс	Методы, которые может имплементировать сервлет
`ServletConfig`	Интерфейс	Объект конфигурации сервлета. Используется контейнером сервлетов для передачи информации сервлету в процессе инициализации
`ServletContext`	Интерфейс	Определяет набор методов, с помощью которых сервлет взаимодействует с контейнером сервлетов, например, для получения типа mime-файла или записи в файл логов
`ServletException`	Класс	Определяет класс исключения, которое возникает, если выполнение сервлета затруднено
`ServletInputStream`	Класс	Поток ввода для чтения двоичных данных запроса клиента, в том числе метод readLine для чтения данных построчно
`ServletOutputStream`	Класс	Поток вывода для отправки двоичных данных клиенту
`ServletRequest`	Интерфейс	Задает объект для работы с запросами клиента
`ServletResponse`	Интерфейс	Создание объекта для отправки ответа клиенту
`SingleThreadModel`	Интерфейс	Для того чтобы сервлет мог обрабатывать только один запрос
`UnavailableException`	Класс	Определяет класс ошибки для того, чтобы сообщить, что сервлет временно или постоянно недоступен

4.2.1. Пакет *servlet.HTTP*

Для того чтобы иметь возможность работать с некоторыми элементами, поддерживаемыми в протоколе HTTP, сервлет создается на основе пакета `javax.servlet.http`. Такой сервлет может получать HTTP-запросы

с использованием стандартных методов, обрабатывать и посылать ответ. Методы запросов следующие:

- `GET`
- `POST`
- `PUT`
- `DELETE`
- `HEAD`
- `TRACE`
- `CONNECT`
- `OPTIONS`

В классе `HttpServlet` эти методы имплементированы. Первое, что мы должны сделать, — это создать сервлет на основе класса `HttpServlet` и переопределить метод `doGet()` и/или метод `doPost()`. Затем нужно будет переопределить методы `doPut()` и `doDelete()`. При использовании мастера создания сервлетов в JBuilder (среда программирования на языке Java от компании Borland) можно просто выбрать переопределяемые методы, а мастер сам предложит заготовки кода.

В табл. 4.2 приведены наиболее часто используемые классы и интерфейсы пакета `javax.servlet.http`.

Таблица 4.2. *Часто используемые классы и интерфейсы пакета* `javax.servlet.http`

Имя	Класс или интерфейс	Описание
`Cookie`	Класс	Создает cookies (информация, посылаемая браузеру, хранимая браузером и затем вновь отправляемая браузером серверу)
`HttpServlet`	Класс	Абстрактный класс для создания сервлетов
`HttpServletRequest`	Интерфейс	Создан на основе интерфейса `ServletRequest` для работы с информацией от клиентских запросов
`HttpServletResponse`	Интерфейс	Создан на основе интерфейса `ServletResponse` для отправки ответов клиенту
`HttpSession`	Интерфейс	Интерфейс для работы с сессиями. Идентификация пользователя, которая сохраняется в процессе перехода с одной Web-страницы на другую

Таблица 4.2 *(окончание)*

Имя	Класс или интерфейс	Описание
`HttpSessionBindingEvent`	Класс	Посылается объекту, имплементирующему интерфейс `HttpSessionBindingListener` при связывании или освобождении объекта от сессии
`HttpSessionBindingListener`	Интерфейс	Приводит к тому, что объекту сообщается о том, связывается ли он с сессией или освобождается от сессии

4.2.2. Жизненный цикл сервлета

Интерфейс `javax.servlet.Servlet` содержит методы, которые используются сервлетом в процессе организации его жизненного цикла. Основные этапы жизни сервлета следующие:

- создание сервлета и вызов метода `init()`, при помощи которого инициализируется сервлет;
- обращение к методу `service()`;
- вывод сервлета из обращения, обращение к методу `destroy()`, сбор мусора.

Создание и инициализация сервлета

При старте контейнера сервлета или тогда, когда сервлет должен отвечать на запрос, контейнер вызывает метод `init()`. Этим он сообщает сервлету о том, что тот начинает работать (вступает в активную фазу). Метод `init()` вызывается только один раз. Выполнение метода `init()` должно быть завершено прежде, чем сервлет получит запрос. Параметр `ServletConfig` в методе `init()` — это объект, который содержит информацию о конфигурации и инициализации сервлета. После инициализации сервлета эта информация доступна и может быть получена при помощи метода `getServletConfig()`.

Обработка клиентских запросов

Контейнер сервлета вызывает метод `service()` с помощью которого организуется отправка ответов на клиентские запросы. Объекты запроса и ответа передаются в качестве параметров методу `service()` после получения клиентского запроса. Сервлет может также имплементировать интерфейсы `ServletRequest` и `ServletResponse`, которые позволят ему обращаться к па-

раметрам запроса и посылаемым в ответ данным. Параметры запроса содержат данные и методы, определенные в протоколе. Данные ответа состоят из заголовка ответа и статус-кода.

Многопоточность и сервлеты

Как правило, сервлеты обладают свойством многопоточности, что позволяет одному и тому же сервлету обрабатывать несколько запросов одновременно. Разработчик должен специально позаботиться о том, чтобы ресурсы, используемые при работе потоков (файлы, сетевые соединения, переменные) использовались безопасным способом. Существует возможность создания сервлета, который может работать только с одним потоком. Для этого следует использовать интерфейс `SingleThreadModel`. При этом сервлет будет способен обрабатывать только один запрос в каждый конкретный момент. Однако на практике эта возможность используется довольно редко.

4.2.3. Сервлеты и HTML

С помощью сервлетов легко создавать HTML-текст. HTML-код создается динамически в процессе работы сервлета. При этом нет необходимости использовать языки сценариев, скрипты. Если, например, Web-сервер располагает средствами SSI (Server-Side Include — вставка программных элементов на стороне сервера), то для обработки HTML-текста можно использовать фрагмент, обозначенный ярлыком `<servlet>`. При этом соответствующий файл, содержащий этот фрагмент, будет иметь расширение `.shtml`. Ярлык `<servlet>` сообщает серверу, что следует загрузить и выполнить сервлет, инициализировав его соответствующими параметрами конфигурации. Результат работы сервлета будет получен в виде отформатированного при помощи HTML-ответа. Как и в ярлыке `<applet>`, мы имеем возможность указать местоположение класса сервлета, задав значения атрибутов `class` и `codebase` в ярлыке `<servlet>`. Параметр `codebase` указывает каталог, в котором располагается файл Java-класса, параметр `code` задает имя Java-класса.

Пример использования ярлыка `<servlet>`:

```
<servlet>
  codebase=""
  code="dbServlet.Servlet1.class"
  param1=in
  param2=out
</servlet>
```

4.2.4. Сервлеты и HTTP

Сервлеты могут быть использованы для обработки запросов, которые передаются по протоколу HTTP. Они могут поддерживать любые методы, указанные в протоколе. Они могут осуществлять переадресацию запросов, посылать сообщения об ошибках. Сервлеты могут работать с параметрами, посылаемыми в составе форм, при этом сервлет может получать методы и адреса наряду с именами и значениями параметров:

```
String method  = request.getMethod();
String uri     = request.getRequestURI();
String name    = request.getParameter("name");
String phone   = request.getParameter("phone");
String address = request.getParameter("address");
String city    = request.getParameter("city");
```

Сервлет может работать с mime-типами, что позволяет ему получать в качестве параметров соответствующую запросу информацию, и посылать в качестве ответа такой тип информации, который необходим.

4.2.5. Как пользоваться сервлетами

Сервлет можно использовать различными способами.

- Как часть системы обработки клиентской информации (например, заказов) при работе с базами данных для получения данных из форм (ярлык `<form>` HTML).
- В сочетании с апплетами как составная часть сети интранет.
- Как часть многопользовательской системы, например, системы сообщений.
- Как часть системы, используемой для распределения загрузки запросов.
- Как HTML-ориентированный сервлет для форматирования динамической информации, отправляемой в ответ на запрос браузера, в том числе, для работы с рекламными баннерами.

4.2.6. Размещение сервлетов

Сервлет может работать самостоятельно на сервере, а также может быть размещен в качестве модуля J2EE. Такой модуль состоит из самого сервлета или нескольких компонентов одного типа (например, Web-компонентов, компонентов EJB, клиентов и т. п.), которые описываются одним дескриптором размещения.

4.2.7. Использование утилиты создания сервлетов на основе интернет-компонентов InternetBeans Express

Технология интернет-компонентов (это не то же самое, что серверные компоненты EJB) объединяет сервлеты и серверные страницы Java, облегчая процесс создания приложений. Интернет-компоненты InternetBeans Express представляют собой набор компонентов и библиотеку ярлыков JSP, которые позволяют быстро и легко создавать Web-приложения, уделяя значительное внимание уровню представления приложения пользователю. Они используют статические страницы-шаблоны и вносят в них динамическое содержание, а затем представляют их клиенту и получают данные от клиента. Например, можно использовать InternetBeans Express для работы с компонентами, которые созданы для легкой работы при создании приложений, позволяющих зарегистрировать пользователя на сайте, или для создания JSP, отображающей результат поиска. InternetBeans Express содержит встроенную поддержку для работы с данными с помощью таких инструментов, как DataExpress, в частности с DataSets и DataModules. Интернет-компоненты могут быть использованы при работе с другими источниками данных, а также при работе с серверными компонентами EJB. Классы и интерфейсы Интернет-компонентов InternetBeans Express можно подразделить на три категории:

- интернет-компоненты InternetBeans, которые являются компонентами, динамически создающими ответы на запросы HTTP;
- обработчики ярлыков JSP, которые вызывают интернет-компоненты внутри себя;
- поддержка работы с данными.

4.3. Структура сервлета

Файл простейшего сервлета уже был приведен ранее. Сейчас будет рассмотрен пример, из которого будет понятна общая структура апплета, который обрабатывает запросы, получаемые с использованием методов `GET`. Этот метод позволяет передавать данные запроса вместе с адресом URL по протоколу HTTP. Наряду с методом `GET` может быть использован метод `POST` (данные передаются отдельным фрагментом в теле HTTP-запроса, при этом данные не являются составной частью URL).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SomeServlet extends HttpServlet {
```

```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
           // Для чтения получаемых данных используется "request",
           // с его помощью читаются заголовки, cookies, данные из форм
           // Для создания ответов используется "response", с его
           // помощью создаются заголовки ответа, а также весь текст
           // ответа, в том числе посылаемый тип данных, cookies.

       PrintWriter out = response.getWriter();
            // для отправки ответа браузеру используется "out"
  }
}
```

Для того чтобы класс представлял собой сервлет, он должен быть создан на основе класса `HttpServlet`, и в нем должны быть переопределены методы `doGet` и `doPost` (один или оба сразу), в зависимости от того, каким методом он получает данные (`GET` или `POST`). Эти методы получают два аргумента: объекты `HttpServletRequest` и `HttpServletResponse`. Интерфейс `HttpServletRequest` описывает методы, которые позволяют читать данные HTML-форм, заголовки HTTP и т. п. Интерфейс `HttpServletResponse` предоставляет методы, которые используется для отправки кодов HTTP (200, 404 и т. п.), заголовков ответов (`Content-Type`, `Set-Cookie` и т. п.), а также позволяет использовать поток вывода `PrintWriter`, с помощью которого HTML-код отправляется клиенту. В простейшем сервлете вывод может быть осуществлен при помощи инструкции `println`. Методы `doGet` и `doPost` могут вызывать появление двух исключительных ситуаций. Следует также помнить, что в класс сервлета необходимо импортировать несколько пакетов, в том числе пакет `java.io` (содержит `PrintWriter` и пр.), пакет `javax.servlet` (содержит `HttpServlet` и пр.), пакет `javax.servlet.http` (содержит `HttpServletRequest` и `HttpServletResponse`). И, наконец, переопределить методы `doGet` и `doPost`.

4.3.1. Сервлет, создающий HTML

Большинство сервлетов выводят не простой текст, а HTML. Чтобы иметь возможность выводить HTML, необходимо первым делом сообщить браузеру о том, что текст, посылаемый ему, — это HTML. Для этого необходимо послать заголовок в соответствии со спецификацией протокола HTTP. Чтобы указать тип данных HTML, заголовок должен содержать "`Content-Type: text/html`". Заголовки должны быть отделены от потока текста пустой строкой. За нас все это сделает метод `setHeader` из интерфейса `HttpServletResponse`, однако существует и другой метод — `setContentType`.

Более того, заголовок можно сформировать и передать вручную. Заголовок должен создаваться перед тем, как будут переданы данные. Например, файл `HelloWWW.java` (листинг 4.2). Здесь класс размещается внутри пакета `paket` (рис. 4.4).

Рис. 4.4. Сервлет выводит HTML

Листинг 4.2. Передача заголовков браузеру

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWWW extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
```

```
                                        "Transitional//EN\">\n" +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

В файле конфигурации Web-сервера пишем:

```
servlet.name: HelloWWW
servlet.HelloWWW.className: paket.HelloWWW
servlet.HelloWWW.url:  /hellowww
```

Файл класса HelloWWW.class помещаем в папку `C:\Blazix\classs\paket`.

Каждый раз выводить информацию с использованием `println` не очень удобно. Учитывая, что заголовки и названия страниц часто содержат весьма однотипную информацию, можно создать класс, который будет упрощать работу со стандартной информацией, выводимой сервлетом. Пусть это будет класс `ServletUtilities.java` (листинг 4.3).

Листинг 4.3. Файл Servletutilities.java

```
package paket;
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletUtilities {
  public static final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">";
  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" +
           "<HTML>\n" +
           "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
  }
  /*
  чтение параметра по имени name, преобразование его в int,
  возвращение этого параметра
  */
  public static int getIntParameter(HttpServletRequest request,
                                    String paramName, int defaultValue) {
```

```
    String paramString = request.getParameter(paramName);
    int paramValue;
    try {
      paramValue = Integer.parseInt(paramString);
    } catch(NumberFormatException nfe) { // обработка null и
                                         // неправильных значений
      paramValue = defaultValue;
    }
    return(paramValue);
  }
  public static String getCookieValue(Cookie[] cookies,
                                      String cookieName,
                                      String defaultValue) {
    for(int i=0; i<cookies.length; i++) {
      Cookie cookie = cookies[i];
      if (cookieName.equals(cookie.getName()))
        return(cookie.getValue());
    }
    return(defaultValue);
  }
  // значения
  public static final int SECONDS_PER_MONTH = 60*60*24*30;
  public static final int SECONDS_PER_YEAR = 60*60*24*365;
}
```

Сейчас, когда есть уже класс с утилитами, программа HelloWWW (рис. 4.5) может быть записана в более удобном виде (листинг 4.4).

Листинг 4.4. Файл HelloWWW2.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWWW2 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```

```
    out.println(ServletUtilities.headWithTitle("Hello WWW") +
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

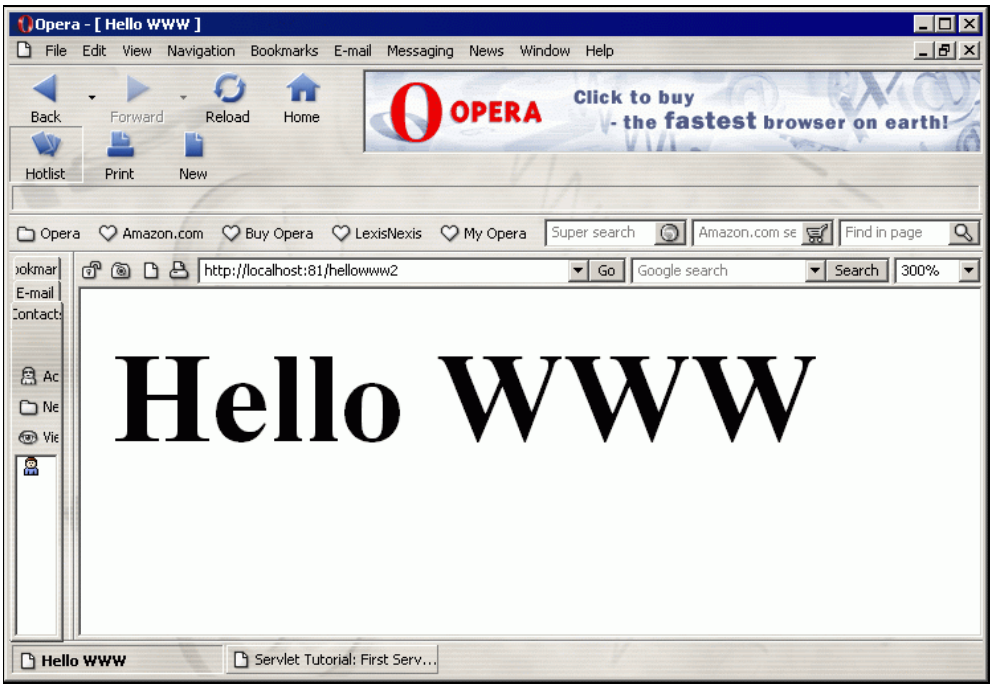


Рис. 4.5. Сервлет HelloWWW2

4.3.2. Обработка данных, полученных из HTML-форм

Сервлеты могут принимать данные, получаемые сервером в составе HTML-форм. Рассмотрим пример, в котором используется класс `ServletUtilities` (листинг 4.5).

Листинг 4.5. Файл ThreeParams.java

```
package paket;
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.util.*;
public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading Three Request Parameters";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY>\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI>param1: "
                + request.getParameter("param1") + "\n" +
                "  <LI>param2: "
                + request.getParameter("param2") + "\n" +
                "  <LI>param3: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

В файл web.ini вставим следующий фрагмент:

```
servlet.name: param
servlet.param.className: paket.ThreeParams
servlet.param.url:  /param
```

Перезапустим Web-сервер и обратимся к сервлету, указав адрес в виде **http://localhost:81/param?param1=Viktoria&param2=parametr2&param3=escheparametr**

В программе описано три параметра:

- param1
- param2
- param3

Это имена, чьи значения указаны в адресе URL после знака равенства. Здесь использован метод `GET` для пересылки параметров. Такой метод подразумевает пересылку строки запроса в составе URL. Браузер отобразит ответ (рис. 4.6).

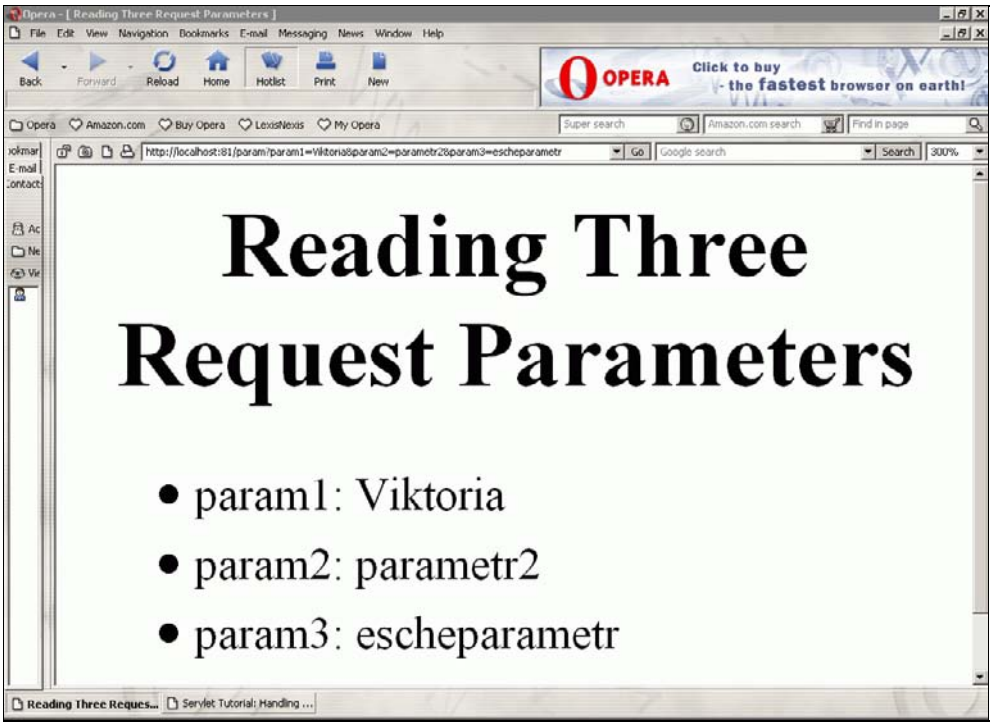


Рис. 4.6. Отображение параметров запросов

Возможна ситуация, когда сервлет не знает, какие имена параметров могут быть переданы с запросом. Ниже приводится пример (листинг 4.6), который просматривает все имена параметров запроса и составляет таблицу. При этом учитываются также имена, у которых нет значения, а также имена, имеющие несколько значений. Имена параметров просматриваются с использованием функции `getParameterNames`. Это метод интерфейса `HttpServletRequest` (точнее, суперинтерфейса `ServletRequest`). Метод возвращает `java.util.Enumeration`. Список имен можно просматривать с использованием проверки наличия следующего элемента при помощи метода `hasMoreElements`, а затем с применением метода `nextElement`. Метод `nextElement` возвращает переменную типа `Object`, затем преобразует результат к `String` и передает его `getParameterValues`. Таким образом, создается массив строк `String`. Если массив получается длиной в один элемент и содержит пустую строку, то сервлет создает следующий текст: "*No Value*". Если

массив имеет несколько элементов, то они будут выведены в виде маркированного списка (рис. 4.7).

Листинг 4.6. Файл ShowParameters.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class ShowParameters extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Reading All Request Parameters";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Parameter Name<TH>Parameter Value(s)");
    Enumeration paramNames = request.getParameterNames();
    while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.println("<TR><TD>" + paramName + "\n<TD>");
      String[] paramValues = request.getParameterValues(paramName);
      if (paramValues.length == 1) {
        String paramValue = paramValues[0];
        if (paramValue.length() == 0)
          out.print("<I>No Value</I>");
        else
          out.print(paramValue);
      } else {
        out.println("<UL>");
        for(int i=0; i<paramValues.length; i++) {
          out.println("<LI>" + paramValues[i]);
```

```
        }
        out.println("</UL>");
      }
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Дополнения, вносимые в файл web.ini:

```
servlet.name: showparam
servlet.showparam.className: paket.ShowParameters
servlet.showparam.url:  /showparam
```

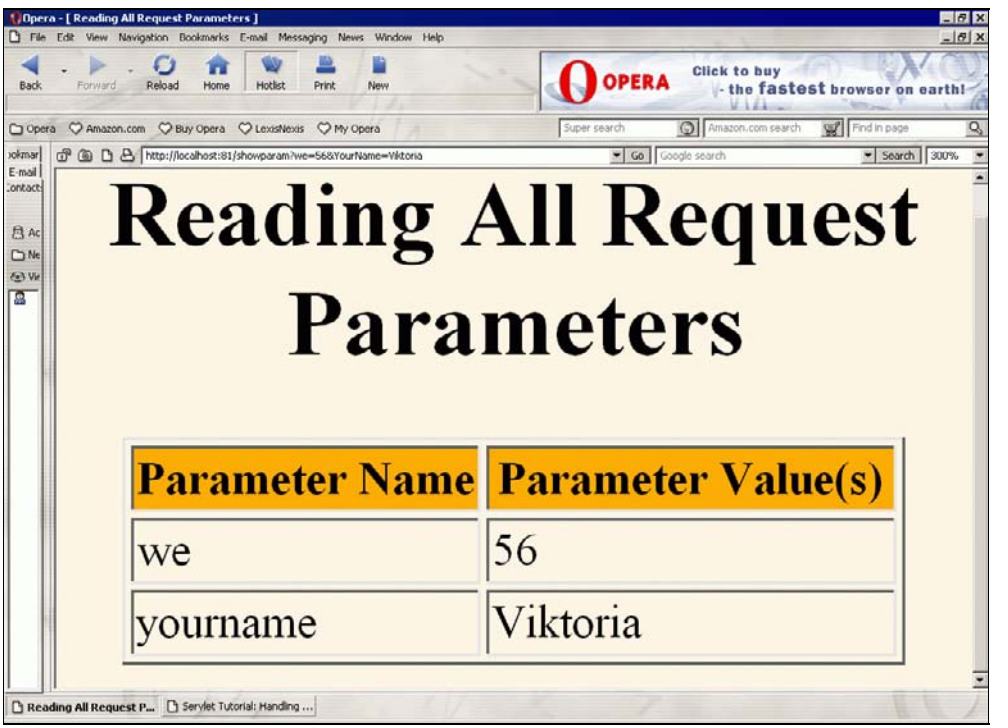


Рис. 4.7. Чтение всех параметров

Для сбора информации от пользователя и передаче его серверу, в том числе, сервлету, используются HTML-формы. В качестве примера приведем файл, содержащий HTML-форму (листинг 4.7).

Листинг 4.7. Файл PostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>A Sample FORM using POST</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>

<FORM ACTION="http://localhost:81/ShowParam"
      METHOD="POST">
  Номер предмета:
  <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Количество:
  <INPUT TYPE="TEXT" NAME="quantity"><BR>
  Цена за штуку:
  <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
  <HR>
  Имя:
  <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Фамилия:
  <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Отчество (одна буква):
  <INPUT TYPE="TEXT" NAME="initial"><BR>
  Адрес доставки:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Кредитная карта:<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Visa">Visa<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Master Card">Master Card<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Amex">American Express<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
```

```
                    VALUE="Discover">Discover<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                    VALUE="Java SmartCard">Java SmartCard<BR>
  Номер карты:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  номер карты повторить:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT" VALUE="Submit Order">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

Собранные данные показаны на рис. 4.8. Обработанные данные представлены на рис. 4.9.

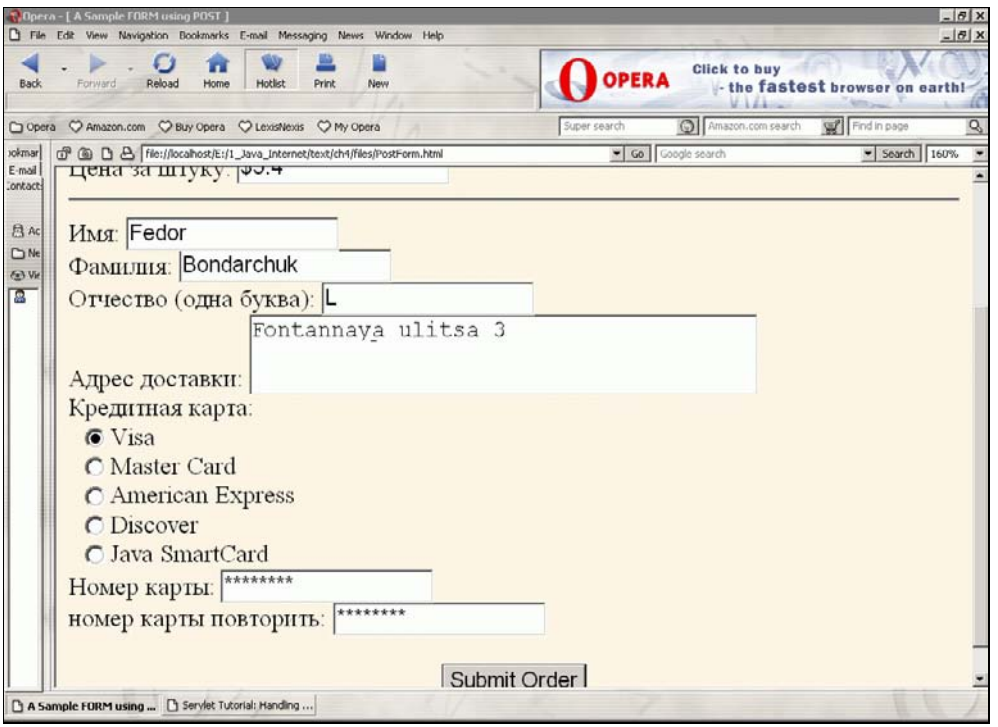


Рис. 4.8. Данные, вводимые пользователем

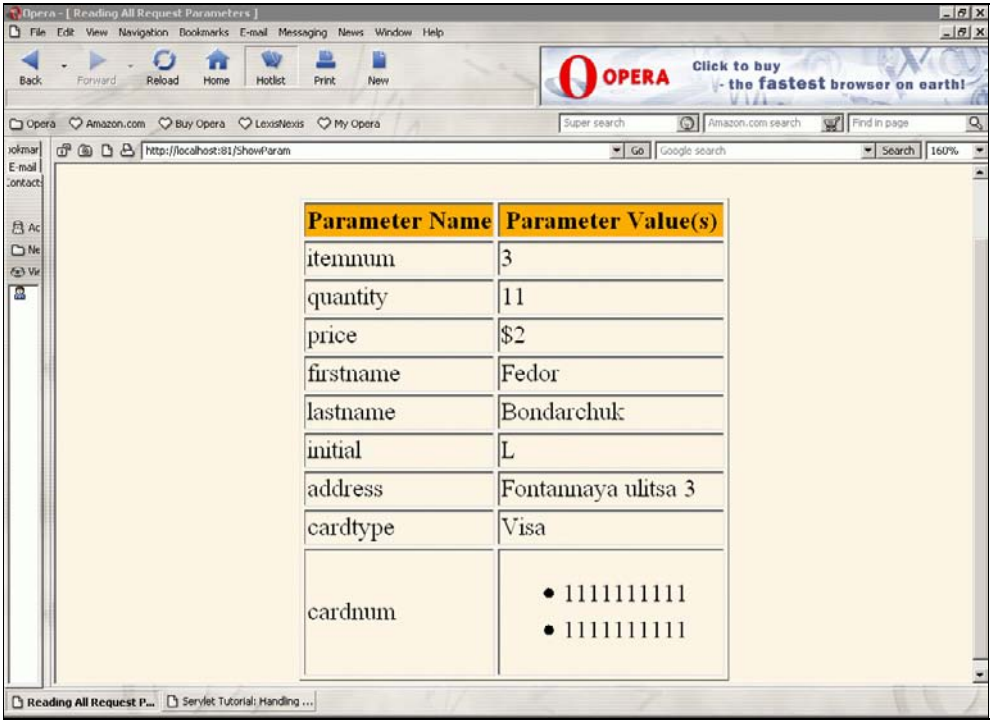


Рис. 4.9. Данные, обработанные сервлетом

При помощи этой формы можно собрать пользовательские данные и получить их на сервере по указанному в качестве значения атрибута `ACTION` адресу. Элемент `<FORM>` также содержит атрибут `METHOD`, который указывает метод отправки данных (рис. 4.8).

4.3.3. HTTP-заголовки

Когда HTTP-клиент (браузер) посылает запрос по протоколу HTTP, то первой строкой посылается заголовок. Клиент может послать несколько заголовков. Почти все из них не являются обязательными, за исключением заголовка `Content-Length`, который используется в том случае, если тип запроса `POST`. Перечислим некоторые наиболее часто употребляемые заголовки.

- `Accept` — MIME-типы, понятные браузеру.
- `Accept-Charset` — наборы символов, понятные браузеру.
- `Accept-Encoding` — тип кодировки (gzip и т. п.).
- `Accept-Language` — язык.
- `Authorization` — информация по аутентификации.

- `Connection` — используются ли персистентные соединения. Если сервлет получает значение `Keep-Alive`, то могут быть использованы персистентные соединения. Это полезно при создании такого ответа, который может содержать заранее неизвестное количество компонентов (рисунков или апплетов). При этом сервер должен будет послать заголовок `Content-Length`. Также удобно пользоваться `ByteArrayOutputStream`.
- `Content-Length` — длина ответа (размер приложения) для метода `POST`.
- `Cookie` — для работы с cookies.
- `From` — для адресов e-mail, не используется в браузерах.
- `Host` — хост и порт.
- `If-Modified-Since` — возвращать документы, более новые, чем указанный срок.
- `Pragma` — не использовать кэш. Сервер должен возвращать только обновленный документ, даже если прокси-сервер содержит локальную копию запрашиваемого документа.
- `Referer` — откуда произошло обращение к данной странице.
- `User-Agent` — тип пользовательского агента, тип браузера.
- `UA-Pixels`, `UA-Color`, `UA-OS`, `UA-CPU` — нестандартные заголовки для задания размеров экрана, глубины отображаемых цветов, типа процессора.

Чтение заголовка — очень простой процесс. Вызывается метод `getHeader HttpServletRequest`, который возвращает `String` или `null`. Существуют и другие методы, в том числе метод `getCookies`, а также методы `getAuthType`, `getRemoteUser`, `getDateHeader`, `getIntHeader`. Метод `getMethod` возвращает основной метод запроса, то есть один из `GET` или `POST`, а также `HEAD`, `PUT`, `DELETE`. Метод `getRequestURI` возвращает URI. Метод `getRequestProtocol` возвращает протокол, например "HTTP/1.0" или "HTTP/1.1".

Ниже приведен пример, в котором сервлет читает все заголовки (листинг 4.8). Сервлет создает таблицу, в которую выводит прочитанную информацию. Он также выводит три компонента строки запроса (метод, URI и строку с названием протокола).

Листинг 4.8. Файл ShowRequestHeader.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
```

```
public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Servlet Example: Showing Request Headers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<B>Request Method: </B>" +
                request.getMethod() + "<BR>\n" +
                "<B>Request URI: </B>" +
                request.getRequestURI() + "<BR>\n" +
                "<B>Request Protocol: </B>" +
                request.getProtocol() + "<BR><BR>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Header Name<TH>Header Value");
    Enumeration headerNames = request.getHeaderNames();
    while(headerNames.hasMoreElements()) {
      String headerName = (String)headerNames.nextElement();
      out.println("<TR><TD>" + headerName);
      out.println("    <TD>" + request.getHeader(headerName));
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Изменения, вносимые в файл конфигурации Web-сервера, следующие:

```
servlet.name: showheaders
servlet.showheaders.className: paket.ShowRequestHeqders
servlet.showheaders.url:  /showheaders
```

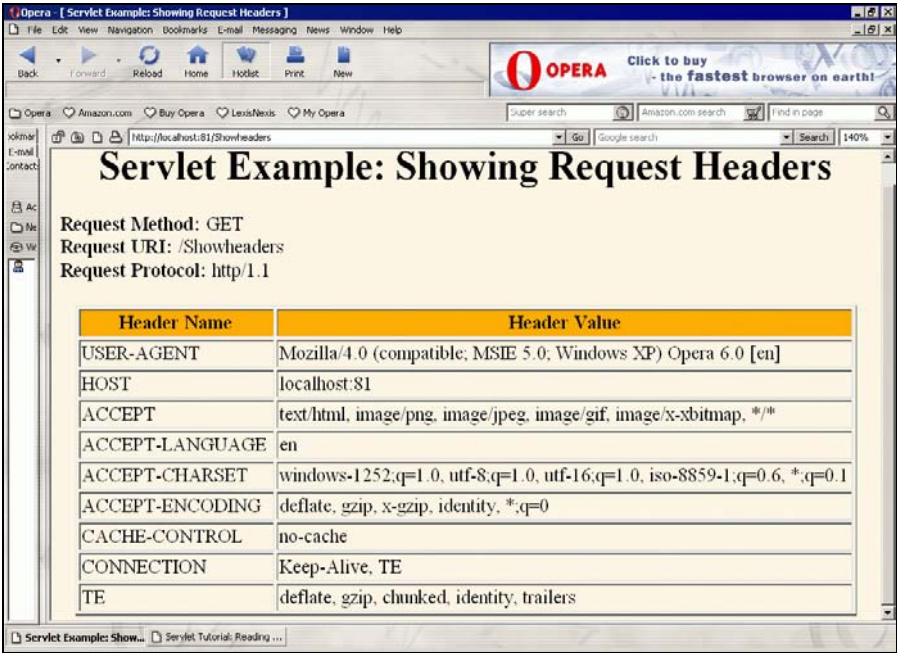


Рис. 4.10. Ответ сервлета на запрос браузера Opera

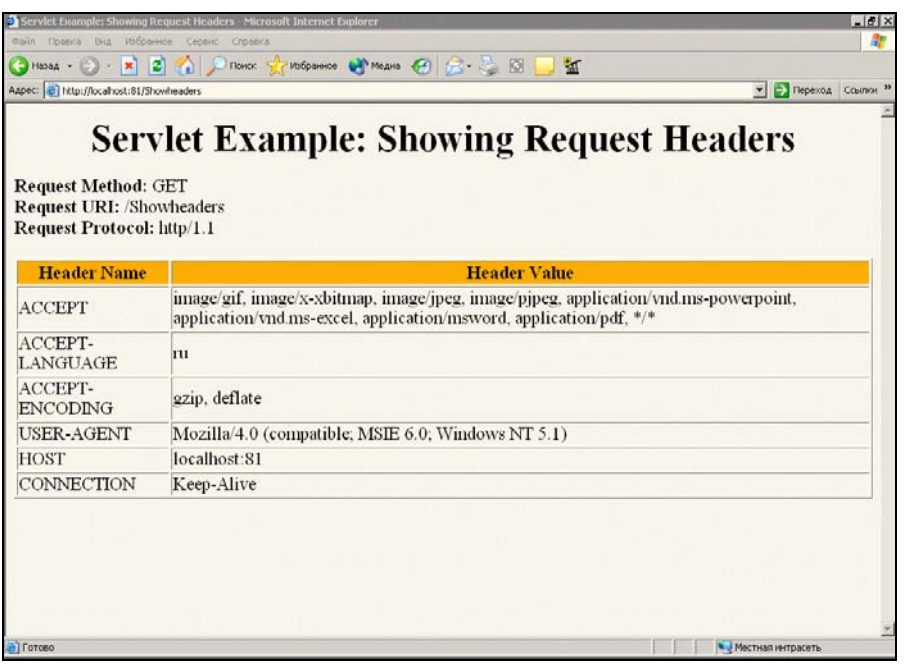


Рис. 4.11. Ответ сервлета на запрос браузера IE6

Ответ, получаемый от сервлета, выглядит так, как показано на рис. 4.10 и рис. 4.11 (для двух типов браузеров).

Далее будет рассмотрено, как можно обращаться к значениям, посылаемым в HTML-форме.

4.3.4. Сервлеты и переменные CGI

При работе с CGI-приложениями мы сталкиваемся с большим количеством переменных, используемых в CGI-программировании. Источники CGI-переменных могут быть различны. Информация собирается из разных мест, включая информацию, содержащуюся в HTTP-запросах (заголовки содержат URI). При этом часть информации получается из сокетов (IP-адреса), часть получается из параметров инсталляции сервера (отображение URL в физический путь). Полезно сопоставить переменные, используемые в CGI, с аналогами Java-сервлетов (табл. 4.3).

Таблица 4.3. *Переменные окружения*

Переменная CGI	Смысл или назначение	Обращение к значению из doGet или из doPost
`AUTH_TYPE`	Если присутствует заголовок `Authorization`, то задается схема (`basic` или `digest`)	request.getAuthType()
`CONTENT_LENGTH`	Для запросов `POST`. Количество байт	эквивалентно `String.valueOf(request.getContentLength())` — строка полезно использовать `request.getContentLength()` — значение типа `int`
`CONTENT_TYPE`	MIME-тип приложения	`Request.getContentType()`
`DOCUMENT_ROOT`	Путь к каталогу, соответствующему http://имя-хоста/	`GetServletContext().getRealPath("/")`, в старых версиях `request.getRealPath("/")`
`HTTP_XXX_YYY`	Доступ к заголовку	`Request.getHeader("Xxx-Yyy")`
`PATH_INFO`	Информация о пути	`Request.getPathInfo()`
`PATH_TRANSLATED`	Путь переведенный в реальный путь к сервлету	`Request.getPathTranslated()`
`QUERY_STRING`	Данные (для запроса GET)	`Request.getQueryString()`

Таблица 4.3 *(окончание)*

Переменная CGI	Смысл или назначение	Обращение к значению из doGet или из doPost
REMOTE_ADDR	IP-адрес в виде строки, например "192.9.48.9"	Request.getRemoteAddr()
REMOTE_HOST	Полное доменное имя, например "java.sun.com"	Request.getRemoteHost()
REMOTE_USER	Пользовательская часть аутентификации, если есть заголовок Authorization	Request.getRemoteUser()
REQUEST_METHOD	Тип запроса (обычно GET или POST, но может быть и HEAD, PUT, DELETE, OPTIONS, TRACE)	Request.getMethod()
SCRIPT_NAME	Путь к сервлету	Request.getServletPath()
SERVER_NAME	Имя Web-сервера	Request.getServerName()
SERVER_PORT	Номер порта	эквивалентно String.valueOf(request.getServerPort()) — строка, полезен метод request.getServerPort() — значение типа int
SERVER_PROTOCOL	Версия протокола HTTP (HTTP/1.1)	Request.getProtocol()
SERVER_SOFTWARE	Информация о Web-сервере	GetServletContext(). getServerInfo()

В качестве иллюстрации к вышеизложенному приведем пример сервлета, который выводит значения переменных окружения (листинг 4.9).

Листинг 4.9. Файл ShowCGI.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
```

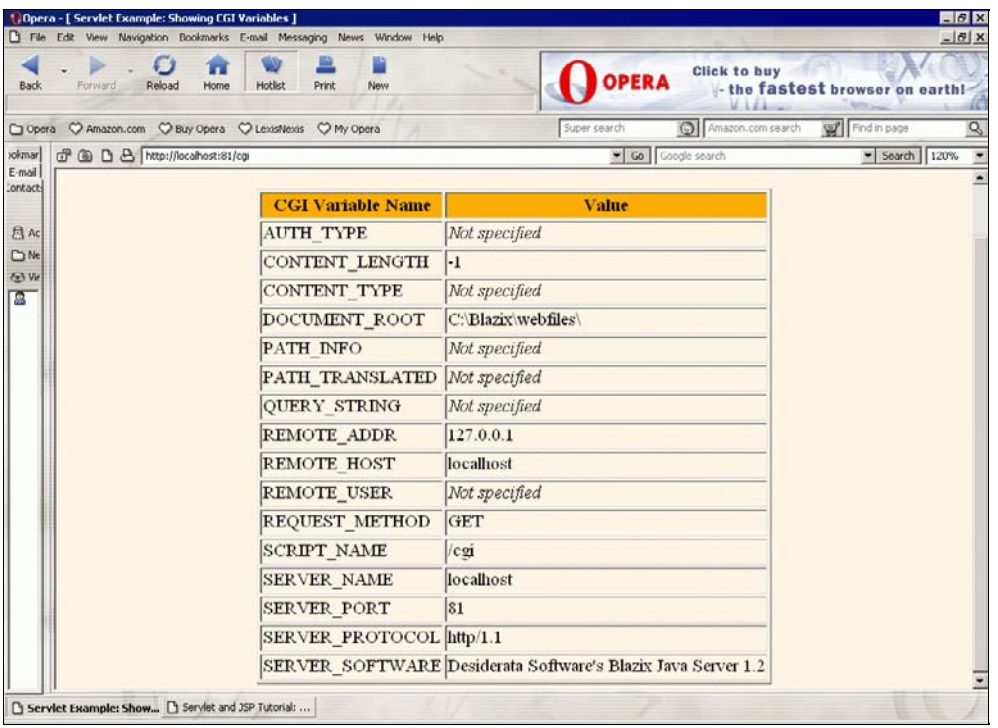
```
public class ShowCGI extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String[][] variables =
      { { "AUTH_TYPE", request.getAuthType() },
        { "CONTENT_LENGTH", String.valueOf(request.getContentLength()) },
        { "CONTENT_TYPE", request.getContentType() },
        { "DOCUMENT_ROOT", getServletContext().getRealPath("/") },
        { "PATH_INFO", request.getPathInfo() },
        { "PATH_TRANSLATED", request.getPathTranslated() },
        { "QUERY_STRING", request.getQueryString() },
        { "REMOTE_ADDR", request.getRemoteAddr() },
        { "REMOTE_HOST", request.getRemoteHost() },
        { "REMOTE_USER", request.getRemoteUser() },
        { "REQUEST_METHOD", request.getMethod() },
        { "SCRIPT_NAME", request.getServletPath() },
        { "SERVER_NAME", request.getServerName() },
        { "SERVER_PORT", String.valueOf(request.getServerPort()) },
        { "SERVER_PROTOCOL", request.getProtocol() },
        { "SERVER_SOFTWARE", getServletContext().getServerInfo() }
      };
    String title = "Servlet Example: Showing CGI Variables";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>CGI Variable Name<TH>Value");
    for(int i=0; i<variables.length; i++) {
      String varName = variables[i][0];
      String varValue = variables[i][1];
      if (varValue == null)
        varValue = "<I>Not specified</I>";
      out.println("<TR><TD>" + varName + "<TD>" + varValue);
    }
    out.println("</TABLE></BODY></HTML>");
```

```
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Вставка в файл web.ini следующая:

```
servlet.name: cgi
servlet.cgi.className: paket.ShowCGI
servlet.cgi.url:  /cgi
```

Ответ, получаемый браузером, показан на рис. 4.12.



CGI Variable Name	Value
AUTH_TYPE	*Not specified*
CONTENT_LENGTH	-1
CONTENT_TYPE	*Not specified*
DOCUMENT_ROOT	C:\Blazix\webfiles\
PATH_INFO	*Not specified*
PATH_TRANSLATED	*Not specified*
QUERY_STRING	*Not specified*
REMOTE_ADDR	127.0.0.1
REMOTE_HOST	localhost
REMOTE_USER	*Not specified*
REQUEST_METHOD	GET
SCRIPT_NAME	/cgi
SERVER_NAME	localhost
SERVER_PORT	81
SERVER_PROTOCOL	http/1.1
SERVER_SOFTWARE	Desiderata Software's Blazix Java Server 1.2

Рис. 4.12. Отображение полученного ответа браузером

4.3.5. Коды состояния

В HTTP используются коды состояния, которые несут некоторую общую мета-информацию о том, как был выполнен запрос. Когда Web-сервер

посылает ответ браузеру или другому клиенту, от которого он получил запрос, то ответ, как правило, включает в себя строку состояния, заголовки, пустую строку, отделяющую заголовки от тела ответа, и сам документ, посылаемый в качестве ответа. В простейшем случае ответ может быть следующим:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, World!
```

Строка состояния включает в себя информацию о версии HTTP, которая есть целое число. Затем следует код состояния. После строки состояния идут строки с заголовками, в частности, строка, задающая mime-тип посылаемого документа. Строки заголовков отделяются от тела ответа пустой строкой. Наиболее распространенные коды состояний приведены ниже.

- 100 — продолжение выполнения запроса.
- 101 — изменение протокола, сервер меняет протокол (на новый HTTP 1.1).
- 200 — OK. Все в порядке. Если не задан другой ответ для строки статуса, то, как правило, получают именно такой заголовок при работе с сервлетами.
- 201 — Created. Сервер создал документ, заголовок `Location` содержит соответсвующий URL.
- 202 — Accepted. Запрос принят к обработке, которая еще не завершена.
- 203 — Несоответствующая информация. Документ возвращается обычным порядком, но некоторые заголовки могут быть неточными.
- 204 — No Content. Нет нового документа, браузер должен продолжать показывать старый документ. Это полезно в том случае, если пользователь периодически перезагружает страницу (обновляет ее). Этот код не работает со страницами, которые производят автоматическое обновление с помощью заголовка `Refresh` или ярлыка `<META HTTP-EQUIV="Refresh" ...>`, поскольку получаемый код статуса прекращает перезагрузку. В таких ситуациях автоматическая перезагрузка с использованием JavaScript вполне работоспособна.
- 205 — Reset Content. Нового документа нет, но браузер должен обновить вид документа. Полезно в тех случаях, когда необходимо очистить поля CGI-формы.
- 206 — Partial Content. Клиент посылает частичный запрос, сервер удовлетворяет его.
- 300 — Multiple Choices. Множественный документ, который может быть обнаружен в нескольких местах. Предпочтительное положение передается браузеру в заголовке `Location`.

- 301 — Удалено навсегда (Moved Permanently). Документ удален, браузер должен перейти по указанному в заголовке `Location` адресу URL.
- 302 — Found. Схоже с 301, но запрашиваемый URL временно удален.
- 303 — See Other. Схоже с 301 и 302, но если запрос использовал метод `POST`, то документ, на который происходит переадресация, будет запрошен с использованием метода `GET`.
- 304 — Not Modified. Клиент сохраняет документ в кэше и выполняет условный запрос (например, с использованием заголовка `If-Modified-Since`). Сервер сообщает клиенту, что старый документ не был изменен, то есть можно использовать документ, хранящийся в кэше.
- 305 — Use Proxy. Запрошенный документ должен быть получен через прокси-сервер, указанный в заголовке `Location`.
- 307 — Temporary Redirect. Идентично 302.
- 400 — Bad Request. Неправильный синтаксис запроса.
- 401 — Unauthorized. Клиент пытается получить защищенный паролем документ. Необходимо иметь заголовок `WWW-Authenticate`.
- 403 — Forbidden. Ресурс недоступен вне зависимости от аутентификации.
- 404 — Not Found. Ресурс не существует.
- 405 — Method Not Allowed. Метод (`GET`, `POST`, `HEAD`, `DELETE`, `PUT`, `TRACE` и т. п.) не разрешен в применении к данному ресурсу.
- 406 — Not Acceptable. Ресурс создает тип mime, который не совместим с теми, что использует клиент (в заголовке `Accept`).
- 407 — Proxy Authentication Required. Подобен 401, но прокси-сервер должен возвращать заголовок `Proxy-Authenticate`.
- 408 — Request Timeout. Клиент слишком долго посылает запрос.
- 409 — Conflict. Как правило, связано с запросами `PUT`, например, при закачке неподдерживаемых типов файлов.
- 410 — Gone. Документ потерян, адрес переадресовки неизвестен. Отличается от 404 тем, что документ потерян навсегда.
- 411 — Length Required. Сервер не сможет обработать запрос, если клиент не пошлет заголовок `Content-Length`.
- 412 — Precondition Failed. Условие, необходимое и заданное в заголовке, не выполнено.
- 413 — Request Entity Too Large. Запрошенный документ оказывается большим, чем может быть обработан сервером. Если сервер полагает, что запрос может быть обработан позже, то он возвращает заголовок `Retry-After header`.

- 414 — Request URI Too Long. Адрес URI слишком длинный.
- 415 — Unsupported Media Type. Неизвестный формат в запросе.
- 416 — Requested Range Not Satisfiable. Клиент использует невыполнимый заголовок `Range`.
- 417 — Expectation Failed. Значение в заголовке `Expect` не выполнено.
- 500 — Internal Server Error. Внутренняя ошибка сервера.
- 501 — Not Implemented. Сервер не поддерживает необходимые для выполнения запроса функции.
- 502 — Bad Gateway. Используется сервером, работающим в качестве прокси-сервера. Показывает, что сервер получил неверный ответ от удаленного сервера.
- 503 — Service Unavailable. Сервер не может ответить в силу состояния технического обслуживания или перегрузки.
- 504 — Gateway Timeout. Используется шлюзом или прокси-сервером. Показывает, что исходный сервер не получил ответа за установленное время.
- 505 — HTTP Version Not Supported. Неподдерживаемая версия HTTP.

Далее приводится пример (листинг 4.10), в котором используются строки состояния с кодами 302 и 404. Код 302 задается при помощи метода `sendRedirect`, а код 404 устанавливается методом `sendError`.

Листинг 4.10. Файл SearchEngines.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
public class SearchEngines extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String searchString =
      URLEncoder.encode(request.getParameter("searchString"));
    String numResults =
      request.getParameter("numResults");
    String searchEngine =
      request.getParameter("searchEngine");
```

```
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    for(int i=0; i<commonSpecs.length; i++) {
      SearchSpec searchSpec = commonSpecs[i];
      if (searchSpec.getName().equals(searchEngine)) {
        String url =
          response.encodeURL(searchSpec.makeURL(searchString,
                                                numResults));
        response.sendRedirect(url);
        return;
      }
    }
    response.sendError(response.SC_NOT_FOUND,
                       "No recognized search engine specified.");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Класс `SearchSpec` описан в следующем файле (листинг 4.11).

Листинг 4.11. Файл SearchSpec.java

```
package paket;
class SearchSpec {
  private String name, baseURL, numResultsSuffix;
  private static SearchSpec[] commonSpecs =
    { new SearchSpec("google",
                     "http://www.google.com/search?q=",
                     "&num="),
      new SearchSpec("infoseek",
                     "http://infoseek.go.com/Titles?qt=",
                     "&nh="),
      new SearchSpec("lycos",
                     "http://lycospro.lycos.com/cgi-bin/pursuit?query=",
                     "&maxhits="),
      new SearchSpec("hotbot",
                     "http://www.hotbot.com/?MT=",
```

```
                    "&DC=")
    };
  public SearchSpec(String name,
                    String baseURL,
                    String numResultsSuffix) {
    this.name = name;
    this.baseURL = baseURL;
    this.numResultsSuffix = numResultsSuffix;
  }

  public String makeURL(String searchString, String numResults) {
    return(baseURL + searchString + numResultsSuffix + numResults);
  }

  public String getName() {
    return(name);
  }

  public static SearchSpec[] getCommonSpecs() {
    return(commonSpecs);
  }
}
```

Фронт-энд описан в HTML-файле (листинг 4.12).

Листинг 4.12. Файл SearchEngines.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Searching the Web</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Searching the Web</H1>
<FORM ACTION="http://localhost:81/search">
  <CENTER>
    Search String:
    <INPUT TYPE="TEXT" NAME="searchString"><BR>
    Results to Show Per Page:
    <INPUT TYPE="TEXT" NAME="numResults"
```

```
                    VALUE=10 SIZE=3><BR>
    <INPUT TYPE="RADIO" NAME="searchEngine"
                    VALUE="google">
    Google |
    <INPUT TYPE="RADIO" NAME="searchEngine"
                    VALUE="infoseek">
    Infoseek |
    <INPUT TYPE="RADIO" NAME="searchEngine"
                    VALUE="lycos">
    Lycos |
    <INPUT TYPE="RADIO" NAME="searchEngine"
                    VALUE="hotbot">
    HotBot
    <BR>
    <INPUT TYPE="SUBMIT" VALUE="Search">
  </CENTER>
</FORM>
</BODY>
</HTML>
```

4.3.6. Заголовки HTTP в ответе сервера

Как правило, ответ сервера включает в себя несколько заголовков, в том числе статус-код. С некоторыми статус-кодами предполагается использование определенного набора заголовков. Итак, статус-код 401 (`Unauthorized`) предполагает наличие заголовка `WWW-Authenticate`. Наиболее удобный способ генерирования заголовков — использование метода `setHeader` (интерфейс `HttpServletResponse`). В качестве параметров метод получает имя заголовка и значение для этого типа заголовка. Существует два специальных метода для работы с заголовками — это метод для работы с датами `setDateHeader` и метод для работы с целыми значениями `setIntHeader`. Первый метод переводит дату в миллисекунды, в том формате, который возвращается методом `System.currentTimeMillis` или методом `getTime` в применении к объекту типа `Date`. Второй метод помогает производить преобразование целого значения в строку.

Интерфейс `HttpServletResponse` содержит набор методов, облегчающих работу с общепринятыми заголовками. Метод `setContentType` задает заголовок `Content-Type`. Метод применяется в большинстве сервлетов. Метод `setContentLength` задает заголовок `Content-Length`. Этот заголовок полезен в том случае, если браузер поддерживает персистентные соединения. Метод `addCookie` задает `Cookies`. Метод `sendRedirect` задает заголовок `Location`.

Общепринятые заголовки и их назначение

Ниже приводится список наиболее употребительных и общепринятых HTTP-заголовков, используемых при генерации ответа сервером.

- `Allow`

  Методы запроса, поддерживаемые сервером (например, `GET`, `POST` и т. п.).

- `Content-Encoding`

  Метод кодирования документа.

- `Content-Length`

  Количество байтов, посылаемых в ответе. Используется при персистентных соединениях.

- `Content-Type`

  Тип mime посылаемого документа.

- `Date`

  Текущее время и дата.

- `Expires`

  Срок годности посылаемого документа.

- `Last-Modified`

  Время последнего изменения посылаемого документа.

- `Location`

  Местоположение документа (при переадресации).

- `Refresh`

  Как часто браузер должен запрашивать измененную страницу, указывается интервал в секундах. (Например, `setHeader("Refresh", "5; URL=http://host/path")`. Аналогичный результат может быть достигнут с использованием ярлыка `<META>`: `<META HTTP-EQUIV="Refresh" CONTENT="5; URL=http://host/path">`.)

- `Server`

  Тип сервера.

- `Set-Cookie`

  Задание Cookies.

- `WWW-Authenticate`

  Тип аутентификации, например `response.setHeader("WWW-Authenticate", "BASIC realm=\"executives\"")`.

Пример периодически загружаемой страницы, содержимое которой меняется со временем, приведен в листинге 4.13. Пользовательский интерфейс

выполнен в виде HTML-страницы. Это самообновляющаяся страница. Страница взаимодействует с сервлетом PrimeNumbers.class (листинг 4.14), вычисляющим простые числа.

Листинг 4.13. Файл PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Поиск простых чисел</H2>
<BR><BR>
<CENTER>
<FORM ACTION="http://localhost:81/PrimeNumbers">
  <B>Сколько простых чисел найти:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Количество десятичных знаков в простом числе:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>
<BR><BR>
Этот пример призван продемонстрировать два свойства сервлетов Java-
servlets:
<UL>
  <LI><B>Способность сервлетов поддерживать персистентную связь.</B>
В этом случае сервлет выполняет низкоприоритетный поток, который
вычисляет простые числа.
Когда клиент запрашивает обновление, сервлет посылает новые данные (если
есть).
Сервлет также поддерживает набор из 30 последних полученных результатов.
Это на тот случай, если клиент запросит то, что уже ранее было вычислено.
  <LI><B>Способность сервлетов работать с заголовками HTTP.</B>
В нашем случае сервлет посылает заголовок <CODE>Refresh</CODE>
в том случае, если процесс вычисления все еще не завершен.
После этого заголовка выводится страница, содержащая промежуточный
результат.
```

```
</UL>
<P>
<HR>
</BODY>
</HTML>
```

Листинг 4.14. Файл PrimeNumbers.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/*
Сервлет, создающий набор последовательных простых чисел общим
количеством  n, каждое число содержит m знаков.
Вычисления производятся в отдельном низкоприоритетном потоке,
результат выводится с отображением только тех чисел,
которые уже вычислены.
Если процесс вычисления не закончен,
то апплет посылает браузеру заголовок Refresh
с указанием времени повторного обращения к серверу
для отображения ранее непросчитанных простых чисел.
Сервер также поддерживает список ранее вычисленных чисел,
которые могут быть возвращены клиенту
при его запросе немедленно без вычислений.
*/
public class PrimeNumbers extends HttpServlet {
  private static Vector primeListVector = new Vector();
  private static int maxPrimeLists = 30;
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    int numPrimes =
      ServletUtilities.getIntParameter(request, "numPrimes", 50);
    int numDigits =
      ServletUtilities.getIntParameter(request, "numDigits", 120);
    PrimeList primeList =
```

```
      findPrimeList(primeListVector, numPrimes, numDigits);
    if (primeList == null) {
      primeList = new PrimeList(numPrimes, numDigits, true);
      synchronized(primeListVector) {
        if (primeListVector.size() >= maxPrimeLists)
          primeListVector.removeElementAt(0);
        primeListVector.addElement(primeList);
      }
    }
    Vector currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
      response.setHeader("Refresh", "5");
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Some " + numDigits + "-Digit Prime Numbers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
                "<H3>Primes found with " + numDigits +
                " or more digits: " + numCurrentPrimes + ".</H3>");
    if (isLastResult)
      out.println("<B>Done searching.</B>");
    else
      out.println("<B>Still looking for " + numPrimesRemaining +
                  " more<BLINK>...</BLINK></B>");
    out.println("<OL>");
    for(int i=0; i<numCurrentPrimes; i++) {
      out.println("  <LI>" + currentPrimes.elementAt(i));
    }
    out.println("</OL>");
    out.println("</BODY></HTML>");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
```

```
      throws ServletException, IOException {
    doGet(request, response);
  }

/*
Проверка того, есть ли уже вычисленные ранее числа
или происходит ли в данный момент вычисление чисел,
совпадающих с условиями нового запроса.
Если числа такие есть, то повторные вычисления производиться не будут.
Поддерживаемый список не должен быть большим,
чтобы не занимать много памяти.
Доступ к списку должен быть синхронизирован
во избежание возникновения одновременных запросов.
*/
  private PrimeList findPrimeList(Vector primeListVector,
                                  int numPrimes,
                                  int numDigits) {
    synchronized(primeListVector) {
      for(int i=0; i<primeListVector.size(); i++) {
        PrimeList primes = (PrimeList)primeListVector.elementAt(i);
        if ((numPrimes == primes.numPrimes()) &&
            (numDigits == primes.numDigits()))
          return(primes);
      }
      return(null);
    }
  }
}
```

Помимо этого файла потребуются еще два файла PrimeList.java (листинг 4.15) и Primes.java (листинг 4.16).

Листинг 4.15. Файл PrimeList.java

```
package paket;
import java.util.*;
import java.math.BigInteger;
/*
создается вектор больших простых чисел
*/
```

```
public class PrimeList implements Runnable {
  private Vector primes;
  private int numPrimes, numDigits;
  // находим простые числа numPrimes; каждое число содержит количество
  // цифр, не меньшее, чем numDigits
  public PrimeList(int numPrimes, int numDigits,
                   boolean runInBackground) {
    // используем Vector, но не ArrayList
    // (для совместимости с JDK 1.1)
    primes = new Vector(numPrimes);
    this.numPrimes = numPrimes;
    this.numDigits = numDigits;
    if (runInBackground) {
      Thread t = new Thread(this);
      // устанавливаем низкий приоритет,
      // чтобы не "затормозить" работу сервера
      t.setPriority(Thread.MIN_PRIORITY);
      t.start();
    } else {
      run();
    }
  }
  public void run() {
    BigInteger start = Primes.random(numDigits);
    for(int i=0; i<numPrimes; i++) {
      start = Primes.nextPrime(start);
      synchronized(this) {
        primes.addElement(start);
      }
    }
  }
  public synchronized boolean isDone() {
    return(primes.size() == numPrimes);
  }
  public synchronized Vector getPrimes() {
    if (isDone())
      return(primes);
```

```
    else
      return((Vector)primes.clone());
  }

  public int numDigits() {
    return(numDigits);
  }
  public int numPrimes() {
    return(numPrimes);
  }
  public synchronized int numCalculatedPrimes() {
    return(primes.size());
  }
}
```

Листинг 4.16. Файл Primes.java

```
package paket;
import java.math.BigInteger;
/*
Утилиты, используемые для создания больших простых чисел.
Случайное большое целое число BigInteger;
следующее простое число должно быть больше этого целого числа.
*/
public class Primes {
  private static final BigInteger ZERO = new BigInteger("0");
  private static final BigInteger ONE = new BigInteger("1");
  private static final BigInteger TWO = new BigInteger("2");
  private static final int ERR_VAL = 100;
  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
```

```
      return(nextPrime(start));
  }
  private static boolean isEven(BigInteger n) {
    return(n.mod(TWO).equals(ZERO));
  }
  private static StringBuffer[] digits =
    { new StringBuffer("0"), new StringBuffer("1"),
      new StringBuffer("2"), new StringBuffer("3"),
      new StringBuffer("4"), new StringBuffer("5"),
      new StringBuffer("6"), new StringBuffer("7"),
      new StringBuffer("8"), new StringBuffer("9") };
  private static StringBuffer randomDigit() {
    int index = (int)Math.floor(Math.random() * 10);
    return(digits[index]);
  }
  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      s.append(randomDigit());
    }
    return(new BigInteger(s.toString()));
  }
  public static void main(String[] args) {
    int numDigits;
    if (args.length > 0)
      numDigits = Integer.parseInt(args[0]);
    else
      numDigits = 150;
    BigInteger start = random(150);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Основная HTML-страница показана на рис. 4.13. Промежуточные результаты показаны на рис. 4.14 и 4.15, а окончательный результат показан на рис. 4.16.

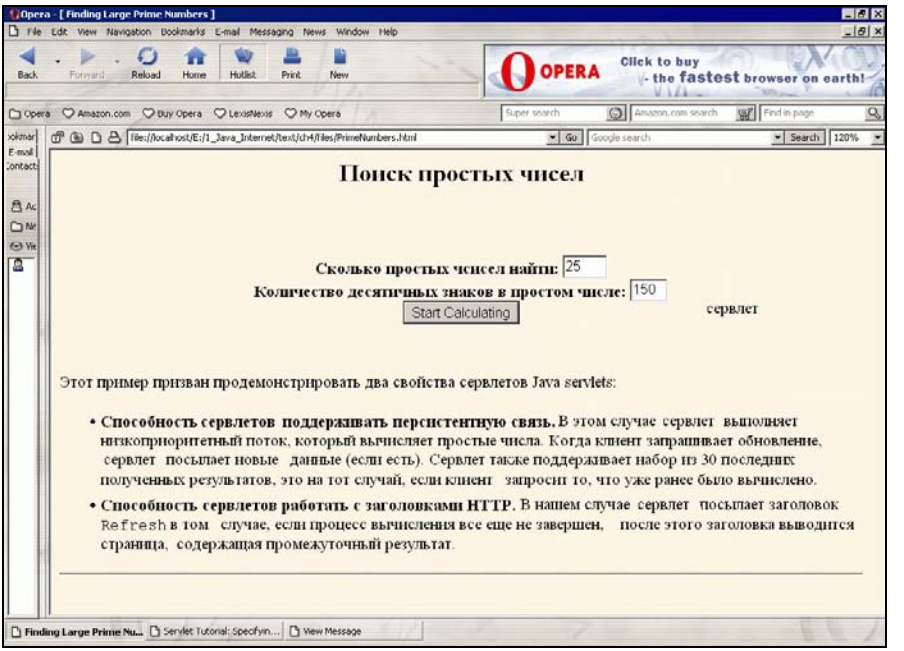


Рис. 4.13. Отображаемая HTML -страничка

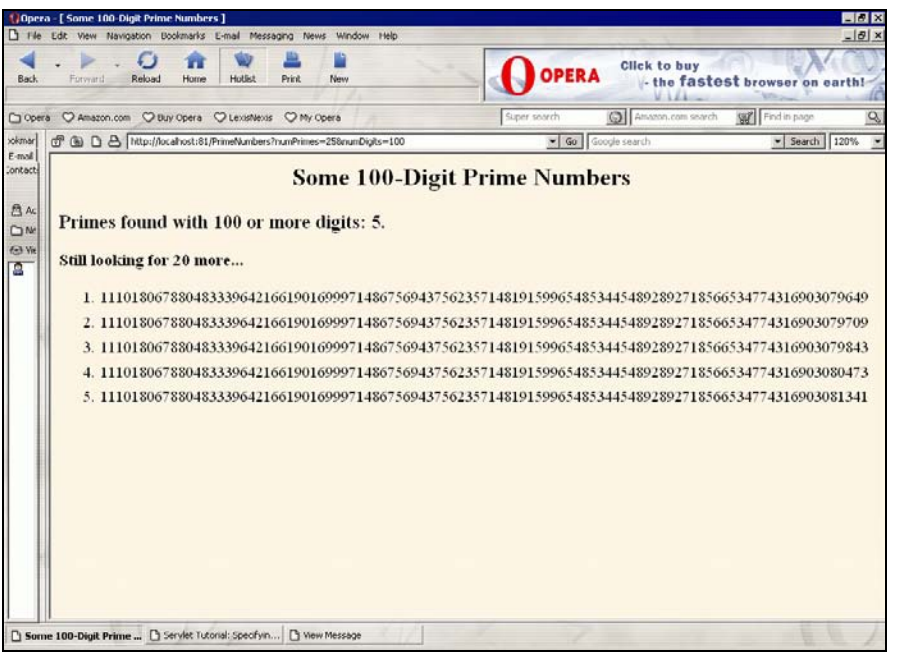


Рис. 4.14. Некоторые числа уже вычислены и отображены в браузере

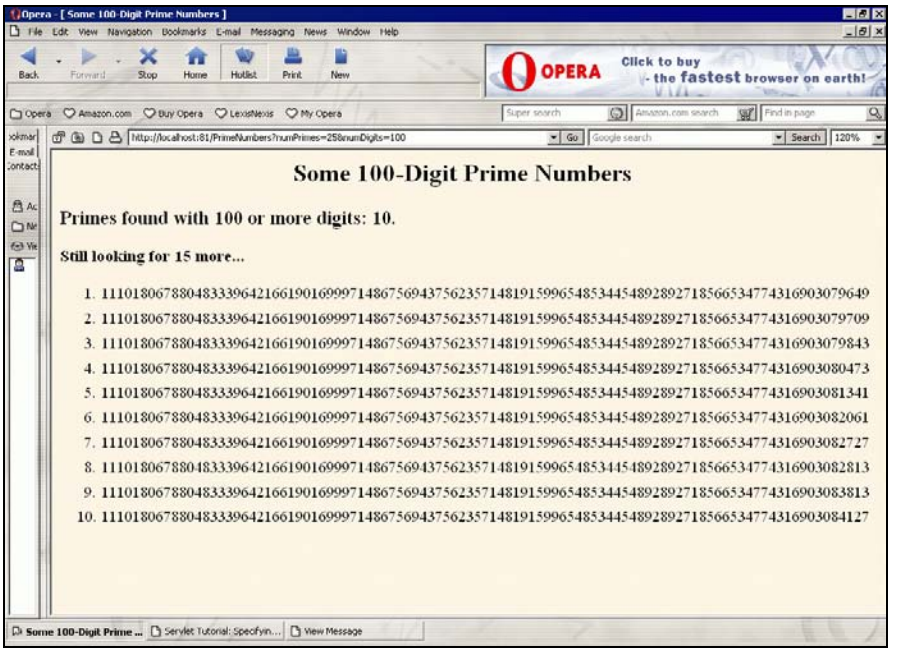


Рис. 4.15. Браузер получил дополнительные промежуточные результаты

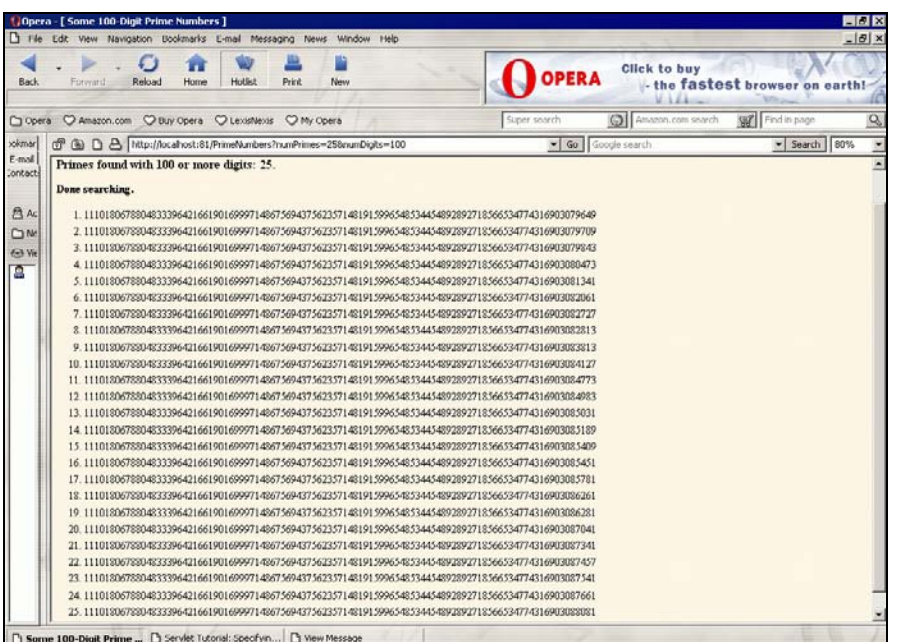


Рис. 4.16. Сервлет завершил рассчет

4.3.7. Работа с Cookies

Cookies — это небольшие фрагменты текстовой информации, которую посылает Web-сервер браузеру, а браузер, когда вновь отображает сайт, возвращает их в первоначальном виде серверу. Cookies используются для решения нескольких типичных задач.

- Идентификация пользователя на протяжении сессии.

  Особенно полезно производить идентификацию сессий на сайтах, занимающихся электронной коммерцией.

- Можно обходиться без имени пользователя и пароля.

  Cookies предоставляют альтернативу: если не требуется высокий уровень безопасности, то вместо паролей можно использовать Cookies.

- Организация сайта в соответствии с пользовательскими предпочтениями.

  Сайт может предлагать производить пользовательскую настройку. Параметры настройки можно передавать в виде Cookies.

- Целенаправленная реклама.

  Cookies могут быть использованы для того, чтобы производить выбор отправляемых баннеров в соответствии с пользовательскими предпочтениями, которые передаются посредством Cookies.

Cookies вполне безопасны. Информация, передаваемая в виде Cookies, никак не может быть использована не по назначению. Cookies никогда никоим образом не интерпретируются, не выполняются, соответственно, они не могут нести в себе вирусов. Помимо этого установлено, что браузер может принимать не более 20 Cookies для каждого сайта, общее количество Cookies не может превышать 300, размер для каждого фрагмента Cookie установлен не более 4 Кбайт. Следовательно, Cookies не смогут занять сколь-либо значительного места на жестком диске и производить какие-либо хакерские атаки.

API для работы с Cookies

Прежде чем послать Cookies клиенту, необходимо создать один или несколько Cookies, указав их имена и задав значения. При этом используется метод `Cookie(name, value)`. Дополнительные атрибуты задаются при помощи `cookie.setXxx`. Заголовок ответа, содержащего `Cookie`, формируется в сервлете с применением метода `response.addCookie(cookie)`. Для чтения возвращаемых клиентом Cookies используется метод `request.getCookies()`. Этот метод возвращает массив объектов типа `Cookie`. Как правило, затем объект просматривается с целью нахождения нужного `Cookie`, для этого используется метод `getName`, значение `Cookie` может быть получено с применением метода `getValue`.

Как создать Cookies

Cookie можно создать при помощи конструктора `Cookie`, которому передается два параметра — имя `Cookie` и значение `Cookie` (оба представляют собой строки). Имя и значение не могут содержать символов пробелов, а также символов:

```
[ ] () =, " / ? @ : ;
```

Задание атрибутов для Cookies

Прежде чем вставить `Cookies` в заголовок, нужно задать атрибуты. При работе с `Cookies` используются следующие атрибуты (указаны соответствующие методы задания значений атрибутов).

- `getComment/setComment`

  Устанавливает/получает комментарий для `Cookie`.

- `getDomain/setDomain`

  Устанавливает/получает домен, для которого предназначен `Cookie`. Домен должен начинаться с точки (например, .e-olymp.com). Для доменов, не относящихся к странам (например, .com, .edu), точек должно быть две, для доменов стран (например, .spb.ru, .edu.es) точек должно быть три.

- `getMaxAge/setMaxAge`

  Устанавливает/получает промежуток времени в секундах до истечения срока годности `Cookie`. Если это значение не установлено, то `Cookie` годятся только для текущей сессии.

- `getName/setName`

  Устанавливает/читает имя `Cookie`.

- `getPath/setPath`

  Устанавливает/получает путь, к которому относится `Cookie`. Если путь не указан, то `cookie` возвращается ко всем URL, расположенным в том же директории, что и текущая страница. Этот метод может быть использован для идентификации с применением `Cookie` самых общих свойств. В частности, метод `someCookie.setPath("/")` устанавливает, что `Cookie` должны возвращаться для всех страниц сервера. Путь должен содержать текущий каталог.

- `getSecure/setSecure`

  Устанавливает и читает значение `Boolean`. В соответствии с ним `Cookie` могут быть посланы только по зашифрованному каналу SSL или нет.

- `getValue/setValue`

  Устанавливает/получает значение для `Cookie`.

- `getVersion/setVersion`

  Устанавливает/получает версию протокола для `Cookie`.

Вставка Cookie в заголовок ответа

Вставка Cookie в заголовок HTTP-ответа происходит с помощью метода `addCookie` (интерфейс `HttpServletResponse`). Например:

```
Cookie userCookie = new Cookie("user", "uid1234");
response.addCookie(userCookie);
```

Чтение Cookie, полученного от клиента

Чтобы направить клиенту Cookie, необходимо создать `Cookie` при помощи метода `addCookie` и послать его в заголовке методом `response.addCookie()`. Для чтения Cookie используется метод `getCookies` интерфейса `HttpServletRequest`. Метод возвращает массив объектов типа `Cookie`. К нему можно применить метод `getName`. По совпадающему имени можно получить значение с использованием метода `getValue`.

Утилиты для работы с Cookies

Файл ServletUtilities.java содержит описание утилит, упрощающих работу с Cookies (листинг 4.17).

Листинг 4.17. Фрагмент файла ServletUtilities.java

```
public static String getCookieValue(Cookie[] cookies,
                                     String cookieName,
                                     String defaultValue) {
  for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookieName.equals(cookie.getName()))
      return(cookie.getValue());
  }
  return(defaultValue);
}
```

Файл LongLivedCookie.java (листинг 4.18) содержит класс, предназначенный для работы с `Cookies`. Срок жизни для `Cookie` устанавливается равным одному году.

Листинг 4.18. Файл LongLivedCookie.java

```
package hall;
import javax.servlet.http.*;
public class LongLivedCookie extends Cookie {
```

```
  public static final int SECONDS_PER_YEAR = 60*60*24*365;
   public LongLivedCookie(String name, String value) {
    super(name, value);
    setMaxAge(SECONDS_PER_YEAR);
  }
}
```

В качестве примера рассмотрим файл SearchEnginesFrontEnd.java (листинг 4.19). В коде использован метод `getCookieValue`. Метод `headWithTitle` создает фрагмент генерируемого HTML-кода. В примере используется класс `LongLivedCookie` (листинг 4.18).

Листинг 4.19. Файл SearchEnginesFrontEnd.java.

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
public class SearchEnginesFrontEnd extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    Cookie[] cookies = request.getCookies();
    String searchString =
      ServletUtilities.getCookieValue(cookies,
                                      "searchString",
                                      "Java Programming");
    String numResults =
      ServletUtilities.getCookieValue(cookies,
                                      "numResults",
                                      "10");
    String searchEngine =
      ServletUtilities.getCookieValue(cookies,
                                      "searchEngine",
                                      "google");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Searching the Web";
    out.println(ServletUtilities.headWithTitle(title) +
```

```
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">Searching the Web</H1>\n" +
                "\n" +
                "<FORM ACTION=\"http://localhost:81/dosearch">\n" +
                "<CENTER>\n" +
                "Search String:\n" +
                "<INPUT TYPE=\"TEXT\" NAME=\"searchString\"\n" +
                "       VALUE=\"" + searchString + "\"><BR>\n" +
                "Results to Show Per Page:\n" +
                "<INPUT TYPE=\"TEXT\" NAME=\"numResults\"\n" +
                "       VALUE=" + numResults + " SIZE=3><BR>\n" +
                "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                "       VALUE=\"google\"" +
                checked("google", searchEngine) + ">\n" +
                "Google |\n" +
                "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                "       VALUE=\"infoseek\"" +
                checked("infoseek", searchEngine) + ">\n" +
                "Infoseek |\n" +
                "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                "       VALUE=\"lycos\"" +
                checked("lycos", searchEngine) + ">\n" +
                "Lycos |\n" +
                "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                "       VALUE=\"hotbot\"" +
                checked("hotbot", searchEngine) + ">\n" +
                "HotBot\n" +
                "<BR>\n" +
                "<INPUT TYPE=\"SUBMIT\" VALUE=\"Search\">\n" +
                "</CENTER>\n" +
                "</FORM>\n" +
                "\n" +
                "</BODY>\n" +
                "</HTML>\n");
  }
  private String checked(String name1, String name2) {
    if (name1.equals(name2))
      return(" CHECKED");
    else
```

```
        return("");
    }
}
```

В этом случае фронт-энд (внешний вид) (рис. 4.17) формируется с помощью файла SearchEnginesFrontEnd.java. Этот сервлет (будем вызывать его по адресу **http://localhost:81/frontend**) посылает собранную информацию другому сервлету (рис. 4.18), код которого расположен в файле CustomizedSearchEngines.java (листинг 4.20), а адрес которого **http://localhost:81/dosearch**.

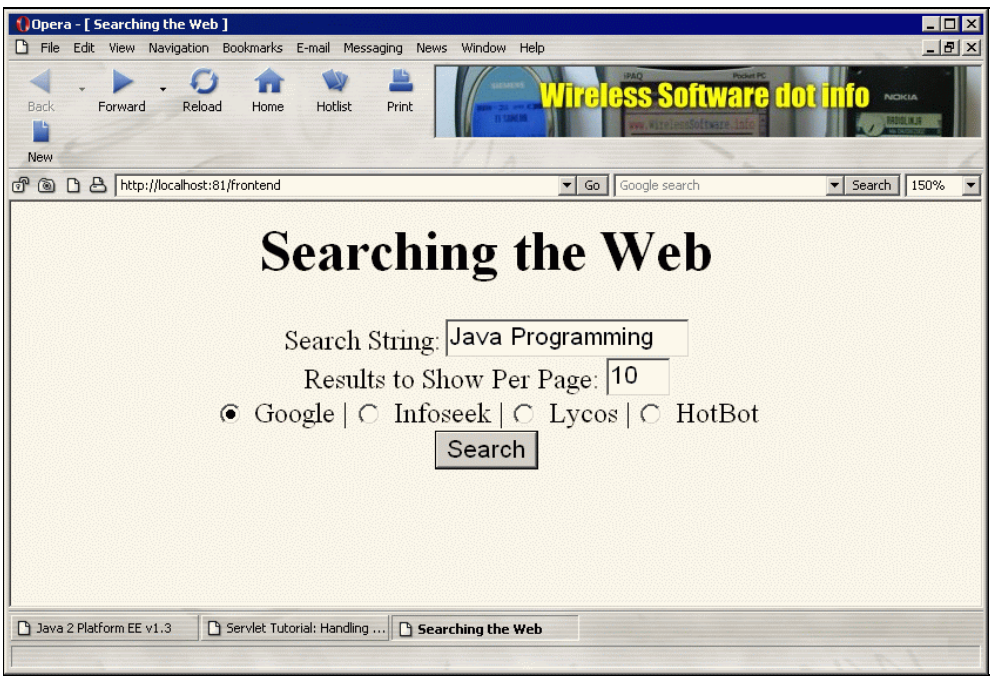


Рис. 4.17. Фронт-энд поискового сервлета

Листинг 4.20. Файл CustomizedSearchEngines.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
public class CustomizedSearchEngines extends HttpServlet {
```

```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    String searchString = request.getParameter("searchString");
    Cookie searchStringCookie =
      new LongLivedCookie("searchString", searchString);
    response.addCookie(searchStringCookie);
    searchString = URLEncoder.encode(searchString);
    String numResults = request.getParameter("numResults");
    Cookie numResultsCookie =
      new LongLivedCookie("numResults", numResults);
    response.addCookie(numResultsCookie);
    String searchEngine = request.getParameter("searchEngine");
    Cookie searchEngineCookie =
      new LongLivedCookie("searchEngine", searchEngine);
    response.addCookie(searchEngineCookie);
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    for(int i=0; i<commonSpecs.length; i++) {
      SearchSpec searchSpec = commonSpecs[i];
      if (searchSpec.getName().equals(searchEngine)) {
        String url =
          searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
      }
    }
    response.sendError(response.SC_NOT_FOUND,
                       "No recognized search engine specified.");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

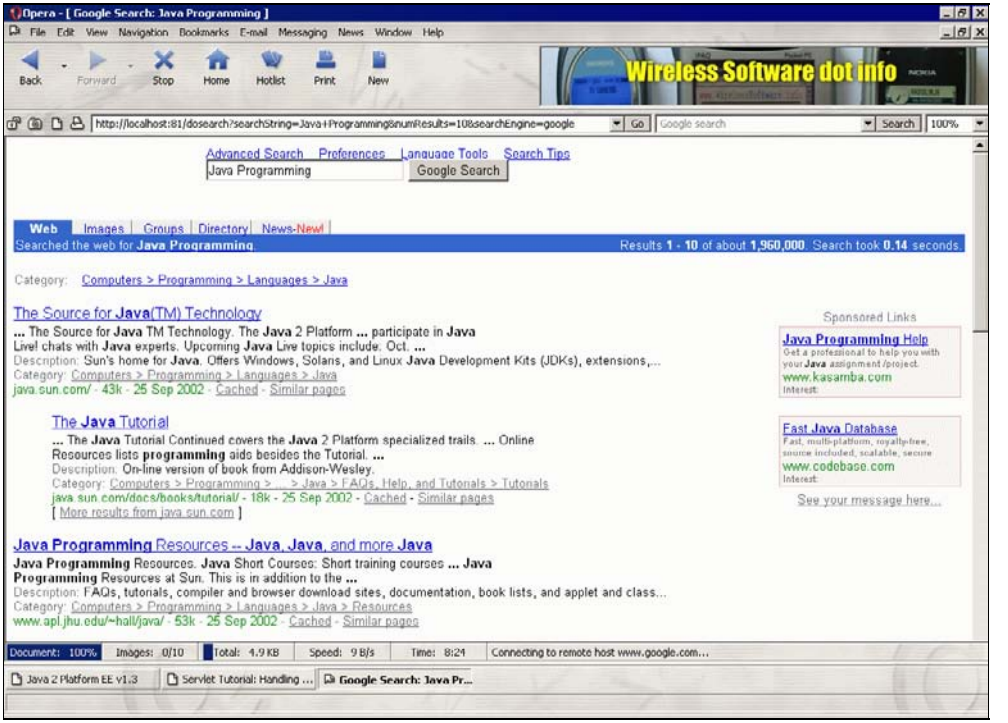


Рис. 4.18. Запрос выполнен

4.3.8. Поддержание сессий

Протокол HTTP не поддерживает сессии автоматически. Для поддержки сессий могут использоваться другие механизмы. Существует, по крайней мере, три стандартных подхода.

- Cookies.

  Для хранения информации, которая служит для идентификации сессии и сбора данных при работе с сессиями, могут быть использованы `Cookie`.

- Запись информации сессии в составе URL.

  Информация о сессии и данные сессии могут быть переданы в составе `URL`. Однако если пользователь покидает сайт, то при повторном его возвращении информация о сессии будет потеряна.

- Скрытые элементы `<input>` в форме.

  Информация о сессии может быть передана посредством скрытых форм в HTML-ярлыке `<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">`.

Сервлеты обладают набором средств для работы с сессиями — `HttpSession`. Этот интерфейс основан на технике использования Cookies и организации сессий с перезаписью URL. Второй вариант используется тогда, когда клиент не поддерживает Cookies или если они отключены. При использовании `HttpSession` нет необходимости касаться всех деталей работы этого механизма. В этом удобство такого интерфейса. При этом работа с сессиями производится довольно прямолинейно и очевидно. Чтобы найти объект `HttpSession`, который работает с данным запросом, вызывается метод `getSession` интерфейса `HttpServletRequest`. Если метод возвращает `null`, то мы имеем возможность создать новую сессию. Сессия может быть создана автоматически. Для этого необходимо в качестве аргумента метода `getSession` указать значение `true`:

```
HttpSession session = request.getSession(true);
```

Объекты типа `HttpSession` располагаются на сервере. Они связаны с конкретными запросами. Объекты эти имеют свою внутреннюю структуру, которая определяется произвольным набором ключей. Значение ключа может быть получено при помощи метода `getValue("key")`. Метод возвращает объект типа `Object`. Далее этот объект должен быть приведен к более конкретному типу, который соответствует запрошенному ключу. Если атрибут не задан, то возвращаемое значение будет `null`. Это устаревший метод, новый метод — `getAttribute`. Соответствующий ему метод `setAttribute` позволяет осуществлять контроль за значениями. Этот метод определен в `HttpSessionBindingListener`. Ниже приводится небольшой пример с корзиной для покупок.

```
HttpSession session = request.getSession(true);
ShoppingCart previousItems =
  (ShoppingCart)session.getValue("previousItems");
if (previousItems != null) {
  doSomethingWith(previousItems);
} else {
  previousItems = new ShoppingCart(...);
  doSomethingElseWith(previousItems);
}
```

Список всех атрибутов можно получить с помощью метода `getValueNames`, который возвращает массив строк `String`. Более новый вариант этого метода — метод `getAttributeNames`.

Существует стандартный набор данных, используемый в сессиях. Перечислим некоторые ключевые методы.

- `getId`

  Метод возвращает уникальный идентификатор сессии. Этот идентификатор может быть использован в качестве ключа, когда сессия содержит

единственное ключевое значение, которое идентифицируется этим идентификатором как ключом.

- `isNew`

  Метод возвращает `true`, если клиент не работал с сессией (сессия только что создана). В противном случае возвращается `false`.

- `getCreationTime`

  Возвращает время создания сессии в миллисекундах. Чтобы получить удобно читаемое выражение, полученное время передается конструктору класса `Date` или методу `setTimeInMillis` из `GregorianCalendar`.

- `getLastAccessedTime`

  Возвращает время последней отправки сессии клиентом в миллисекундах.

- `getMaxInactiveInterval`

  Задает интервал в секундах. По истечении этого времени сессия будет прекращена. Отрицательное значение соответствует отсутствию заданного времени уничтожения сессии.

Информация, связанная с сессией, может быть получена с использованием методов `getValue` или `getAttribute`. Для задания информации и вставки ее в сессию, используются методы `putValue` или `setAttribute`. Например,

```
HttpSession session = request.getSession(true);
session.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart previousItems =
  (ShoppingCart)session.getValue("previousItems");
if (previousItems == null) {
  previousItems = new ShoppingCart(...);
}
String itemID = request.getParameter("itemID");
previousItems.addEntry(Catalog.getEntry(itemID));
session.putValue("previousItems", previousItems);
```

Далее вашему вниманию предлагается пример страницы, которая выводит информацию о текущей сессии (листинг 4.21). Результат выполнения этого кода можно увидеть на рис. 4.19.

Листинг 4.21. Файл ShowSession.java

```
package paket;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.net.*;
import java.util.*;
public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Searching the Web";
    String heading;
    Integer accessCount = new Integer(0);;
    if (session.isNew()) {
      heading = "Welcome, Newcomer";
    } else {
      heading = "Welcome Back";
      Integer oldAccessCount =
        (Integer)session.getValue("accessCount");
      if (oldAccessCount != null) {
        accessCount =
          new Integer(oldAccessCount.intValue() + 1);
      }
    }
    session.putValue("accessCount", accessCount);
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
                "<H2>Information on Your Session:</H2>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "  <TH>Info Type<TH>Value\n" +
                "<TR>\n" +
                "  <TD>ID\n" +
                "  <TD>" + session.getId() + "\n" +
                "<TR>\n" +
                "  <TD>Creation Time\n" +
                "  <TD>" + new Date(session.getCreationTime()) + "\n" +
                "<TR>\n" +
                "  <TD>Time of Last Access\n" +
                "  <TD>" + new Date(session.getLastAccessedTime()) + "\n" +
```

```
                "<TR>\n" +
                "  <TD>Number of Previous Accesses\n" +
                "  <TD>" + accessCount + "\n" +
                "</TABLE>\n" +
                "</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Рис. 4.19. Вывод информации о текущей сессии

4.3.9. Еще раз о JSP

Серверные страницы Java представляют собой разновидность технологии сервлетов. При помощи JSP мы имеем возможность создавать такие страницы, в которых динамический компонент отделен от статического

HTML-кода. Выполняемый код вставляется в страницу между знаками <% и %>, например

```
Спасибо за заказ
<I><%= request.getParameter("title") %></I>
```

Как правило, файл, содержащий JSP-вставки, имеет расширение .jsp и располагается в основном каталоге Web-сервера, то есть там, куда помещаются обычные нединамические (не исполняемые на сервере) страницы. JSP-страница преобразуется в сервлет. Статический HTML выводится неизменным в поток вывода. На сервере Blazix временный файл Java для промежуточного сервлета будет сохранен на время компиляции в каталоге Blazix\jspdir. После компиляции временный файл будет автоматически удален. При желании и необходимости в промежуточный файл .java можно посмотреть. Пусть, например, исходный JSP-файл будет test1.jsp:

```
<HTML>
<head>
<title>Simple test.</title>
</head>
<body>
<h1>
Just simplest jsp test
</h1>
Current time is
<font size=+2>
<%= new java.util.Date() %>
</font>

<HTML> <BODY>
<%
System.out.println("Evaluating date now");
java.util.Date date = new java.util.Date();
%>
Hello! The time is now <%= date %>
</BODY>
 </HTML>
</BODY>
</HTML>
```

Соответствующий ему промежуточный Java-файл, описывающий сервлет, будет выглядеть следующим образом (листинг 4.22).

Листинг 4.22. Файл desisoft_jsp_test1_jsp1030370508327

```
// JSP Page automatically generated by Desiderata Software JSP generator
// Mon Aug 26 18:01:48 MSD 2002
import javax.servlet.*;
import java.io.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;


public class desisoft_jsp_test1_jsp1030370508327 extends
desisoft.server.JspBaseClass {


       int _ds_jsp_lnum = -1;
       public void _jspService(HttpServletRequest request,
                               HttpServletResponse response)
                        throws ServletException, IOException
         { JspFactory factory = JspFactory.getDefaultFactory();
           PageContext pageContext = factory.getPageContext(this,
                                          request, response,
                                          null, true,
                                          JspWriter.DEFAULT_BUFFER,
                                          true
                                         );

              HttpSession session = pageContext.getSession();
              JspWriter out = pageContext.getOut();
              ServletConfig config = getServletConfig();
              ServletContext application = config.getServletContext();
              Object page = this;
              try {
                     _ds_jsp_lnum = 1;
                     out.println("<HTML>");
                     _ds_jsp_lnum = 2;
                     out.println("<head>");
                     _ds_jsp_lnum = 3;
                     out.println("<title>Simple test.</title>");
                     _ds_jsp_lnum = 4;
                     out.println("</head>");
                     _ds_jsp_lnum = 5;
```

```
                    out.println("<body>");
                    _ds_jsp_lnum = 6;
                    out.println("<h1>");
                    _ds_jsp_lnum = 7;
                    out.println("Just simplest jsp test");
                    _ds_jsp_lnum = 8;
                    out.println("</h1>");
                    _ds_jsp_lnum = 9;
                    out.println("Current time is");
                    _ds_jsp_lnum = 10;
                    out.println("<font size=+2>");
                    _ds_jsp_lnum = 11;
                                    out.print( new java.util.Date() );
                                    out.println(" ");
                    _ds_jsp_lnum = 12;
                    out.println("</font>");
                    _ds_jsp_lnum = 13;
                    out.println();
                    _ds_jsp_lnum = 14;
                    out.println("<HTML> <BODY> ");
                    _ds_jsp_lnum = 15;
//***** Start Scriptlet
System.out.println("Evaluating date now");
java.util.Date date = new java.util.Date();

                    //***** End Scriptlet
                    _ds_jsp_lnum = 21;
                    out.println(" ");
                    _ds_jsp_lnum = 22;
                    out.print("Hello! The time is now ");
                                        out.print( date );
                                        out.println(" ");
                    _ds_jsp_lnum = 23;
                    out.println("</BODY>");
                    _ds_jsp_lnum = 24;
                    out.println(" </HTML>");
                    _ds_jsp_lnum = 25;
                    out.println("</BODY>");
```

```
                    _ds_jsp_lnum = 26;
                    out.println("</HTML>");
                    _ds_jsp_lnum = 27;
              } catch (Exception ex) {
                    out.clearBuffer();
                    request.setAttribute(
                                "javax.servlet.jsp.jspException", ex);
                    request.setAttribute(
                "desisoft.jsp.lineNumber", new Integer(_ds_jsp_lnum));
                    pageContext.handlePageException(ex);
              } finally {
                    factory.releasePageContext(pageContext);
              }
        }
        public String[] _desisoft_jsp_getDependentFiles()
          { return _desisoft_jsp_dependencyList; }
        public long[] _desisoft_jsp_getModTimes()
          { return _desisoft_jsp_dependencyModTimes; }

    static String[] _desisoft_jsp_dependencyList = { "test1.jsp" };
    static long [] _desisoft_jsp_dependencyModTimes = { 1028784400000L };
}
```

Имя файла генерируется автоматически. Код также генерируется сервером автоматически. Затем он автоматически компилируется в соответствующий класс.

Основные элементы синтаксиса JSP

- Выражение JSP.

  Синтаксис: `<%= expression %>`

  Выражение вычисляется и помещается в поток вывода. XML-эквивалент следующий:

  ```
  <jsp:expression>
  expression
  </jsp:expression>.
  ```

  Определены переменные `request`, `response`, `out`, `session`, `application`, `config`, `pageContext`.

- Скриплет JSP.

  Синтаксис: `<% code %>`

  `Code` вставляется в метод `service`. XML-эквивалент следующий:

  ```
  <jsp:scriptlet>
  code
  </jsp:scriptlet>.
  ```

- Декларация JSP.

  Синтаксис: `<%! code %>`

  `Code` вставляется в класс сервлета вне метода `service`. XML-эквивалент следующий:

  ```
  <jsp:declaration>
  code
  </jsp:declaration>.
  ```

- Директива JSP.

  Синтаксис: `<%@ page att="val" %>`

  Директива установки общих свойств. XML-эквивалент следующий:

  ```
  <jsp:directive.page att="val"\>.
  ```

  Используются атрибуты:

  - `import="package.class"`
  - `contentType="MIME-Type"`
  - `isThreadSafe="true|false"`
  - `session="true|false"`
  - `buffer="sizekb|none"`
  - `autoflush="true|false"`
  - `extends="package.class"`
  - `info="message"`
  - `errorPage="url"`
  - `isErrorPage="true|false"`
  - `language="java"`

- Директива вставки файла.

  Синтаксис:

  ```
  <%@ include file="url" %>
  ```

  Вставляется файл, расположенный в локальной файловой системе. Вставка происходит в процессе преобразования в сервлет. XML-эквивалент следующий:

  ```
  <jsp:directive.include  file="url"\>
  ```

Указывается относительный URL. Для вставки файла во время обращения (а не во время трансляции) можно использовать `jsp:include`.

- Комментарий.

  Синтаксис:

  ```
  <%-- comment --%>
  ```

  При трансляции в сервлет комментарии игнорируются. Для вставки HTML-комментариев используются знаки `<- — комментарий -->`.

- Действие `jsp:include`.

  Синтаксис:

  ```
  <jsp:include
  page="relative URL"
  flush="true"/>
  ```

  Вставка файла во время запроса страницы.

- Действие `jsp:useBean`.

  Синтаксис:

  ```
  <jsp:useBean att=val*/>
  ```

  или

  ```
  <jsp:useBean att=val*>
  ...
  </jsp:useBean>
  ```

  Найти или создать компонент.

  Допустимые атрибуты:

  - `id="name"`
  - `scope="page|request|session|application"`
  - `class="package.class"`
  - `type="package.class"`
  - `beanName="package.class"`

- Действие `jsp:setProperty`.

  Синтаксис:

  ```
  <jsp:setProperty att=val*/>
  ```

  Задает свойства компонентов в явном виде или в виде параметров.

  Допустимые атрибуты:

  - `name="beanName"`
  - `property="propertyName|*"`
  - `param="parameterName"`
  - `value="val"`

- Действие `jsp:getProperty`.

  Синтаксис:

  ```
  <jsp:getProperty
  name="propertyName"
  value="val"/>
  ```

  Получает и выводит свойства компонента.

- Действие `jsp:forward`.

  Синтаксис:

  ```
  <jsp:forward    page="relative URL"/>
  ```

  Переадресует запрос на другую страницу.

- Действие `jsp:plugin`.

  Синтаксис:

  ```
  <jsp:plugin   attribute="value"*>
    ...
  </jsp:plugin>
  ```

  Создает ярлык `OBJECT` или `EMBED` в соответствии с тем, что требуется браузеру, запрашивая возможность использовать клиентский плагин Java.

Некоторые аспекты применения JSP

Статический текст на JSP-странице выглядит как обычный HTML-код. Он составляет неизменяемую основу, шаблон. Статический текст отделяется от JSP-вставок комбинацией символов `<%`. Если такая же комбинация должна быть выведена в составе статического HTML, то ее следует записать так: "`<\%`". Элементы перечислены выше.

Выражения JSP используются для вставки значений в поток вывода (`<%= Java Expression %>`). Выражение вычисляется, преобразуется в строку и вставляется в поток вывода в скриплете. Для того чтобы облегчить работу с выражениями, существует набор заранее определенных переменных, часть из которых приводится ниже:

- `request` (из `HttpServletRequest`);
- `response` (из `HttpServletResponse`);
- `session` (из `HttpSession`, связанной с запросом);
- `out` — поток `PrintWriter` для вывода клиенту.

Например:

```
Your hostname: <%= request.getRemoteHost() %>
```

Если кто-либо предпочитает XML, то JSP-выражения могут быть записаны в альтернативном виде:

```
<jsp:expression>
Java Expression
</jsp:expression>
```

Важно помнить, что XML чувствителен к регистру.

Технология JSP в подробностях освещена в *главе 3*. Приведем пример простой JSP-страницы (листинг 4.23).

Листинг 4.23. Файл jsptest4.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using JavaServer Pages</TITLE>

<META NAME="keywords"
      CONTENT="JSP,JavaServer Pages,servlets">
<META NAME="description"
      CONTENT="A quick example of the four main JSP tags.">
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
      VLINK="#551A8B" ALINK="#FF0000">

<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
  <TR><TH CLASS="TITLE">
      Using JavaServer Pages</TABLE>
</CENTER>
<P>

Some dynamic content created using various JSP mechanisms:
<UL>
  <LI><B>Expression.</B><BR>
```

```
      Your hostname: <%= request.getRemoteHost() %>.
  <LI><B>Scriptlet.</B><BR>
      <% out.println("Attached GET data: " +
                     request.getQueryString()); %>
  <LI><B>Declaration (plus expression).</B><BR>
      <%! private int accessCount = 0; %>
      Accesses to page since server reboot: <%= ++accessCount %>
  <LI><B>Directive (plus expression).</B><BR>
      <%@ page import = "java.util.*" %>
      Current date: <%= new Date() %>
</UL>
</BODY>
</HTML>
```

Глава 5

Серверные компоненты EJB

В идеале гармоническое колебание — это нечто неизменно повторяющееся на протяжении всего времени от самого удаленного прошлого до самого удаленного будущего. В некотором смысле оно существует sub specie aeternitatis. Начало и окончание звука неизбежно связаны с изменением его частотного состава, которое может быть и невелико, но всегда реально. Звук, длящийся лишь ограниченное время, разлагается на целую полосу простых гармонических колебаний, и ни одно из этих колебаний не может рассматриваться как единственно существующее. Уточнение положения звука на шкале времени связано с увеличением неточности в значении его частоты, и, наоборот, более точное указание частоты влечет за собой большую неопределенность во времени. Эти соображения имеют отнюдь не только чисто теоретическую ценность — они реально ограничивают возможности музыканта. Никто не сможет сыграть жигу на нижнем регистре органа.

Н. Винер. Я — математик

5.1. Серверные компоненты EJB и среда J2EE

Серверные компоненты EJB Java — технология, реализованная в пакете J2EE (Java 2 Enterprise Edition — Java 2 для предприятий) от компании Sun. Серверные компоненты EJB предназначены для создания распределенных приложений. Их использование позволяет создавать надежные распределенные приложения, а также реализовывать механизмы быстрой обработки запросов, разрабатывать системы защиты пользователей от несанкционированного

доступа к данным и пользовательской информации, передаваемой во время запросов, создавать постоянно растущие и усложняющиеся приложения по мере роста нужд заказчика, для которого разрабатываются приложения.

Разработка приложений на платформе J2EE подразумевает использование встроенных серверов. Существует несколько вариантов серверов от различных разработчиков, в том числе сервера от компаний Sun, IBM, Borland.

Технология разработки клиент-серверных приложений *(см. главу 2)* стремительно развивалась еще в начале 90-х. Многие информационные системы того времени были основаны на архитектуре клиент-сервер. Программа, реализующая пользовательский интерфейс, как правило, устанавливалась на обычном персональном компьютере и осуществляла функции клиента, представляя собой пользовательский уровень. Данные поставщика услуг и серверные данные были доступны клиенту при обращении к серверу. Сервер хранил данные в виде базы данных (рис. 5.1).

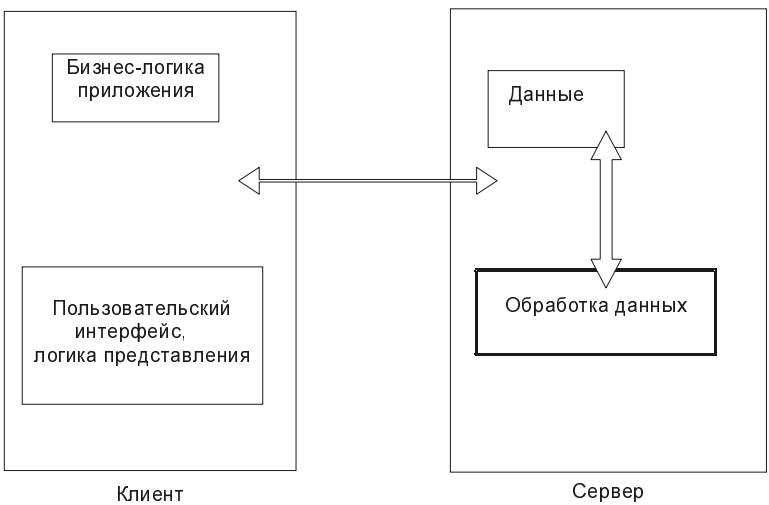


Рис. 5.1. Архитектура клиент-сервер

По мере накопления опыта стало понятно, что разработка гибких приложений, позволяющих учитывать запросы как крупных, так и небольших заказчиков, становится достаточно трудной задачей, если использовать модель простого взаимодействия между клиентом и сервером. Например, логика приложения осуществляется клиентом так, как это показано на рисунке, а сервер содержит только базу данных. В случае, когда логика приложения должна претерпевать изменения, клиентские приложения должны быть вновь установлены на все компьютеры предприятия заново. Обслуживание приложений, основанных на подобной архитектуре, превращается в изнурительно однообразный и неэффективный род деятельности. При этом следует

учитывать тот факт, что помимо требований эффективности работы самого приложения и требований к безопасности проводимых операций, необходимо обеспечить простой и понятный пользовательский интерфейс. Попытайтесь ответить на вопрос, много ли найдется людей, обладающих достаточным талантом во всех этих областях одновременно? Современные требования предприятий часто выходят за рамки того, что может предложить классическая архитектура клиент-сервер.

Поскольку недостатки клиент-серверной архитектуры довольно скоро стали очевидны, появилась необходимость создания новой, более совершенной модели. Появилась модель многоуровневой архитектуры (рис. 5.2).

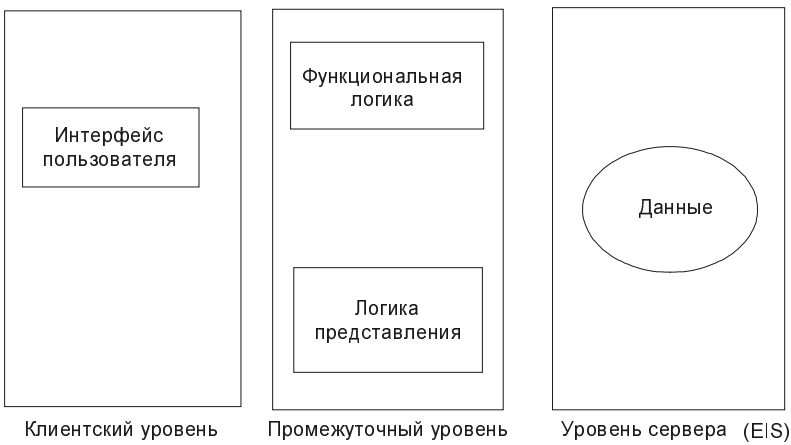


Рис. 5.2. Многоуровневая архитектура

В модели многоуровневой архитектуры логика приложения отделена от клиентской части, она выделяется в самостоятельный промежуточный ярус. Логика приложения включает в себя как бизнес-логику, т. е. функциональную логику приложения, так и логику представления пользовательского интерфейса. При возникновении необходимости внесения изменений в логику приложения, требуемые улучшения могут быть внесены быстро и эффективно, при этом нет необходимости производить повторную установку всех клиентов. Более подробно многоуровневая архитектура показана на рис. 5.3.

Клиентом в приложении J2EE может быть HTML-страница, содержащая апплет, или просто страница без апплета. Клиентом также может быть Java-приложение. Это приложение может быть мультиплатформенным и может быть запущено с любого устройства, в том числе с карманного компьютера или с сотового телефона.

В качестве приложений среднего звена могут использоваться, например, серверные страницы Java, сервлеты, которые работают на том или ином Web-сервере и осуществляют бизнес-логику и реализуют пользовательский

интерфейс. Серверные компоненты EJB выполняются в контейнере EJB. Компоненты EJB осуществляют основные функции бизнес-логики приложения. Контейнер предоставляет компонентам EJB набор сервисов, эти сервисы всегда доступны, разработчик программ не должен вновь и вновь заботиться о них, они уже готовы для использования.

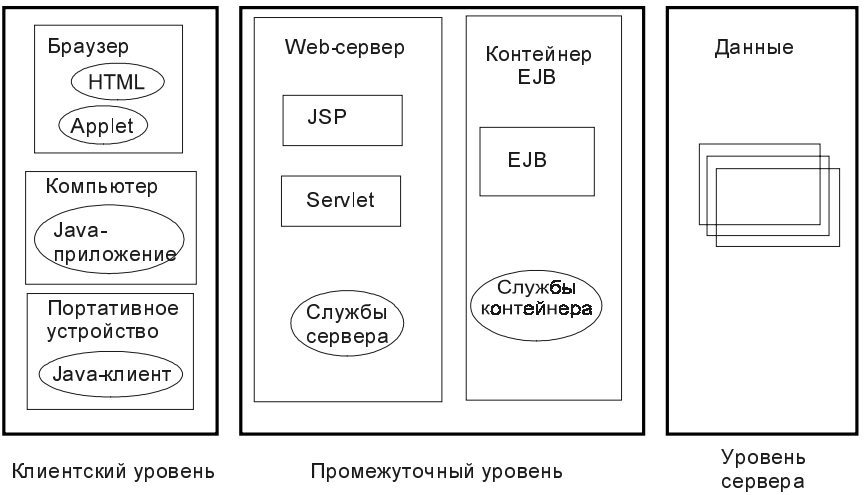


Рис. 5.3. Детализированная модель многоуровневой архитектуры

Серверный уровень, соответствующий информационному серверу предприятия (Enterprise Information Server, EIS), — это уровень, на котором расположены данные. Как правило, он содержит базы данных. Не все разрабатываемые приложения будут включать в себя все перечисленные компоненты. Приложение создается таким, чтобы максимально удовлетворить нужды предприятия наиболее эффективным образом.

Каковы преимущества распределенных многоуровневых приложений? Разработчики J2EE отмечают следующие достоинства распределенной модели. Распределенные приложения позволяют уменьшить уровень сложности создаваемых приложений за счет упрощения архитектуры приложения и распределения ролей между разработчиками. К примеру, бизнес-логика приложения может быть осуществлена в контейнере EJB и на Web-сервере. Общий набор функций может быть достаточно большим и таким образом задача становится сложной. Но EJB-контейнер будет обрабатывать транзакции, создавать и хранить экземпляры объектов автоматически, без участия программиста. В то же время Web-сервер может работать с сервлетами, потоками, сокетами. В этом случае программист должен позаботиться не столько том, как написать код, но как наиболее эффективно осуществить связь между частями приложения, причем эта задача главным образом будет решаться во время размещения приложения на сервере.

Каждый разработчик обычно является специалистом в той или иной области и решает свою собственную задачу. Содержимое HTML-страницы, работа со стилями CSS (Cascading Style Sheets — каскадные листы стилей) — это область применения опыта и знаний Web-мастера или дизайнера. Старший разработчик может отвечать за создание и воплощение бизнес-логики приложения, включая разработку приложения на основе серверных компонентов, серверных компонентов EJB. Web-программист отвечает за разработку пользовательского интерфейса и логики представления приложения пользователю, при этом он может использовать серверные страницы Java, а также сервлеты. Координатор проекта может отвечать за сборку всех частей проекта в общее целое, при этом могут создаваться самостоятельные файловые единицы, например, архивы предприятия, EAR (Enterprise Archive). Администратор проекта ответственен за установку приложения. Разделение задач разработки приложения между специалистами позволяет использовать опыт людей наиболее эффективным способом.

Еще одним преимуществом создания приложений с использованием J2EE является то, что такие приложения оказываются легко изменяемыми, можно без существенных затрат изменить приложение. Когда система разрастается, средний уровень приложения, осуществляющий бизнес-логику и логику представления, может быть переделан или создан заново, при этом все же нет необходимости устанавливать приложение на каждую машину заново.

Преимуществом также является то, что создаваемое приложение может использовать уже существующие информационные ресурсы и системы. Java поддерживает работу с базами данных посредством JDBC. Этот механизм представляет собой набор средств для работы с базами данных на основе языка SQL. Интерфейс имен JNDI (Java Naming and Directory Interface) может использовать сервис имен. Сервис сообщений JMS (Java Messaging Service) представляет собой интерфейс для отправки и получения сообщений. Сервисы, предоставляемые технологией CORBA, доступны посредством использования технологии JavaIDL (Java Interface Definition Language — язык описания интерфейса для Java).

J2EE предоставляет улучшенные возможности защиты информации от несанкционированного доступа. Кроме того, разработчик имеет возможность сделать выбор из широкого ассортимента средств, предоставляемых компанией Sun. Одним из наиболее популярных средств разработки Java-приложений является JBuilder от компании Borland.

Среда разработки JBuilder Enterprise Edition включает в себя большой набор средств разработки приложений J2EE. Наиболее популярные из них мы рассматриваем в этой книге. К клиентской части относятся апплеты. Апплеты — это Java-приложения, которые загружаются на компьютер клиента в браузер и выполняются на компьютере клиента. Среда JBuilder имеет ряд возможностей, облегчающих создание пользовательских интерфейсов Java

(находящихся в меню **Application Wizard | UI designer**). При создании клиентских приложений полезным бывает использование готовых или созданных самостоятельно компонентов EJB Java (не путать с серверными компонентами EJB). Компоненты EJB помогут облегчить процесс создания приложения (апплета).

Уровень бизнес-логики и пользовательского интерфейса, включая Web-сервер и серверные компоненты EJB, может быть реализован с использованием контейнера сервлетов — набора средств, носящих название Tomcat. Сервер приложений, работающих с серверными компонентами EJB, — это сервер Borland Enterprise Server. Среда разработки позволяет создавать приложения не только для этого сервера, но и для ряда других Java-серверов, таких как Web Logic, WebSphere, iPlanet и других.

Сервер позволяет создавать приложения среднего уровня (уровня бизнес-логики и реализации пользовательского интерфейса) на основе популярных технологий, включая сервлеты и серверные страницы Java. Разработка сервлета в JBuilder осуществляется с использованием проводника создания сервлета Servlet Wizard, а серверных страниц — с помощью Java Server Page Wizard.

JBuilder содержит библиотеку компонентов, упрощающих создание сервлетов и JSP-страниц. Эта библиотека состоит из интернет-компонентов IB. Интернет-компоненты IB облегчают процесс создания приложений, работающих с данными, также они помогают организовать представление данных пользователю наиболее удобным образом.

Уровень бизнес-логики может включать в себя контейнер серверных компонентов EJB. JBuilder 7 оснащен набором средств работы с серверными компонентами EJB спецификации EJB 1 и EJB 2.0. Разработка приложений на основе серверных компонентов EJB упрощается, если использовать проводника серверных компонентов EJB (**File | New | EJB**). После создания серверного компонента EJB создается дескриптор размещения компонента EJB. Дескрипторы можно редактировать в редакторе дескрипторов.

Все Web-приложения и их компоненты, а также серверные компоненты EJB, могут быть объединены в один файл архива EAR. Для этого можно использовать проводник создания архива EAR Wizard.

J2EE позволяет работать с базами данных посредством драйверов JDBC (Java Database Connectivity). JDBC — это стандарт доступа к базам данных, используемых в Java для осуществления доступа к данным на сервере Enterprise Information Systems (EIS). На основе JDBC создаются SQL-запросы к базам данных. В среде JBuilder содержится библиотека `DataExpress`, облегчающая работу с базами даных. В JBuilder содержится также база данных `JDataStore`. Сервис сообщений JMS служит для передачи сообщений от одного компонента и процесса распределенного приложения к другому. Существуют различные реализации сервиса сообщений. JBuilder

использует средства SonicMQ 3.5. Кроме того, существует возможность создания серверных компонентов EJB третьего типа, компонентов EJB, основанных на сообщениях (компонентов EJB типа message-driven). Этот тип серверных компонентов EJB использует сервис сообщений, встраивая его внутрь компонента EJB.

Для нахождения объектов в распределенных приложениях используется интерфейс имен JNDI. В J2EE также широко применяется язык XML. Так, Web компоненты и серверные компоненты EJB имеют дескрипторы размещения, записанные в файле XML и содержащие информацию о расположении и поведении компонентов.

5.1.1. Метод разработки EJB

Создатели пакетов для работы с J2EE предлагают схемы работы, упрощающие процесс создания и поддержания работоспособности приложений. Спецификация серверных компонентов EJB определяет модель серверных компонентов EJB и программный интерфейс Java-серверов. Разработчик создает компоненты EJB, которые содержат функциональную логику приложения, реализуемую предприятием, для которого создается приложение. Серверный компонент EJB работает в контейнере серверных компонентов EJB, контейнер предоставляет набор сервисов, например, таких, как обработка транзакций, обеспечение безопасности компонентов EJB. Разработчик не будет касаться деталей устройства этих сервисов, требующих решения задач низкоуровневого программирования. Разработчику необходимо решить задачи организации предоставляемых контейнером средств внутри компонента EJB, компоновки системы компонента EJB в целом, принимая во внимание то, что при необходимости использования того или иного сервиса контейнера, всегда можно воспользоваться этим сервисом. Популярная в свое время архитектура клиент-сервер может хорошо работать только в том случае, когда ею пользуется не очень большое количество клиентов. Управление такой системой достаточно неэффективно. Серверные компоненты EJB решают задачу эффективности как управления системой, так и эффективности использования системы. Упрощается разработка приложения за счет распределения задач между различными группами разработчиков, специализирующихся в той или иной области разработки приложений. Распределение ролей при разработке приложений может быть осуществлено, например, таким способом, как предлагают разработчики системы JBuilder, когда каждая часть работы выполняется узким специалистом в этой области. Вся задача делится в общем случае на шесть частей (в том числе две задачи, предназначенные для разработчиков приложения). Каждый разработчик должен в деталях представлять то, как будет работать приложение. Первый разработчик — это разработчик компонентов EJB. Он (один или группа разработчиков) разрабатывает код компонентов EJB, создает функциональную логику компонентов EJB, описывает методы, используемые

компонентами EJB, создает эти методы. Помимо этого разработчик описывает удаленный домашний интерфейс (или локальный домашний интерфейс), а также удаленный и локальный интерфейсы (при необходимости). При этом разработчик компонентов EJB может не знать о том, как будут использоваться компоненты EJB, как они будут размещены. Он создает компоненты EJB, интерфейсы, имплементирует (реализует) методы.

Компоновщик приложения создает приложение, которое использует уже готовые компоненты EJB. Эти приложения помимо компонентов EJB могут включать в себя элементы пользовательского интерфейса в виде интерфейсов пользовательского клиента, а также сервлеты и серверные страницы Java. Эти компоненты организуются в виде распределенного приложения. Компоновщик, или сборщик кода, добавляет инструкции о сборке готовых элементов в дескриптор (описатель) размещения компонентов EJB. Сборщик кода должен знать, какие методы имплементированы в серверных компонентах EJB для того, чтобы он мог использовать эти методы при создании приложения. Но при этом сборщик кода может не знать о том, как имплементированы эти методы.

Без поддержания необходимой инфраструктуры ни серверные компоненты EJB, ни приложение не будут работать. Для работы компонентов необходимо иметь в рабочем состоянии ряд служб. Предлагается разделить службы поддержки инфраструктуры на две части. Обе части, как правило, должны быть реализованы на основе систем, предоставленных одним и тем же разработчиком. В отношении серверных компонентов EJB предлагается разделить инфраструктуру на две части: провайдер сервера EJB и провайдер контейнера EJB.

Провайдер (менеджер) сервера EJB — это специалист (или специалисты) по работе с распределенными транзакциями, распределенными объектами, сервисами низкого уровня. Они ответственны за поддержание средств, обеспечивающих работу контейнеров серверных компонентов EJB. Провайдер сервера EJB должен отвечать за работу по крайней мере сервиса имен и сервиса транзакций, без которых невозможна работа серверных компонентов EJB.

Провайдер контейнера серверных компонентов EJB ответственен за работу и предоставление средств размещения серверных компонентов EJB, за поддержку работоспособности контейнера во время работы компонентов EJB. Контейнер предоставляет службы, которые могут работать с одним или несколькими серверными компонентами EJB.

Помимо разработчиков и ответственных за поддержание инфраструктуры, существуют части работы, связанные с размещением компонентов и администрированием системы. Завершающий этап разработки приложения — это размещение приложения и поддержание системы в работоспособном состоянии, включая работу приложения и инфраструктуру сети. Часть работы выполняется ответственным за размещение приложения: он размещает готовое

приложение в рабочем системном окружении в сети. При необходимости он изменяет те или иные параметры компонентов EJB, при этом используются средства, предоставляемые провайдером контейнера компонентов EJB. В частности, при размещении приложения могут быть установлены те или иные свойства безопасности обработки транзакций с использованием компонентов EJB, эти свойства задаются в описателе размещения компонентов EJB. Помимо этого может потребоваться состыковка приложения с существующим программным обеспечением системы. После того как приложение размещено, ответственность за работоспособность и функционирование системы в целом возлагается на администратора. Администратор должен предпринять меры по устранению неполадок в том случае, если приложение будет функционировать неправильно. Администратор приложения должен отвечать за конфигурирование и администрирование всей системы сети предприятия, включая сервер EJB и контейнер EJB.

5.1.2. Архитектура серверных компонентов EJB

Часто общая архитектура приложений построена таким образом, что клиентское приложение работает на той или иной локальной машине. Средний уровень приложений, которые осуществляют функциональную логику приложений, выполняется на сервере, который работает совместно с сервером информационной системы предприятия, реализованном в виде сервера EIS (Enterprise Information Server). Сервер предприятия может включать в себя базы данных, прочие приложения, включая ранее использовавшиеся приложения,

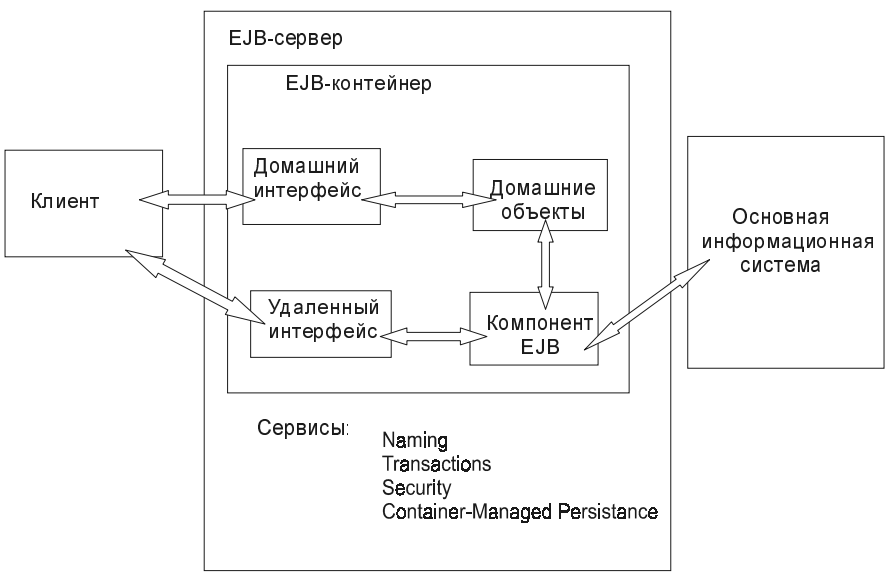


Рис. 5.4. Архитектура приложения, использующего серверные компоненты EJB

созданные еще для устаревших систем, но не исчерпавшие актуальности, и т. п. Общую структуру приложения с использованием серверных компонентов EJB можно представить различными способами, например так, как показано на рис. 5.4.

При изучении методов создания серверных компонентов EJB интерес концентрируется на среднем уровне, реализующем функциональную логику приложения.

Сервер компонентов EJB

Сервер компонентов EJB предоставляет системные службы для работы серверных компонентов EJB, управляет контейнерами серверных компонентов EJB, в которых работают серверные компоненты EJB. Сервер должен предоставлять доступ к службе имен JNDI и сервису транзакций. Помимо этого сервер EJB может предоставлять ряд дополнительных сервисов.

Контейнер компонентов EJB

Контейнер серверных компонентов EJB — это система, в которой работают серверные компоненты EJB. Посредством контейнера происходит взаимодействие компонентов EJB с сервером компонентов EJB. Контейнер осуществляет проведение транзакций, обеспечивает безопасность, управление распределенными системами. Контейнер также предоставляет средства разработки кода серверных компонентов EJB для определенных условий. Контейнер предоставляет средства размещения компонентов EJB, средства управления приложением. Контейнер и сервер обеспечивают среду, в которой работают компоненты EJB.

Функционирование серверных компонентов EJB

В задачи разработчика компонентов EJB, который занимается созданием программного кода, входит разработка следующих интерфейсов и классов. В первую очередь, это удаленный домашний и/или локальный домашний интерфейсы. Домашний интерфейс содержит методы, которые использует клиент для создания экземпляров серверных компонентов EJB, нахождения необходимых компонентов EJB, уничтожения ненужных экземпляров серверных компонентов EJB. Он также создает удаленный или локальный интерфейс для компонентов EJB. Эти интерфейсы содержат методы, обеспечивающие функциональную логику компонента EJB. Клиент имеет доступ к этим методам посредством удаленного интерфейса. Также должен быть создан класс компонента EJB. Класс компонента EJB имплементирует функции компонента EJB. Созданные в классе компонента EJB методы доступны для клиента посредством удаленного интерфейса компонента EJB.

После размещения компонента EJB в контейнере компонента EJB компонент EJB можно инициализировать вызовом из клиента метода `create()`,

этот метод задается в домашнем интерфейсе. В компоненте EJB не содержится имплементации домашнего интерфейса, но он имеет свое воплощение в контейнере компонента EJB. Кроме метода, инициализирующего компонент EJB, домашний интерфейс позволяет клиенту находить экземпляры компонентов EJB и удалять экземпляры компонентов EJB, если они уже более не нужны. Могут существовать также и другие методы, набор методов изменяется по мере развития данной технологии.

После того как компонент EJB инициализирован, клиент имеет доступ к методам самого компонента EJB. Доступ этот не осуществляется непосредственно, клиент никогда не имеет доступа к методам компонента EJB напрямую. Те методы, которые доступны клиенту, описываются в удаленном или локальном интерфейсе и реализуются в контейнере компонентов EJB. Если клиент посылает запрос, то этот запрос будет получен контейнером и передан экземпляру компонента EJB.

5.1.3. Типы серверных компонентов EJB

Существует три типа серверных компонентов EJB. Это компоненты EJB-сессий (session bean), компоненты EJB-сущностей (entity bean) и компоненты EJB на основе сообщений (message-driven bean). Рассмотрим характерные различия, которые отличают один тип компонентов EJB от другого.

Компоненты EJB-сессий (Session bean)

Этот тип компонентов EJB распадается на два подтипа. Серверные компоненты EJB-сессий могут быть либо компонентами EJB с заданным состоянием, либо компонентами EJB без состояния. Компоненты EJB без состояния не имеют возможности поддерживать информацию о состоянии того или иного конкретного клиента. Поскольку компоненты EJB без поддержки состояния взаимодействия с клиентом не имеют информации о текущем состоянии клиента, то такие компоненты EJB не могут поддерживать работу с несколькими клиентами одновременно.

Компоненты EJB-сессий с поддержкой состояний являются своего рода представителями клиента во взаимоотношениях с сервером компонентов EJB. Такой компонент EJB имеет информацию о текущем состоянии клиента. В качестве классического примера компонента EJB с поддержкой информации о состоянии клиента используется компонент EJB, который представляет собой покупательскую корзину для отдельного покупателя в онлайновом интернет-магазине. В этом примере компонент EJB поддерживает текущее состояние. После того как покупатель помещает в корзину очередной товар, компонент EJB изменяет список товаров, помещенных в корзину. Компонент EJB-сессий может быть организован так, что после того, как клиент завершает сессию, компонент EJB прекращает свое существование (не компонент EJB, как программный модуль, но компонент EJB

в смысле "экземпляр компонента EJB"). По завершении сессии клиент уничтожает экземпляр компонента EJB.

Компоненты EJB-сущности (Entity bean)

Компонент EJB-сущности часто представляет собой объект с данными, получаемыми из базы данных. Этот тип компонентов EJB можно назвать *насущными* компонентами EJB. Как правило, он содержит строку, соответствующую записи, хранимой в реляционной базе. Обычно компонент EJB-сущности может работать одновременно с несколькими клиентами. В отличие от компонентов EJB-сессий, компонент EJB-сущности, как правило, бывает долгоживущим, его жизненный цикл не ограничивается периодом активности клиентской сессии. Компонент EJB-сущности поддерживает постоянное соединение с базой данных и существует столько времени, сколько в базе данных существуют данные. Он потому и насущен, что его существование необходимо всегда. Насущный компонент EJB поддерживает свое долговременное бытие с помощью контейнера или своими собственными средствами. Если существование компонента EJB-сущности поддерживается средствами самого компонента EJB, то разработчик кода сам должен создать фрагменты кода обращения к базе данных.

Компоненты EJB на основе сообщений (Message-driven bean)

Компоненты EJB на основе сообщений используют сервисы службы JMS. Эти компоненты EJB ведут себя так, как прослушиватели сообщений JMS, которые производят обработку асинхронных сообщений. Компоненты EJB с сообщениями подобны компонентам EJB-сессий без поддержки состояний. Эти компоненты EJB не хранят информацию о клиенте. В отличие от компонентов EJB-сущностей и компонентов EJB-сессий, компоненты EJB-сообщений недоступны для клиентов. Они не имеют интерфейсов, в них присутствует только класс компонента EJB как такового. Один компонент EJB-сообщений может обрабатывать сообщения от нескольких клиентов. По своей сути компоненты EJB-сообщений представляют собой фрагменты программного кода, который выполняется в том случае, когда на определенный адрес JMS поступает сообщение.

5.1.4. Удаленный и локальный доступ

Компоненты EJB доступны как локально, так и удаленным образом. При обращении к удаленному компоненту EJB клиенты используют удаленный интерфейс компонента EJB и домашний удаленный интерфейс. Часто для простоты удаленный домашний интерфейс называют просто домашним интерфейсом. Клиент удаленного доступа может работать на удаленной машине, отличной от той, где расположен компонент EJB, и выполняться на

независимой виртуальной машине Java. При удаленных обращениях к методам компонента EJB параметры передаются своими значениями.

Клиент локального доступа должен выполняться на той же машине Java, на которой работает компонент EJB. Локальный клиент не может быть удаленным приложением, локальный клиент это, как правило, либо Web-компонент, либо другой локальный компонент EJB. При обращении к локальному компоненту EJB параметры передаются по ссылке. Локальный интерфейс, так же как и удаленный интерфейс, обеспечивает возможность обращения к методам компонента EJB. Локальный домашний интерфейс отвечает за доступ к методам управления жизненным циклом компонента EJB. Часто возникают ситуации, когда несколько компонентов EJB-сущностей, управляемых контейнером, располагаются локально и у них есть возможность локального доступа друг к другу.

5.2. Создание компонентов EJB

В этой части главы будет подробно рассмотрена работа с компонентами EJB.

5.2.1. Компоненты EJB-сущности

Компонент EJB-сущности — это объект, который существует постоянно, вне зависимости от работы программы. Она может создать компонента EJB-сущности, а затем может быть прервана и запущена вновь, но созданный компонент EJB-сущности будет существовать все это время без перерыва. После того как программа будет запущена вновь, она сможет найти ранее созданный компонент EJB и использовать его в дальнейшей работе.

Компонент EJB-сущности рассчитан на работу в сети. Стандартные объекты Java как правило используются только в пределах одной программы. Компонент EJB-сущности может быть использован любой программой, работающей в сети. Программе для этого необходимо лишь найти этот компонент EJB-сущности. Компонент EJB-сущности выполняется удаленно. Методы компонентов EJB выполняются на сервере. После вызова методов серверного компонента EJB прекращается работа текущего потока до тех пор, пока запрос не будет передан компоненту EJB и приложение не получит от него ответ. После получения ответа текущий поток возобновит свою работу.

Компонент EJB-сущности имеет первичный ключ, который служит идентификатором компонента EJB. Первичный ключ является уникальным. Каждый компонент EJB однозначно идентифицируется на основе своего первичного ключа.

5.2.2. Компоненты EJB-сессий простым языком

Компоненты EJB-сессий не являются постоянными. Это их первое отличие от компонентов EJB-сущностей. Компоненты EJB-сессий, как правило, не могут быть поделены между несколькими клиентами. Тем не менее, существует возможность использования компонента EJB несколькими клиентами посредством меток (handles) компонентов EJB. Компоненты EJB-сессий используются при необходимости решения распределенных задач, требующих работы с конкретным клиентом. Каждый компонент EJB-сессии используется при выполнении отдельной задачи для конкретного клиента. Клиент может работать с несколькими компонентами EJB-сессий. Задачи могут быть распределены таким образом, что соответствующие им компоненты EJB будут располагаться на различных компьютерах и самостоятельных виртуальных машинах Java.

Можно провести аналогию между функционированием компонентов EJB-сессий и обращающихся к ним клиентам с тем, как работают браузеры и Web-серверы. Web-сервер располагается в определенном месте на определенной машине. К нему осуществляют доступ различные браузеры, расположенные в различных местах. Каждый сервер предназначен для того, чтобы выполнять определенный набор действий (передавать определенный набор информации). Браузеры могут соединяться с произвольным числом серверов для получения необходимой информации. Конечно, задачи, решаемые серверным компонентом EJB-сессий могут быть более специфичны, чем задачи предоставления информации, решаемые Web-сервером. Они более связаны с программированием. Например, это могут быть определенные вычисления. Компоненты EJB могут также включать в себя методы представления произведенных вычислений, то есть методы отображения полученных результатов. Клиент обращается к компонентам EJB, вызывая их методы. Примером такой задачи может служить банковское приложение. Учет всех транзакций производится в одном центре. Java-приложение — аналог банкомата. Это приложение обращается к серверному компоненту EJB-сессии, который работает на центральном сервере и который производит операции со счетом, вычисляя его баланс, меняя текущий баланс и тому подобное в соответствии с теми операциями, которые будут указаны клиентом. Компонент EJB-сессии не имеет первичного ключа. Компонент EJB-сессии не используется для разделения и совместного использования несколькими клиентами. К нему нет необходимости обращаться повторно при новом запуске клиентского приложения.

5.2.3. Интерфейсы серверных компонентов EJB

Здесь будут более подробно рассматриваться интерфейсы серверных компонентов EJB.

Удаленный интерфейс (Remote Interface)

Удаленный интерфейс серверного компонента EJB — это функциональный язык компонента EJB. Удаленный интерфейс серверного компонента EJB содержит набор методов, предоставляемых этим компонентом EJB.

Домашний интерфейс (Home Interface)

Домашний интерфейс помогает клиентам создавать новые экземпляры компонентов EJB, а также находить уже существующие экземпляры компонентов EJB. Поиск существующих экземпляров компонентов EJB для компонентов EJB-сессий не актуален.

Имплементация интерфейсов

Домашний интерфейс не требует имплементации, задачи домашнего интерфейса достаточно однородны и конкретны, имплементация домашнего интерфейса автоматически осуществляется контейнером компонентов EJB.

Удаленный интерфейс должен быть имплементирован, так как он зависит от решаемых компонентом EJB задач.

Далее в настоящей главе будут рассмотрены описанные здесь задачи и способы их решения более подробно.

5.2.4. Компонент EJB-сущности

Серверные компоненты EJB для своей работы требуют наличия контейнера серверных компонентов EJB. Контейнером EJB, как правило, является часть сервера, поддерживающего работу с рядом технологий, в том числе работу с серверными компонентами EJB. В качестве примера возьмем сервер Blazix. Этот сервер содержит проводника создания серверных компонентов EJB. Проводник создания компонентов EJB позволит сконцентрироваться на основных принципиальных моментах, избавит от ненужных на первых шагах деталей. Общие принципы могут быть в дальнейшем использованы при работе и с другими серверами компонентов EJB. При работе с сервером Blazix необходимо убедиться в том, что переменная окружения `PATH` содержит путь к каталогу, в котором установлен сервер Blazix, а также в том, что в значении переменной `CLASSPATH` содержится файл blazix.jar.

Еще раз напомним, из чего состоят компоненты EJB-сущности. Компонент EJB-сущности требует наличия удаленного интерфейса. Этот интерфейс не производит непосредственно полезной работы, он служит для помощи клиенту, предоставляя ему ряд сервисов. Компонент EJB должен иметь домашний интерфейс. Перед тем как компонент EJB может быть использован, он должен быть найден или создан. Домашний интерфейс предоставляет методы для нахождения и создания компонентов EJB. Компонент EJB-сущности также имеет уникальный первичный ключ.

Проводник создания серверных компонентов EJB для сервера Blazix носит название Blizzard. Приступим к созданию первого серверного компонента EJB. Создадим компонент EJB-сущности. Он будет работать с базой данных. Для этого понадобится простая таблица. Используем средства Microsoft Access и создадим таблицу, состоящую из двух столбцов. Первый столбец — `key`, второй — `value`, имеющий тип `VARCHAR (255)` (рис. 5.5). Таблицу назовем `nashaTablitsa`, базу данных назовем db.mdb.

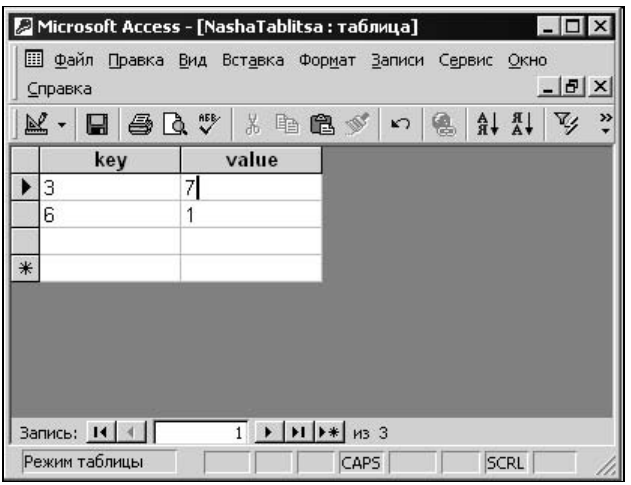


Рис. 5.5. Создание базы данных

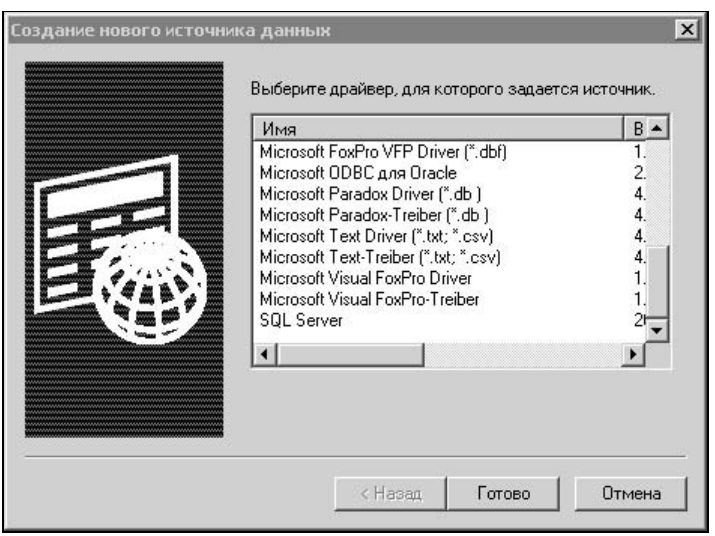


Рис. 5.6. Создание источника данных

Можно создать таблицу при помощи любой доступной базы данных. Для простоты воспользуемся MS Access. Создадим в Windows источник данных. Для этого откроем контрольную панель, найдем там пункт администрирования баз данных, закладку создания нового источника данных. Расположение этого сервиса может изменяться от одной системы к другой (рис. 5.6).

Выбираем драйвер для MS Access и переходим к следующему шагу. Задаем имя источника ODBC. Пусть это будет nashODBC (рис. 5.7).

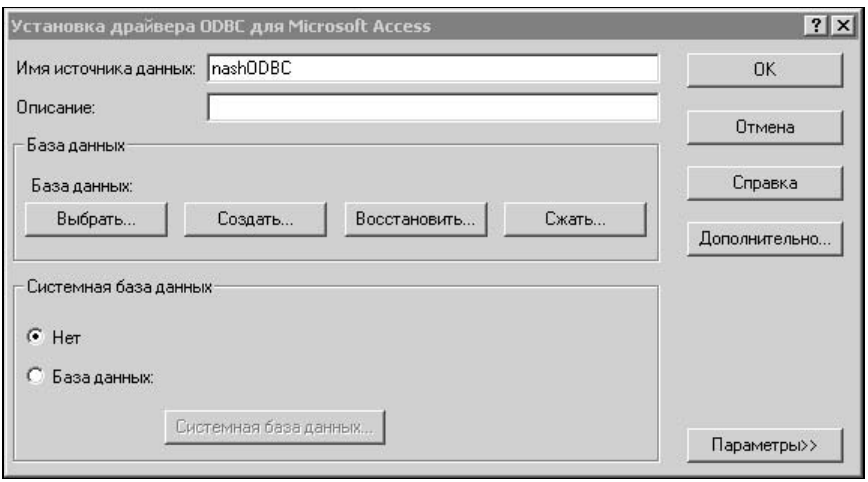


Рис. 5.7. Задание имени источника данных

Переходим к следующему диалоговому окну, нажимая кнопку **OK**. Здесь необходимо указать файл базы данных — это файл db.mdb (рис. 5.8).

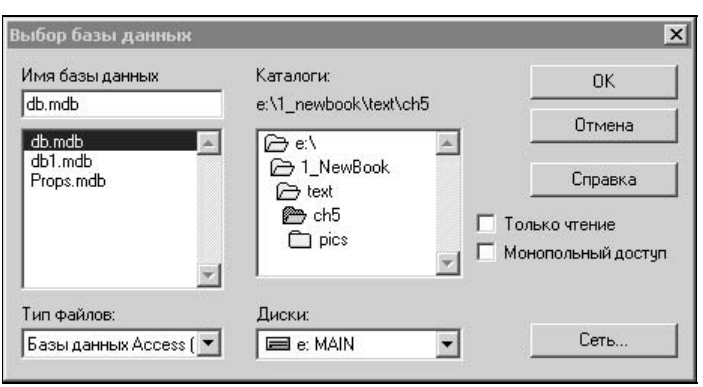


Рис. 5.8. Задание файла базы данных

На следующем шаге не меняем ничего, закрываем настройки, нажав снова кнопку **OK** (рис. 5.9).

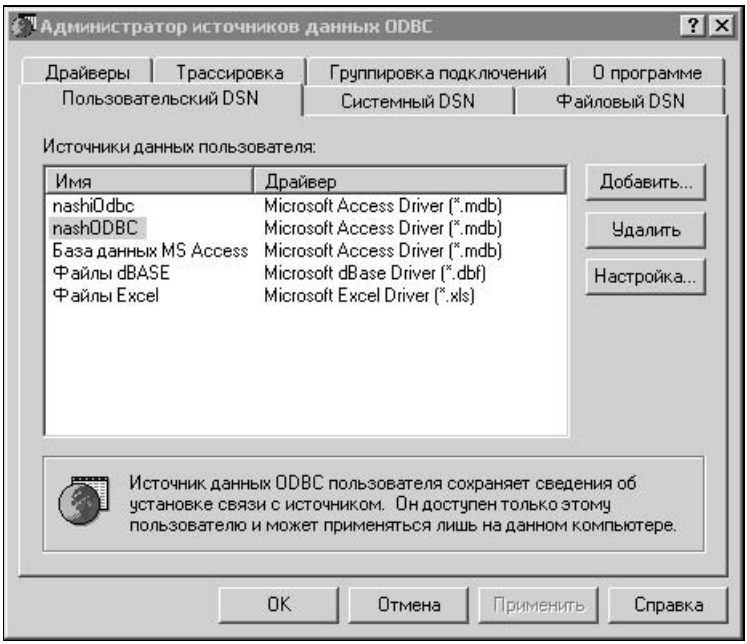


Рис. 5.9. Имя источника данных задано

Переходим к конфигурированию компонентов EJB на сервере EJB. Необходимо внести созданный источник данных в файл ejb.ini. Это сделать несложно, нужно вставить в этот файл две строчки.

```
dataSource.name: nashiDannye

dataSource.nashiDannye.odbc: nashODBC
```

Затем вызовем проводник Blizzard. Появляется окно проводника создания компонента EJB. Выбираем создание серверного компонента EJB (рис. 5.10). В окне присутствует выпадающий список, с помощью которого можно выбрать базу данных, с которой будет работать компонент EJB. В данном случае это `nashiDannye`. Если источник данных доступен без указания имени пользователя пароля, то пропускаем этот шаг. Если имя и пароль требуются (они задаются во время создания источника данных или позже при изменении параметров источника данных), то указываем их и переходим к следующему шагу.

Текущая база данных содержит единственную таблицу, которая появится в окне проводника. Оставим ее в качестве той таблицы, с которой работает компонент EJB (рис. 5.11) и перейдем к следующему шагу. Проводник будет открывать отдельное окно для настройки каждого столбца в указанной таблице.

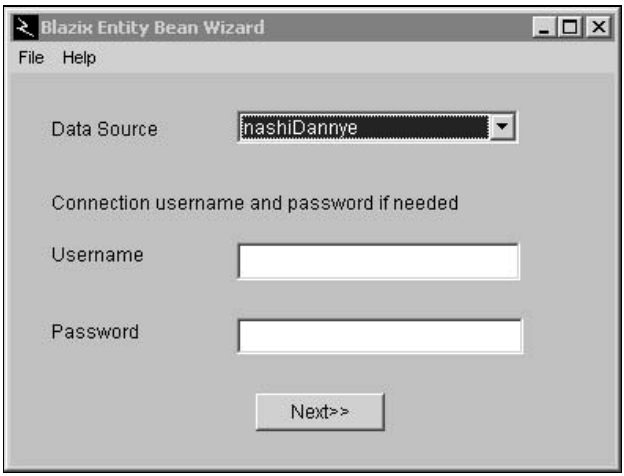


Рис. 5.10. Выбор источника данных для компонента EJB-сущности

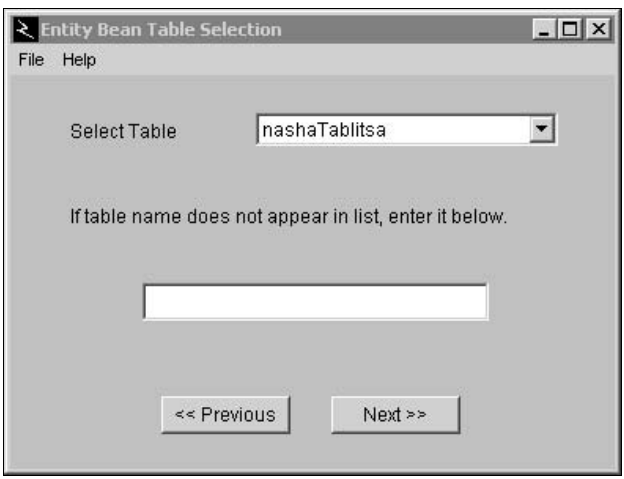


Рис. 5.11. Выбор таблицы

Укажем, что столбец `key` будет использоваться как первичный ключ, поставив переключатель напротив пункта **Use as the primary key**. Тип значений данного столбца будет `java.lang.String`, что указывается в списке ниже (рис. 5.12).

Следующим столбцом в таблице будет столбец `value`. Он не будет использоваться в качестве ключа (рис. 5.13), но мы укажем, что для него генерируется специальная функция `setter` (установим переключатель в положение **Generate Setter for this property**). Задаем тот же тип значений, что и для предыдущего столбца, то есть `java.lang.String`.

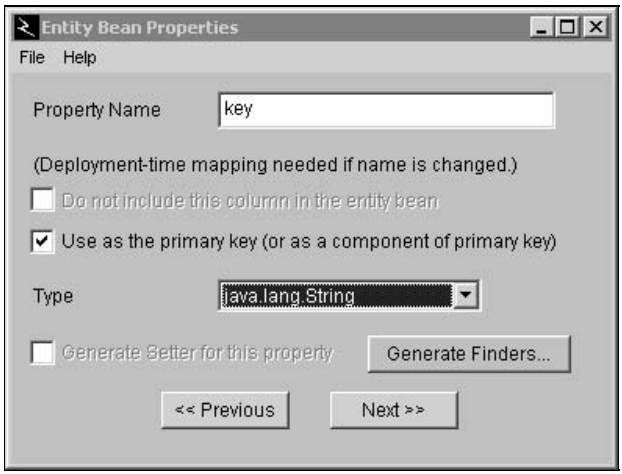


Рис. 5.12. Задание параметров столбца `key` в таблице

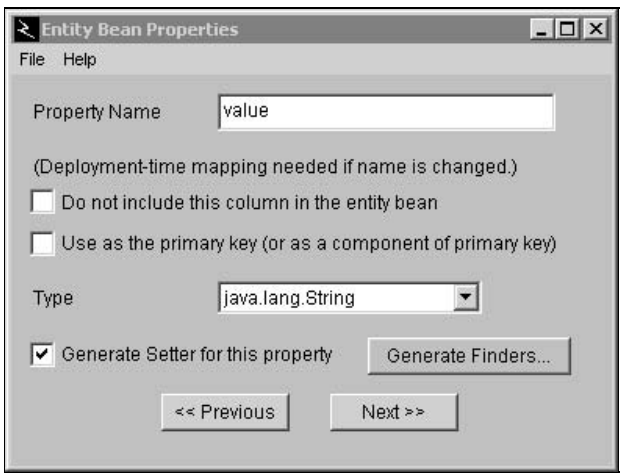


Рис. 5.13. Задание значений параметров для столбца `value`

Переходим к последнему диалоговому окну проводника создания серверного компонента EJB. Здесь мы можем указать название компонента EJB и пакета. Здесь же указываем папку, в которой будут помещены файлы компонента EJB. Пусть это будет каталог c:\ejb (рис. 5.14). В качестве названия компонента EJB и пакета укажем `NashBob` и `NashPaket` соответственно.

После нажатия кнопки **Build** проводник генерирует необходимые файлы. Обычно это занимает несколько секунд или даже меньше. По завершении генерации в консоли проводника будет выведено сообщение о том, как происходит процесс создания файлов компонента EJB (рис. 5.15).

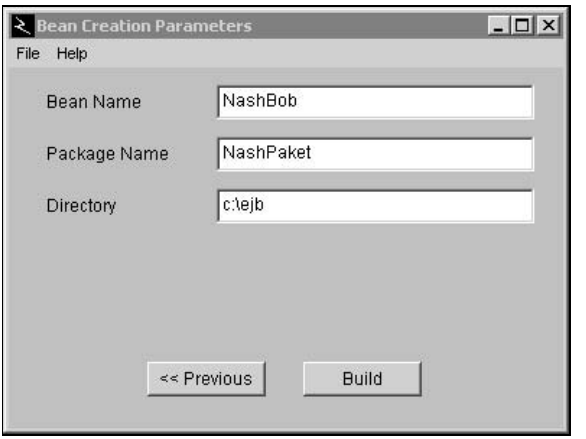


Рис. 5.14. Завершающий шаг работы с проводником по созданию серверного компонента EJB

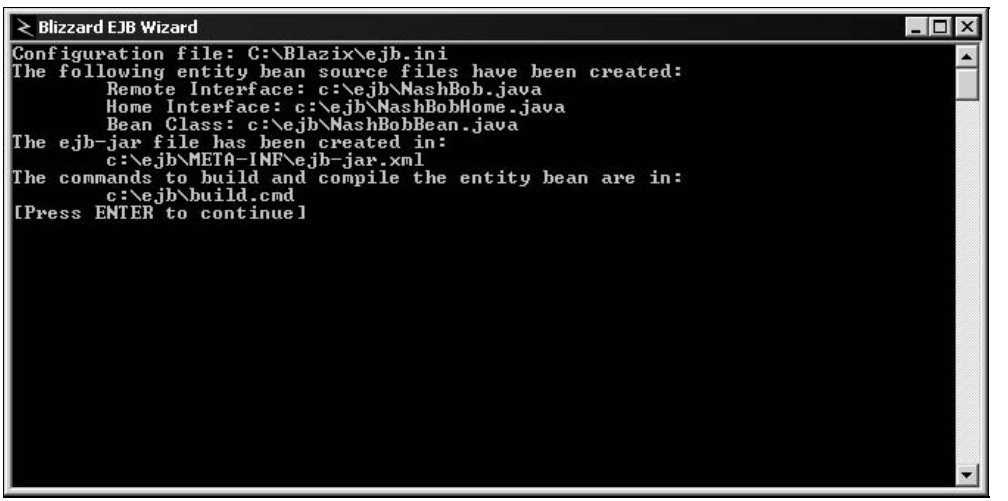


Рис. 5.15. Сообщение о создании файлов компонента EJB

В папке c:\ejb находятся несколько сгенерированных файлов. Файл build.cmd содержит команды для компиляции классов. Код компонента EJB хранится в файлах NashBobHome.java (листинг 5.1), NashBobBean.java (листинг 5.2), NashBob.java (листинг 5.3). Помимо этого в папке META-INF расположен файл ejb-jar.xml.

Листинг 5.1. Файл NashBobHome.java

```
package NashPaket;
import javax.ejb.*;
```

```
import java.rmi.*;
import java.util.*;

public interface NashBobHome extends javax.ejb.EJBHome {
  NashBob create(
  java.lang.String key,
  java.lang.String value
) throws javax.ejb.CreateException, java.rmi.RemoteException;

  NashBob findByPrimaryKey(java.lang.String pkey) throws
javax.ejb.FinderException, java.rmi.RemoteException;
}
```

Файл с описанием домашнего интерфейса достаточно прост. Класс самого компонента EJB тоже не очень сложен.

Листинг 5.2. Файл NashBobBean.java

```
package NashPaket;
import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;

public class NashBobBean implements javax.ejb.EntityBean {
  //Переменные
  public java.lang.String key = null;
  public java.lang.String value = null;
  // Контекст (для работы с метками handles и пр.)
  javax.ejb.EntityContext ejbEntityContext = null;
  //Методы get/set
  public java.lang.String getKey() throws java.rmi.RemoteException {
    return key;
  }
  public java.lang.String getValue() throws java.rmi.RemoteException
  {
    return value;
  }
  public void setValue(java.lang.String value)
  throws java.rmi.RemoteException
  {
```

```
    this.value = value;
  }
  //  Имплементация дополнительных удаленных методов.
  //  По умолчанию – метод ejbCreate.
  public java.lang.String ejbCreate(
  java.lang.String key,
  java.lang.String value
)
  throws javax.ejb.CreateException, java.rmi.RemoteException
  {
    this.key = key;
    this.value = value;
    return null;
  }
  //  при необходимости имплементируются прочие дополнительные методы —
  //  работа разработчика
  public void setEntityContext(javax.ejb.EntityContext ejbEntityContext)
  throws RemoteException {
    this.ejbEntityContext = ejbEntityContext;
  }
  public void unsetEntityContext()
  throws RemoteException {
    this.ejbEntityContext = null;
  }
  public void ejbPostCreate(
  java.lang.String key,
  java.lang.String value
)
  {
  // Создается разработчиком:
  // функции, используемые после создания экземпляра компонента EJB.
  }
  public void ejbRemove()
  throws java.rmi.RemoteException, javax.ejb.RemoveException
  {
  // Создается разработчиком:
  // функции, выполняемые при уничтожении экземпляра компонента EJB.
  }
```

```
  public void ejbActivate()
  throws java.rmi.RemoteException
  {
  // Создается разработчиком:
  // функции, используемые при активации экземпляра компонента EJB.
  }
  public void ejbPassivate()
  throws java.rmi.RemoteException
  {
  // Создается разработчиком:
  // функции, выполняемые при пассивации компонента EJB.
  }
  public void ejbLoad()
  throws java.rmi.RemoteException
  {
  // Создается разработчиком:
  // загрузка прочих данных помимо тех, что предоставлены
  // контейнером компонентов EJB.
  }
  public void ejbStore()
  throws java.rmi.RemoteException
  {
  //  Создается разработчиком:
  //  сохранение прочих данных, кроме тех, с которыми работает
  //  контейнер компонента EJB.
  }
}
```

Файл NashBob.java (листинг 5.3) по размеру небольшой.

Листинг 5.3. Файл NashBob.java

```
package NashPaket;
import javax.ejb.*;
import java.rmi.*;

public interface NashBob extends javax.ejb.EJBObject {
  java.lang.String getKey() throws java.rmi.RemoteException;
  java.lang.String getValue() throws java.rmi.RemoteException;
```

```
  void setValue(java.lang.String value) throws java.rmi.RemoteException;
  // Создается разработчиком:
  // методы удаленного интерфейса.
}
```

Отметим, что во всех перечисленных файлах использован пакет для работы с серверными компонентами EJB `javax.ejb.*`, а также пакет для работы с вызовом удаленных процедур `java.rmi.*`.

Файл build.cmd (листинг 5.4) состоит из нескольких команд.

Листинг 5.4. Файл build.cmd

```
javac -d "c:\ejb" "c:\ejb\NashBobHome.java" "c:\ejb\NashBob.java"
"c:\ejb\NashBobBean.java"


jar -cvf NashBob.jar -C "c:\ejb" META-INF/ejb-jar.xml
"NashPaket/NashBobHome.class" "NashPaket/NashBob.class"
"NashPaket/NashBobBean.class"
blxejbc NashBob.jar NashBobEjb.jar
```

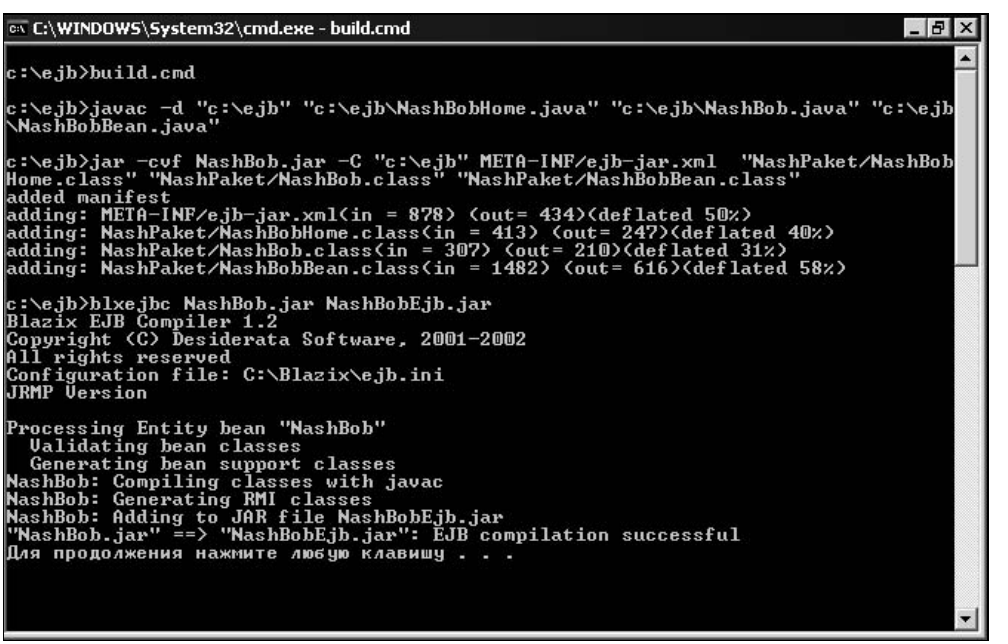


Рис. 5.16. Создаем и упаковываем классы компонента EJB

В файле build.cmd содержатся три команды. Первая команда — это инструкция компиляции `javac`. Команда `jar -cvf` служит для создания файла JAR на основе полученных в результате выполнения предыдущей команды файлов классов. Наконец, blxejbc — это компилятор сервера Blazix для создания файла серверного компонента EJB, т. е. файла JAR, который готов к тому, чтобы быть размещенным на сервере. Для того чтобы команда `blxejbc` успешно выполнила все свои функции, необходимо в переменной `PATH` задать путь к папке, где установлен сервер Blazix.

Выполним (рис. 5.16) из консоли файл build.cmd (если `javac` не находит классы, убедитесь в том, что все необходимые пакеты установлены на вашем компьютере).

После компиляции в папке c:\ejb появится новая папка NashPacket, содержащая файлы трех классов, а также два новых файла в папке c:\ejb — файл NashBob.jar и файл NashBobEjb.jar.

Размещение компонента EJB на сервере

Для того чтобы разместить вновь созданный компонент EJB на сервере, необходимо изменить файл ejb.ini, вставив в него три строки.

Листинг 5.5. Вставка в файл ejb.ini

```
ejbJar: c:\ejb\NashBobEjb.jar
ejb.NashBob.dataSource: nashiDannye
ejb.NashBob.table: NashaTablitsa
```

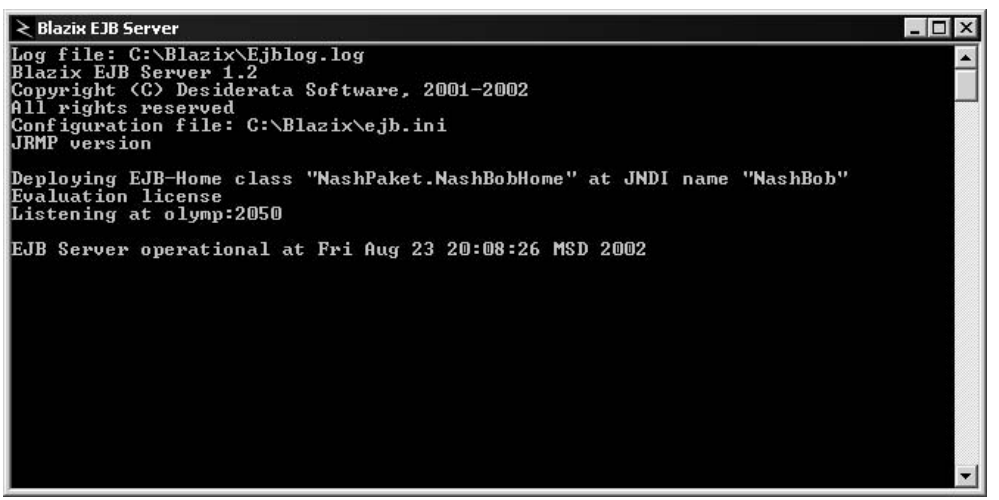


Рис. 5.17. Сообщение о размещении компонента EJB на сервере

Теперь можно запустить сервер EJB. Если ошибки в конфигурировании серверных компонентов EJB не были допущены (файл ejb.ini содержит указанную выше информацию), то в консоли появится сообщение о том, что компонент EJB NashBob благополучно был размещен на сервере (рис. 5.17).

На практике может оказаться неудобным остановка и повторный запуск сервера EJB в целях размещения вновь созданного компонента EJB. Существуют методы размещения компонента EJB на сервере без остановки сервера. Сейчас мы не будем останавливаться на этом.

Обращение к серверному компоненту EJB

Удаленный интерфейс описан в файле NashBob.java. Удаленный интерфейс описывает методы взаимодействия с компонентом EJB-сущности. Прежде чем можно будет взаимодействовать с компонентом EJB, необходимо найти этот компонент EJB или же создать новый компонент EJB. Интерфейс, который позволяет обнаруживать или создавать новый компонент EJB, располагается в файле NashBobHome.java.

Сначала получаем ссылку на домашний объект следующим образом:

```
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.*;
import java.util.*;
import props.*;


  Properties env = new Properties();
  env.put("java.naming.factory.initial",
  "desisoft.ejb.client.JRMPFactory");
  env.put("desisoft.ejb.nameServer1",
  "localhost:2050");
  Context ctx = new InitialContext(env);
  NashBobHome home =
  (NashBobHome) ctx.lookup("NashBob");
```

При работе с сервером Blazix, когда ссылка ищется из страниц JSP или из сервлета, передача свойств `env` конструктору `InitialContext` не требуется, поскольку окружение уже автоматически будет инициировано. Вместо этого нужно указать префикс поиска "java:comp/env/ejb/":

```
NashBobHome home = (NashBobHome) ctx.lookup("java:comp/env/ejb/Props");
```

Более корректный вариант последней строчки кода выглядит так, как показано ниже. Этот вариант портируем, так как в нем используется `PortableRemoteObject`:

```
Propshome home = (PropsHome)
```

```
  PortableRemoteObject.narrow(
   ctx.lookup("NashBob"),
   NashBobHome.class);
```

Рекомендуется всегда при приведении типов использовать `PortableRemoteObject`. Отметим, что каждый конкретный контейнер серверных компонентов EJB будет по-своему осуществлять поиск домашнего интерфейса. В данном случае сервер Blazix работает с классами `desisoft.ejb.*`.

Теперь, когда мы располагаем ссылкой на домашний интерфейс, процесс нахождения или создания компонента EJB будет весьма прост: для этого нужно лишь вызвать соответствующий метод. После создания или нахождения компонента EJB мы получим удаленный интерфейс. Использование компонента EJB сводится к вызову методов этого удаленного интерфейса. Все вышесказанное можно реализовать при помощи кода, который приводится ниже (листинг 5.6).

Листинг 5.6. Файл клиента ClientBobSuschnosti.java

```
// пример обращения к компоненту EJB-сущности
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.*;
import java.util.*;
import NashPaket.*;

public class ClientBobSuschnosti {
  static public void main(String[] args)
  {
    checkArgs(args);
    try {
    // получение домашнего интерфейса компонента EJB
    Properties env = new Properties();
    env.put("java.naming.factory.initial",
    "desisoft.ejb.client.JRMPFactory");
    env.put("desisoft.ejb.nameServer1",
    "localhost:2050");
    Context ctx = new InitialContext(env);
    NashBobHome home = (NashBobHome) PortableRemoteObject.narrow
    (ctx.lookup("NashBob"), NashBob.class);
```

```
    NashBob bean;
    if (doGet) {
      try {
        bean = home.findByPrimaryKey(prop);
        System.out.println(prop + ": " + bean.getValue());
      }
      catch (FinderException notFound) {
       System.out.println("Property \"" + prop + "\" was not found");
      }
    }
    else {
      try {
        bean = home.create(prop, value);
      }
      catch (DuplicateKeyException exists) {
        bean = home.findByPrimaryKey(prop);
      }
    bean.setValue(value);
    System.out.println("Set value of \"" + prop + "\" to \"" + value +
    "\"");
    }
  }
  catch (Exception ex) {
    ex.printStackTrace();
  }
}
static void usage()
{
  System.out.println("Usage: java ClientBobSuschnosti put <prop-name>
  <value>");
  System.out.println("  java ClientBobSuschnosti get <prop-name>");
  System.exit(0);
}
static void checkArgs(String[] args)
{
  if (args.length < 2)
    usage();
  prop = args[1];
  if (args[0].equalsIgnoreCase("get")) {
```

```
    if (args.length > 2)
      usage();
    doGet = true;
  }
  else
    if (args[0].equalsIgnoreCase("put")) {
     if (args.length != 3)
      usage();
    doGet = false;
    value = args[2];
    }
    else
      usage();
  }
  static boolean doGet = false;
  static String prop = null;
  static String value = null;
}
```

Этот код может быть использован только для jdk1.3, но не для jdk1.2. Для того чтобы скомпилировать этот код, необходимо указать в значении переменной `CLASSPATH` файл NashBobEjb.jar. Поместим его в отдельную папку, где будут храниться приложения, пусть это будет папка c:\ejbtest, и скомпилируем, выполнив консольную команду `javac ClientBobSuschnosti.java` (рис. 5.18).

Рис. 5.18. Файл ClientBobSuschnosti.java скомпилирован

После компиляции появился класс `ClientBobSuschnosti.class`. Укажем путь к этому классу в качестве одного из путей, содержащихся в переменной `CLASSPATH`, а затем выполним команду `java ClientBobSuschnosti` (рис. 5.19). Мы неправильно обратились к компоненту EJB, но в ответ получили инструкции, как следует к нему обращаться. Исправим ошибку, повторим обращение к компоненту EJB:

```
Java ClientBobSuschnosti put imya znacheniye
```

Получаем сообщения об ошибках (рис. 5.20).

Рис. 5.19. Синтаксическая ошибка при обращении к компоненту EJB

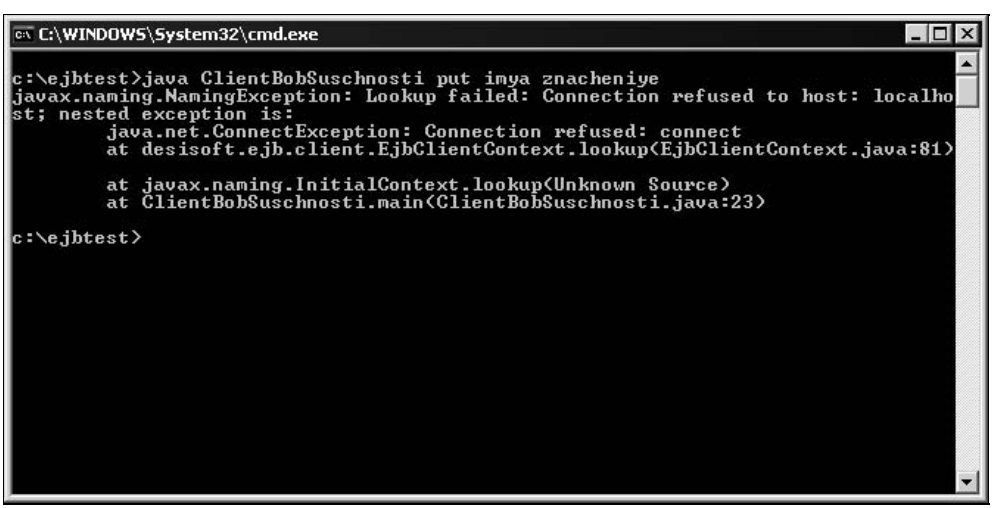


Рис. 5.20. Ошибка при обращении к компоненту EJB

Приведенные в качестве примера ошибки легко устранить — нужно запустить сервер компонентов EJB (сервер не был запущен). Запускаем сервер и повторяем команду

```
Java ClientBobSuschnosti put imya znacheniye
```

Опять получаем сообщения об ошибках (рис. 5.21). На этот раз получаем ошибку при обращении к базе данных.

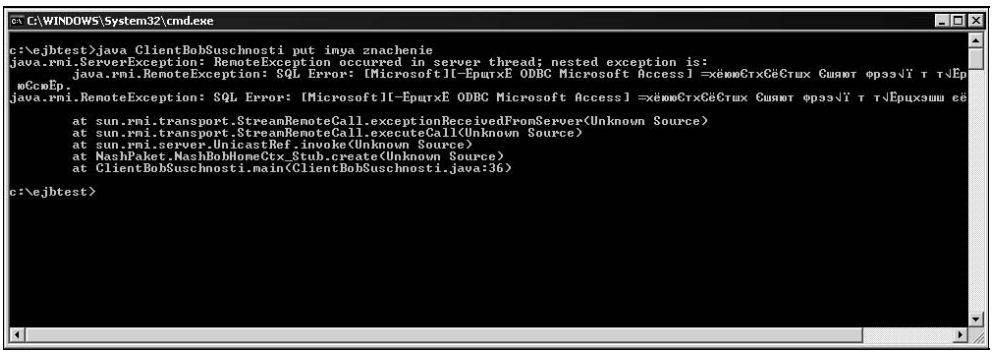


Рис. 5.21. Ошибка при работе с базой данных

Вспомним, что в качестве значений полей при создании таблицы базы данных использовались числовые значения. Возникает подозрение, что строковые значения не могут быть преобразованы в числовые значения, которые хранит база данных. Вместо строк используем числа, как это сделано ниже.

```
Java ClientBobSuschnosti put 101 777
```

Таким образом попытка завершилась удачно (рис. 5.22).

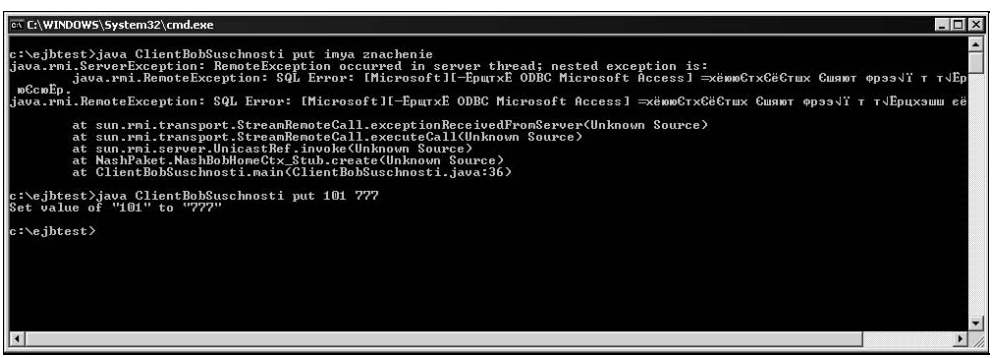


Рис. 5.22. Удачное обращение к компоненту EJB

Если сейчас попытаться существенно изменить структуру таблицы или базы данных, например, попытаться удалить таблицу, то система выдаст сообщение о том, что таблица используется (компонент EJB имеет постоянное под-

ключение к таблице) и не может быть удалена. Выключим сервер компонентов EJB. Только сейчас таблицу можно модифицировать (например, удалить). Изменим таблицу: пусть значениями полей станут строки (рис. 5.23).

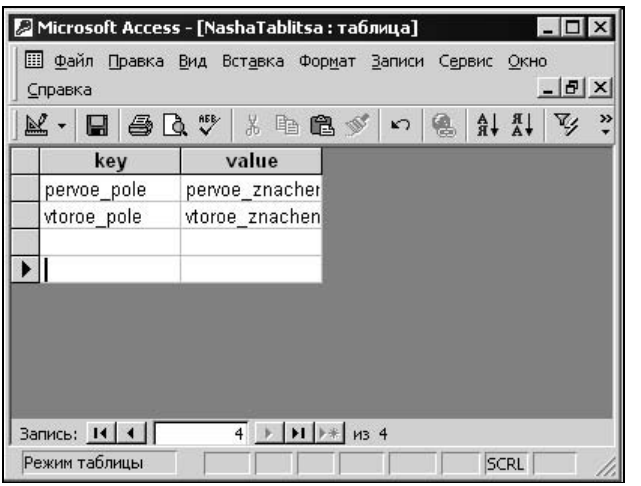


Рис. 5.23. Измененная таблица NashaTablitsa

Вновь запустим сервер компонентов EJB и выполним команду

```
Java ClientBobSuschnosti put imya znacheniye
```

В этом случае не возникнет никаких проблем (рис. 5.24).

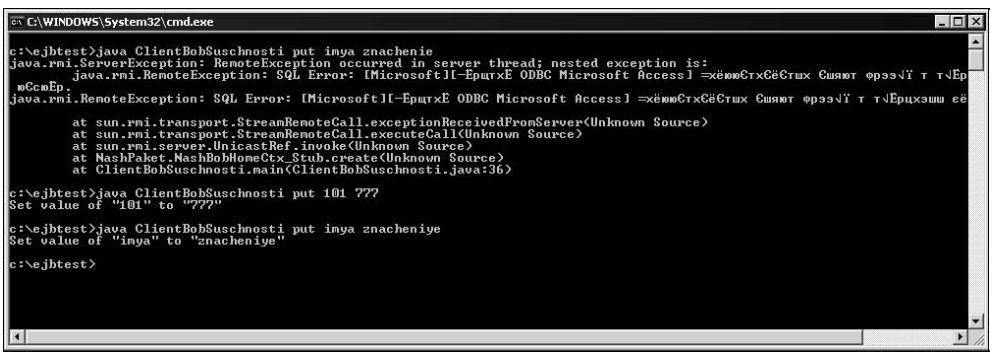


Рис. 5.24. Все неполадки устранены

Чтобы посмотреть значение параметра, выполняем команду (рис. 5.25).

```
Java ClientBobSuschnosti get imya
```

В примере, который рассматривался выше, компонент EJB находился по указанному значению первичного ключа. При этом всегда находится единственный компонент EJB, так как для данного первичного ключа существует

лишь одна запись в базе данных. Существует и другой метод поиска компонента EJB. При этом в проводнике создания компонента EJB следует нажать кнопку **Generate Finders** (рис. 5.26).

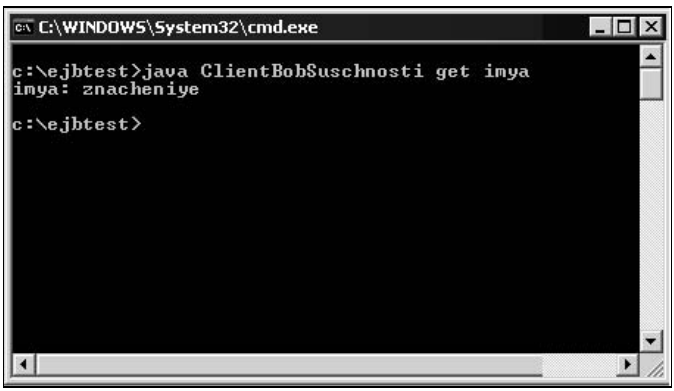


Рис. 5.25. Получение значения параметра от компонента EJB

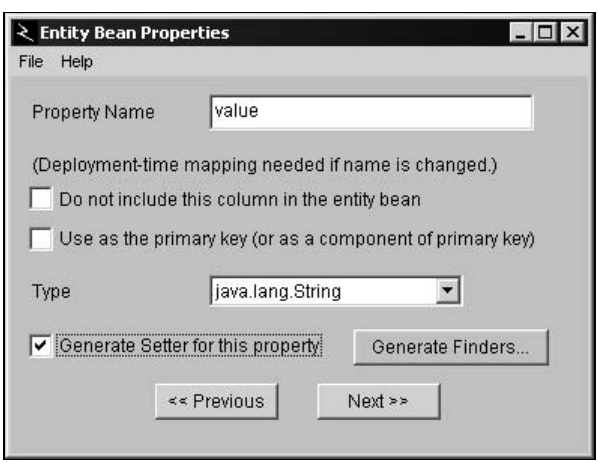


Рис. 5.26. Альтернативный метод поиска компонента EJB

Следующий шаг — выбор критерия поиска (рис. 5.27).

Первый переключатель следует установить в том случае, если будут созданы компоненты EJB для всех значений столбца `value`, которые окажутся меньше, чем заданное значение. Второй переключатель соответствует выбору всех значений, больших, чем заданное. Третий переключатель соответствует созданию компонентов EJB при совпадении значения поля указанному значению. Последний четвертый переключатель соответствует поиску всех значений, расположенных между двух указанных значений. Выберем третий вариант и перейдем к завершающему шагу (рис. 5.28).

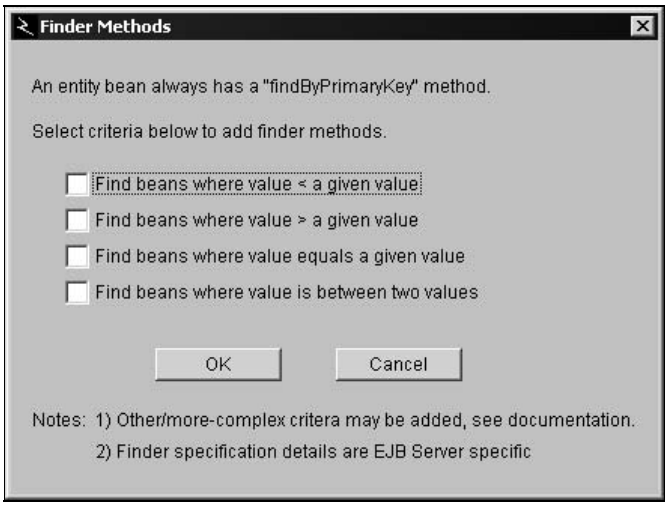


Рис. 5.27. Критерии поиска компонента EJB

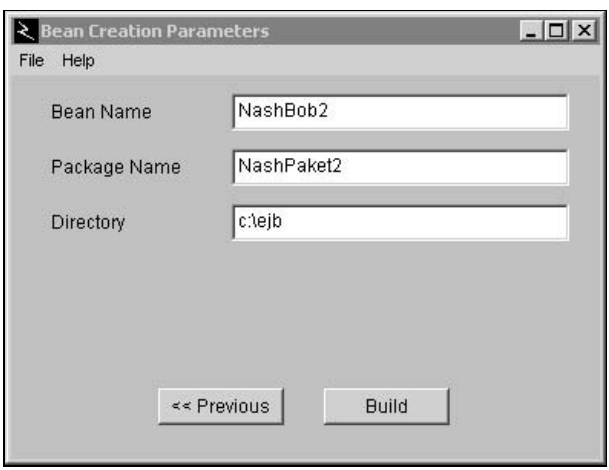


Рис. 5.28. Задание имени компонента EJB и пакета

В качестве имени компонента EJB указываем `NashBob2`, пакет будет называться `NashPaket2`, каталог c:\ejb.

Новые методы поиска отражены в файле, содержащем домашний интерфейс (листинг 5.7).

Листинг 5.7. Файл NashBob2Home.java

```
package NashPaket2;
import javax.ejb.*;
```

```
import java.rmi.*;
import java.util.*;

public interface NashBob2Home extends javax.ejb.EJBHome {
  NashBob2 create(
  java.lang.String key,
  java.lang.String value
)
throws javax.ejb.CreateException, java.rmi.RemoteException;
  NashBob2 findByPrimaryKey(java.lang.String pkey) throws
javax.ejb.FinderException, java.rmi.RemoteException;
  java.util.Collection findValueEquals(java.lang.String value) throws
javax.ejb.FinderException, java.rmi.RemoteException;
}
```

5.2.5. Компонент EJB-сессии

Создание компонента EJB-сессии представляется еще более простым. Компонент EJB-сессии не имеет первичного ключа, обладает удаленным и домашним интерфейсом, а также не связан с базами данных. Чтобы создать компонент EJB-сессии, снова воспользуемся проводником Blizzard. Компонент EJB будет выдавать биржевые котировки, располагаясь на том компьютере, где информация о котировках будет храниться в виде файла, расположенного в фиксированном каталоге. Пусть это будет файл c:\StockPrices.txt. Клиент будет обращаться к компоненту EJB, указывая название акций, в ответ же он будет получать котировки акций.

Запускаем проводника создания серверных компонентов EJB Blizzard, выбираем пункт **Create a Session Bean**, задаем название компонента EJB `BobKotirovki`, название пакета, где будут храниться файлы компонента EJB `PaketKotirovki`, и каталог, куда будут помещены файлы c:\ejb\kotirovki (рис. 5.29). Также вставим в компонент EJB данные `stock` и `price`.

Данные `stock` должны иметь тип `string`, а цена `price` — тип `float`. Методы задания параметра для данных цены (**Generate Setter Methods**) не потребуются, но они нужны для данных `stock`. Помимо этого отметим, что компонент EJB будет поддерживать состояние сессии (отметить **stateful**), а транзакции будут регулироваться средствами контейнера (отметить **let the container manage transacton boundaries**). После совершения всех операций нажимаем кнопку **Build**.

Через несколько мгновений получаем уже знакомый набор файлов и папок в каталоге c:\ejb\kotirovki. Это файлы `BobKotirovki.java` (листинг 5.8) — класс компонента EJB, `BobKotirovkiHome.java` (листинг 5.9) — домашний

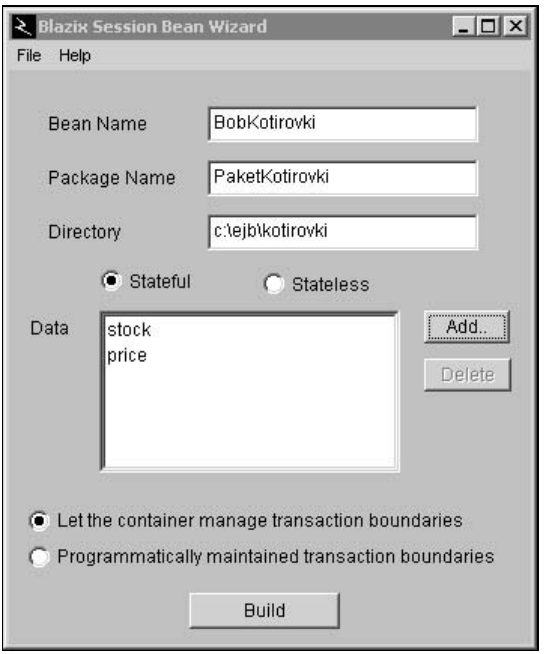


Рис. 5.29. Создание компонента EJB-сессии

интерфейс и `BobKotirovkiBean.java` (листинг 5.10) — удаленный интерфейс. В сгенерированных файлах не содержится первичного ключа, в них не будет метода `findByPrimaryKey`.

Листинг 5.8. Удаленный интерфейс BobKotirovki.java — сгенерирован Blizzard

```
package PaketKotirovki;
// File generated by Desiderata Software's Blazix session bean wizard
// Удаленный интерфейс для компонента EJB-сессии "BobKotirovki".

import javax.ejb.*;
import java.rmi.*;

public interface BobKotirovki extends javax.ejb.EJBObject {
  String getStock() throws java.rmi.RemoteException;
  void setStock(String stock) throws java.rmi.RemoteException;
  float getPrice() throws java.rmi.RemoteException;
  // Задача разработчика:
  // прочие методы удаленного интерфейса.
}
```

Листинг 5.9. Домашний интерфейс BobKotirovkiHome.java — сгенерирован Blizzard

```
package PaketKotirovki;
// File generated by Desiderata Software's Blazix session bean wizard
// Домашний интерфейс для компонента EJB-сессии "BobKotirovki".

import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface BobKotirovkiHome extends javax.ejb.EJBHome {
  BobKotirovki create(
  String stock,
  float price
) throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```

Листинг 5.10. Файл класса компонента EJB BobKotirovkiBean.java — сгенерирован Blizzard

```
package PaketKotirovki;
// File generated by Desiderata Software's Blazix session bean wizard
// Класс компонента EJB-сессии "BobKotirovki".

import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;

public class BobKotirovkiBean implements javax.ejb.SessionBean {
  public String stock = null;
  public float price = 0.0F;
  // контекст сессии
  javax.ejb.SessionContext ejbSessionContext = null;
  public String getStock() throws java.rmi.RemoteException
  {
  return stock;
  }
  public void setStock(String stock) throws java.rmi.RemoteException
  {
```

```
  this.stock = stock;
  }
  public float getPrice() throws java.rmi.RemoteException
  {
  return price;
  }
  // Задача разработчика:
  // прочие методы.

  // Метод ejbCreate по умолчанию.
  public void ejbCreate(
  String stock,
  float price
)
  throws javax.ejb.CreateException, java.rmi.RemoteException
  {
  this.stock = stock;
  this.price = price;
  }

  // Задача разработчика:
  // прочие методы, выполняемые во время создания компонента EJB-сессии.

public void setSessionContext(javax.ejb.SessionContext ejbSessionContext)
  throws RemoteException
  {
  this.ejbSessionContext = ejbSessionContext;
  }
  public void unsetSessionContext()
  throws RemoteException
  {
  this.ejbSessionContext = null;
  }
  public void ejbRemove()
  throws java.rmi.RemoteException, javax.ejb.EJBException
  {
  // Задача разработчика:
  // прочие методы, выполняемые при удалении экземпляра компонента
```

```
  // EJB-сессии
  }

  public void ejbActivate()
  throws java.rmi.RemoteException
  {
  // Задача разработчика:
  // обращение к ресурсам.
  }

  public void ejbPassivate()
  throws java.rmi.RemoteException
  {
  // Задача разработчика
  // сохранение ресурсов.
  }
}
```

Нам будет необходим ресурс, то есть файл с котировками c:\StockProces.txt (листинг 5.11). Формат информации, представленной в файле, следующий:

название_акции : цена

Листинг 5.11. Вспомогательный файл c:\StockPrices.txt

```
AOL: 151.03
AWE: 127.30
IBM: 190.25
SUNW: 213.11
ORCL: 152.62
INTC: 271.85
MSFT: 170.34
DELL: 226.35
JDSU: 192.47
T: 90.62
LU: 28.12
EMC: 365.97
GOU: 107.11
GE: 4.27
CSCO: 20.42
```

Сейчас необходимо внести изменения в файл класса компонента EJB-сессии `StockQuotesBean.java`. Вставим инструкцию импорта пакета `java.io.*` (будем работать с файлом StockProces.txt). Удалим переменную `price`, а также связанные с этой переменной действия, а именно, аргумент `price` в методе `ejbCreate`, и фрагмент, где задается значение для `price`. В листинге выше все удаляемые фрагменты выделены полужирным шрифтом.

Кроме того, необходимо создать заново метод `getPrice()`. Новый метод будет выглядеть так, как показано ниже (листинг 5.12).

Листинг 5.12. Метод `getPrice()`

```
public float getPrice() throws java.rmi.RemoteException,
               java.io.IOException
{
 BufferedReader reader;
 reader = new BufferedReader(
 new FileReader("C:\\StockPrices.txt"));
 String line;
 String prefix = stock.toLowerCase() + ":";
 while ((line = reader.readLine()) != null) {
   if (line.toLowerCase().startsWith(prefix)) {
    line = line.substring(prefix.length());
    reader.close();
    return Float.parseFloat(line.trim());
   }
 }
 reader.close();
 throw new java.rmi.RemoteException("Not found");
 // оставим без создания класса обработки ошибки
}
```

Таким образом, были внесены изменения в метод `getPrice()`, которые потребуют внесения изменений в файле `BobKotirovki.java`, где нужно внести в список исключений функции `getPrice()` исключение `java.io.IOException`. Такое же изменение нужно произвести в файле `BobKotirovkiHome.java`, кроме того, в этом файле необходимо удалить переменную `price`.

Окончательный вид исходных файлов приведен ниже (листинги 5.13–5.15).

Листинг 5.13. Удаленный интерфейс — окончательный вариант (BobKotirovki.java)

```
package PaketKotirovki;
// File generated by Desiderata Software's Blazix session bean wizard
// Удаленный интерфейс для компонента EJB-сессии BobKotirovki.

import javax.ejb.*;
import java.rmi.*;

public interface BobKotirovki extends javax.ejb.EJBObject {
  String getStock() throws java.rmi.RemoteException;
  void setStock(String stock) throws java.rmi.RemoteException;
  float getPrice() throws java.rmi.RemoteException, java.io.IOException ;
  // прочие методы
}
```

Листинг 5.14. Домашний интерфейс — окончательный вариант (BobKotirovkiHome.java)

```
package PaketKotirovki;
// File generated by Desiderata Software's Blazix session bean wizard
// Домашний интерфейс для компонента EJB-сессии "BobKotirovki".

import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface BobKotirovkiHome extends javax.ejb.EJBHome {
  BobKotirovki create(
   String stock
   // float price
) throws javax.ejb.CreateException, java.rmi.RemoteException,
java.io.IOException;
}
```

Листинг 5.15. Класс компонента EJB — окончательный вариант (BobKotirovkiBean.java)

```
package PaketKotirovki;
// File generated by Desiderata Software's Blazix session bean wizard
```

```
// Класс компонента EJB сессии "BobKotirovki".

import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;
import java.io.* ;

public class BobKotirovkiBean implements javax.ejb.SessionBean {

  //переменные
  public String stock = null;
  // контекст сессии
  javax.ejb.SessionContext ejbSessionContext = null;
  // методы get и set
  public String getStock() throws java.rmi.RemoteException
  {
  return stock;
  }
  public void setStock(String stock) throws java.rmi.RemoteException
  {
  this.stock = stock;
  }
  /*  public float getPrice() throws java.rmi.RemoteException
  {
  return price;
  }
  */
  public float getPrice() throws java.rmi.RemoteException,
  java.io.IOException
  {
  BufferedReader reader;
  reader = new BufferedReader(
  new FileReader("C:\\StockPrices.txt"));
  String line;
  String prefix = stock.toLowerCase() + ":";
  while ((line = reader.readLine()) != null) {
  if (line.toLowerCase().startsWith(prefix)) {
  line = line.substring(prefix.length());
  reader.close();
```

```
return Float.parseFloat(line.trim());
}
}
reader.close();
throw new java.rmi.RemoteException("Not found");
// оставим без создания класса обработки ошибки
}
// Дополнительные методы
// Метод ejbCreate по умолчанию
public void ejbCreate(
String stock //,
//  float price
)
throws javax.ejb.CreateException, java.rmi.RemoteException
{
this.stock = stock;
//  this.price = price;
}
// Дополнительные методы на момент создания компонента EJB
// прочие методы компонента EJB-сущности
public void setSessionContext
 (javax.ejb.SessionContext ejbSessionContext)
throws RemoteException
{
this.ejbSessionContext = ejbSessionContext;
}
public void unsetSessionContext()
throws RemoteException
{
this.ejbSessionContext = null;
}
public void ejbRemove()
throws java.rmi.RemoteException, javax.ejb.EJBException
{
// методы на момент удаления экземпляра компонента EJB
}
public void ejbActivate()
throws java.rmi.RemoteException
{
```

```
  // обращение к ресурсам
  }
  public void ejbPassivate()
  throws java.rmi.RemoteException
  {
  // сохранение ресурсов
  }
}
```

После внесения всех изменений запускаем командный файл build.com, компонент EJB будет благополучно создан (рис. 5.30).

Для того чтобы разместить вновь созданный компонент EJB на сервере компонентов EJB, нужно либо скопировать файл `BobKotirovkiEjb.jar` в каталог ejbdir сервера Blazix, либо указать в файле ejb.ini:

```
ejbJar: c:\ejb\kotirovki\BobKotirovkiEjb.jar
```

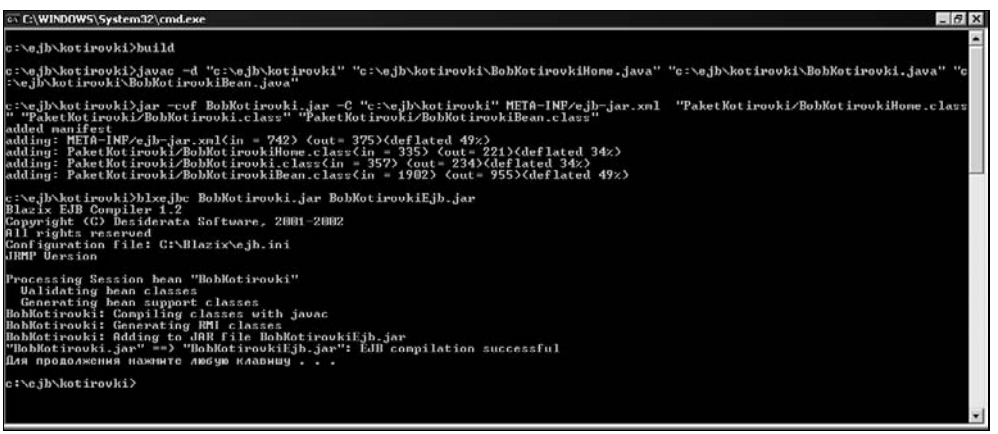


Рис. 5.30. Создание компонента EJB-сессии завершено

Рис. 5.31. Компонент EJB-сессии успешно размещен на сервере

Запускаем сервер компонентов EJB и убеждаемся, что новый компонент EJB был успешно размещен (рис. 5.31).

Сейчас необходимо постараться использовать созданный компонент EJB-сессии. Для этого создадим класс клиента (листинг 5.16).

Листинг 5.16. Клиентская программа ClientKotirovki.java

```
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;
import java.util.*;
import java.io.*;
import PaketKotirovki.*;

public class ClientKotirovki {
  static public void main(String[] args)
  {
  try {
  // поиск компонента EJB
  Properties env = new Properties();
  env.put("java.naming.factory.initial",
  "desisoft.ejb.client.JRMPFactory");
  env.put("desisoft.ejb.nameServer1",
  "localhost:2050");
  Context ctx = new InitialContext(env);
  BobKotirovkiHome home = (BobKotirovkiHome)
  PortableRemoteObject.narrow(ctx.lookup("BobKotirovki"),
  BobKotirovkiHome.class);
  BufferedReader input = new BufferedReader(new
  InputStreamReader(System.in));
  for (;;) {
   System.out.print("Stock symbol: ");
   String stock = input.readLine();
  if (stock.equals(""))
   continue;
  if (stock.equalsIgnoreCase("quit"))
   break;
  BobKotirovki bean = home.create(stock);
  try {
```

```
    System.out.println("Price = " + bean.getPrice());
   } catch (Exception ex) {
    ex.printStackTrace();
   }
   }
   } catch (Exception ex) {
   ex.printStackTrace();
   }
   }
}
```

Компилируем программу и запускаем ее.

```
Java ClientKotirovki
```

Перед компиляцией не забудьте указать путь к файлу BobKotirofkiEjb.jar в переменной `CLASSPATH`. После запуска программа приглашает ввести название акции. После ввода названия акции программа выводит котировку. Если название введено неправильно, то появляется сообщение об ошибке. В наших примерах обработка сообщений об ошибках не предусмотрена (рис. 5.32).

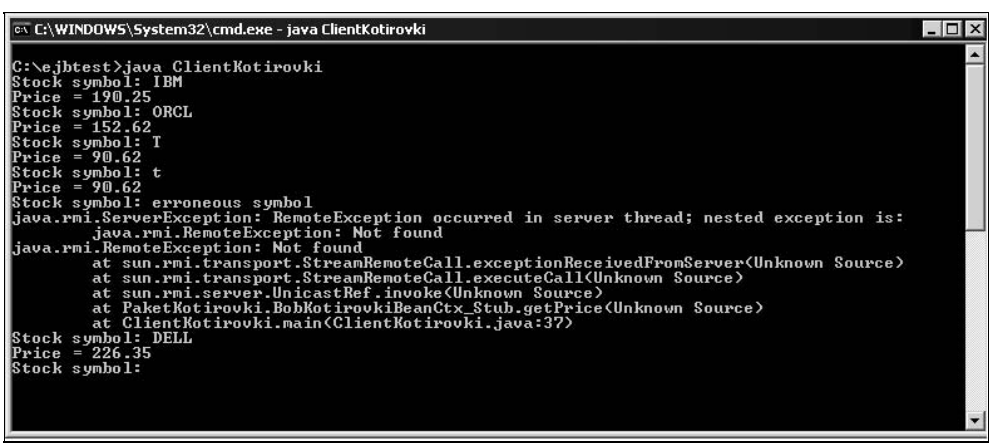


Рис. 5.32. Работа с компонентом EJB-сессии

Если запустить программу без наличия файла StockPrices.txt, то также будет выдано сообщение об ошибках.

К компоненту EJB можно обращаться из нескольких клиентов одновременно. При этом будет создано столько экземпляров компонентов EJB, сколько независимых запросов получено от разных клиентов (рис. 5.33).

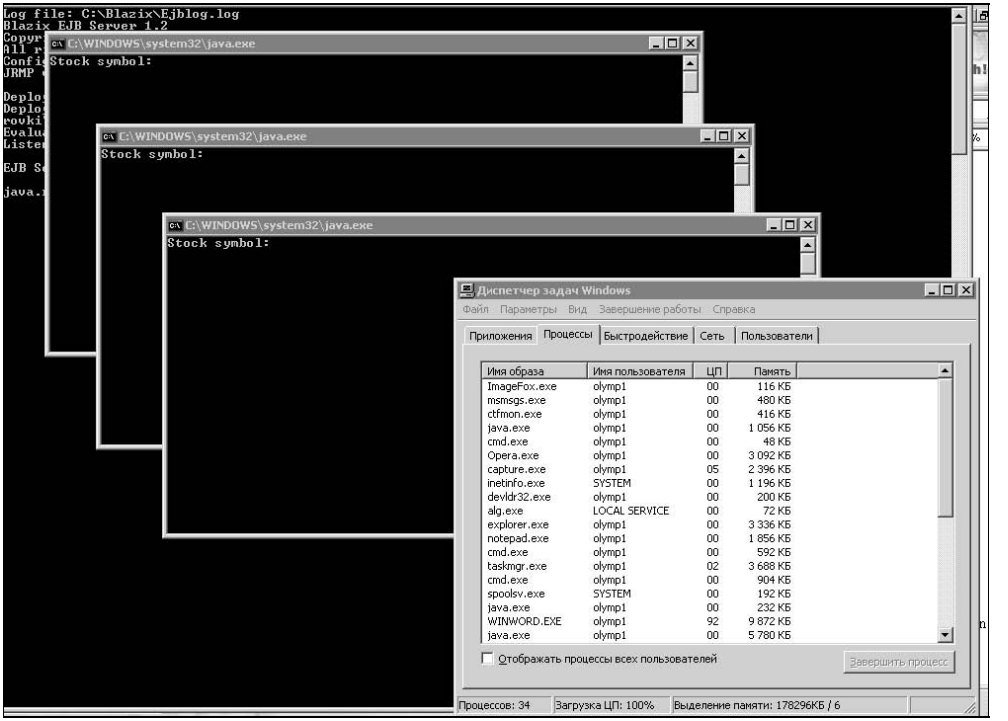


Рис. 5.33. Компонент EJB одновременно работает с несколькими клиентами

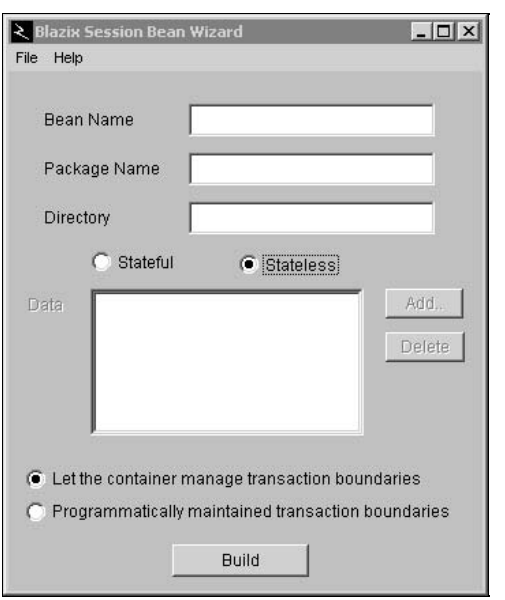


Рис. 5.34. Создание компонента EJB-сессии без поддержки состояния

При создании компонента EJB мы указали `stateful`, т. е. компонент EJB поддерживает состояние при работе с клиентом. При создании компонента EJB для работы с ценами акций можно было создать такой компонент EJB, который бы не поддерживал текущее состояние клиента. При этом в зависимости от сервера компонентов EJB, многочисленное обращение из разных клиентов к такому компоненту EJB может либо приводить к появлению нескольких самостоятельных экземпляров компонентов EJB, либо может оказаться достаточным использование единственного экземпляра компонента EJB. При создании компонента EJB без поддержки состояния в проводнике Blizzard следует отметить пункт `stateless` (рис. 5.34).

Удаленный интерфейс в этом случае будет содержать один метод:

```
public float getPrice(String stock)
  throws java.rmi.RemoteException, java.io.IOException;
```

Файл класса компонент EJB будет содержать переменную

```
int count = 0;
```

Имплементация метода `getPrice` будет иметь следующий вид (листинг 5.17).

Листинг 5.17. Метод getPrice() компонента EJB без поддержки состояния

```
public float getPrice(String stock)
  throws java.rmi.RemoteException,
  java.io.IOException
  {
    count++;
    System.out.println("Count = " + count);
    BufferedReader reader;
    reader = new BufferedReader(
    new FileReader("C:\\StockPrices.txt"));
    String line;
    stock = stock.toLowerCase() + ":";
    while ((line = reader.readLine()) != null) {
      if (line.toLowerCase().startsWith(stock)) {
       line = line.substring(stock.length());
       reader.close();
       return Float.parseFloat(line.trim());
      }
    }
    reader.close();
    throw new java.rmi.RemoteException("Not found");
 }
```

5.2.6. Метки компонентов EJB

В примерах, которые мы использовали ранее, был осуществлен поиск компонента EJB и его создание. Поиск компонента EJB осуществлялся стандартным образом. В компонентах EJB-сессий нет методов поиска компонента EJB, компонент EJB исчезает в момент прекращения работы программы. Однако для получения доступа к компонентам EJB помимо поиска существует возможность использования меток серверных компонентов EJB.

Объекты компонентов EJB и домашние объекты располагают своими метками. Эти метки могут быть затребованы из компонента EJB или из домашнего объекта. Метки не используются во время выполнения программы, они нужны только во время выхода из программы и при повторном ее запуске. Метка существует постоянно. При выходе из программы необходимо иметь возможность сохранения метки. Метку можно записать в файл. Метки создаются на основе класса `java.io.Serializable`, а значит, их действительно можно сохранять в виде файлов.

Код создания метки объекта и получения объекта по его метке достаточно прост. Пример приводится ниже.

```
Handle handle = stock.getHandle();
ObjectOutputStream p = new ObjectOutputStream(
  new FileOutputStream("C:\\FajlMetki.dat"));
p.writeObject(handle);
p.close();
```

Чтение метки осуществимо следующим образом.

```
ObjectInputStream p = new ObjectInputStream(
  new FileInputStream("C:\\FajlMetki.dat"));
Handle handle = (Handle) p.readObject();
BobKotirovki stock = (BobKotirovki) handle.getEJBObject();
```

Необходимо отметить, что полученный по метке объект должен быть приведен к необходимому типу.

5.2.7. Размещение компонентов EJB

При создании серверного компонента EJB автоматически создается файл ejb-jar.xml. Для последнего примера этот файл выглядит так, как показано ниже. Это файл описания размещения компонента EJB (листинг 5.18).

Листинг 5.18. Файл ejb-jar.xml

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

```
<ejb-jar>
  <description>
  ejb-jar.xml file for session bean BobKotirovki
  </description>
  <enterprise-beans>
  <session>
  <description>
  Session bean BobKotirovki
  Framework automatically generated by
  Blizzard, Desiderata Software's Blazix EJB
  Wizard
   </description>
  <ejb-name>
   BobKotirovki
   </ejb-name>
  <home>
   PaketKotirovki.BobKotirovkiHome
   </home>
  <remote>
   PaketKotirovki.BobKotirovki
   </remote>
  <ejb-class>
  PaketKotirovki.BobKotirovkiBean
  </ejb-class>
  <session-type>
  Stateful
   </session-type>
  <transaction-type>
  Container
   </transaction-type>
  </session>
  </enterprise-beans>
</ejb-jar>
```

Корневым XML-элементом в файле размещения является элемент `<ejb-jar>`, в этот элемент вкладывается элемент `<enterprise-beans>`. В одном файле может быть описано несколько компонентов EJB. Элемент `<enterprise-beans>` в качестве своих дочерних элементов может содержать

элементы <session> и <entity>, что соответствует компоненту EJB-сессии и сущности. Далее для каждого элемента <session> и <entity> указывается имя компонента EJB в элементе <ejb-bean>, а также имена классов с указанием пакета, соответствующих домашнему интерфейсу (в элементе <home>) и удаленному интерфейсу (в элементе <remote>). Элемент <ejb-class> содержит класс компонента EJB. Для компонента EJB-сессий задается элемент <session-type>, который содержит тип компонента EJB-сессии.

Описатель размещения для компонента EJB-сущности несколько отличается от описателя компонента EJB-сессии (листинг 5.19).

Листинг 5.19. Описатель компонента EJB-сущности

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar>
  <description>
  ejb-jar.xml file for entity bean NashBob2
  Framework automatically generated by Blizzard,
  Desiderata Software's Blazix EJB Wizard
  </description>
  <enterprise-beans>
  <entity>
   <description>
  Entity bean NashBob2
  </description>
   <ejb-name>
   NashBob2
   </ejb-name>
  <home>
   NashPaket2.NashBob2Home
    </home>
   <remote>
   NashPaket2.NashBob2
   </remote>
   <ejb-class>
   NashPaket2.NashBob2Bean
  </ejb-class>
```

```
   <persistence-type>
   Container
   </persistence-type>
   <prim-key-class>
   java.lang.String
   </prim-key-class>
   <reentrant>
   True
  </reentrant>
   <cmp-field>
   <field-name>
  key
  </field-name>
  </cmp-field>
   <cmp-field>
   <field-name>
  value
  </field-name>
  </cmp-field>
   <primkey-field>
  key
  </primkey-field>
  </entity>
  </enterprise-beans>
</ejb-jar>
```

В описателе компонента EJB-сессии мы находим и другие элементы, которые являются дочерними по отношению к элементу `<entity>`. Элемент `<persistence-type>` в нашем случае указывает на то, что ответственность за постоянное существование компонента EJB возлагается на контейнер серверных компонентов EJB. В случаях, когда необходимо использовать методы `ejbLoad` и `ejbStore`, в этом элементе следует указать `user`.

Элемент `<prim-key-class>` задает тип первичного ключа. Здесь же содержатся элементы `<cmp-field>` — эти поля создаются только для компонентов EJB, постоянное существование которых поддерживается контейнером. CMP — это аббревиатура для Container Managed Persistence. Здесь же есть элемент, описывающий первичный ключ. Детальное описание структуры файла описания размещения достаточно объемно, его можно найти в документации по EJB.

5.2.8. Взаимодействие серверных компонентов EJB друг с другом

В примерах, которые мы рассмотрели ранее, обращение к серверному компоненту EJB осуществлялось из независимого клиента. Серверный компонент EJB располагается на сервере, а клиент может находиться где угодно. Месторасположение компонента EJB (адреса и порта сервера компонентов EJB) указывается в клиентской программе. Существует возможность использования в качестве клиентов для серверных компонентов EJB таких модулей, как серверные страницы Java. Может возникнуть ситуация, когда серверный компонент EJB потребуется вызвать из другого серверного компонента EJB. Технология EJB специально предусматривает такую возможность.

Предположим, что компонент EJB-client является клиентом по отношению к компоненту EJB-1, тогда описатель размещения компонента EJB для клиентского компонента EJB должен содержать в себе ссылку на компонент EJB-1. Ссылка на компонент EJB-сущности описывается в элементе `<ejb-ref>`. Если обратить внимание на дерево вложенных друг в друга элементов в XML-файле, то можно заметить, что элемент `<ejb-ref>` находится на том же самом уровне вложения элементов (для компонента EJB-сущности), что и элементы `<reentrant>`, `<cmp-field>`, `<primkey-field>`, `<env-entry>`, и элемент `<ejb-ref>` расположен после них, но перед элементами `<security-role-ref>` и `<resource-ref>`. Не все перечисленные сейчас элементы будут с необходимостью использованы в файле описателя, но в случае, если эти элементы присутствуют, последовательность их появления должна быть такой, как указано выше. Перечислим элементы, используемые в компоненте EJB-сущности, внутри элемента `<ejb-ref>`, сведя информацию в таблицу (табл. 5.1).

Таблица 5.1. *Дочерние элементы для элемента* `<ejb-ref>` *в описателе компонента EJB- сущности*

Элемент	Описание
`<description>`	Описание компонента EJB, необязательный элемент
`<ejb-ref-name>`	Имя, используемое в ссылке на компонент EJB, должно начинаться с `ejb/`, например `<ejb-ref-name>ejb/stocks</ejb-ref-name>`
`<ejb-ref-type>`	Тип компонента EJB: `Session` или `Entity`
`<home>`	Полное имя домашнего интерфейса компонента EJB, на который есть ссылка
`<remote>`	Полное имя удаленного интерфейса компонента EJB

Таблица 5.1 (окончание)

Элемент	Описание
`<ejb-link>`	Это необязательный элемент, его содержимое зависит от контейнера серверных компонентов EJB. Элемент используется для задания имени компонента EJB, на который есть ссылка, причем имя указывается таким, какое имя соответствует этому компоненту EJB в контейнере

Пример того, как может выглядеть элемент `<ejb-ref>`:

```
<ejb-ref>
<description>Poluchenie ssylki na dannye tsin aktsij</description>
<ejb-ref-name>ejb/BobSessii</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>kotirovki.BobSessiiHome</home>
<remote>kotirovki.BobSessii</remote>
<ejb-link>BobSessii</ejb-link>
</ejb-ref>
```

Если в файл размещения клиентского компонента EJB вставлена ссылка на компонент EJB, к которому будет производиться обращение из этого компонента EJB, то есть в компонент EJB-Client вставлена ссылка на компонент EJB `BobSessii` (который выполняет роль компонента EJB-1), то обращение к компоненту EJB `BobSessii` из клиентского компонента EJB можно осуществить при помощи следующего кода:

```
Context ctx = new InitialContext();
BobSessiiHome home = (BobSessiiHome)
   ctx.lookup("java:comp/env/ejb/kotirovki");
```

В результате получим объект EJB, который затем можно использовать для поиска компонентов EJB, создания компонентов EJB и т. п. Приставка `java:comp/env` используется всегда, когда необходимо произвести поиск с использованием JNDI в пределах доступного окружения.

5.2.9. Базы данных в серверных компонентах EJB

В предыдущем разделе рассматривался поиск компонента EJB в доступном окружении. Аналогичным способом можно работать с базами данных JDBC. Это стандартный метод доступа к базам JDBC, предоставляемый технологией EJB. Это позволяет контейнеру компонентов EJB наиболее эффективно работать, осуществляя доступ к базе данных наиболее продуктивно. Также это предоставляет возможность использования нескольких самостоятельных компонентов EJB при работе с транзакциями. В начале работы с компонентом EJB-сущности в файл инициализации компонентов EJB (файл ejb.ini)

вставлялись сведения об источнике данных `dataSource` с указанием его имени в виде `nashiDannye`. Этот источник данных будет автоматически доступен для JNDI, если записать следующую строчку:

```
java:comp/env/jdbc/NashiDannye
```

Но это лишь общий принцип. Конкретный вид обращения к базе данных будет зависеть от контейнера компонентов EJB. При подобном обращении мы получаем объект типа `javax.sql.DataSource`. Далее мы будем иметь возможность пользоваться методами этого объекта, в том числе методом `getConnection(username, password)`, который возвращает привычный объект типа `java.sql.Connection`.

В качестве примера создадим компонент EJB-сессии, поддерживающий состояние, который будет работать с базой данных, откуда он будет получать информацию о стоимости акций. Создадим таблицу в базе данных Stocks с тремя столбцами, которые назовем `ImyaAktsii`, `TsenaAktsii` и `KolichastvoAktsij`. Таблицу можно создать либо в уже использующейся базе данных, либо создать новую базу данных. Запишем в таблицу несколько строк, при этом одна и та же акция может быть записана несколько раз по разной цене.

Удаленный интерфейс будет содержать следующие дополнительные методы:

```
int getNumber() throws java.rmi.RemoteException,
  java.sql.SQLException,
  javax.naming.NamingException;
boolean buy(float maxPrice, int maxAmount)
  throws java.rmi.RemoteException,
  java.sql.SQLException,
  javax.naming.NamingException;
int getNumPurchased() throws java.rmi.RemoteException;
float getPurchasePrice() throws java.rmi.RemoteException;
```

Эти методы необходимо вставить вручную.

Метод `getPrice` уже присутствует, но мы будем использовать `java.sql.SQLException` и `javax.naming.NamingException`, поэтому необходимо изменить соответствующим образом код в файле удаленного интерфейса. Мы вставим в этот метод запросы к базе данных с использованием инструкции `SQL SELECT` с выводом минимальной цены для данной акции. Метод `getNumber` будет возвращать количество акций, предлагаемых по минимальной цене. Метод `buy()` будет использоваться для покупки указанного количества акций по указанной цене или дешевле. Если указанное количество покупаемых акций более, чем предложено к продаже, то будет куплено столько акций, сколько доступно. Количество купленных акций и стоимость покупки в дальнейшем могут быть определены при помощи методов

getNumPurchased() и getPurchasePrice(), наличием этих методов объясняется то, что данный компонент EJB будет иметь тип stateful.

Имплементация компонента EJB может быть представлена в следующем виде (листинг 5.20).

Листинг 5.20. Файл BobAktsii.java

```
package stocks;
import java.io.*;
import java.sql.*;
import javax.sql.*;

// класс компонента EJB-сессии "BobAktsii"
import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;

public class BobAktsiiBean implements javax.ejb.SessionBean {
  // переменные
  public String stock = null;
  int nPurchased = 0;
  float purchasePrice = 0.0F;
  // контекст сессии
  javax.ejb.SessionContext ejbSessionContext = null;
  //методы get и set
  public String getStock() throws java.rmi.RemoteException
  {
  return stock;
  }
  public void setStock(String stock) throws java.rmi.RemoteException
  {
  this.stock = stock;
  }
  public float getPrice() throws java.rmi.RemoteException,
   java.io.IOException, NamingException, SQLException
  {
  DataSource dataSource = (DataSource) (new
InitialContext()).lookup("java:comp/env/jdbc/NashiDannye");
  Connection con = dataSource.getConnection();
```

```
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT MIN(TsenaAktsii) FROM Stocks
WHERE ImyaAktsii='" + stock + "'");
  if (! rs.next()) {
  stmt.close();
  con.close();
  throw new java.rmi.RemoteException("Not found");
  }
  float result = rs.getFloat(1);
  stmt.close();
  con.close();
  if (result == 0.0)
  throw new java.rmi.RemoteException("Not found");
  return result;
  }
  public int getNumber() throws java.rmi.RemoteException,
NamingException, SQLException
  {
  DataSource dataSource = (DataSource) (new
InitialContext()).lookup("java:comp/env/jdbc/nashiDannye");
  Connection con = dataSource.getConnection();
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT KolichestvoAktsij FROM Stocks
WHERE ImyaAktsii='" + stock + "' AND TsenaAktsii=(SELECT MIN(TsenaAktsii)
FROM Stocks WHERE ImyaAktsii='" + stock + "')");
  if (! rs.next()) {
  stmt.close();
  con.close();
  return 0;
  }
  int result = rs.getInt(1);
  stmt.close();
  con.close();
  return result;
  }
  public boolean buy(float maxPrice, int maxAmount) throws
java.rmi.RemoteException, NamingException, SQLException
  {
  // контейнер отвечает за транзакции
```

```
  DataSource dataSource = (DataSource) (new
InitialContext()).lookup("java:comp/env/jdbc/nashiDannye");
  Connection con = dataSource.getConnection();
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT KolichestvoAktsij,TsenaAktsii
FROM Stocks WHERE TsenaAktsii < " + maxPrice + " AND ImyaAktsii='" +
stock + "' AND TsenaAktsii=(SELECT MIN(TsenaAktsii) FROM Stocks WHERE
ImyaAktsii='" + stock + "')");
  if (! rs.next()) {
  stmt.close();
  con.close();
  return false;
  }
  int number = rs.getInt(1);
  purchasePrice = rs.getFloat(2);
  rs.close();
  if (number > maxAmount) {
  int newNumber = number – maxAmount;
  stmt.executeUpdate("UPDATE Stocks SET KolichestvoAktsij=" + newNumber +
" WHERE TsenaAktsii=" + purchasePrice + " AND ImyaAktsii='" + stock +
"'");
  nPurchased = maxAmount;
  } else {
  stmt.executeUpdate("DELETE FROM Stocks WHERE TsenaAktsii=" +
purchasePrice + " AND ImyaAktsii='" + stock + "'");
  nPurchased = number;
  }
  stmt.close();
  con.close();
  return true;
  }
  public int getNumPurchased() throws java.rmi.RemoteException
  {
  return nPurchased;
  }
  public float getPurchasePrice() throws java.rmi.RemoteException
  {
  return purchasePrice;
  }
```

```
  // метод ejbCreate по умолчанию
  public void ejbCreate(
  String stock
)
  throws javax.ejb.CreateException, java.rmi.RemoteException
  {
  this.stock = stock;
  }
  // прочие методы для ejbCreate

  // прочие методы для компонента EJB-сущности

  public void setSessionContext(javax.ejb.SessionContext
    ejbSessionContext)
  throws RemoteException
  {
  this.ejbSessionContext = ejbSessionContext;
  }
  public void unsetSessionContext()
  throws RemoteException
  {
  this.ejbSessionContext = null;
  }
  public void ejbRemove()
  throws java.rmi.RemoteException, javax.ejb.EJBException
  {
  // методы во время удаления экземпляра компонента EJB-сущности
  }
  public void ejbActivate()
  throws java.rmi.RemoteException
  {
  // восстановление данных
  }
  public void ejbPassivate()
  throws java.rmi.RemoteException
  {
  // сохранение данных
  }
}
```

5.2.10. Транзакции и серверные компоненты EJB

Что такое транзакция? Этот термин часто встречается при работе с базами данных. Этот термин часто используется при проведении различных банковских операций со счетами. Транзакция подразумевает под собой одну или несколько операций, объединяемых в один целый блок, когда выполнение одной операции зависит от результата выполнения другой операции. Все операции, включенные в транзакцию, составляют одно общее целое. Если транзакция проводится успешно, выполняются все условия, то есть все операции, составляющие транзакцию, будут выполнены. Если транзакция отклоняется, то никакие изменения не будут внесены ни в базы данных, ни в любые другие источники. Все операции транзакции, в случае отклонения транзакции, будут отменены. В частности, если после выбора товара по удовлетворяющей покупателя цене начинается выполнение транзакции, то во время проведения этой транзакции необходимо удостовериться в том, что во время проведения транзакции покупки цена на товар не изменится. Для этого все операции, связанные с покупкой, объединяются в одно целое, что и составляет транзакцию.

Предположим, что покупатель собирается купить некоторое количество акций компании MMM, этот покупатель имеет счет в банке и будет осуществлять покупку акций через банк, деньги будут сниматься с его банковского счета автоматически. Предположим, что он хотел бы купить 500 акций за $10. При обработке такого заказа банку потребуется проверять состояние счета покупателя во время покупки, так, чтобы не произошло чрезмерного снятия денег через банк или через банковский автомат во время проведения транзакции, после которого на счете могло бы остаться такое количество денег, которое было бы недостаточно для оплаты купленных акций. Все указанные действия составляют одну транзакцию. Если все проверки проходят успешно, то сделка совершается, баланс счета уменьшается на сумму стоимости купленных акций. Если же что-то не так, например, покупка акций привела бы к отрицательному значению баланса, а счет не позволяет кредитование, то транзакция будет отменена. Состояние баланса не будет изменено, он останется таким, как если бы никакие действия, связанные с транзакцией, не производились вовсе.

При работе с транзакциями в компоненте EJB `BobAktsii` нам потребуется имплементировать метод `buy()`, соответствующий покупке акций. При этом может образоваться множество запросов к базе данных. Например, к базе данных могут обратиться два независимых клиента практически в одно и то же время. При этом один покупатель обратиться к базе данных для определения цены акций, а к моменту покупки, то есть спустя несколько мгновений, содержимое базы данных уже будет изменено в связи с тем, что другой клиент также производит покупку этих акций, и минимальная цена на эти акции уже будет другой. Чтобы избежать ошибок в таких ситуациях, следует

использовать транзакции. Приятным сюрпризом является то, что программист не должен обеспечивать все тонкости работы с транзакциями своими усилиями. Технология серверных компонентов EJB имеет встроенные возможности для работы с транзакциями. Транзакции можно организовать так, что в одной транзакции будут участвовать несколько серверных компонентов EJB и несколько баз данных, связанных с транзакцией ресурсов.

При выполнении транзакции наступает такой момент, когда необходимо (средствами программы) принять решение о том, следует ли принять транзакцию (commit), либо условия не позволяют принять транзакцию, тогда транзакция будет отменена (abort). В этот момент контейнер серверных компонентов EJB связывается с клиентом на предмет отмены транзакции. Если клиент не принимает решение об отмене транзакции (все условия транзакции при этом должны быть выполнены, решение только за клиентом), то транзакция будет принята. Если транзакция отклоняется, то все базы данных, в которые были бы внесены изменения в случае проведения транзакции, будут возвращены в исходное состояние.

Поддержка работы с транзакциями описывается в файле дескриптора размещения компонента EJB (описатель размещения), файле ejb-jar.xml. Для того чтобы указать, что данный метод работает в режиме транзакций, необходимо использовать элемент `<trans-attribute>`. Атрибут транзакций может принимать несколько значений (табл. 5.2).

Таблица 5.2. *Значения атрибута транзакций (элемент* `<trans-attribute>`*)*

Значение	Объяснение
`NotSupported`	В данном методе транзакции не поддерживаются
`Supports`	Этот метод может быть вызван как часть транзакции или как независимый метод
`Required`	Этот метод может быть вызван только как часть транзакции. Если методы вызывается вне транзакции, то автоматически создается новая транзакция
`RequiresNew`	При вызове данного метода создается новая транзакция
`Mandatory`	Этот метод может быть вызван только в составе транзакции
`Never`	Этот метод не может быть использован в составе транзакции

Метод покупки `buy()` должен приводить к тому, что создается новая транзакция, то есть ему будет соответствовать значение `RequiresNew`.

В дескрипторе размещения транзакции описываются внутри элемента `<assembly-descriptor>`. Существует несколько ярлыков описания транзакции. Элемент `<assembly-descriptor>` находится на том же уровне, что и элемент `<enterprise-beans>` (листинг 5.21).

Листинг 5.21. Пример описания транзакции

```
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>BobAktsii</ejb-name>
<method-name>buy</method-name>
</method>
<trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
</assembly-descriptor>
```

После того как описатель размещения компонента EJB `BobAktsii` будет содержать код, приведенный выше, нет нужды волноваться о том, что база данных будет изменена в процессе покупки другим клиентом.

Как отменить транзакцию? Пусть, например, имеется компонент EJB `BankBob`, для возможности осуществления покупки акций удаленный интерфейс будет содержать метод `buyAktsii()`:

```
boolean buyAktsii(String name, float price, int number);
```

Этот метод в описателе размещения должен быть помечен как `RequiresNew`. Этот метод будет обращаться к компоненту EJB `BobAktsii` для проведения покупки. Если в процессе покупки выяснится, что покупка не может быть совершена, то компонент EJB `BankBob` сообщит контейнеру о том, что транзакцию следует отменить. Чтобы информировать контейнер об отмене транзакции, следует использовать интерфейс `SessionSynchronization`.

Вот как это можно сделать. В компонент EJB `BankBob` вставим переменную `(boolean) fail`. В имплементации метода `buyAktsii()` зададим значение переменной, равное `false` (иными словами, транзакция удалась). Затем производим изменение баланса счета, уменьшая его на сумму, уплаченную за покупаемые акции, это делается при помощи SQL-инструкции `UPDATE`, потом вызываем компонент EJB `BobAktsii`, который покупает акции. Если в процессе покупки акций компонентом EJB `BobAktsii` транзакция покупки не проходит, то изменяем значение переменной `fail` на новое значение `true`. После покупки возвращаем все неиспользованные деньги обратно на счет заказчика. В описание компонента EJB `BankBob` необходимо добавить `implements javax.ejb.SessionSynchronization`.

Компонент EJB будет содержать следующие методы, основное внимание уделим методу `beforeCompletion()`:

```
public void afterBegin() throws javax.ejb.EJBException,
   java.rmi.RemoteException
```

```
{
}
public void beforeCompletion() throws javax.ejb.EJBException,
   java.rmi.RemoteException
{
  if (fail) {
  ejbSessionContext.setRollbackOnly();
  fail = false;
  }
}
public void afterCompletion() throws javax.ejb.EJBException,
   java.rmi.RemoteException
{
}
```

В методе `beforeCompletion()` в случае отмены транзакции мы вызываем метод контекста компонента EJB `setRollbackOnly()`, который приводит к тому, что никакие изменения в базах данных не будет произведены, то есть базы данных останутся такими, как до транзакции.

Конечно, транзакции можно организовать и вручную, при этом следует описать тип транзакции в элементе `<transaction-type>` иным образом, а именно, вместо `Container` написать `Bean`. Для этого случая существует интерфейс `javax.transaction.UserTransaction`.

В заключение скажем несколько слов о компонентах EJB-сущности, которые поддерживают свое долговременное существование собственными средствами. Такая ситуация может возникнуть тогда, когда требуется создать, например, особую конфигурацию при работе с базами данных, при работе с центральным сервером, которому передаются нестандартные данные, при работе с базами данных, которые не поддерживают JDBC. При этом создаются компоненты EJB, которые имеют тип поддержания своего жизненного цикла, называемый Bean-Managed Persistence (поддержка работы с базой данных средствами компонента EJB).

Сервер Bllizard создает следующие функции при работе с компонентами EJB-сущности, самостоятельно поддерживающими свой жизненный цикл: `ejbCreate()`, `ejbPostCreate()`, `ejbLoad()`, `ejbStore()`. Однако эти функции пусты, их необходимо имплементировать. Эти функции вызываются на разных этапах жизненного цикла компонента EJB. Функция `ejbCreate` (листинг 5.22) вызывается в ответ на запрос домашнего интерфейса о создании компонента EJB. Функция `ejbPostCreate` вызывается сразу после того, как экземпляр компонента EJB был создан. Функция `ejbRemove` вызывается тогда, когда необходимо удалить компонент EJB. Особое значение для поддержания жизненного цикла компонента EJB типа Bean-Managed Persistence

имеют функции `ejbLoad` (листинг 5.23) и `ejbStore` (листинг 5.24). Эти функции ответственны за чтение и сохранение данных, хранимых в источниках данных (не обязательно в базах данных). Кроме этого, в компонентах EJB с самостоятельной поддержкой жизненного цикла необходимо определить методы нахождения компонента EJB, соответствующие описанным в домашнем интерфейсе методам нахождения компонента EJB, при этом может потребоваться метод нахождения компонента EJB по первичному ключу (или набору ключей), список этих методов обязательно содержит по крайней мере один метод, а именно метод `findByPrimaryKey`.

Листинг 5.22. Метод `ejbCreate`

```
public java.lang.String ejbCreate(String key, String value)
  throws javax.ejb.CreateException, java.rmi.RemoteException
{
 this.key = key;
 this.value = value;
 boolean duplicateKey = false;
 try {
  InitialContext ctx = new InitialContext();
  DataSource ds = (DataSource)
  ctx.lookup("java:comp/env/jdbc/NashiDannye");
  Connection conn = ds.getConnection();
  PreparedStatement stmt = conn.prepareStatement(
  "SELECT * FROM PropsTable WHERE KEY=?");
  stmt.setString(1, key);
  ResultSet rs = stmt.executeQuery();
  duplicateKey = rs.next();
  rs.close();
  stmt.close();
  if (! duplicateKey) {
    stmt = conn.prepareStatement(
    "INSERT INTO NashaTablitsa (KEY,\"VALUE\") VALUES (?,?)");
    stmt.setString(1, key);
    stmt.setString(2, value);
    stmt.executeUpdate();
    stmt.close();
   }
  conn.close();
  }
```

```
  catch (Exception ex) {
    throw new java.rmi.RemoteException("ejbCreate Error", ex);
  }
 if (duplicateKey)
   throw new javax.ejb.DuplicateKeyException();
 return null;
}
```

Листинг 5.23. Метод `ejbLoad`

```
public void ejbLoad()
 throws java.rmi.RemoteException
{
  boolean found = false;
  try {
  InitialContext ctx = new InitialContext();
  DataSource ds = (DataSource)
  ctx.lookup("java:comp/env/jdbc/NashiDannye");
  Connection conn = ds.getConnection();
  PreparedStatement stmt = conn.prepareStatement(
  "SELECT \"VALUE\" FROM NashaTablitsa WHERE KEY=?");
  stmt.setString(1, key);
  ResultSet rs = stmt.executeQuery();
  if (rs.next()) {
    found = true;
    value = rs.getString(1);
  }
  rs.close();
  stmt.close();
  conn.close();
  }
  catch (Exception ex) {
    throw new java.rmi.RemoteException("ejbLoad Error", ex);
  }
  if (! found)
   throw new java.rmi.RemoteException("Bean not found");
 }
```

Имплементация метода осуществляется похожим способом.

Листинг 5.24. Фрагмент метода `ejbStore`

```
PreparedStatement stmt = conn.prepareStatement(
"UPDATE PropsTable SET \"VALUE\"=? WHERE KEY=?");
stmt.setString(1, value);
stmt.setString(2, key);
if (stmt.executeUpdate() > 0)
found = true;
```

5.2.11. Безопасность серверных компонентов EJB

Вопросы безопасности серверных компонентов EJB решаются в описателе, в файле-дескрипторе размещения. Параметры безопасности легко могут быть изменены, при этом не возникнет необходимость внесения изменений в программный код. Безопасность компонентов EJB задается путем указания ролей `roles`. Роли задают разные уровни доступа к компонентам EJB. Администратор имеет один уровень доступа, а клиент — другой. Могут существовать такие методы, которые, скажем, доступны администратору, но не доступны клиенту. И если пользователь меняет роль клиента на роль администратора, то ему становятся доступны методы, предназначенные для администратора.

Доступ к любому методу может быть ограничен с указанием ролей. Ограничения описываются в элементе `<assembly-descriptor>`. В начале необходимо перечислить все используемые роли как таковые. Затем для каждого метода указывается возможность доступа путем перечисления ролей из ранее созданного перечня ролей. Продемонстрируем это на примере (листинг 5.25).

Листинг 5.25. Задание ролей

```
<assembly-descriptor>
<security-role>
<role-name>administrator</role-name>
</security-role>
<security-role>
<role-name>client</role-name>
</security-role>
<method-permission>
<role-name>administrator</role-name>
<role-name>client</role-name>
<method>
<ejb-name>BobAktsii</ejb-name>
<method-name>buy</method-name>
```

```
</method>
<method>
<ejb-name>BobAktsii</ejb-name>
<method-name>getPrice</method-name>
</method>
</method-permission>
</assembly-descriptor>
```

Внутри каждого элемента `<method-permission>` может быть размещено несколько ролей, за которыми указываются один или несколько методов, которые будут доступны для указанных ролей. При указании методов разрешается использование символа `*`. Детальные объяснения описания уровней доступа к методам можно найти в документации по EJB.

5.2.12. Резюме

Подведем итоги. Серверные компоненты EJB — составная часть спецификации Java 2 Enterprise Edition (J2EE). Компоненты EJB предоставляют готовую архитектуру распределенных компонентов, основанных на использовании объектов. Компонент EJB применяется в качестве представителя "бизнес-объекта" или же бывает связан с таблицей в базе данных. Часть разрабатываемого приложения представляет собой, как правило, набор EJB для работы с базами данных, в то время как другая часть — это набор EJB, не образующих постоянных персистентных соединений, но реализующих бизнес-логику приложения.

Архитектура EJB

Базовая архитектура EJB представлена набором, состоящим из сервера EJB, контейнеров EJB, EJB-клиентов, компонентов EJB, используется также сервис имен, например, JNDI. Компонент EJB обслуживается окружением, создаваемым контейнером EJB, контейнер EJB располагается на EJB-сервере (рис. 5.35). Клиент не может вызвать методы компонента EJB непосредственно на самом серверном компоненте EJB, контейнер служит своеобразным посредником между компонентом EJB и клиентом (табл. 5.3).

Таблица 5.3. *Функции элементов архитектуры EJB*

Элемент архитектуры	Назначение
Сервер EJB	EJB-сервер — это набор программных средств. В чем-то он сродни объектному брокеру в архитектуре CORBA. Сервер предоставляет клиенту возможность взаимодействовать с компонентами EJB. Сервер предоставляет сервис имен

Таблица 5.3 (окончание)

Элемент архитектуры	Назначение
Контейнер EJB	Располагается на сервере EJB как его составной элемент. Предоставляет средства для обращения к методам компонентов EJB. Клиент EJB посылает запрос вызова метода компонента EJB контейнеру компонентов EJB. Контейнер EJB вызывает соответствующий метод компонента EJB. Контейнер обеспечивает безопасность, постоянную работоспособность объектов, управление ресурсами, поддержку работы с транзакциями
Клиенты EJB	Посредством сервиса имен EJB клиенты осуществляют поиск компонентов EJB. После обнаружения компонента EJB клиент посылает сообщение контейнеру компонентов EJB, контейнер определяет доступные для клиента методы
Компоненты EJB	Существует три типа серверных компонентов EJB: компоненты EJB-сессий, компоненты EJB-сущности, компоненты EJB, основанные на сообщениях. Компоненты EJB работают в окружении, предоставляемом контейнером серверных компонентов EJB
Сервис имен	Сервис имен и директориев предоставляет возможность обнаружения зарегистрированных объектов. При помощи сервиса имен можно найти соответствие между именем, например, компонента EJB, и домашним интерфейсом серверного компонента EJB

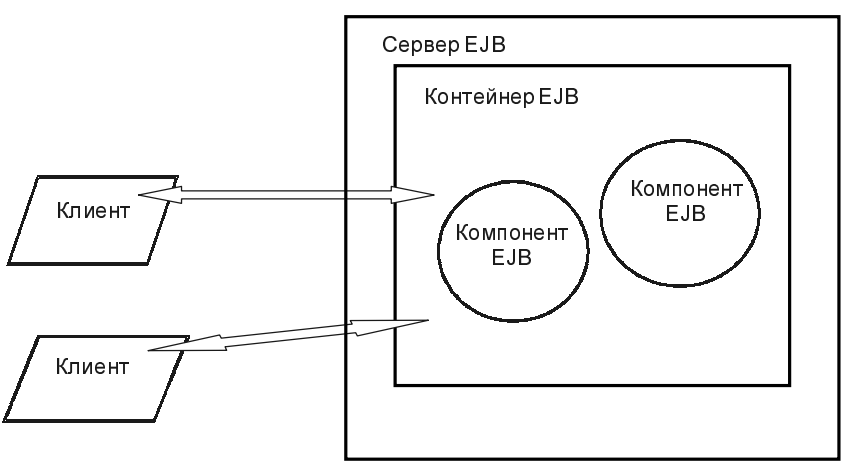


Рис. 5.35. Базовая архитектура EJB

Типы компонентов EJB

Компонент EJB-сессии — Session Bean

Компоненты EJB-сессии используются для выполнения бизнес-логики, которая не требует создания постоянных соединений. Часто такие компоненты EJB ассоциируются с тем или иным вполне конкретным клиентом. При остановке работы клиента, останавливается и работающий серверный компонент EJB-сессии. Компоненты EJB-сессии могут иметь доступ и модифицировать данные, однако для этого более приспособлены компоненты EJB-сущности.

Компонент EJB-сущности — Entity Bean

Как правило, компонент EJB-сущности используется для того, чтобы осуществлять доступ к базам данных. Компонент EJB-сущности может обслуживать несколько клиентов, при прекращении работы клиента, компонент EJB-сущности продолжает работать. Компонент EJB-сущности создает запросы к базе данных — это наиболее частое его назначение и представляет таблицу в базе данных. Свойства компонента EJB-сущности соответствуют различным столбцам этой таблицы. При создании экземпляра компонента EJB его можно представлять как временный образ строки таблицы, с которой ассоциируется компонент EJB.

Компонент EJB, основанный на сообщениях — Message-driven Bean

Этот тип серверных компонентов EJB появился в спецификации EJB 2.0. Компонент EJB этого типа используется для выполнения фрагмента кода при получении сообщения от клиента.

Пассивация и активация компонентов EJB

Контейнер компонентов EJB имеет возможность манипулировать занимаемой памятью путем пассивации и активации компонентов EJB, которые обслуживаются этим контейнером. При пассивации состояние компонента EJB записывается в постоянное хранилище данных, например, в базу данных или при необходимости в метку компонента EJB, что, однако, не есть пассивация как таковая, а просто дополнительный способ сохранения компонента EJB для последующего возобновленного его использования, затем компонент EJB удаляется из памяти. Активация противоположна пассивации: состояние компонента EJB считывается, и компонент EJB загружается в память. Активация и пассивация как таковые применяются к компонентам EJB-сущностей с поддержкой сессий. Именно состояние сессии записывается при пассивации, для этого существует специальный механизм. Перед пассивацией контейнер вызывает метод `ejbPassivate()`, а после активации — метод `ejbActivate()`, эти методы используются для того, чтобы произвести все необходимые настройки перед пассивацией или после активации.

Постоянные соединения

Компонент EJB-сессий без поддержки состояний и компонент EJB-сущности являются постоянными, перманентными. Контейнер поддерживает их работу и постоянное существование. Для сохранения таких компонентов EJB используются метки компонентов EJB — объекты типа `javax.ejb.Handle`. Для их создания применяется метод компонента EJB `getHandle()`. Восстановление компонента EJB осуществляется с применением метода `getEJBObject()` объекта `Handle`.

Основные компоненты компонентов EJB

Серверные компоненты EJB состоят из четырех компонентов: домашнего интерфейса, удаленного интерфейса, класса компонента EJB и описателя размещения компонента EJB (рис. 5.36).

Домашний интерфейс определяет методы создания, нахождения и уничтожения экземпляра компонента EJB.

Удаленный интерфейс содержит методы, определяющие бизнес-логику компонента EJB. Методы удаленного интерфейса должны быть имплементированы в классе компонента EJB.

Класс компонента EJB содержит логику работы самого компонента EJB. В нем должны быть имплементированы методы удаленного интерфейса компонента EJB и методы, соответствующие методам домашнего интерфейса для нахождения, создания и удаления экземпляра компонента EJB.

Описатель размещения — это XML-файл, который всегда носит имя `ejb-jar.xml`. Дескриптор размещения содержит описание компонента EJB, используемое контейнером серверных компонентов EJB. Каждый компонент EJB может использовать отдельный файл дескриптора размещения, но, как правило, EJB размещается в комплекте компонентов EJB, образующих приложение и упакованных в файл `EAR`.

Принятые названия и имена

Если компонент EJB назван `Primer`, то удаленный интерфейс компонента EJB будет описан в файле `Primer.java`, домашний интерфейс в файле `PrimerHome.java`, класс компонента EJB описан в файле `PrimerBean.java` или `PrimerEJB.java`. Имя дескриптора размещения всегда будет `ejb-jar.xml` (табл. 5.4).

Таблица 5.4. *Назначение элементов компонента EJB*

Элемент компонента EJB	Описание
Домашний интерфейс	Методы создания экземпляра компонента EJB, нахождения существующего компонента EJB, уничтожения экземпляра компонента EJB

Таблица 5.4 (окончание)

Элемент компонента EJB	Описание
Удаленный интерфейс	Содержит описания методов бизнес-логики компонента EJB
Класс компонента EJB	Логика компонента EJB как такового, имплементация методов удаленного интерфейса
Описатель размещения	Файл XML, содержащий информацию о размещении компонента EJB и пр.

Размещение компонента EJB

Для размещения компонента EJB на сервере необходимо обладать удаленным интерфейсом, домашним интерфейсом, классом компонента EJB и дескриптором размещения. Интерфейсы и класс компонента EJB размещаются в отдельном каталоге, образуя самостоятельный пакет, а дескриптор размещения помещается в каталог META-INF. Пакет с интерфейсами и классом компонента EJB, а также дескриптор размещения META-INF/ejb-jar.inf затем упаковываются в JAR-файл.

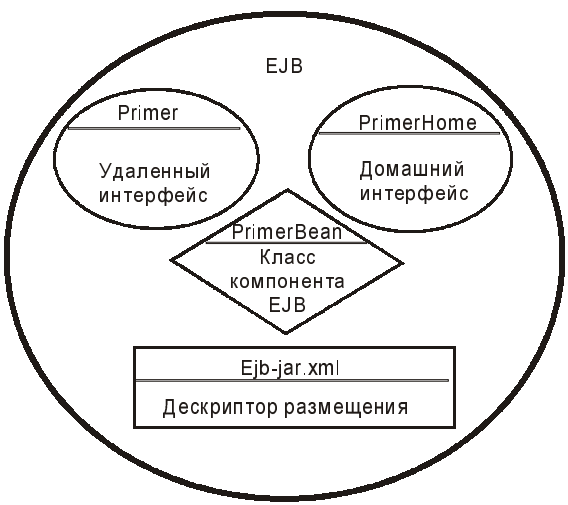


Рис. 5.36. Структура компонента EJB

5.2.13. Принципы работы EJB

Взаимодействие клиента с компонентом EJB

Когда компонент EJB размещается с помощью контейнера EJB, то контейнер запрашивает имя компонента EJB и регистрирует домашний интерфейс.

При запросе клиентом имени домашнего интерфейса, компонент EJB получает стаб домашнего интерфейса. *Стаб* — это часть технологии вызова удаленных процедур (RMI, Remote Method Invocation). Клиент использует стабы для удаленного обращения к доступным методам объекта, который располагается в месте, удаленном от клиента. Стаб получает значение, возвращаемое методом от удаленного объекта и передает его клиенту (если значение возвращается).

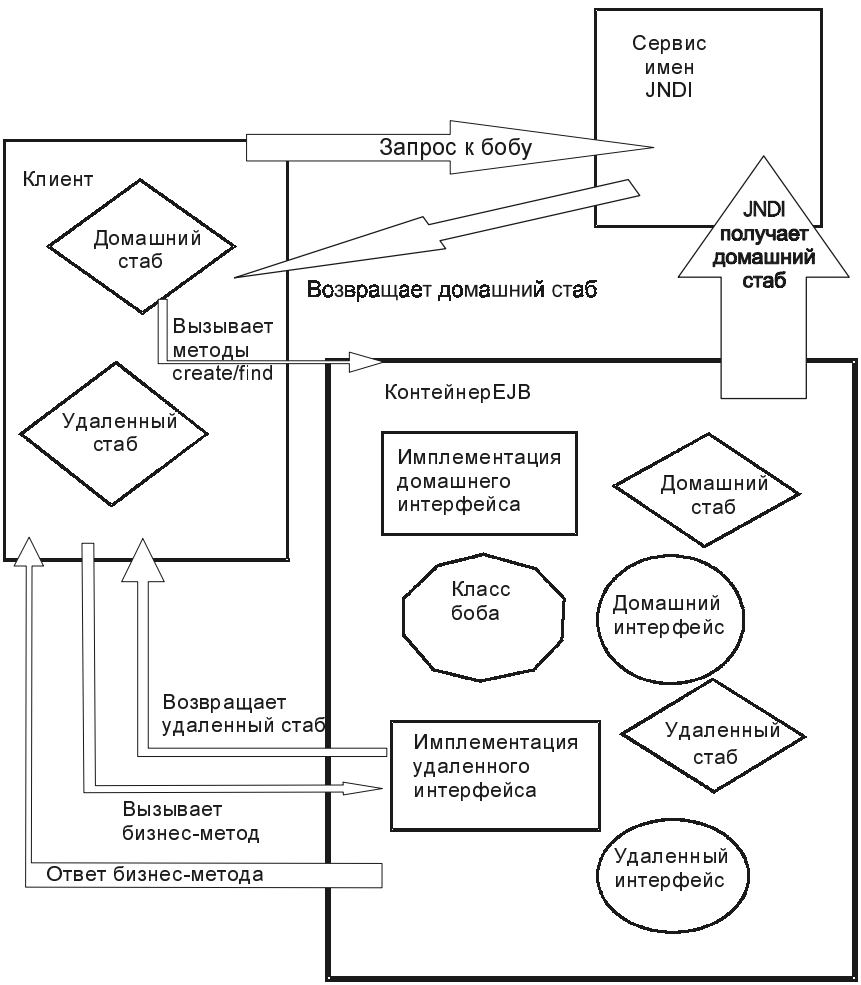


Рис. 5.37. Взаимодействие клиента с компонентом EJB

Стаб домашнего интерфейса не осуществляет никаких взаимодействий с домашним интерфейсом. Стаб взаимодействует с имплементацией домашнего интерфейса, которая находится в контейнере. Клиент использует стаб

для вызова методов создания, нахождения и удаления экземпляра компонента EJB. Когда вызов метода создания компонента EJB (или нахождения компонента EJB) выполняется удачно, тогда контейнер создает экземпляр серверного компонента EJB (экземпляр класса серверного компонента EJB). Имплементация домашнего интерфейса при этом возвращает стаб удаленного интерфейса, который соответствует имплементации удаленного интерфейса. Клиент может использовать удаленный стаб для отправки запросов к серверному компоненту EJB (рис. 5.37).

Взаимодействие с компонентами EJB

Клиент должен произвести следующие действия.

- Задание службы имен

```
System.setProperty("java.naming.factory.initial",
"org.jnp.interfaces.NamingContextFactory");
System.setProperty("java.naming.provider.url", "localhost:1099");
```

- Получение контекста службы имен (в блоке `try/catch`)

```
InitialContext jndiContext = new InitialContext();
```

- Получение ссылки на домашний стаб (в блоке `try/catch`)

```
Object reference = jndiContext.lookup("EJBName");
```

- Получение домашнего стаба (в блоке `try/catch`)

```
com.masslight.ExampleEJB.Home home = (com.masslight.ExampleEJB.Home)
  PortableRemoteObject.narrow (reference,
com.masslight.ExampleEJB.Home.class);
```

- Получение удаленного стаба (в блоке `try/catch`)

```
com.masslight.ExampleEJB.Remote remote = home.create();
```

- Вызов бизнес-метода в экземпляре компонента EJB (в блоке `try/catch`)

```
remote.businessMethodOne(...);
```

5.2.14. Дескриптор размещения EJB

Структура дескриптора размещения описана в файле описания типа документа: **http://java.sun.com/dtd/ejb-jar_2_0.dtd**, как это указывается в начале дескриптора размещения:

```
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

Это обязательные ярлыки файла дескриптора размещения.

Ярлыки файла размещения ejb-jar.xml

Дескриптор размещения может описывать более одного компонента EJB в одном файле. Ярлык `<session>` используется для описания компонента EJB-сессии, ярлык `<entity>` описывает компонент EJB-сущности. Ярлык `<display-name>` задает имя, которое будет отображено при использовании утилит отображения компонентов EJB.

```
<ejb-jar>
  <display-name>ExampleEJBs</display-name>
  <enterprise-beans>
  ...
  <session>
  ...
  </session>
  ...
  <entity>
  ...
  </entity>
  ...
  </enterprise-beans>
</ejb-jar>
```

Имя компонента EJB

Ярлыки, приведенные ниже, используются для описания компонентов EJB-сессий и компонентов EJB-сущностей. `<display-name>` задает имя, показывается утилитой отображения компонентов EJB. Имя `<ejb-name>` будет зарегистрировано службой имен. Это имя будет использоваться клиентом для нахождения компонента EJB.

```
<display-name>Hello</display-name>
<ejb-name>Hello</ejb-name>
```

Классы компонента EJB

Следующие ярлыки должны быть расположены внутри тела элемента описания компонента EJB-сессии или компонента EJB-сущности. Здесь задаются имена файлов классов, в которых описаны домашний интерфейс, удаленный интерфейс, класс компонента EJB. Все три имени должны быть указаны полностью.

```
<home>com.masslight.HelloEJBClasses.HelloHome</home>
<remote>com.masslight.HelloEJBClasses.Hello</remote>
<ejb-class>com.masslight.HelloEJBClasses.HelloBean</ejb-class>
```

Ярлыки компонента EJB-сессий

Эти ярлыки обязаны быть расположены внутри тела описания размещения компонента EJB-сессии. Ярлык `<session-type>` задает тип компонента EJB-

сессии и может описывать компонент EJB с поддержкой состояния `Stateful` или компонент EJB без поддержки состояния `Stateless`.

```
<session-type>Stateless</session-type>
<transaction-type>Bean</transaction-type>
```

Ярлыки компонента EJB-сущности

В описании компонента EJB-сущности задается тип поддержания постоянного существования компонента EJB (средствами компонента EJB или средствами контейнера) `<persistence-type>`. Ярлык `<prim-key-class>` — полное имя класса `Java`, соответствующее типу первичного ключа компонента EJB.

```
<persistence-type>Bean</persistence-type>
<prim-key-class>java.lang.Integer</prim-key-class>
<reentrant>False</reentrant>
```

5.2.15. Компоненты EJB-сессий

Компоненты EJB с поддержкой состояния

Компоненты EJB-сессий могут быть компонентами EJB с поддержкой состояний или без поддержки состояний. Компоненты EJB с поддержкой состояния должны быть определены как таковые, например, как компонент EJB типа `Stateful` в описателе размещения. Компоненты EJB без поддержки состояния не имеют внутреннего состояния, поэтому для них не требуется проведение процедуры пассивации и активации, они попросту не нужны, так как сохранять нечего. Компонент EJB без поддержки сессии может работать с несколькими клиентами, в то время как компонент EJB с поддержкой сессии работает только с одним клиентом (табл. 5.5).

Таблица 5.5. *Различия компонентов EJB с поддержкой состояния и без поддержки состояния*

Компонент EJB-сессии без поддержки состояния	Компонент EJB-сессии с поддержкой состояния
В дескрипторе размещения:	В дескрипторе размещения:
Stateless Session Bean	Stateful Session Bean
Много клиентов одновременно	Один клиент на экземпляр компонента EJB

Имплементация компонента EJB-сессий

Компонент EJB-сессии требует наличия домашнего интерфейса, удаленного интерфейса, класса компонента EJB и XML-дескриптора размещения. Все четыре компонента строятся по определенным правилам. Домашний интерфейс является наследником интерфейса `javax.ejb.EJBHome`, удаленный

интерфейс расширяет интерфейс `javax.ejb.EJBObject`. Класс компонента EJB имплементирует интерфейс `javax.ejb.SessionBean` (листинг 5.26).

Листинг 5.26. Имплементация интерфейса `javax.ejb.EJBObject`

```
public interface SessionBean extends EnterpriseBean {
public abstract void ejbActivate() throws java.rmi.RemoteException;
public abstract void ejbPasivate() throws java.rmi.RemoteException;
public abstract void ejbRemove() throws java.rmi.RemoteException;
public abstract void setSessionContext(SessionContext ctx) throws
java.rmi.RemoteException;
}
```

В табл. 5.6.–5.8 приведены детали, необходимые при работе с удаленным интерфейсом, домашним интерфейсом, классом компонента EJB для компонента EJB-сессии с поддержкой состояний и для компонента EJB-сессии без поддержки состояний.

Таблица 5.6. *Домашний интерфейс*

`Home interface extends javax.ejb.EJBHome`		
Метод	**Возвращаемое значение**	**Описание**
`Create() throws java.rmi.RemoteException, javax.ejb.CreateException`	стаб удаленного интерфейса	При получении контейнером запроса от клиента, контейнер создает экземпляр компонента EJB-сессии. Метод `create()` может содержать параметры. Метод соответствует имплементации метода `ejbCreate()` в классе компонента EJB. Должен существовать, по крайней мере, один метод `create()`

Таблица 5.7. *Удаленный интерфейс*

`Remote interface extends javax.ejb.EJBObject`		
Метод	**Возвращаемое значение**	**Описание**
`ExampleMethod (int GNPofFranceInBillionsOfPounds);`	`Int`	Все определяемые методы удаленного интерфейса зависят от конкретных методов данного компонента EJB

Таблица 5.7 *(окончание)*

Remote interface extends javax.ejb.EJBObject		
Метод	**Возвращаемое значение**	**Описание**
`ExampleMethod (int GNPofFranceInBillionsOfPounds);`	`Int`	Каждый объявленный в интерфейсе метод должен быть имплементирован в компоненте EJB. В таблице приводится только пример, метод `PrimerMetod()`

Таблица 5.8. *Класс компонента EJB*

Bean class implements javax.ejb.SessionBean		
Метод	**Возвращаемое значение**	**Описание**
`EjbCreate() throws javax.ejb.CreateException`	Void	Соответствует методу `create` домашнего интерфейса. Контейнер вызывает метод `ejbCreate` после того, как клиент вызовет метод `create`. Обязателен
`EjbRemove()`	Void	Этот метод вызывается контейнером для предупреждения экземпляра компонента EJB о том, что его собираются уничтожить. Экземпляр компонента EJB при этом производит зачистку и уничтожается. Обязателен
`EjbActivate()`	Void	Контейнер вызывает этот метод для сообщения экземпляру компонента EJB о его активации. При этом экземпляр компонента EJB становится "активным", восстанавливает свое состояние (сетевые связи, метки) и становится годным к использованию. Обязателен

Таблица 5.8 (окончание)

Bean class implements javax.ejb.SessionBean		
Метод	**Возвращаемое значение**	**Описание**
`EjbPassivate()`	Void	Контейнер вызывает этот метод для сообщения экземпляру компонента EJB о пассивации, при этом компонент EJB освобождает ресурсы, например, сетевые связи, метки. Обязателен
`SetSessionContext()`	Void	Контейнер вызывает этот метод после того, как компонент EJB был создан, но перед тем, как будут вызваны методы `ejbCreate`. Компонент EJB отвечает за сохранность объекта типа `javax.ejb.SessionContext`. Обязателен
`PrimerMethod(int Chtolinbo)`	Int	Метод — пример прочих методов, используемых в классе компонента EJB, если такие методы описаны в удаленном интерфейсе. Необязателен

5.2.16. Компоненты EJB-сущности

Поддержка транзакций: средствами контейнера и средствами компонента EJB

Компонент EJB-сущности может поддерживать транзакции. Транзакции могут поддерживаться либо средствами компонента EJB, либо средствами контейнера. При поддержании транзакций средствами контейнера задача поддержания взаимодействий с базой данных, входящих в транзакцию, возлагается на контейнер. При поддержке транзакций средствами компонента EJB, класс компонента EJB должен содержать необходимые инструкции для поддержания транзакций. Метод поддержания транзакций задается в дескрипторе размещения.

Имплементация компонента EJB-сущности

Компонент EJB-сущности состоит из домашнего интерфейса, удаленного интерфейса, класса компонента EJB и XML-файла дескриптора размещения. Все четыре компонента строятся по определенным правилам. Домашний интерфейс является наследником интерфейса `javax.ejb.EJBHome`, удаленный интерфейс расширяет интерфейс `javax.ejb.EJBObject`. Класс компонента EJB имплементирует интерфейс `javax.ejb.EntityBean`. Компонент EJB-сущности предназначен для работы с базами данных. В нем всегда есть свойство, которое соответствует столбцу в таблице базы данных. В табл. 5.9 приводятся свойства компонентов EJB-сущности.

Таблица 5.9. *Свойства компонента EJB-сущности*

SQL-Фрагменты	Соответствующие свойства компонента EJB
`CREATE TABLE DEMO (`	нет соответствия
`ID integer(8) not null,`	`Private long id;`
`STOBETS_ODIN varchar(32) not null,`	`Private String stolbetsOdin;`
`primary key(ID));`	нет соответствия

В классе компонента EJB должны быть имплементированы все методы, которые объявлены в домашнем интерфейсе, удаленном интерфейсе и интерфейсе `javax.ejb.EntityBean`. Некоторые имплементируемые методы имеют несовпадающие с методами интерфейса имена, это относится к методам домашнего интерфейса. Например, метод `create` (параметры) домашнего интерфейса соответствует методу `ejbCreate` (те же параметры) в классе компонента EJB. Ниже приведены методы, которые используются в классе компонента EJB.

```
public interface EnterpriseBean extends java.io.Serializable {}
public interface EntityBean extends EnterpriseBean {
public abstract void ejbActivate() throws java.rmi.RemoteException;
public abstract void ejbLoad() throws java.rmi.RemoteException;
public abstract void ejbPassivate() throws java.rmi.RemoteException;
public abstract void ejbRemove() throws java.rmi.RemoteException;
public abstract void ejbStore() throws java.rmi.RemoteException;
public abstract void setEntityContext(EntityContext ctx) throws
java.rmi.RemoteException;
public abstract void unsetEntityContext() throws
java.rmi.RemoteException;
}
```

Ниже в табл. 5.10—5.12 приводится описание элементов удаленного и домашнего интерфейсов, класса компонента EJB-сущности с поддержкой транзакций средствами компонента EJB и средствами контейнера.

Таблица 5.10. *Удаленный интерфейс* `remote interface extends javax.ejb.EJBObject`

Метод	Возвращаемое значение	Описание
`PrimerMethod(int Chtolibo);`	`Int`	Все описываемые здесь методы имеют отношение к конкретному компоненту EJB. Каждый описанный метод должен быть имплементирован в классе компонента EJB. Необязателен

Таблица 5.11. *Домашний интерфейс* `home interface extends javax.ejb.EJBHome`

Метод	Возвращаемое значение	Описание
`create(...) throws java.rmi.RemoteException, javax.ejb.CreateException`	Экземпляр стаба, связанный с удаленным интерфейсом экземпляра класса компонента EJB	Создает новую строку в таблице, если ряд с указанным первичным ключом еще не существует. При получении запроса контейнер создает компонент EJB-сущности. Метод `create` содержит параметры. Может быть создано несколько методов `create()`. Обязателен
`FindByPrimaryKey (primary_key_type primary_key) throws java.rmi.RemoteException, javax.ejb.FinderException`	Экземпляр стаба, связанный с удаленным интерфейсом экземпляра класса компонента EJB с заданным первичным ключом `primary_key`	Возвращает ряд в таблице с первичным ключом `primary_key`. Компонент EJB должен содержать метод `ejbFindByPrimaryKey`. Обязателен
`public void remove() throws FinderException, RemoteException`	`Void`	Удаляет текущий ряд из таблицы. Должен соответствовать методу компонента EJB `EjbRemove`. Необязателен
`findAll()` — или подобное имя	Коллекция всех первичных ключей таблицы	Возвращает все первичные ключи таблицы. Необязателен

Таблица 5.12. *Класс компонента EJB* `Bean class implements javax.ejb.EntityBean`

Метод	Возвращаемое значение	Описание
`EjbCreate(...) throws javax.ejb.CreateException`	`Primary_key_type`	Вставляет строку в таблицу. Свойства компонента EJB задаются равными значениям атрибутов рядов (названиям рядов) в таблице. Должен существовать хотя бы один метод `ejbCreate` (или более). Обязателен
`EjbPostCreate(...)`	`Void`	Сообщает компоненту EJB, что контейнер прекратил вызов метода `ejbCreate`. Обязателен
`EjbFindByPrimaryKey (primary_key_type primary_key) throws javax.ejb.FinderException`	`Primary_key_type`	Возвращает ряд из таблицы базы данных по первичному ключу `primary_key`. Если ряд не существует, то образуется `FinderException`. Обязателен
`EjbFindAll()`	`Collection`	Возвращает все первичные ключи в таблице. Соответствует методу домашнего интерфейса `findAll`. Необязателен
`ejbLoad()`	`Void`	Загружает ряд в компонент EJB. Обязателен
`EjbStore()`	`Void`	Обновляет содержимое таблицы базы данных, записывая ряд в базу. Обязателен
`EjbRemove()`	`Void`	Контейнер вызывает этот метод перед тем, как компонент EJB будет удален из контейнера. Этот метод должен удалить строку из таблицы, которая соответствует этому компоненту EJB. Обязателен
`EjbActivate()`	`Void`	Контейнер вызывает метод при активации компонента EJB
`EjbPassivate()`	`Void`	Контейнер вызывает метод при пассивации компонента EJB. Обязателен

Таблица 5.12 *(окончание)*

Метод	Возвращаемое значение	Описание
`SetEntityContext (EntityContext ctx)`	`Void`	Контейнер вызывает этот метод после создания компонента EJB, но перед вызовом метода компонента EJBCreate. Компонент EJB ответственен за поддержание своего состояния. Обязателен
`UnsetEntityContext()`	`Void`	Контейнер обращается к методу перед уничтожением компонента EJB. Обязателен
`PrimerMethod(int Chtolibo)`	`Int`	Этот метод следует имплементировать в том случае, если он определен в удаленном интерфейсе. Необязателен

Интерфейсы и классы пакета *javax.ejb*

Интерфейс *javax.ejb.EJBHome*

Домашний интерфейс создается на основе интерфейса `EJBHome` (табл. 5.13).

```
public interface EJBHome extends java.rmi.Remote {
public abstract EJBMetaData getEJBMetaData() throws
java.rmi.RemoteException;
public abstract void remove(Object primaryKey) throws
java.rmi.RemoteException, RemoveException;
public abstract void remove(Handle handle) throws
java.rmi.RemoteException, RemoveException;
}
```

Таблица 5.13. *Интерфейс* `interface javax.ejb.EJBHome`

Метод	Описание
`Public abstract EJBMetaData getEJBMetaData() throws java.rmi.RemoteException;`	Возвращает объект EJBMetaData, который может быть использован в утилитах разработки EJB

Таблица 5.13 *(окончание)*

Метод	Описание
`Public abstract void remove(Object primaryKey) throws java.rmi.RemoteException, RemoveException;`	Удаляет ссылку на компонент EJB в контейнере компонентов EJB
`Public abstract void remove(Handle handle) throws java.rmi.RemoteException, RemoveException;`	Удаляет ссылку на компонент EJB в контейнере компонентов EJB

Интерфейс *javax.ejb.EJBObject*

Удаленный интерфейс основан на интерфейсе `EJBObject` (табл.5.14—5.18)

```
public interface EJBObject extends java.rmi.Remote {
public abstract EJBHome getEJBHome() throws java.rmi.RemoteException;
public abstract Handle getHandle() throws java.rmi.RemoteException;
public abstract void getPrimaryKey() throws java.rmi.RemoteException;
public abstract boolean isIdentical(EJBObject obj) throws
java.rmi.RemoteException;
public abstract void remove() throws java.rmi.RemoteException,
RemoveException;
}
```

Таблица 5.14. *Интерфейс* `interface javax.ejb.EJBObject`

Метод	Описание
`public abstract EJBHome getEJBHome() throws java.rmi.RemoteException;`	Возвращает стаб, соответствующий домашнему интерфейсу
`public abstract Handle getHandle() throws java.rmi.RemoteException;`	Возвращает метку удаленного интерфейса
`public abstract void getPrimaryKey() throws java.rmi.RemoteException;`	Возвращает первичный ключ компонента EJB, если это компонент EJB-сущности
`Public abstract boolean isIdentical(EJBObject obj) throws java.rmi.RemoteException;`	Сравнивает два удаленных стаба, проверяя на идентичность
`public abstract void remove() throws java.rmi.RemoteException, RemoveException;`	Удаляет ссылку на удаленный интерфейс

Интерфейс *javax.ejb.EJBContext*

```
public interface EJBContext extends java.rmi.Remote {
public abstract java.security.Identity getCallerIdentity();
public abstract EJBHome getEJBHome();
public abstract java.util.Properties getEnvironment();
public abstract boolean getRollbackOnly();
public abstract javax.jts.UserTransaction getUserTransaction() throws
java.lang.IllegalStateException;
public abstract boolean IsCallerInRole(java.security.identity role);
public abstract void setRollbackOnly();
}
```

Таблица 5.15. *Интерфейс* `interface javax.ejb.EJBContext`

Метод	Описание
`Public abstract java.security.Identity getCallerIdentity();`	Позволяет идентифицировать клиента, вызывающего компонента EJB
`public abstract EJBHome getEJBHome();`	Возвращает метку удаленного интерфейса
`public abstract java.util.Properties getEnvironment();`	Возвращает список свойств, которые доступны компоненту EJB и предоставлены ему контейнером
`Public abstract boolean getRollbackOnly();`	Сообщает компоненту EJB о том, работает ли он в контексте транзакции, которой дан откат
`Public abstract javax.jts.UserTransaction getUserTransaction() throws java.lang.IllegalStateException;`	Возвращает имплементацию (содержится в контейнере) интерфейса `javax.transaction.UserTransaction`
`Public abstract boolean IsCallerInRole (java.security.identity role);`	Проверяет, имеет ли вызывающий клиент подходящую роль
`Public abstract void setRollbackOnly();`	Задает, что текущая транзакция (если она существует) помечается для отката

Интерфейс *javax.ejb.Handle*

```
public interface Handle{
public abstract EJBObject getEJBObject() throws java.rmi.RemoteException;
}
```

Таблица 5.16. *Интерфейс* `interface javax.ejb.Handle`

Метод	Описание
`public abstract EJBObject getEJBObject() throws java.rmi.RemoteException;`	Возвращает удаленную ссылку объекта `EJBObject`

Интерфейс *javax.ejb.EntityContext*

```
public interface EntityContext extends javax.ejb.EJBContext{
public abstract EJBObject getEJBObject() throws
java.lang.IllegalStateException;
public abstract Object getPrimaryKey() throws
java.lang.IllegalStateException;
}
```

Таблица 5.17. *Интерфейс* `interface javax.ejb.EntityContext`

Метод	Описание
`Public abstract EJBObject getEJBObject() throws java.lang.IllegalStateException;`	Возвращает стаб удаленного интерфейса
`public abstract Object getPrimaryKey() throws java.lang.IllegalStateException;`	Возвращает первичный ключ строки, связанной с компонентом EJB

Интерфейс *javax.ejb.SessionContext*

```
public interface SessionContext extends javax.ejb.EJBContext {
public abstract EJBObject getEJBObject() throws
java.lang.IllegalStateException;
}
```

Таблица 5.18. *Интерфейс* `interface javax.ejb.SessionContext`

Метод	Описание
`public abstract EJBObject getEJBObject() throws java.lang.IllegalStateException;`	Возвращает стаб удаленного интерфейса

Интерфейс *javax.ejb.EJBMetaData*

Интерфейс может использоваться для работы с утилитами разработки компонентов EJB (табл. 5.19)

```
public interface EJBMetaData {
```

```
public abstract EJBHome getEJBHome();
public abstract Class getHomeInterfaceClass();
public abstract Class getPrimaryKeyClass();
public abstract Class getRemoteInterfaceClass();
public abstract boolean isSession();
}
```

Таблица 5.19. *Интерфейс* `interface javax.ejb.EJBMetaData`

Метод	Описание
`public abstract EJBHome getEJBHome();`	Возвращает стаб домашнего интерфейса
`Public abstract Class getHomeInterfaceClass();`	Возвращает класс домашнего интерфейса
`Public abstract Class getPrimaryKeyClass();`	Возвращает класс первичного ключа для компонента EJB-сущности
`Public abstract Class getRemoteInterfaceClass();`	Возвращает класс удаленного интерфейса
`public abstract boolean isSession();`	Возвращает информацию, является ли компонент EJB компонентом EJB-сессии

В заключение приведем пример приложения, работающего на сервере Tomcat.

5.2.17. Пример приложения с использованием компонента EJB

Приложение работает на сервере Tomcat. Структура каталогов приложения выглядит следующим образом (табл. 5.20).

Таблица 5.20. *Структура каталогов приложения*

code/StatefulMakeApp/MakeAppClient		
	Build.xml	
	Index.jsp	
	WEB-INF	
		Web.xml
		App.tld
		Struts-bean.tld

Таблица 5.20 *(окончание)*

code/StatefulMakeApp/MakeAppClient							
		Struts-form.tld					
		Struts-html.tld					
		Struts-logic.tld					
		Struts-template.tld					
		Struts.tld					
		Struts-config.xml					
		Classes					
			Code				
				Masslight			
					Beans		
						MakeAppClientBean.java	
					Actions		
						GetBeanAction.java	
		Lib					
			struts.jar (*)				

Клиентский код включает в себя класс GetBeanAction — файл code/StatefulMakeApp/MakeAppClient/WEB-INF/classes/code/eolymp/actions/GetBeanAction.java (листинг 5.27).

Листинг 5.27. Файл GetBeanAction.java

```
package code.eolymp.actions;
import java.util.Vector;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.util.Locale;
import java.util.Hashtable;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.util.*;
import code.eolymp.beans.MakeAppClientBean;
```

```
public final class GetBeanAction extends Action {

public GetBeanAction() {}
public ActionForward perform(ActionMapping mapping,
                ActionForm form,
                HttpServletRequest request,
                HttpServletResponse response)
 throws IOException, ServletException {

  HttpSession session = request.getSession();
  MakeAppClientBean clientbean = new MakeAppClientBean();
  clientbean.startup();
  session.setAttribute("greeting", clientbean.sayGreeting("MakeApp from a
stateful EJB!"));
    return (mapping.findForward("success"));
  }
}
```

Приложение использует клиентский компонент — класс `MakeAppClientBean`, файл code/MakeApp/MakeAppClient/WEB-INF/classes/code/eolymp/beans/MakeAppClientBean.java (листинг 5.28).

Листинг 5.28. Файл MakeAppClientBean/java

```
package code.eolymp.beans;

import javax.naming.*;
import java.util.Hashtable;
import javax.rmi.PortableRemoteObject;
import javax.ejb.*;

public class MakeAppClientBean
{
  code.eolymp.MakeAppEJBClasses.MakeApp MakeApp = null;
  code.eolymp.MakeAppEJBClasses.MakeAppHome home = null;
  public MakeAppClientBean() {
  }
  /*
  Создает экземпляр компонента MakeApp на сервере и вызывает метод
  'sayMakeApp()', выводит результат в виде String.
```

```
  */
  public void startup () {
    System.setProperty("java.naming.factory.initial",
               "org.jnp.interfaces.NamingContextFactory");
    System.setProperty("java.naming.provider.url",
               "localhost:1099");
    try
    {
      InitialContext jndiContext = new InitialContext();
      System.out.println("Got context \n");
      Object ref  = jndiContext.lookup("MakeApp");
      System.out.println("Got reference \n");
      home = (code.eolymp.MakeAppEJBClasses.MakeAppHome)
        PortableRemoteObject.narrow (ref,
        code.eolymp.MakeAppEJBClasses.MakeAppHome.class);
      MakeApp = home.create();
    }
    catch(Exception e)
    {
      System.out.println(e.toString());
    }
  }
  public String sayGreeting(String greeting) {
    String output = new String("");
    try {
      output += MakeApp.sayMakeApp(greeting);
    }
    catch(Exception e)  {
      System.out.println(e.toString());
    }
    return(output);
  }
}
```

Серверный компонент EJB располагается в каталоге MakeAppEJB, который имеет следующую структуру (табл. 5.21).

Таблица 5.21. Структура компонента EJB

Code/StatefulMakeApp/MakeAppEJB				
	Build.xml			
	META-INF			
		ejb-jar.xml		
	Code			
		Eolymp		
			MakeAppEJBClasses	
				MakeAppBean.java
				MakeApp.java
				MakeAppHome.java

Домашний интерфейс описан в файле code/StatefulMakeApp/MakeAppEJB/code/eolymp/MakeAppEJBClasses/MakeAppHome.java (листинг 5.29).

Листинг 5.29. Файл MakeAppHome.java

```
package code.eolymp.MakeAppEJBClasses;
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

/*
Задание домашнего интерфейса для 'MakeApp'.
*/
public interface MakeAppHome extends EJBHome
{
/*
Экземпляр класса 'MakeAppBean' на сервере
*/
MakeApp create() throws RemoteException, CreateException;
}
```

Файл code/StatefulMakeApp/MakeAppEJB/code/eolymp/MakeAppEJBClasses/MakeApp.java содержит удаленный интерфейс (листинг 5.30).

Листинг 5.30. Файл MakeApp.java

```
package code.eolymp.MakeAppEJBClasses;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface MakeApp extends EJBObject
{
  public void setGreeting(String newGreeting) throws RemoteException;
  public String getGreeting() throws RemoteException;
  public String sayMakeApp() throws RemoteException;
}
```

Класс компонента EJB содержит методы `getGreeting` и `setGreeting`, а также переменную `greeting`, которая служит для хранения информации о состоянии сессии — файл code/StatefulMakeApp/MakeAppEJB/code/eolymp/MakeAppEJBClasses/MakeAppBean.java (листинг 5.31).

Листинг 5.31. Файл MakeAppBean.java

```
package code.eolymp.MakeAppEJBClasses;
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class MakeAppBean implements SessionBean
{
  String greeting;
  public void setGreeting(String newGreeting) {
    greeting = newGreeting;
  }
  public String getGreeting() {
    return (greeting);
  }
  public String sayMakeApp()
  {
    return(getGreeting());
  }
  public MakeAppBean() {}
```

```
  public void ejbCreate() {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {}
}
```

Файл описания размещения ejb-jar.xml представлен ниже (листинг 5.32).

Листинг 5.32. Файл ejb-jar.xml

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
  '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
  'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>MakeApp</display-name>
  <enterprise-beans>
  <session>
    <display-name>MakeApp</display-name>
    <ejb-name>MakeApp</ejb-name>
    <home>code.eolymp.MakeAppEJBClasses.MakeAppHome</home>
    <remote>code.eolymp.MakeAppEJBClasses.MakeApp</remote>
    <ejb-class>code.eolymp.MakeAppEJBClasses.MakeAppBean</ejb-class>
    <session-type>Stateful</session-type>
    <transaction-type>Bean</transaction-type>
    <security-identity>
    <description></description>
    <use-caller-identity></use-caller-identity>
    </security-identity>
  </session>
  </enterprise-beans>
</ejb-jar>
```

Скрипт build.xml служит для задания информации, используемой утилитами code/StatefulMakeApp/MakeAppClient/build.xml (листинг 5.33).

Листинг 5.33. Файл build.xml

```
<project name="MakeAppClient" default="dist" basedir=".">

 <!-- set global properties for this build -->
 <property environment="env"/>
 <property name="top" value="."/>
 <property name="src" value="."/>
 <property name="build" value="build"/>
 <property name="dist" value="dist"/>
 <property name="war_dir" value="${dist}/lib"/>
 <property name="war_file" value="${war_dir}/MakeAppClient.war"/>

 <property name="webinf" value="${top}/WEB-INF"/>
 <property name="web.xml" value="${webinf}/web.xml"/>
 <property name="classes" value="${webinf}/classes"/>
 <property name="lib" value="${top}/WEB-INF/lib"/>
 <property name="struts.jar" value="${env.STRUTS_HOME}/lib/struts.jar"/>
 <property name="servlet.jar"
value="${env.TOMCAT_HOME}/lib/servlet.jar"/>
 <property name="ejb.src" value="${top}/../MakeAppEJB"/>
 <property name="ejb.jar" value="${env.JBOSS_HOME}/lib/ext/jboss-
j2ee.jar"/>
 <property name="deploy" value="${env.JBOSS_HOME}/deploy"/>

 <target name="clean">
   <!-- Delete our the ${build} and ${dist} directory trees -->
   <delete dir="${build}"/>
   <delete dir="${dist}"/>
   <delete dir="${war_dir}"/>
 </target>

 <target name="init">
   <!-- Create the build directory structure used by compile and dist -->
   <mkdir dir="${build}"/>
   <mkdir dir="${dist}"/>
   <mkdir dir="${war_dir}"/>
 </target>
```

```
 <target name="compile" depends="init">
   <!-- Compile the java code from ${src} into ${build} -->
   <javac
   srcdir="${ejb.src}"
   destdir="${build}"
   classpath="${ejb.jar}"/>
   <javac
   srcdir="${top}/${src}"
   destdir="${build}"
   classpath="${servlet.jar}:${struts.jar}:${ejb.jar}"/>
 </target>


 <target name="dist" depends="compile">
   <!-- Put everything in a war file -->
   <war warfile="${war_file}" webxml="${web.xml}">
   <!-- include all JSPs in root level, and all .properties files
anywhere -->
   <fileset dir="${top}/${src}">
     <include name="*.jsp"/>
     <include name="**/*.properties"/>
   </fileset>


   <!-- include all tag libraries in WEB-INF, and all .xml config files,
but not web.xml (that's handled separately) -->
   <webinf dir="${webinf}">
     <include name="*.tld"/>
     <include name="*.xml"/>
     <exclude name="web.xml"/>
   </webinf>


   <!-- include all libraries in WEB-INF/lib (like struts.jar) -->
   <lib dir="${lib}"/>


   <!-- include all compiled classes -->
   <classes dir="${build}"/>
   </war>
 </target>


 <target name="deploy">
```

```
   <!-- Copy the war file to the JBoss deploy directory -->
   <copy file="${war_file}" todir="${deploy}"/>
 </target>

 <target name="all" depends="clean,dist,deploy"/>

</project>
```

Скрипт конфигурации code/StatefulMakeApp/MakeAppEJB/build.xml представлен ниже (листинг 5.34).

Листинг 5.34. Файл MakeAppEJB/build.xml

```
<project name="MakeAppEJB" default="dist" basedir=".">

 <!-- set global properties for this build -->
 <property environment="env"/>
 <property name="top" value="."/>
 <property name="src" value="."/>
 <property name="build" value="build"/>
 <property name="dist" value="dist"/>
 <property name="jar_dir" value="${dist}/lib"/>
 <property name="jar_file" value="${jar_dir}/MakeAppEJB.jar"/>

 <property name="webinf" value="${top}/WEB-INF"/>
 <property name="web.xml" value="${webinf}/web.xml"/>
 <property name="classes" value="${webinf}/classes"/>
 <property name="lib" value="${top}/WEB-INF/lib"/>
 <property name="ejb.jar" value="${env.JBOSS_HOME}/lib/ext/
jboss-j2ee.jar"/>
 <property name="deploy" value="${env.JBOSS_HOME}/deploy"/>

 <target name="clean">
   <!-- Delete our the ${build} and ${dist} directory trees -->
   <delete dir="${build}"/>
   <delete dir="${dist}"/>
   <delete dir="${jar_dir}"/>
 </target>

 <target name="init">
```

```
    <mkdir dir="${build}"/>
    <mkdir dir="${dist}"/>
    <mkdir dir="${jar_dir}"/>
  </target>

  <target name="compile" depends="init">
    <!-- Compile the java code from ${src} into ${build} -->
    <javac
    srcdir="${top}/${src}"
    destdir="${build}"
    classpath="${ejb.jar}"/>
  </target>

  <target name="dist" depends="compile">
    <!-- Put everything in a jar file -->
    <jar jarfile="${jar_file}">
    <!-- include all compiled class files -->
    <fileset dir="${build}">
      <include name="**/*.class"/>
    </fileset>

    <!-- include ejb-jar.xml -->
    <fileset dir="${top}">
      <include name="META-INF/ejb-jar.xml"/>
    </fileset>
    </jar>
  </target>

  <target name="deploy">
    <!-- Copy the jar file to the JBoss deploy directory -->
    <copy file="${jar_file}" todir="${deploy}"/>
  </target>

  <target name="all" depends="clean,dist,deploy"/>

</project>
```

И, наконец, скрипт конфигурации code/StatefulMakeApp/build.xml (листинг 5.35).

Листинг 5.35. Файл code/StatefulMakeApp/build.xml

```
<project name="StatefulMakeApp" default="all" basedir=".">

<target name="all">
  <ant dir="MakeAppClient" target="all"/>
  <ant dir="MakeAppEJB" target="all"/>
</target>

<target name="clean">
  <ant dir="MakeAppClient" target="clean"/>
  <ant dir="MakeAppEJB" target="clean"/>
</target>
</project>
```

Все приложение компилируется и размещается на сервере с применением утилиты ant, которая запускается из каталога StatefulMakeApp (ant all). Приложение запускается путем обращения из браузера к ресурсу **http://localhost:8080/MakeAppClient/getEJB.do**.

Приложение 1

Краткая справка по компонентам EJB

Посвящается Бобе Иисусовичу Рабиновичу

Пакет *javax.ejb*

В большинстве приложений, в которых применяется технология EJB, используются следующие пакеты:

- `javax.ejb.*`
- `javax.rmi.*`
- `javax.naming.*`

Пакет `javax.ejb.*` относится к Java 2 Platform, Enterprise Edition.

Этот пакет содержит классы серверных компонентов EJB и интерфейсы, определяющие взаимодействие между компонентами EJB и клиентами и компонентами EJB и контейнером компонентов EJB.

Интерфейсы и классы пакета *javax.ejb*

Интерфейсы

- `public interface EJBContext`
- `public interface EJBHome`
- `public interface EJBLocalHome`
- `public interface EJBLocalObject`
- `public interface EJBMetaData`
- `public interface EJBObject`
- `public interface EnterpriseBean`
- `public interface EntityBean`
- `public interface EntityContext`
- `public interface Handle`

- public interface HomeHandle
- public interface MessageDrivenBean
- public interface MessageDrivenContext
- public interface SessionBean
- public interface SessionContext
- public interface SessionSynchronization
- public interface TimedObject
- public interface Timer
- public interface TimerHandle
- public interface TimerService

Классы (исключения)

- public class AccessLocalException
- public class CreateException
- public class DuplicateKeyException
- public class EJBException
- public class FinderException
- public class NoSuchEntityException
- public class NoSuchObjectLocalException
- public class ObjectNotFoundException
- public class RemoveException
- public class TransactionRequiredLocalException
- public class TransactionRolledbackLocalException

Интерфейсы

- EJBContext

  Предоставляет экземпляру компонента EJB средства доступа к контейнеру компонентов EJB.

- EJBHome

  Используется для создания домашнего интерфейса компонента EJB.

- EJBLocalHome

  Этот интерфейс используется для создания локального домашнего интерфейса компонента EJB.

- EJBLocalObject

  Интерфейс EJBLocalObject используется для создания локального интерфейса компонента EJB.

- `EJBMetaData`

  Интерфейс, позволяющий клиенту получать мета-данные о компоненте EJB.

- `EJBObject`

  Интерфейс, на основе которого создается удаленный интерфейс компонента EJB.

- `EnterpriseBean`

  Интерфейс, на основе которого создается класс компонента EJB.

- `EntityBean`

  Интерфейс, который имплементируется для всех компонентов EJB-сущности.

- `EntityContext`

  Интерфейс `EntityContext` предоставляет контекст контейнера, в котором выполняется компонент EJB-сущности.

- `Handle`

  Этот интерфейс имплементируется при создании меток компонентов EJB `handles`.

- `HomeHandle`

  Этот интерфейс имплементируется при создании меток `handles` домашних объектов.

- `MessageDrivenBean`

  Интерфейс `MessageDrivenBean` имплементируется при создании класса компонента EJB на основе сообщений.

- `MessageDrivenContext`

  При помощи интерфейса `MessageDrivenContext` осуществляется доступ экземпляра компонента EJB, основанного на сообщениях, к контейнеру компонента EJB во время выполнения компонента EJB.

- `SessionBean`

  Этот интерфейс имплементируется при создании класса компонента EJB-сессии.

- `SessionContext`

  Интерфейс `SessionContext` предоставляет компоненту EJB-сессии доступ к окружению, обеспечиваемому контейнером компонентов EJB, во время работы компонента EJB.

- `SessionSynchronization`

  Интерфейс `SessionSynchronization` используется для того, чтобы с помощью контейнера компонентов EJB сообщать компоненту EJB-сессии о транзакции, с которой связывается экземпляр компонента EJB.

Исключительные ситуации

- `AccessLocalException`

  Исключение `AccessLocalException` возникает в том случае, если вызывающий клиент не имеет права обращаться к вызываемому им методу.

- `CreateException`

  Исключение `CreateException` используется при возникновении ошибки во всех случаях при работе с созданием домашних интерфейсов компонентов EJB на основе методов `create`.

- `DuplicateKeyException`

  Исключение `DuplicateKeyException` возникает, если компонент EJB-сущности с указанным ключом не может быть создан, так как компонент EJB с таким ключом уже существует.

- `EJBException`

  Сообщение выдается компонентом EJB контейнеру в том случае, когда запрашиваемый метод не может быть выполнен из-за непредвиденной ошибки.

- `FinderException`

  Это исключение возникает при невозможности обнаружить компонент EJB. Используется во всех методах, осуществляющих поиск компонента EJB в домашних интерфейсах.

- `NoSuchEntityException`

  Исключение `NoSuchEntityException` возникает в компоненте EJB-сущности тогда, когда запрошенный метод не может быть выполнен из-за того, что требуемые данные были удалены из базы данных.

- `NoSuchObjectLocalException`

  Исключение возникает при попытке обратиться к объекту, которого уже не существует.

- `ObjectNotFoundException`

  Исключение возникает в методе поиска тогда, когда заданный серверный компонент EJB уже не существует.

- `RemoveException`

  Возникает тогда, когда осуществляется попытка удалить серверный компонент EJB, но удаление компонента EJB невозможно (запрещает либо компонент EJB, либо контейнер).

- `TransactionRequiredLocalException`

  Сообщение об ошибке возникает тогда, когда в запросе не присутствует контекст транзакции, а объект запроса требует наличие контекста транзакции.

- `TransactionRolledbackLocalException`

  Это сообщение возникает тогда, когда транзакции, ответственной за обработку полученного запроса, дан откат, или эта транзакция была помечена для отката (отмены проведения транзакции).

Интерфейс *EJBContext*

```
public interface EJBContext
```

Интерфейс `EJBContext` предоставляет доступ экземпляру компонента EJB к окружению, предоставляемому контейнером компонентов EJB. Дополнительные методы содержатся в интерфейсах, основанных на этом интерфейсе, а именно, в интерфейсах `SessionContext`, `EntityContext`, `MessageDrivenContext`, при помощи которых осуществляется доступ к окружению контейнера из компонентов EJB-сессии и компонентов EJB-сущности соответственно.

Методы интерфейса

- `getCallerPrincipal()`

  Возвращает `java.security.Principal`, от которого получен запрос.

  ```
  public java.security.Principal getCallerPrincipal()
  ```

- `getEJBHome()`

  Возвращает удаленный домашний интерфейс.

  ```
  public EJBHome getEJBHome()
  ```

  Исключение `java.lang.IllegalStateException` — возникает в случае отсутствия удаленного домашнего интерфейса для компонента EJB.

- `getEJBLocalHome()`

  Возвращает локальный домашний интерфейс.

  ```
  public EJBLocalHome getEJBLocalHome()
  ```

  Исключение `java.lang.IllegalStateException` — возникает в случае, если компонент EJB не обладает локальным домашним интерфейсом.

- `getRollbackOnly()`

  ```
  public boolean getRollbackOnly()
                  throws java.lang.IllegalStateException
  ```

  Проверяет, помечена ли транзакция для отката. Метод используется только в компоненте EJB с транзакциями, которые поддерживаются средствами контейнера. Возвращает `true`, если транзакция помечена для отката, или `false` в противном случае.

  Исключение `java.lang.IllegalStateException` — исключение возникает в контейнере, если данный метод не может быть использован (в ком-

понентах EJB с поддержкой транзакций средствами самого компонента EJB).

- `isCallerInRole(java.lang.String roleName)`

  Проверяет, обладает ли клиент указанной ролью безопасности.

  ```
  public boolean isCallerInRole(java.lang.String roleName)
  ```

  Где `roleName` — имя роли. Именем роли может быть одна из ролей, указанная в описателе размещения компонента EJB. Возвращает `true`, если клиент обладает указанной ролью, `false` в противном случае.

- `setRollbackOnly()`

  Помечает транзакцию для отката.

  ```
  public void setRollbackOnly()
              throws java.lang.IllegalStateException
  ```

  После того как транзакция помечена для отката, эта отметка остается навсегда. Такая транзакция не может быть выполнена. Этот метод может быть применен только для компонентов EJB, в которых используются транзакции, поддерживаемые контейнером.

  Исключение `java.lang.IllegalStateException` — возникает в контейнере компонентов EJB тогда, когда указанный метод не может быть использован (в компонентах EJB, поддерживающих транзакции своими средствами).

Интерфейс *EntityContext*

```
public interface EntityContext
```

Создан на основе `EJBContext`. Интерфейс предоставляет доступ к контексту окружения, предоставляемому компоненту EJB контейнером во время работы. Интерфейс передается экземпляру компонента EJB-сущности контейнером после создания компонента EJB-сущности.

Методы интерфейса

- `getEJBLocalObject()`

  ```
  public EJBLocalObject getEJBLocalObject()
  ```

  Исключение `java.lang.IllegalStateException`.

  Возвращает локальный объект, связанный с экземпляром компонента EJB. Используется в методах `ejbActivate`, `ejbPassivate`, `ejbPostCreate`, `ejbRemove`, `ejbLoad`, `ejbStore`.

- `getEJBObject`

  ```
  public EJBObject getEJBObject()
  ```

  Возвращает объект компонента EJB, связанный с данным экземпляром.

Исключение `java.lang.IllegalStateException` — возникает тогда, когда обращение к методу запрещено, или если экземпляр компонента EJB не имеет удаленного интерфейса.

- `getPrimaryKey`

  `public java.lang.Object getPrimaryKey()`

  Возвращает первичный ключ, связанный с экземпляром компонента EJB.

  Исключение `java.lang.IllegalStateException`.

  Возникает в случае невозможности вызова метода.

Интерфейс *SessionContext*

`public interface SessionContext`

Интерфейс создан на основе `EJBContext`. Предоставляет средства доступа к контексту окружения во время работы компонента EJB-сессии. Контекст окружения передается контейнером компоненту EJB-сессии сразу после создания экземпляра компонента EJB. Контекст остается связан с экземпляром компонента EJB до тех пор, пока существует этот экземпляр компонента EJB.

Методы интерфейса

- `getEJBLocalObject`

  `public EJBLocalObject getEJBLocalObject()`

  Возвращает локальный объект, соответствующий экземпляру компонента EJB. Используется в методах `ejbActivate`, `ejbPassivate`, `ejbPostCreate`, `ejbRemove`, `ejbLoad`, `ejbStore`.

  Исключение `java.lang.IllegalStateException` — возникает при невозможности выполнить метод.

- `getEJBObject`

  `public EJBObject getEJBObject()`

  Возвращает объект компонента EJB, используется в методах `ejbCreate` `ejbRemove` или между этими методами.

  Исключение `java.lang.IllegalStateException` — возникает при невозможности выполнить метод (например, при отсутствии удаленного интерфейса).

Интерфейс *MessageDrivenContext*

`public interface MessageDrivenContext`

Создан на основе `EJBContext`. Контекст, предоставляемый контейнером экземпляру компонента EJB, основанного на сообщениях, возникающих

после создания экземпляра компонента EJB. Контекст связан с экземпляром компонента EJB в течение всего жизненного цикла компонента EJB.

Интерфейс *EJBHome*

```
public interface EJBHome
```

Создан на основе `java.rmi.Remote`. На основе этого интерфейса создаются удаленные домашние интерфейсы компонентов EJB. Эти интерфейсы содержат методы, позволяющие клиенту создавать, находить и удалять объекты компонентов EJB, а также использовать домашние методы компонентов EJB. Методы интерфейса имплементируются в контейнере компонентов EJB.

Методы интерфейса

- `remove`

  ```
  public void remove(Handle handle)
  ```

  Удаляет объект компонента EJB по метке компонента EJB `handle`.

  Исключения:

  `RemoveException`

  Возникает при невозможности удаления компонента EJB (не позволяет компонент EJB или контейнер).

  `java.rmi.RemoteException`

  Возникает, если нет возможности удалить компонент EJB в силу действий системы.

- `remove`

  ```
  public void remove(java.lang.Object primaryKey)
  ```

  Удаляет экземпляр компонента EJB по первичному ключу компонента EJB. Используется только для удаления компонентов EJB-сущности. При попытке использовать этот метод для удаления компонента EJB-сессии возникает исключение `RemoteException`.

  Исключения:

  `RemoveException` — возникает в случае, если компонент EJB или контейнер не позволяют удалить экземпляр компонента EJB.

  `java.rmi.RemoteException` — возникает в случае невозможности удалить компонент EJB в силу действий системы.

- `getEJBMetaData`

  ```
  public EJBMetaData getEJBMetaData()
  ```

  Возвращает интерфейс `EJBMetaData`.

  Исключение `java.rmi.RemoteException` — возникает ввиду системных ошибок.

- `getHomeHandle`

  `public HomeHandle getHomeHandle()`

  Возвращает метку удаленного домашнего объекта. Эта метка может затем использоваться для получения ссылки на удаленный домашний объект даже на другой машине Java.

  Исключение `java.rmi.RemoteException` — возникает при ошибке в работе системы.

Интерфейс *EJBLocalHome*

`public interface EJBLocalHome`

На основе этого интерфейса создаются все локальные домашние интерфейсы компонентов EJB. Интерфейс содержит методы, с помощью которых клиент находит, создает и удаляет объекты компонентов EJB, а также использует домашние методы компонентов EJB. Интерфейс имплементируется в контейнере серверных компонентов EJB.

Метод *remove* интерфейса *EJBLocalHome*

`public void remove(java.lang.Object primaryKey)`

Удаляет объект компонента EJB по первичному ключу. Может быть использован только локальными клиентами для удаления компонентов EJB-сущности. При использовании с компонентом EJB-сессии возникнет исключение `EJBException`.

Исключения:

- `RemoveException`

  Возникает при невозможности удалить компонент EJB (запрещает контейнер или компонент EJB).

- `EJBException`

  Возникает при невозможности удалить компонент EJB в силу ошибок системы.

Интерфейс *EJBLocalObject*

`public interface EJBLocalObject`

Интерфейс используется для создания локальных интерфейсов компонентов EJB. Этот интерфейс предоставляет возможность клиенту доступ к компонентам EJB. Локальный интерфейс предоставляет возможность обращения к бизнес-методам. Интерфейс имплементируется в контейнере.

Метод *getEJBLocalHome*

```
public EJBLocalHome getEJBLocalHome()
```

Возвращает ссылку на локальный домашний интерфейс. Локальный домашний интерфейс содержит методы создания, нахождения и удаления компонентов EJB, а также бизнес-методы, которые доступны клиенту.

Исключение `EJBException` — возникает при ошибке в работе системы.

Метод *getPrimaryKey*

```
public java.lang.Object getPrimaryKey() throws EJBException
```

Возвращает первичный ключ локального объекта EJB.

Метод может быть вызван только в компоненте EJB-сущности. Попытка обратиться к этому методу из компонента EJB-сессии приведен к возникновению ошибки `EJBException`.

Исключение `EJBException`.

Метод *remove*

- `public void remove() throws RemoveException, EJBException`

  Удаляет локальный объект компонента EJB.

  Исключения: `RemoveException` и `EJBException`.

- `public boolean isIdentical(EJBLocalObject obj) throws EJBException`

  Проверяет идентичность заданного локального компонента EJB вызванному компоненту EJB. Возвращает `true` или `false`.

  Исключение `EJBException`.

Интерфейс *EJBMetaData*

```
public interface EJBMetaData
```

Интерфейс позволяет получать мета-данные о серверном компоненте EJB. Мета-данные используется при создании приложения и размещении приложения на сервере, а также при использовании языков сценариев с доступом к серверным компонентам EJB. Интерфейс `EJBMetaData` не является удаленным интерфейсом. Класс, имплементирующий этот интерфейс, должен иметь тип RMI/IDL и быть сериализуемым (serializable).

Методы интерфейса

- `getEJBHome`

  `public EJBHome getEJBHome()`

  Получает удаленный домашний интерфейс компонента EJB.

- `getHomeInterfaceClass`

  `public java.lang.Class getHomeInterfaceClass()`

  Получает объект типа `Class` для удаленного домашнего интерфейса компонента EJB.

- `getRemoteInterfaceClass`

  `public java.lang.Class getRemoteInterfaceClass()`

  Получает объект типа `Class` для удаленного домашнего интерфейса компонента EJB.

- `getPrimaryKeyClass`

  `public java.lang.Class getPrimaryKeyClass()`

  Получает объект типа `Class` для класса первичного ключа компонента EJB.

- `isSession`

  `public boolean isSession()`

  Проверяет, является ли компонент EJB компонентом EJB-сессии.

- `isStatelessSession`

  `public boolean isStatelessSession()`

  Проверяет, является ли компонент EJB компонентом EJB-сессии без поддержки состояния.

Интерфейс *EJBObject*

`public interface EJBObject`

Создан на основе `java.rmi.Remote`.

Этот интерфейс используется при создании всех удаленных интерфейсов серверных компонентов EJB. Удаленный интерфейс позволяет клиенту "увидеть" серверный компонент EJB. Удаленный интерфейс предоставляет удаленному клиенту доступ к бизнес-методам компонента EJB. Удаленный интерфейс должен основываться на интерфейсе `javax.ejb.EJBObject`, в нем должны содержаться специфичные для серверных компонентов EJB методы. Удаленный интерфейс компонентов EJB имплементируется в контейнере компонентов EJB.

Методы интерфейса

- `getEJBHome`

  `public EJBHome getEJBHome() throws java.rmi.RemoteException`

  Получает удаленный домашний интерфейс. В нем определены методы `create`, `finder`, `remove` и домашние бизнес-методы.

Возвращает ссылку на домашний интерфейс компонента EJB.

Исключение `java.rmi.RemoteException`.

- `getPrimaryKey`

```
public java.lang.Object getPrimaryKey()
            throws java.rmi.RemoteException
```

Получает первичный ключ объекта компонента EJB.

Используется в компоненте EJB-сущности, при вызове из компонента EJB-сессии возникает ошибка `RemoteException`. Возвращает первичный ключ.

- `remove`

```
public void remove()
        throws java.rmi.RemoteException, RemoveException
```

Удаляет объект EJB.

Исключения:

`java.rmi.RemoteException` и `RemoveException`.

- `getHandle`

```
public Handle getHandle()
         throws java.rmi.RemoteException
```

Получает метку объекта EJB. В дальнейшем метка может быть использована для получения ссылки на объект даже на другой машине Java. Возвращает ссылку на метку объекта EJB.

Исключение `java.rmi.RemoteException`.

- `isIdentical`

```
public boolean isIdentical(EJBObject obj)
       throws java.rmi.RemoteException
```

Проверка идентичности компонента EJB.

Исключение `java.rmi.RemoteException`.

Интерфейс *EnterpriseBean*

```
public interface EnterpriseBean
```

Создан на основе `java.io.Serializable`.

Используется при создании класса серверного компонента EJB. Это суперинтерфейс для создания интерфейсов компонентов EJB `SessionBean`, `EntityBean` и `MessageDrivenBean`.

Интерфейс *EntityBean*

```
public interface EntityBean
```

Создан на основе `EnterpriseBean`.

Интерфейс `EntityBean` имплементируется при создании компонентов EJB-сущности.

Методы интерфейса

- `setEntityContext`

  ```
  public void setEntityContext(EntityContext ctx)
          throws EJBException, java.rmi.RemoteException
  ```

  Задает контекст компонента EJB-сущности. После создания объекта компонента EJB-сущности контейнер вызывает соответствующие методы. Методы выполняются в контексте окружения.

  Параметр `ctx` — интерфейс контекста компонента EJB сущности.

  Исключения: `EJBException`, `java.rmi.RemoteException`.

- `unsetEntityContext`

  ```
  public void unsetEntityContext()
         throws EJBException, java.rmi.RemoteException
  ```

  Метод вызывается перед удалением компонента EJB-сущности.

  Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbRemove`

  ```
  public void ejbRemove()
          throws RemoveException, EJBException,
                 java.rmi.RemoteException
  ```

  Контейнер вызывает этот метод для удаления компонента EJB.

  Исключения: `RemoveException`, `EJBException` и `java.rmi.RemoteException`.

- `ejbActivate`

  ```
  public void ejbActivate()
          throws EJBException, java.rmi.RemoteException
  ```

  Активация объекта контейнером.

  Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbPassivate`

  ```
  public void ejbPassivate()
          throws EJBException, java.rmi.RemoteException
  ```

Контейнер пассивирует экземпляр компонента EJB.

Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbLoad`

```
public void ejbLoad()
        throws EJBException, java.rmi.RemoteException
```

Контейнер синхронизирует экземпляр компонента EJB. Компонент EJB обновляет свое состояние.

Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbStore`

```
public void ejbStore()
        throws EJBException, java.rmi.RemoteException
```

Компонент EJB сохраняет свое состояние.

Исключения: `EJBException` и `java.rmi.RemoteException`.

Интерфейс *SessionBean*

```
public interface SessionBean
```

Создан на основе `EnterpriseBean`. Интерфейс имплементируется при создании компонентов EJB-сессии. Контейнер использует методы интерфейса при работе с экземплярами компонентов EJB-сессии на протяжении их жизненного цикла.

Методы интерфейса

- `setSessionContext`

```
public void setSessionContext(SessionContext ctx)
        throws EJBException, java.rmi.RemoteException
```

Задает связь с контекстом сессии. Контейнер вызывает метод после создания экземпляра компонента EJB.

Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbRemove`

```
public void ejbRemove()
        throws EJBException, java.rmi.RemoteException
```

Метод вызывается перед удалением объекта компонента EJB-сессии.

Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbActivate`

```
public void ejbActivate()
        throws EJBException, java.rmi.RemoteException
```

Экземпляр компонента EJB активизируется контейнером.

Исключения: `EJBException` и `java.rmi.RemoteException`.

- `ejbPassivate`

```
public void ejbPassivate()
        throws EJBException, java.rmi.RemoteException
```

Контейнер выводит экземпляр компонента EJB из сервиса.

Исключения: `EJBException` и `java.rmi.RemoteException`.

Интерфейс *MessageDrivenBean*

```
public interface MessageDrivenBean
```

Создан на основе `EnterpriseBean`.

Имплементируется при создании компонента EJB, основанного на сообщениях. Контейнер использует методы интерфейса для взаимодействия с компонентом EJB в течение жизненного цикла компонента EJB.

Методы интерфейса

- `setMessageDrivenContext`

```
public void setMessageDrivenContext(MessageDrivenContext ctx)
        throws EJBException
```

Задает контекст.

Исключение `EJBException`.

- `ejbRemove`

```
public void ejbRemove() throws EJBException
```

Вызывается для удаления объекта компонента EJB.

Исключение `EJBException`.

Интерфейс *SessionSynchronization*

```
public interface SessionSynchronization
```

Этот интерфейс используется контейнером компонентов EJB для поддержания сессий, для информирования о пределах распространения транзакций. Класс компонента EJB-сессии должен имплементировать этот интерфейс только в том случае, если требуется синхронизация работы компонента EJB в пределах транзакций.

Методы интерфейса

- `afterBegin`

  ```
  public void afterBegin()
          throws EJBException, java.rmi.RemoteException
  ```

  Метод сообщает компоненту EJB о том, что начата новая транзакция, последующие методы компонента EJB будут вызываться в контексте транзакции.

  Исключения: `EJBException` и `java.rmi.RemoteException`.

- `beforeCompletion`

  ```
  public void beforeCompletion()
          throws EJBException, java.rmi.RemoteException
  ```

  Сообщает компоненту EJB, что транзакция почти завершена, например, для того, чтобы записать кэшированные данные в базу.

  Исключения: `EJBException` и `java.rmi.RemoteException`.

- `afterCompletion`

  ```
  public void afterCompletion(boolean committed)
          throws EJBException, java.rmi.RemoteException
  ```

  Сообщает, что транзакция проведена (протокол проведения транзакции успешно завершен) или о том, что транзакции дан откат.

  `committed`: `true` — транзакция проведена, `false` — транзакции дан откат.

  Исключения: `EJBException` и `java.rmi.RemoteException`.

Интерфейс *Handle*

```
public interface Handle
```

Создан на основе `java.io.Serializable`.

Имплементируется всеми метками компонентов EJB.

Методы интерфейса

- `getEJBObject`

  ```
  public EJBObject getEJBObject() throws java.rmi.RemoteException
  ```

  Получает ссылку на объект по метке компонента EJB.

  Исключение `java.rmi.RemoteException`.

Интерфейс *HomeHandle*

```
public interface HomeHandle
```

Создан на основе `java.io.Serializable`.

Имплементируется всеми метками домашних объектов.

Метод *getEJBHome*

```
public EJBHome getEJBHome() throws java.rmi.RemoteException
```

Получает домашний объект по метке домашнего объекта.

Приложение 2

Краткая справка по сервлетам и JSP

Интерфейс сервлетов

Для создания сервлетов используется интерфейс `Servlet`. Существуют два стандартных класса, имплементирующих этот интерфейс. Это классы `GenericServlet` и `HttpServlet`. В большинстве случаев для разработки сервлетов используется класс `HttpServlet`.

Методы обработки запросов

Базовый интерфейс `Servlet` определяет метод `service`, в котором происходит обработка клиентских запросов. Этот метод вызывается для каждого запроса, поступившего к объекту сервлета. Как правило, сервлет разрабатывается таким образом, чтобы была возможность осуществлять одновременную обработку нескольких запросов. При этом сервлет должен поддерживать многопоточность. Контейнер сервлета обрабатывает одновременно поступающие запросы путем последовательного вызова методы `service` в соответствующие моменты времени.

Методы обработки HTTP-запросов

Абстрактный класс `HttpServlet` содержит дополнительные методы обработки запросов в дополнение к методам интерфейса `Servlet`, которые автоматически вызываются при обращении к методу `service` класса `HttpServlet` и которые предназначены для обработки HTTP-запросов. Перечислим эти методы.

- `doGet`

  Метод обработки HTTP-запросов типа `GET`.

- `doPost`

  Метод обработки HTTP-запросов типа `POST`.

- `doPut`

  Метод обработки HTTP-запросов типа `PUT`.

- `doDelete`

  Метод обработки HTTP-запросов типа `DELETE`.

- `doHead`

  Метод обработки HTTP-запросов типа `HEAD`.

- `doOptions`

  Метод обработки HTTP-запросов типа `OPTIONS`.

- `doTrace`

  Метод обработки HTTP-запросов типа `TRACE`.

При разработке сервлетов разработчик, как правило, использует методы `doGet` и `doPost`. Прочие методы нужны в том случае, когда разработчик использует возможности, предоставляемые протоколом HTTP. Методы `doPut` и `doDelete` позволяют использовать возможности протокола HTTP/1.1. Метод `doHead` класса `HttpServlet` — это специальный случай метода `doGet`, который предназначен для возвращения заголовков, создаваемых методом `doGet`. Сервлеты поддерживают также метод `doOptions`. Метод `doTrace` генерирует ответ, который содержит все элементы заголовка, посланного в запросе `TRACE`.

Поддержка условного выполнения метода *GET*

Интерфейс `HttpServlet` содержит метод `getLastModified`, с помощью которого выполняется условный вызов метода `GET`. Ресурс будет вызван и выполнен только в том случае, если он был изменен в течение указанного времени. В некоторых ситуациях использование этого метода может помочь значительно более эффективно использовать сеть.

Количество экземпляров сервлета

Если сервлет описан в дескрипторе размещения, то дескриптор содержит информацию о том, как контейнер сервлета осуществляет создание экземпляров сервлета. Если сервлет размещен не в распределенном окружении, контейнер должен использовать только один экземпляр сервлета. Однако если сервлет имплементирует интерфейс `SingleThreadModel`, то контейнер сервлета может создать несколько объектов сервлета, что может быть следствием большой загрузки сервера, когда запросы будут распределяться между несколькими объектами сервлета. Если сервлет размещается в составе приложения, которое помечено в качестве распределенного приложения, то может существовать только один объект сервлета на одной виртуальной машине Java. Если же сервлет в составе распределенного приложения создан

на основе интерфейса `SingleThreadModel`, тогда контейнер может инициализировать несколько объектов этого сервлета.

Однопоточный сервлет

Если сервлет создан на основе интерфейса `SingleThreadModel`, то такой сервлет может выполнять только один вызов метода `service()` для каждого момента времени. Это справедливо для каждого экземпляра сервлета в отдельности. Контейнер, тем не менее, может создать несколько объектов такого сервлета.

Жизненный цикл сервлета

Жизненный цикл сервлета строго определен, задан способ загрузки сервлета, создание и инициализация экземпляра сервлета, методы обработки запросов, методы вывода сервлета из работы. Жизненный цикл сервлета определяется методами `init`, `service`, `destroy` интерфейса `javax.servlet.Servlet`. Эти методы должны быть имплементированы либо явно либо опосредованно через классы `GenericServlet` или `HttpServlet`.

Загрузка сервлета

Загрузка сервлета производится контейнером сервлетов. Загрузка сервлетов происходит во время запуска контейнера или тогда, когда контейнер обнаружит необходимость обращения к сервлету, когда появится необходимость обработать запрос. Контейнер загружает класс сервлета, используя обычную процедуру загрузки Java-класса, используемую в Java. Загрузка может быть осуществлена как из локальной файловой системы, так и с использованием сети. После загрузки класса контейнер создает экземпляр этого класса.

Инициализация сервлета

При создании экземпляра класса сервлета контейнер должен инициализировать этот объект, прежде чем объект получит клиентские запросы. Во время инициализации сервлет может читать данные, используемые для создания постоянных соединений, инициализировать соединения с ресурсами (например, с базами данных JDBC), производить другие одноразовые действия. Контейнер производит инициализацию объекта сервлета путем вызова функции `init` интерфейса `Servlet`. При этом создается один объект интерфейса `ServletConfig`. Объект конфигурации позволяет сервлету осуществлять доступ к параметрам инициализации, представленным в виде пар имен и значений, задаваемым в описании конфигурации Web-приложения. Объект конфигурации также позволяет сервлету осуществлять доступ к объекту интерфейса `ServletContext`, который содержит информацию об окружении сервлета.

Ошибки инициализации

В процессе инициализации могут возникнуть ошибки `UnavailableException` и `ServletException`. При этом сервлет не сможет нормально работать, и должен быть удален из контейнера сервлетов. При этом метод `destroy` не может быть вызван. После ошибки инициализации контейнер может создать и инициализировать новый объект сервлета. Если возникает исключение `UnavailableException` с указанием минимального времени недоступности сервлета, то контейнер должен подождать в течение этого времени перед созданием и инициализацией нового экземпляра сервлета.

Обработка запросов

После того как сервлет будет благополучно создан и инициализирован, контейнер сервлета может использовать его для обработки клиентских запросов. Запросы представляются в виде объектов типа `ServletRequest`. Сервлет создает ответ на запрос путем вызова методов объекта `ServletResponse`. Эти два объекта передаются в виде параметров методу `service`, определенному в интерфейсе `Servlet`. В случае HTTP-запроса в качестве объектов контейнер предоставляет объекты типов `HttpServletRequest` и `HttpServletResponse`.

Экземпляр сервлета, созданного и инициализированного контейнером, может не обработать ни разу ни одного запроса. Контейнер сервлетов может одновременно послать несколько запросов методу `service` сервлета. Чтобы сервлет мог выполнять запросы параллельно, разработчик должен позаботиться о правильной работе одновременно выполняемых потоков, нескольких одновременно выполняемых методов `service`. Существует возможность создания однопотоковых сервлетов, для этого разработчик должен имплементировать интерфейс `SingleThreadModel`. Контейнер сервлетов будет либо сериализовать запросы к сервлету, либо создаст несколько экземпляров сервлета. Если сервлет является частью Web-приложения, которое отмечено как распределенное, то контейнер сервлетов будет создавать пул объектов сервлетов на каждой виртуальной машине Java, которая участвует в работе распределенного приложения. Если сервлет не имплементирует интерфейс `SingleThreadModel` и если метод `service` (или такие методы, как `doGet` или `doPost`, которые выполняются в составе метода `service`) определены внутри блока `Synchronized`, то контейнер сервлетов не сможет создать пул объектов сервлета, при этом он будет сериализовать запросы. Рекомендуется не синхронизовать метод `service` и методы, которые используются в нем, поскольку это существенным образом отражается на скорости выполнения приложения.

Исключения, возникающие во время обработки запросов

В процессе обработки запроса могут возникнуть исключения `ServletException` и `UnavailableException`. Ошибка `ServletException` сообщает о наличии исключительной ситуации, появляющейся при обра-

ботке запроса. Контейнер должен предпринять соответсвующие действия для удаления запроса. Ошибка `UnavailableException` сообщает о невозможности выполнить запрос временно или постоянно. Если запрос не может быть выполнен на постоянной основе, то контейнер должен удалить сервлет, выведя его из сервиса при помощи обращения к методу `destroy`. Все запросы, которые были удалены контейнером, должны привести к созданию ответа с кодом состояния `SC_NOT_FOUND` (404). Если сервлет недоступен временно (ошибка `UnavailableException`), то контейнер отклоняет запрос и возвращает код состояния `SC_SERVICE_UNAVAILABLE` (503), при этом посылается заголовок `Retry-After`, в котором указывается предполагаемое время, через которое сервлет станет доступен. Контейнер может игнорировать разницу между постоянным и временным состоянием недоступности сервлета, при этом вскоре возникновение ошибки `UnavailableExceptions` будет восприниматься как состояние, когда сервлет недоступен постоянно, следовательно, объект сервлета будет удаляться.

Прекращение работы сервлета

Контейнер сервлетов не должен хранить объекты сервлетов в течение определенного времени. Экземпляр сервлета может быть активен в течение нескольких миллисекунд или в течение всего времени работы контейнера (дни, месяцы, годы), а также в течение любого промежутка времени между этими двумя крайностями. После того как контейнер определяет, что экземпляр сервлета должен быть удален, контейнер вызывает метод `destroy`, описанный в интерфейсе `Servlet`. Это может произойти, например, при необходимости высвободить ресурсы памяти или при выключении самого контейнера.

Перед тем как будет выполнен метод `destroy`, необходимо позволить всем потокам, выполняющим метод `service`, завершить свою работу (либо превысить время, отведенное на выполнение этого метода). После выполнения метода `destroy` контейнер не сможет более направлять запросы этому объекту сервлета. Если контейнеру потребуется возобновить работу сервлета, то он сможет это сделать путем создания нового объекта сервлета. После завершения работы метода `destroy`, контейнер должен высвободить экземпляр сервлета, сделав его доступным для сборки мусора.

Сообщения HTTP

В этом разделе содержится подробная информация о сообщениях HTTP в соответствии с описанием протокола HTTP.

Типы сообщений

HTTP-сообщения состоят из запросов клиента серверу и ответов сервера клиенту.

```
HTTP-message   = Request | Response     ; HTTP/1.1 messages
```

Запросы и ответы состоят из строки, открывающей запрос или ответ, одного или нескольких заголовков, пустой строки, отделяющей заголовки от тела сообщения (строка, которая ничего не содержит перед CRLF) и тело сообщения, которое может отсутствовать:

```
generic-message = start-line
                  *message-header
                  CRLF
   [ message-body ]

  start-line    = Request-Line | Status-Line
```

Заголовки сообщений

Заголовки сообщений бывают заголовками общего назначения, заголовками запросов, заголовками ответов или заголовками сущностей (вставляемых в сообщение данных). Каждый заголовок состоит из имени, за которым ставится двоеточие, после которое следует значение. Имена чувствительны к регистру. Перед значением может быть вставлено произвольное количество пробелов (LWS, SP). Поля в заголовках могут занимать несколько строк, каждая дополнительная строка в поле заголовка должна начинаться с одного или нескольких SP или HT.

Формат заголовков

```
message-header = field-name ":" [ field-value ] CRLF
field-name     = token
field-value    = *(field-content | LWS)

field-content  = <значение поля — *TEXT или комбинация
                 из token, tspecials, quoted-string>
```

Порядок следования заголовков не имеет значения. Однако, как правило, заголовки общего назначения вставляют в первую очередь, затем используются заголовки запросов, заголовки ответов и заголовки сущностей. Заголовки с одним и тем же именем могут встречаться в сообщении только в том случае, если поле значения заголовка состоит из списка значений, разделяемых запятыми. Такие заголовки могут быть сведены в один заголовок без изменения семантики. При этом может быть значимым порядок появления одинаковых заголовков, если значим порядок появления значений, указываемых в таком заголовке через запятую.

Тело сообщения

Тело заголовка, если оно присутствует, используется для вложения в него сущности (данных), связанной с сообщением. Тело сообщения отличается от тела сущности только в том случае, если используется кодировка, тогда это должно быть отражено в заголовке `Transfer-Encoding`.

```
message-body = entity-body
    | <entity-body закодировано в соответствии с Transfer-Encoding>
```

Наличие тела сообщения в запросе связано с необходимостью использовать заголовки `Content-Length` и `Transfer-Encoding`. Именно их появление сигнализирует о наличии тела сообщения. Тело сообщения может быть вставлено в ответ в том случае, если запрос позволяет использование тел сущностей.

Возможность включения в ответ тела сообщения зависит от того, каков был запрос и каков статус ответа. Ответы на запросы типа `HEAD` не могут содержать тела сообщения. Это справедливо даже в том случае, если заголовок выглядит так, что сообщение содержит тело. Все ответы, содержащие статусы ответа `1xx` (*informational*), `204` (*no content*) и `304` (*not modified*) не могут содержать тело ответа. Все прочие ответы содержат тело ответа, которое, однако, может иметь нулевую длину.

Длина сообщения

При вставке в ответ тела сообщения, его длина определяется в соответствии со следующими правилами (в порядке старшинства).

Все ответы, которые не должны включать в себя тело сообщения (статускоды ответов `1xx`, `204`, `304` и ответы на запросы `HEAD`), заканчиваются с обнаружением первой пустой строки после заголовков вне зависимости от того, содержит ли заголовок поля сущности.

Если присутствует заголовок `Transfer-Encoding`, который указывает на кодировку "`chunked`", то длина определяется этой кодировкой.

Если присутствует заголовок `Content-Length`, то длина сообщения считается равной значению этого заголовка.

Если сообщение использует тип "`multipart/byteranges`", то длина сообщения определяется в соответствии с прилагаемым телом сообщения. Этот заголовок следует использовать только в том случае, если клиент умеет анализировать такой ответ.

Заголовки общего типа

Существует несколько заголовков общего типа, которые могут быть применимыми как к запросам, так и к ответам. Эти заголовки не относятся к сущностям, передаваемым вместе с сообщениями.

Заголовки общего типа

```
general-header = Cache-Control
               | Connection
               | Date
               | Pragma
               | Transfer-Encoding
               | Upgrade
               | Via
```

Запросы серверу

Запросы посылаются клиентом серверу, запрос посылает метод обращения к ресурсу, идентификатор ресурса и версию используемого протокола.

Формат запроса

```
Request       = Request-Line
                *(general-header
                 | request-header
                 | entity-header)
                CRLF
                [ message-body ]
```

Request-Line

`Request-Line` начинается с указания метода, за которым указывается `Request-URI` и версия протокола, после чего следует CRLF. Элементы отделены друг от друга пробелом (или несколькими пробелами) SP. Использовать CR или LF не разрешается (за исключением завершающего CRLF).

Формат запроса

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Методы

Метод соответствует методу, который должен быть вызван в запрашиваемом ресурсе для обработки запроса. Методы чувствительны к регистру.

```
Method         = "OPTIONS"
               | "GET"
               | "HEAD"
```

```
                | "POST"
                | "PUT"
                | "DELETE"
                | "TRACE"
                | extension-method
extension-method = token
```

Разрешенные для использования методы должны быть указаны в заголовке `Allow`. Статус-код ответа всегда содержит информацию о том, разрешен ли используемый метод на данном ресурсе. Если метод не разрешен, то сервер обязан возвратить заголовок с указанием статус-кода `405` (`Method Not Allowed`). Если метод известен, но не допускается его использование для ресурса запроса, если метод не распознан, то возвращается статус-код `501` (`Not Implemented`). Методы `GET` и `HEAD` должны поддерживаться всеми серверами.

Request-URI

Request-URI — это идентификатор запрашиваемого ресурса:

```
Request-URI  = "*" | absoluteURI | abs_path
```

Пример запроса

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
```

Заголовки запросов

Заголовки запросов позволяют клиенту передавать дополнительную информацию серверу вместе с запросом. Эта информация может включать в себя данные о клиенте, а также дополнительные данные, касающиеся самого запроса.

Формат заголовков запроса

```
request-header = Accept
               | Accept-Charset
               | Accept-Encoding
               | Accept-Language
               | Authorization
               | From
               | Host
               | If-Modified-Since
               | If-Match
               | If-None-Match
               | If-Range
```

```
                | If-Unmodified-Since
                | Max-Forwards
                | Proxy-Authorization
                | Range
                | Referer
                | User-Agent
```

Ответы сервера

Поле обработки запроса сервер отправляет клиенту ответ.

Формат ответа

```
Response      = Status-Line
                *(general-header
                 | response-header
                 | entity-header)
                CRLF
                [ message-body ]
```

Строка статус-кода

Эта строка используется для указания статус-кода.

Формат строки статус-кода

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Статус-код *Status Code* и пояснение *Reason-Phrase*

Статус-код — это трехзначное число. Пояснение `Reason-phrase` — строка, поясняющая значение этого кода. Первая цифра статус-кода имеет следующий смысл:

- `1xx: Informational` — запрос получен, вычисления продолжаются;
- `2xx: Success` — запрос получен, распознан и принят к исполнению;
- `3xx: Redirection` — для завершения запроса необходимо предпринять дополнительные действия;
- `4xx: Client Error` — запрос содержит неверный синтаксис и не может быть выполнен;
- `5xx: Server Error` — сервер не смог выполнить запрос.

Статус-коды и строка пояснения

```
Status-Code     = "100"   ; Continue
                | "101"   ; Switching Protocols
                | "200"   ; OK
                | "201"   ; Created
                | "202"   ; Accepted
                | "203"   ; Non-Authoritative Information
                | "204"   ; No Content
                | "205"   ; Reset Content
                | "206"   ; Partial Content
                | "300"   ; Multiple Choices
                | "301"   ; Moved Permanently
                | "302"   ; Moved Temporarily
                | "303"   ; See Other
                | "304"   ; Not Modified
                | "305"   ; Use Proxy
                | "400"   ; Bad Request
                | "401"   ; Unauthorized
                | "402"   ; Payment Required
                | "403"   ; Forbidden
                | "404"   ; Not Found
                | "405"   ; Method Not Allowed
                | "406"   ; Not Acceptable
                | "407"   ; Proxy Authentication Required
                | "408"   ; Request Time-out
                | "409"   ; Conflict
                | "410"   ; Gone
                | "411"   ; Length Required
                | "412"   ; Precondition Failed
                | "413"   ; Request Entity Too Large
                | "414"   ; Request-URI Too Large
                | "415"   ; Unsupported Media Type
                | "500"   ; Internal Server Error
                | "501"   ; Not Implemented
                | "502"   ; Bad Gateway
                | "503"   ; Service Unavailable
                | "504"   ; Gateway Time-out
                | "505"   ; HTTP Version not supported
```

```
                | extension-code

extension-code = 3DIGIT

Reason-Phrase  = *<TEXT, excluding CR, LF>
```

Заголовки ответа

Заголовки ответа позволяют серверу отправлять дополнительную информацию клиенту.

Заголовки ответов

```
response-header = Age
                | Location
                | Proxy-Authenticate
                | Public
                | Retry-After
                | Server
                | Vary
                | Warning
                | WWW-Authenticate
```

Сущности Entity

И запросы, и ответы в своем составе могут передавать сущности, если наличие сущности в сообщения не противоречит используемому методу. Сущность состоит из заголовков и тела сущности.

Заголовки сущностей

Заголовки сущностей содержат дополнительную метаинформацию о теле сущности, если оно есть, или о ресурсе, с которым связан запрос.

Заголовки для сущностей

```
entity-header  = Allow
               | Content-Base
               | Content-Encoding
               | Content-Language
               | Content-Length
               | Content-Location
               | Content-MD5
```

```
                | Content-Range
                | Content-Type
                | ETag
                | Expires
                | Last-Modified
                | extension-header
extension-header = message-header
```

Тело сущности

Тело сущности передается в следующем формате:

```
entity-body    = *OCTET
```

- **Тип данных сущности**

  Если в сообщении присутствует тело сущности, то необходимо указать тип данных, используемых в теле сущности, для этого используются заголовки `Content-Type` и `Content-Encoding`. Схема модели выглядит следующим образом:

  ```
  entity-body := Content-Encoding(Content-Type(data))
  ```

  `Content-Type` описывает тип, а `Content-Encoding` описывает дополнительно использованный тип кодировки, примененный к данным. Не существует кодировки, используемой по умолчанию.

- **Длина сущности**

  Длина тела сущности — это длина тела сообщения.

Методы HTTP

Ниже приведены методы, описанные в спецификации протокола HTTP/1.1.

Безопасные и идемпотентные методы

Имплементация методов осуществляется таким образом, что методы `GET` и `HEAD` могут быть использованы только для получения ответов ресурсов. Такие методы являются безопасными. Методы `POST`, `PUT` и `DELETE` принципиально могут быть потенциально небезопасными, поскольку могут выполнять действия, которые в некотором смысле могут содержать угрозу безопасности. Конечно, нельзя быть полностью уверенным в том, что реализация даже того же метода `GET` не приведет к небезопасным действиям. Важным здесь является то, что безопасные методы не могут вызвать небезопасных "побочных" эффектов при обработке соответствующих им клиентских запросов.

Большинство методов должно обладать свойством идемпотентности, при этом "побочный" эффект множественных запросов эквивалентен эффекту,

возникающему от одного запроса. К идемпотентным методам относятся методы GET, HEAD, PUT и DELETE.

Метод *OPTIONS*

Запрос OPTIONS запрашивает информацию о возможностях, предоставляемых для обмена запросами и ответами для данного ресурса. Клиент может использовать этот метод для того, чтобы определить опции и требования для взаимодействия с сервером. Серверный ответ не должен включать в себя информацию о сущности, кроме той, которая может касаться информации о коммуникации (т. е. заголовок Allow может быть использован, но заголовок Content-Type использовать нельзя). Ответы на этот запрос не кэшируются.

Если в качестве идентификатора ресурса используется звездочка (*), то запрос OPTIONS относится к серверу в целом. Ответ с кодом 200 должен содержать заголовки, в которых указываются свойства сервера, включая поддержку таких расширений, которые не содержатся в стандартной спецификации.

Если в качестве идентификатора ресурса используется не звездочка, то запрос относится к конкретному ресурсу. В этом случае ответ с кодом 200 должен содержать информацию о возможностях сервера, применимых к данному ресурсу.

Метод *GET*

Метод GET предполагает получение информации от сервера, которая передается в форму сущности. Если идентификатор ресурса указывает на процесс, создающий данные, то создаваемые данные передаются клиенту в виде сущности. При этом передается результат работы процесса, но не программный код, соответствующий процессу. Метод может иметь опцию условного выполнения, если запрос содержит заголовки с указанием If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match и If-Range. Такой метод будет выполнен только в том случае, если указанное условие выполняется. Ответы, посылаемые на запрос GET, кэшируются, это уменьшает загрузку сети. Метод GET может быть выполнен частично, если запрос содержит заголовок Range.

Запрос *HEAD*

Запрос HEAD идентичен запросу GET с той разницей, что сервер не может включить в ответ тело сообщения. Этот метод может быть использован для того, чтобы протестировать гиперссылки на доступность, работоспособность и наличие модификаций. Ответы могут кэшироваться, чтобы обновить информацию, полученную по такому запросу ранее.

Запрос *POST*

Метод `POST` предполагает, что сервер получит от клиента информацию, которая помещается в тело сообщения в виде сущности и передается для обработки указанному идентификатору ресурса. Этот метод используется для решения таких задач, как

- аннотирование существующих ресурсов;
- размещение сообщений на досках объявлений, группах новостей, списках рассылки и т. п.;
- отправка блоков данных, например, отправка данных из HTML-форм процессам, обрабатывающим данные;
- внесение изменений в базы данных.

Код ответа сервера может быть `200`, что означает, что запрошенный ресурс существует на сервере, если ресурс не возвращает в ответе сущности, то используется код `204` (`No Content`). Если ресурс на сервере будет создан заново, то возвращается код `201` (`Created`), при этом возвращается сущность, содержащая статус и заголовок `Location`. Ответы не кэшируются, если ответ не содержит заголовков `Cache-Control` или `Expiress`.

Запрос *PUT*

Метод `PUT` предполагает, что передаваемая сущность будет сохранена под указанным идентификатором `Request-URI`. Если запрос с таким идентификатором уже существует, то передаваемая сущность будет считаться обновленной версией ресурса. Если новый ресурс будет создан на сервере, то сервер должен послать ответ с кодом `201` (`Created`). Если модифицирован существующий ресурс, то должен быть возвращен код `200` (`OK`) или код `204` (`No Content`), что обозначает успешное создание нового ресурса.

Запрос *DELETE*

Запрос `DELETE` используется для удаления ресурса по указанному `Request-URI`. Автор запроса не может быть уверен в том, что ресурс удален даже тогда, когда получен положительный статус-код в ответе. Операция на сервере может быть удалена вручную, однако при положительном ответе на запрос сервер удаляет ресурс, перемещая его в недоступное место. В качестве ответа могут быть отправлены коды `200` (`OK`), `202` (`Accepted`) или `204` (`No Content`). Ответ не кэшируется.

Запрос *TRACE*

Запрос `TRACE` позволяет клиенту увидеть, что получает сервер на противоположном конце, и использовать эту информацию для диагностики и тестирования.

Пакеты. Интерфейсы. Классы

Пакет *javax.servlet*

Пакет `javax.servlet` содержит классы и интерфейсы, определяющие взаимодействие между сервлетами (классами сервлетов) и окружением контейнера сервлетов (табл. П2.1).

Таблица П2.1. *Интерфейсы пакета* `javax.servlet`

Интерфейс	Описание
`Filter`	Фильтр — объект, который производит фильтрацию запроса к ресурсу, ответа клиенту, или и того и другого. Фильтрация происходит при помощи метода `doFilter`
`FilterChain`	`FilterChain` — это объект, который предоставляется контейнером сервлетов для работы с цепочкой обращений из фильтрованного запроса к ресурсу
`FilterConfig`	Объект конфигурации фильтра, используется контейнером сервлетов для сбора информации во время инициализации и передач ее фильтру
`RequestDispatcher`	Определяет объект, который получает запросы клиента и перенаправляет их другим ресурсам, например, сервлетам, HTML-файлам и JSP-страницам
`Servlet`	Содержит методы, которые должны быть имплементированы во всех сервлетах
`ServletConfig`	Объект конфигурации сервлета, используется контейнером сервлета для передачи информации сервлету во время инициализации
`ServletContext`	Содержит методы, при помощи которых сервлет общается с контейнером для получения MIME-типов файлов, перенаправления запросов, записи информации в файлы обращений к серверу
`ServletContextAttributeListener`	Объект получает информацию об изменениях в списке атрибутов контекста сервлета или Web-приложения
`ServletContextListener`	Объект получает информацию об изменениях в контексте сервлета или Web-приложения, частью которого он является

Таблица П2.1 *(окончание)*

Интерфейс	Описание
`ServletRequest`	Определяет объект, который передает информацию клиентского запроса сервлету
`ServletResponse`	Определяет объект, который передает информацию от сервлета в ответ на запрос
`SingleThreadModel`	Используется для создания сервлетов, которые позволяют работать исключительно в однопоточном режиме

Список классов, входящих в состав пакета `javax.servlet`, приведен в табл. П2.2.

Таблица П2.2. *Классы пакета* `javax.servlet`

Класс	Описание
`GenericServlet`	Определяет независимый от протокола сервлет
`ServletContextAttributeEvent`	Класс, содержащий события, используемые для уведомления об изменениях, происходящих с атрибутами контекста сервлета или Web-приложения
`ServletContextEvent`	Класс, содержащий события, используемые для уведомления об изменениях в контексте сервлета или в Web-приложении
`ServletInputStream`	Поток ввода для чтения двоичных данных клиентского запроса, в том числе метод `readLine` для чтения одной строки
`ServletOutputStream`	Поток вывода для отправки двоичных данных клиенту
`ServletRequestWrapper`	Удобная имплементация интерфейса `ServletRequest`, которая может быть использована для создания подклассов для работы с запросами к сервлету
`ServletResponseWrapper`	Имплементация интерфейса `ServletResponse` для работы с ответами сервлета клиенту

Табл. П2.3 содержит информацию об исключительных ситуациях, возникающих при работе с сервлетами.

Таблица П2.3. *Исключительные ситуации в пакете* `javax.servlet`

Исключение	Описание
`ServletException`	Общая ошибка, произошедшая во время работы сервлета
`UnavailableException`	Ошибка показывает, что сервлет временно или постоянно неработоспособен

Пакет *javax.servlet.http*

Пакет `javax.servlet.http` содержит классы и интерфейсы, определяющие взаимодействие класса сервлета с контейнером сервлетов при работе с протоколом HTTP (табл. П2.4).

Таблица П2.4. *Интерфейсы пакета* `javax.servlet.http`

Интерфейс	Описание
`HttpServletRequest`	Расширение интерфейса `ServletRequest` для работы с информацией HTTP-запросов
`HttpServletResponse`	Расширение интерфейса `ServletResponse` для работы с информацией ответов на HTTP-запросы
`HttpSession`	Интерфейс для работы с сессиями
`HttpSessionActivationListener`	Объекты интерфейса связываются с сессией и прослушивают события контейнера об активации сессии и выходе из сессии
`HttpSessionAttributeListener`	Интерфейс прослушивателя имплементируется для создания объектов, получающих сообщения об изменениях в списке атрибутов сессии или Web-приложения
`HttpSessionBindingListener`	Заставляет сообщать объекту о наложении связи с определенной сессией или о снятии этой связи
`HttpSessionContext`	Устарел
`HttpSessionListener`	Объекты получают сообщения об изменениях в списке активных сессий или Web-приложений

Перечень классов пакета `javax.servlet.http` приводится в табл. П2.5.

Таблица П2.5. *Классы пакета* `javax.servlet.http`

Класс	Описание
`Cookie`	Создает `cookie`
`HttpServlet`	Абстрактный класс для создания классов HTTP-сервлетов, используемых на Web-сайтах
`HttpServletRequestWrapper`	Имплементация интерфейса `HttpServletRequest`
`HttpServletResponseWrapper`	Имплементация интерфейса `HttpServletResponse`
`HttpSessionBindingEvent`	События этого типа направляются объекту, имплементирующему интерфейс `HttpSessionBindingListener` во время установления связи с сессией, или удаления связи с сессией, или объекту, имплементирующему `HttpSessionAttributeListener` во время изменения атрибутов (связь с сессией, удаление связи с сессией)
`HttpSessionEvent`	Класс событий, которые возникают при внесении изменений в объект сессий во время работы Web-приложения
`HttpUtils`	Устарел

Табл. П2.6 содержит описания исключительных ситуаций, возникающих при работе с объектами пакета `javax.servlet.http`.

Таблица П2.6. *Исключительные ситуации в пакете* `javax.servlet.http`

Исключение	Описание
`JspException`	Общая исключительная ситуация
`JspTagException`	Ошибка, которая может возникнуть во время работы обработчика ярлыков

Пакет *javax.servlet.jsp*

Пакет `javax.servlet.jsp` содержит интерфейсы и классы для осуществления взаимодействия класса JSP-страницы и JSP-контейнера (табл. П2.7).

Таблица П2.7. *Интерфейсы пакета* `javax.servlet.jsp`

Интерфейс	Описание
`HttpJspPage`	Интерфейс `HttpJspPage` описывает работу класса, имплементирующего JSP-страницу при использовании протокола HTTP
`JspPage`	Интерфейс `JspPage` описывает функциональность класса, имплементирующего JSP-страницу вне зависимости от протокола

В пакете `javax.servlet.jsp` описан набор интерфейсов, используемых при работе с jsp (табл. П2.8).

Таблица П2.8. *Интерфейсы пакета* `javax.servlet.jsp`

Интерфейс	Описание
`JspEngineInfo`	Абстрактный класс `JspEngineInfo` содержит текущую информацию о JSP
`JspFactory`	Абстрактный класс `JspFactory` определяет методы, которые могут использоваться во время выполнения JSP-страницы для создания экземпляров интерфейсов и классов, используемых при работе
`JspWriter`	Вывод информации может быть осуществлен с использованием объекта `JspWriter`
`PageContext`	Объект `PageContext` предоставляет доступ ко всем пространствам имен на JSP-странице, в том числе к атрибутам страницы

При работе с пакетом `javax.servlet.jsp` могут возникнуть исключительные ситуации, описанные в табл. П2.9.

Таблица П2.9. *Исключительные ситуации в пакете* `javax.servlet.jsp`

Исключение	Описание
`JspException`	Общая исключительная ситуация
`JspTagException`	Ошибка, появляющаяся во время работы обработчика ярлыков

Интерфейс `servlet` содержит методы, вызываемые в те или иные моменты жизненного цикла сервлета, в его состав входят также вспомогательные методы, в том числе методы, помогающие сервлету взаимодействовать с окружением, предоставляемым контейнером сервлетов (табл. П2.10).

Таблица П2.10. *Методы интерфейса* `Servlet`

Тип	Метод
`void`	`destroy()` Вызывается контейнером сервлета перед выводом сервлета из работы
`ServletConfig`	`getServletConfig()` Возвращает объект `ServletConfig`, содержащий параметры инициализации сервлета
`java.lang.String`	`getServletInfo()` Возвращает информацию о сервлете: автора, версию сервлета, copyright

Таблица П2.10 *(окончание)*

Тип	Метод
`void`	`init(ServletConfig config)` Вызывается контейнером сервлета при инициализации сервлета
`void`	`service(ServletRequest req, ServletResponse res)` Вызывается контейнером сервлета для обработки запроса и отправки ответа

Класс `GenericServlet` предоставляет основные методы сервлета (П2.11).

Таблица П2.11. Методы класса `GenericServlet`

Тип	Метод
`Void`	`destroy()` Вызывается контейнером сервлета для уничтожения сервлета
`java.lang.String`	`getInitParameter(java.lang.String name)` Возвращает строку `String`, содержащую значение параметра инициализации или `null`
`java.util.Enumeration`	`getInitParameterNames()` Возвращает имена параметров инициализации в виде `Enumeration` объектов типа `String` или пустой `Enumeration`
`ServletConfig`	`getServletConfig()` Возвращает объект `ServletConfig`
`ServletContext`	`getServletContext()` Возвращает ссылку на `ServletContext` для сервлета
`java.lang.String`	`getServletInfo()` Возвращает информацию о сервлете (автор, версия, copyright)
`java.lang.String`	`getServletName()` Возвращает имя экземпляра сервлета
`Void`	`init()` Метод может быть переопределен, используется для того, чтобы не вызывать метод `super.init(config)`
`Void`	`init(ServletConfig config)` Метод вызывается контейнером сервлета при вводе сервлета в работу
`Void`	`log(java.lang.String msg)` Записывает сообщение в файл логов сервлета, сообщение начинается с имени сервлета

Таблица П2.11 *(окончание)*

Тип	Метод
`Void`	`log(java.lang.String message, java.lang.Throwable t)` Пишет сообщение с объяснением исключительной ситуации в файл логов
`Abstract void`	`service(ServletRequest req, ServletResponse res)` Вызывается контейнером сервлета для выполнения запроса и создания ответа

Интерфейс `ServletRequest` содержит методы для работы с клиентским запросом (табл. П2.12).

Таблица П2.12. Методы интерфейса `ServletRequest`

Тип	Метод
`java.lang.Object`	`getAttribute(java.lang.String name)` Возвращает значение указанного атрибута в виде объекта `Object` или `null`
`java.util.Enumeration`	`getAttributeNames()` Возвращает `Enumeration` с именами атрибутов сервлета
`java.lang.String`	`getCharacterEncoding()` Имя кодировки символов
`Int`	`getContentLength()` Длина (в байтах) тела запроса. `(-1)` — если длина неизвестна
`java.lang.String`	`getContentType()` Возвращает тип MIME, используемый в теле запроса, или `null`
`ServletInputStream`	`getInputStream()` Получает тело запроса с использованием `ServletInputStream`
`java.util.Locale`	`getLocale()` Возвращает предпочтительный `Locale`, заданный в клиенте в соответствии с заголовком `Accept-Language`
`java.util.Enumeration`	`getLocales()` Возвращает `Enumeration` объектов типа `Locale` в убывающем порядке от наиболее предпочтительного на основе заголовка `Accept-Language`

Таблица П2.12 *(продолжение)*

Тип	Метод
`java.lang.String`	`getParameter(java.lang.String name)` Возвращает значение параметра в виде строки `String`, или `null`
`java.util.Map`	`getParameterMap()` Возвращает `java.util.Map` для параметров запроса
`java.util.Enumeration`	`getParameterNames()` Возвращает `Enumeration` объектов `String` с именами параметров запроса
`java.lang.String[]`	`getParameterValues(java.lang.String name)` Возвращает массив объектов `String` со всеми значениями для указанного имени параметра или `null`, если параметр не существует
`java.lang.String`	`getProtocol()` Возвращает имя и версию проткола в форме *protocol/majorVersion.minorVersion*, например, HTTP/1.1
`java.io.BufferedReader`	`getReader()` Получает тело запроса в виде символов с использованием `BufferedReader`
`java.lang.String`	`getRealPath(java.lang.String path)` Устарел. Используется `ServletContext.getRealPath(java.lang.String)`
`java.lang.String`	`getRemoteAddr()` Возвращает адрес IP-клиента, пославшего запрос
`java.lang.String`	getRemoteHost`()` Возвращает полное имя клиента, пославшего запрос
`RequestDispatcher`	`getRequestDispatcher(java.lang.String path)` Возвращает объект `RequestDispatcher`, который используется вместе с ресурсом по указанному пути
`java.lang.String`	`getScheme()` Возвращает имя `scheme` для выполнения запроса (например, `http`, `https`, `ftp`)
`java.lang.String`	`getServerName()` Возвращает имя хоста сервера, получившего запрос
`int`	`getServerPort()` Возвращает номер порта, на котором был получен запрос

Таблица П2.12 *(окончание)*

Тип	Метод
`boolean`	`isSecure()` Возвращает логическое значение, показывающее, был ли получен запрос по безопасному каналу, например, по HTTPS
`void`	`removeAttribute(java.lang.String name)` Удаляет атрибут из запроса
`void`	`setAttribute(java.lang.String name, java.lang.Object o)` Задает атрибут в запросе
`void`	`setCharacterEncoding(java.lang.String env)` Переустанавливает имя кодировки запроса

В интерфейсе `ServletResponce` содержатся методы для работы с ответом сервлета клиенту (табл. П2.13).

Таблица П2.13. Методы интерфейса `ServletResponce`

Тип	Метод
`void`	`flushBuffer()` Отсылает содержимое буфера клиенту
`int`	`getBufferSize()` Возвращает реальный размер буфера, посылаемого клиенту
`java.lang.String`	`getCharacterEncoding()` Возвращает название набора символов, используемого в теле ответа
`java.util.Locale`	`getLocale()` Возвращает объект `Locale`, связанный с ответом
`ServletOutputStream`	`getOutputStream()` Возвращает `ServletOutputStream` для записи двоичных данных ответа
`java.io.PrintWriter`	`getWriter()` Возвращает объект `PrintWriter` для отправки символов клиенту
`boolean`	`isCommitted()` Возвращает логическое значение, отправлен ли ответ
`void`	`reset()` Очищает буфер от всех данных, в том числе от статуса кода и заголовков

Таблица П2.13 *(окончание)*

Тип	Метод
`void`	`resetBuffer()` Очищает буфер ответа, но оставляет заголовки и статус-код
`void`	`setBufferSize(int size)` Устанавливает предпочтительный размер буфера для тела ответа
`void`	`setContentLength(int len)` Устанавливает длину тела в ответе HTTP, этот метод устанавливает значение, используемое в заголовке `Content-Length`
`void`	`setContentType(java.lang.String type)` Устанавливает тип содержимого, посылаемого в ответе клиенту
`void`	`setLocale(java.util.Locale loc)` Устанавливает `locale` для ответа, в том числе устанавливает значения и других заголовков

Диспетчер запросов перенаправляет клиентский запрос для обработки с помощью тех или иных ресурсов (табл. П2.14).

Таблица П2.14. Методы интерфейса `RequestDispatcher`

Тип	Метод
`void`	`forward(ServletRequest request, ServletResponse response)` Перенаправляет запрос от сервлета другому ресурсу (сервлету, странице JSP, файлу HTML)
`void`	`include(ServletRequest request, ServletResponse response)` Вставляет в ответ содержимое ресурса (сервлета, страницы JSP, страницы HTML)

Класс `HttpServlet` содержит набор методов для обработки различных типов HTTP-запросов (табл. П2.15).

Таблица П2.15. Методы класса `HttpServlet`

Тип	Метод
`protected void`	`doDelete(HttpServletRequest req, HttpServletResponse resp)` Вызывается сервером посредством метода `service` для обработки запросов `DELETE`

Таблица П2.15 *(окончание)*

Тип	Метод
protected void	doGet(HttpServletRequest req, HttpServletResponse resp) Используется для обработки запросов GET
protected void	doHead(HttpServletRequest req, HttpServletResponse resp) Обрабатывает запрос HEAD
protected void	doOptions(HttpServletRequest req, HttpServletResponse resp) Обрабатывает запрос OPTIONS
protected void	doPost(HttpServletRequest req, HttpServletResponse resp) Обрабатывает запрос POST
protected void	doPut(HttpServletRequest req, HttpServletResponse resp) Обрабатывает запрос PUT
protected void	doTrace(HttpServletRequest req, HttpServletResponse resp) Обрабатывает запрос TRACE
protected long	getLastModified(HttpServletRequest req) Возвращает время последней модификации объекта HttpServletRequest в миллисекундах с 1 января 1970 года
protected void	service(HttpServletRequest req, HttpServletResponse resp) Выполняет обработку стандартных HTTP-запросов, перенаправляя их соответствующему методу do*XXX*
void	service(ServletRequest req, ServletResponse res) Перенаправляет запрос

Интерфейс HttpServletRequest используется для работы с HTTP-ответами (табл. П2.16).

Таблица П2.16. *Переменные интерфейса* HttpServletRequest

Тип	Переменная
static java.lang.String	BASIC AUTH Базовая аутентификация
static java.lang.String	CLIENT CERT AUTH Базовая аутентификация

Таблица П2.16 *(окончание)*

Тип	Переменная
`static java.lang.String`	`DIGEST_AUTH` Идентификатор базовой аутентификации
`static java.lang.String`	`FORM_AUTH`

Методы интерфейса `HttpServletRequest` служат для работы с HTTP-запросом, в том числе используются при работе с различными компонентами HTTP-запроса (табл. П2.17).

Таблица П2.17. *Методы интерфейса* `HttpServletRequest`

Тип	Метод
`java.lang.String`	`getAuthType()` Возвращает тип аутентификации
`java.lang.String`	`getContextPath()` Возвращает часть `URI`, которая указывает на контекст запроса
`Cookie[]`	`getCookies()` Возвращает массив объектов `Cookie`, посылаемых клиентом с запросом
`long`	`getDateHeader(java.lang.String name)` Возвращает значение соответствующего заголовка в виде `long`
`java.lang.String`	`getHeader(java.lang.String name)` Возвращает заголовок запроса в виде строки `String`
`java.util.Enumeration`	`getHeaderNames()` Возвращает `enumeration` со всеми именами заголовков, содержащимися в запросе
`java.util.Enumeration`	`getHeaders(java.lang.String name)` Возвращает все значения соответствующего заголовка в виде `Enumeration` объектов `String`
`int`	`getIntHeader(java.lang.String name)` Возвращает значение заголовка как `int`
`java.lang.String`	`getMethod()` Возвращает HTTP-метод запроса (например, `GET`, `POST` и `PUT`)
`java.lang.String`	`getPathInfo()` Возвращает дополнительную информацию, содержащуюся в URL

Таблица П2.17 *(продолжение)*

Тип	Метод
`java.lang.String`	`getPathTranslated()` Возвращает дополнительную информацию после имени сервлета, но до строки запроса
`java.lang.String`	`getQueryString()` Возвращает строку запроса, которая содержит часть URL после пути
`java.lang.String`	`getRemoteUser()` Возвращает имя `login`, если пользователь идентифицирован, или `null`
`java.lang.String`	`getRequestedSessionId()` Возвращает идентификатор сессии, указанный клиентом
`java.lang.String`	`getRequestURI()` Возвращает часть URL начиная с имени протокола до строки запроса в первой строке HTTP-запроса
`java.lang.StringBuffer`	`getRequestURL()` Восстанавливает URL, использованный клиентом для создания запроса
`java.lang.String`	`getServletPath()` Возвращает часть URL, которая вызывает сервлет
`HttpSession`	`getSession()` Возвращает текущую сессию, связанную с запросом, если запрос еще не имеет сессии, то сессия будет создана
`HttpSession`	`getSession(boolean create)` Возвращает текущую сессию `HttpSession`, связанную с запросом, если сессии нет и `create` имеет значение `true`, то создает новую сессию
`java.security.Principal`	`getUserPrincipal()` Возвращает объект `java.security.Principal` с именем пользователя
`boolean`	`isRequestedSessionIdFromCookie()` Проверяет, является ли запрошенный идентификатор сессии ID полученным посредством `cookie`
`boolean`	`isRequestedSessionIdFromUrl()` Устарел. Используется `isRequestedSessionIdFromURL()`
`boolean`	`isRequestedSessionIdFromURL()` Проверяет, получен ли идентификатор сессии `ID` в запросе в виде составляющей части URL

Таблица П2.17 *(окончание)*

Тип	Метод
`boolean`	`isRequestedSessionIdValid()` Проверят, является ли запрошенный идентификатор сессии действительным
`boolean`	`isUserInRole(java.lang.String role)` Возвращает логическое значение, истина возвращается тогда, когда опознанный пользователь обладает указанной ролью

Интерфейс `HttpServletResponce` служит для работы с HTTP-ответом (табл. П2.18).

Таблица П2.18. *Переменные интерфейса* `HttpServletResponce`

Тип	Переменные (значения для статус-кодов)
`static int`	`SC_ACCEPTED` Код 202. Запрос принят, но обработка не завершена
`static int`	`SC BAD GATEWAY` Код 502. Неверный ответ прокси-сервера или шлюза
`static int`	`SC BAD REQUEST` Код 400. Клиентский запрос синтаксически неверен
`static int`	`SC CONFLICT` Код 409. Запрос не может быть завершен ввиду текущего состояния ресурса
`static int`	`SC_CONTINUE` Код 100. Клиент должен продолжить работу с запросом
`static int`	`SC_CREATED` Код 201. Запрос обработан, на сервер создан новый ресурс
`static int`	`SC_EXPECTATION_FAILED` Код 417. Сервер не удовлетворяет условиям, указанным в заголовке Expect
`static int`	`SC_FORBIDDEN` Код 403. Сервер понял запрос, но отказался его выполнять
`static int`	`SC GATEWAY TIMEOUT` Код 504. Сервер не получил своевременного ответа от другого сервера (при работе с прокси или шлюзом)
`static int`	`SC GONE` Код 410. Ресурс недоступен, переадресация не установлена
`static int`	`SC_HTTP_VERSION_NOT_SUPPORTED` Код 505. Сервер не поддерживает или отказывается работать с указанной версией протокола HTTP

Таблица П2.18 *(продолжение)*

Тип	Переменные (значения для статус-кодов)
`static int`	`SC_INTERNAL_SERVER_ERROR` Код 500. Ошибка сервера, запрос не выполнен
`static int`	`SC_LENGTH_REQUIRED` Код 411 запрос не может быть выполнен без указания *`Content-Length`*
`static int`	`SC_METHOD_NOT_ALLOWED` Код 405. Метод, указанный в *`Request-Line`* в указанном ресурсе (адрес *`Request-URI`*), не разрешен
`static int`	`SC_MOVED_PERMANENTLY` Код 301. Ресурс постоянно удален в новое место
`static int`	`SC_MOVED_TEMPORARILY` Код 302. Ресурс временно удален в новое место
`static int`	`SC_MULTIPLE_CHOICES` Код 300. Запрошенный ресурс соответствует любому из набора представлений, каждое из которых располагается по собственному адресу
`static int`	`SC_NO_CONTENT` Код 204. Запрос выполнен, но новой информации для возврата нет
`static int`	`SC_NON_AUTHORITATIVE_INFORMATION` Код 203. Метаинформация, посланная сервером, не относится к серверу (создана не сервером)
`static int`	`SC_NOT_ACCEPTABLE` Код 406. Ресурс не может создавать ответ, который бы соответствовал посланным клиентом заголовкам `Accept`
`static int`	`SC_NOT_FOUND` Код 404. Запрошенный ресурс недоступен
`static int`	`SC_NOT_IMPLEMENTED` Код 501. Сервер не поддерживает функций, требуемых для выполнения запроса
`static int`	`SC_NOT_MODIFIED` Код 304. Обнаружен условный запрос `GET`, ресурс доступен, но не изменен
`static int`	`SC_OK` Код 200. Запрос благополучно выполнен
`static int`	`SC_PARTIAL_CONTENT` Код 206. Сервер выполнил частичный запрос `GET`
`static int`	`SC_PAYMENT_REQUIRED` Код 402. Заразервировано
`static int`	`SC_PRECONDITION_FAILED` Код 412. Условия, полученные с заголовками, привели к появлению значений `false` при обработке сервером

Таблица П2.18 *(окончание)*

Тип	Переменные (значения для статус-кодов)
static int	SC_PROXY_AUTHENTICATION_REQUIRED Код 407. Клиент должен быть идентифицирован
static int	SC_REQUEST_ENTITY_TOO_LARGE Код 413. Сервер отказывается обработать запрос ввиду того, что сущность в запросе превышает допустимый размер
static int	SC_REQUEST_TIMEOUT Код 408. Клиент не направил запрос в течение промежутка времени, которое было отведено серверу для ожидания
static int	SC_REQUEST_URI_TOO_LONG Код 414. Сервер не обработал запрос, т. к. адрес Request-URI оказался длиннее того, что сервер в состоянии интерпретировать
static int	SC_REQUESTED_RANGE_NOT_SATISFIABLE Код 416. Сервер не может обработать запрос
static int	SC_RESET_CONTENT Код 205. Агент должен обновить вид документа, который отправляет запрос
static int	SC_SEE_OTHER Код 303. Ответ на запрос может быть получен по другому URI
static int	SC_SERVICE_UNAVAILABLE Код 503. HTTP-сервер временно перегружен и не может выполнить запрос
static int	SC_SWITCHING_PROTOCOLS Код 101. Сервер меняет протокол в соответствии с заголовком Upgrade
static int	SC_TEMPORARY_REDIRECT Код 307. Запрошенный ресурс временно расположен по другому URI
static int	SC_UNAUTHORIZED Код 401. Запрос требует аутентификации
static int	SC_UNSUPPORTED_MEDIA_TYPE Код 415. Сервер отказывается обработать запрос, поскольку сущность запроса записана в формат, не поддерживается ресурсом с учетом указанного метода
static int	SC_USE_PROXY Код 305. Запрошенный ресурс должен быть доступен только через прокси-сервер, адрес которого указан в заголовке Location

Методы интерфейса HttpServletResponce используются для создания различных элементов, посылаемых в составе HTTP-ответа (табл. П2.19).

Таблица П2.19. *Методы интерфейса* `HTTPServletResponce`

Тип	Метод
`void`	`addCookie(Cookie cookie)` Вставляет Cookie в ответ
`void`	`addDateHeader(java.lang.String name, long date)` Вставляет заголовок ответа с именем и датой
`void`	`addHeader(java.lang.String name, java.lang.String value)` Вставляет в ответ заголовок с указанным именем и значением
`void`	`addIntHeader(java.lang.String name, int value)` Вставляет в ответ заголовок с именем и целым значением
`boolean`	`ContainsHeader(java.lang.String name)` Возвращает логическое значение, существует ли заголовок с указанным именем
`java.lang.String`	`encodeRedirectUrl(java.lang.String url)` Устарел. Используется `encodeRedirectURL(String url)`
`java.lang.String`	`encodeRedirectURL(java.lang.String url)` Кодирует указанный URL для использования в методе `sendRedirect`, если кодировка не требуется, то оставляет его неизмененным
`java.lang.String`	`encodeUrl(java.lang.String url)` Устарел. Используется `encodeURL(String url)`
`java.lang.String`	`encodeURL(java.lang.String url)` Кодирует URL
`void`	`sendError(int sc)` Посылает клиенту ответ об ошибке с указанием заданного кода ошибки и очищает буфер
`void`	`sendError(int sc, java.lang.String msg)` Посылает клиенту ответ об ошибке с указанием заданного кода ошибки и очищает буфер. Посылает сообщение
`void`	`sendRedirect(java.lang.String location)` Посылает адрес для перенаправления запроса
`void`	`setDateHeader(java.lang.String name, long date)` Устанавливает заголовок ответа с заданным именем и датой
`void`	`setHeader(java.lang.String name, java.lang.String value)` Устанавливает заголовок ответа с именем и значением
`void`	`setIntHeader(java.lang.String name, int value)` Устанавливает заголовок ответа с заданным именем и целым значением

Таблица П2.19 *(окончание)*

Тип	Метод
`void`	`setStatus(int sc)` Устанавливает статус-код для ответа
`void`	`setStatus(int sc, java.lang.String sm)` Устарел. Используется `setStatus(int)`

В табл. П2.20 приведены методы класса `ServletRequestWrapper`. Этот класс имплементирует интерфейс `ServletRequest`.

Таблица П2.20. *Методы класса* `ServletRequestWrapper`

Тип	Метод
`java.lang.Object`	`getAttribute(java.lang.String name)` Возвращает значение указанного атрибута в виде объекта `Object`, или `null`
`java.util.Enumeration`	`getAttributeNames()` Возвращает `Enumeration` с именами атрибутов сервлета
`java.lang.String`	`getCharacterEncoding()` Имя кодировки символов
`Int`	`getContentLength()` Длина (в байтах) тела запроса. `(-1)` — если длина неизвестна
`java.lang.String`	`getContentType()` Возвращает тип MIME, используемый в теле запроса, или `null`
`ServletInputStream`	`getInputStream()` Получает тело запроса с использованием `ServletInputStream`
`java.util.Locale`	`getLocale()` Возвращает предпочтительный `Locale`, заданный в клиенте в соответствии с заголовком `Accept-Language`
`java.util.Enumeration`	`getLocales()` Возвращает `Enumeration` объектов типа `Locale` в убывающем порядке от наиболее предпочтительного на основе заголовка `Accept-Language`
`java.lang.String`	`getParameter(java.lang.String name)` Возвращает значение параметра в виде строки `String`, или `null`
`java.util.Map`	`getParameterMap()` Возвращает `java.util.Map` для параметров запроса

Таблица П2.20 *(продолжение)*

Тип	Метод
`java.util.Enumeration`	`getParameterNames()` Возвращает `Enumeration` объектов `String` с именами параметров запроса
`java.lang.String[]`	`getParameterValues(java.lang.String name)` Возвращает массив объектов `String` со всеми значениями для указанного имени параметра или `null`, если параметр не существует
`java.lang.String`	`getProtocol()` Возвращает имя и версию протокола в форме *protocol/majorVersion.minorVersion*, например, HTTP/1.1
`java.io.BufferedReader`	`getReader()` Получает тело запроса в виде символов с использованием `BufferedReader`
`java.lang.String`	`getRealPath(java.lang.String path)` Устарел. Используется `ServletContext.getRealPath(java.lang.String)`
`java.lang.String`	`getRemoteAddr()` Возвращает адрес IP клиента, пославшего запрос
`java.lang.String`	`getRemoteHost()` Возвращает полное имя клиента, пославшего запрос
`RequestDispatcher`	`getRequestDispatcher(java.lang.String path)` Возвращает объект `RequestDispatcher`, который используется с ресурсом по указанному пути
`java.lang.String`	`getScheme()` Возвращает имя `scheme` для выполнения запроса (например `http`, `https`, `ftp`)
`java.lang.String`	`getServerName()` Возвращает имя хоста сервера, получившего запрос
`int`	`getServerPort()` Возвращает номер порта, на котором был получен запрос
`boolean`	`isSecure()` Возвращает логическое значение, показывающее, был ли получен запрос по безопасному каналу, например, по HTTPS
`void`	`removeAttribute(java.lang.String name)` Удаляет атрибут из запроса

Таблица П2.20 *(окончание)*

Тип	Метод
`void`	`setAttribute(java.lang.String name, java.lang.Object o)` Задает атрибут в запросе
`void`	`setCharacterEncoding(java.lang.String env)` Переустанавливает имя кодировки запроса

В табл. П2.21 приведены методы класса `ServletResponceWrapper`. Класс `ServletResponceWrapper` имплементирует методы интерфейса `ServletResponce`.

Таблица П2.21. *Методы класса* `ServletResponceWrapper`

Тип	Метод
`void`	`flushBuffer()` Отсылает содержимое буфера клиенту
`int`	`getBufferSize()` Возвращает реальный размер буфера, посылаемого клиенту
`java.lang.String`	`getCharacterEncoding()` Возвращает название набора символов, используемого в теле ответа
`java.util.Locale`	`getLocale()` Возвращает объект `Locale`, связанный с ответом
`ServletOutputStream`	`getOutputStream()` Возвращает `ServletOutputStream` для записи двоичных данных ответа
`java.io.PrintWriter`	`getWriter()` Возвращает объект `PrintWriter` для отправки символов клиенту
`boolean`	`isCommitted()` Возвращает логическое значение, отправлен ли ответ
`void`	`reset()` Очищает буфер от всех данных, в том числе от статус-кода и заголовков
`void`	`resetBuffer()` Очищает буфер ответа, но оставляет заголовки и статус- код
`void`	`setBufferSize(int size)` Устанавливает предпочтительный размер буфера для тела ответа

Таблица П2.21 *(окончание)*

Тип	Метод
`void`	`setContentLength(int len)` Устанавливает длину тела в ответе HTTP, этот метод устанавливает значение, используемое в заголовке `Content-Length`
`void`	`setContentType(java.lang.String type)` Устанавливает тип содержимого, посылаемого в ответе клиенту
`void`	`setLocale(java.util.Locale loc)` Устанавливает `locale` для ответа, в том числе устанавливает значения и других заголовков

Табл. П2.22 содержит методы класса `HttpServletRequestWrapper`. Класс `HttpServletRequestWrapper` имплементирует интерфейс `HttpServletRequeset`.

Таблица П2.22. *Методы класса* `HttpServletRequestWrapper`

Тип	Метод
`java.lang.String`	`getAuthType()` Возвращает тип аутентификации
`java.lang.String`	`getContextPath()` Возвращает часть URI, которая указывает на контекст запроса
`Cookie[]`	`getCookies()` Возвращает массив объектов `Cookie`, посылаемых клиентом с запросом
`long`	`getDateHeader(java.lang.String name)` Возвращает значение соответствующего заголовка в виде `long`
`java.lang.String`	`getHeader(java.lang.String name)` Возвращает заголовок запроса в виде строки `String`
`java.util.Enumeration`	`getHeaderNames()` Возвращает `enumeration` со всеми именами заголовков, содержащимися в запросе
`java.util.Enumeration`	`getHeaders(java.lang.String name)` Возвращает все значения соответствующего заголовка в виде `Enumeration` объектов `String`
`int`	`getIntHeader(java.lang.String name)` Возвращает значение заголовка как `int`

Таблица П2.22 *(продолжение)*

Тип	Метод
`java.lang.String`	`getMethod()` Возвращает HTTP метод запроса (например, `GET`, `POST`, `PUT`)
`java.lang.String`	`getPathInfo()` Возвращает дополнительную информацию, содержащуюся в URL
`java.lang.String`	`getPathTranslated()` Возвращает дополнительную информацию после имени сервлета, но до строки запроса
`java.lang.String`	`getQueryString()` Возвращает строку запроса, которая содержит часть URL после пути
`java.lang.String`	`getRemoteUser()` Возвращает имя `login`, если пользователь идентифицирован, или `null`
`java.lang.String`	`getRequestedSessionId()` Возвращает идентификатор сессии, указанный клиентом
`java.lang.String`	`getRequestURI()` Возвращает часть URL начиная с имени протокола до строки запроса в первой строке HTTP-запроса
`java.lang.StringBuffer`	`getRequestURL()` Восстанавливает URL, использованный клиентом для создания запроса
`java.lang.String`	`getServletPath()` Возвращает часть URL, которая вызывает сервлет
`HttpSession`	`getSession()` Возвращает текущую сессию, связанную с запросом, если запрос еще не имеет сессии, то сессия будет создана
`HttpSession`	`getSession(boolean create)` Возвращает текущую сессию `HttpSession`, связанную с запросом, если сессии нет и `create` имеет значение `true`, то создает новую сессию
`java.security.Principal`	`getUserPrincipal()` Возвращает объект `java.security.Principal` с именем пользователя
`boolean`	`isRequestedSessionIdFromCookie()` Проверяет, является ли запрошенный идентификатор сессии `ID` полученным посредством `Cookie`

Таблица П2.22 *(окончание)*

Тип	Метод
boolean	isRequestedSessionIdFromUrl() Устарел. Используется isRequestedSessionIdFromURL()
boolean	isRequestedSessionIdFromURL() Проверяет, получен ли идентификатор сессии ID в запросе в виде составляющей части URL
boolean	isRequestedSessionIdValid() Проверят, является ли запрошенный идентификатор сессии действительным
boolean	isUserInRole(java.lang.String role) Возвращает логическое значение, истина возвращается тогда, когда опознанный пользователь обладает указанной ролью

Табл. П2.23 содержит методы класса HttpServletResponceWrapper. Класс HttpServletResponceWrapper имплементирует интерфейс HttpServletResponce, описание методов которого приведено в табл. П2.19.

Таблица П2.23. *Методы класса HttpServletResponceWrapper*

Тип	Метод
void	addCookie(Cookie cookie) Вставляет Cookie в ответ
void	addDateHeader(java.lang.String name, long date) Вставляет заголовок ответа с именем и датой
void	addHeader(java.lang.String name, java.lang.String value) Вставляет в ответ заголовок с указанным именем и значением
void	addIntHeader(java.lang.String name, int value) Вставляет в ответ заголовок с именем и целым значением
boolean	ContainsHeader(java.lang.String name) Возвращает логическое значение, существует ли заголовок с указанным именем
java.lang.String	encodeRedirectUrl(java.lang.String url) Устарел. Используется encodeRedirectURL(String url)
java.lang.String	encodeRedirectURL(java.lang.String url) Кодирует указанный URL для использования в методе sendRedirect, если кодировка не требуется, то оставляет его неизмененным

Таблица П2.23 *(окончание)*

Тип	Метод
`java.lang.String`	`encodeUrl(java.lang.String url)` Устарел. Используется `encodeURL(String url)`
`java.lang.String`	`encodeURL(java.lang.String url)` Кодирует URL
`void`	`sendError(int sc)` Посылает клиенту ответ об ошибке с указанием заданного кода ошибки и очищает буфер
`void`	`sendError(int sc, java.lang.String msg)` Посылает клиенту ответ об ошибке с указанием заданного кода ошибки и очищает буфер. Посылает сообщение
`void`	`sendRedirect(java.lang.String location)` Посылает адрес для перенаправления запроса
`void`	`setDateHeader(java.lang.String name, long date)` Устанавливает заголовок ответа с заданным именем и датой
`void`	`setHeader(java.lang.String name, java.lang.String value)` Устанавливает заголовок ответа с именем и значением
`void`	`setIntHeader(java.lang.String name, int value)` Устанавливает заголовок ответа с заданным именем и целым значением
`void`	`setStatus(int sc)` Устанавливает статус-код для ответа
`void`	`setStatus(int sc, java.lang.String sm)` Устарел. Используются `setStatus(int)`

Вывод информации сервлет осуществляет при помощи методов, описанных в классе `ServletOutputStream` (табл. П2.24).

Таблица П2.24. *Методы класса* `ServletOutputStream`

Тип	Метод
`void`	`print(boolean b)` Посылает клиенту логическое значение без символов перевода (CRLF) на конце
`void`	`print(char c)` Посылает клиенту символ без (CRLF) на конце
`void`	`print(double d)` Посылает клиенту величину типа `double` без символов (CRLF) на конце

Таблица П2.24 *(окончание)*

Тип	Метод
`void`	`print(float f)` Посылает клиенту величину типа `float` без символов (CRLF) на конце
`void`	`print(int i)` Посылает клиенту величину типа `int` без символов (CRLF) на конце
`void`	`print(long l)` Посылает клиенту значение типа `long` без символов (CRLF) на конце
`void`	`print(java.lang.String s)` Посылает клиенту величину типа `String` без символов (CRLF) на конце
`void`	`println()` Посылает клиенту символ перевода строки (CRLF)
`void`	`println(boolean b)` Посылает значение типа `boolean` с символами (CRLF) на конце
`void`	`println(char c)` Посылает символ типа `char` с символами (CRLF) на конце
`void`	`println(double d)` Посылает значение типа `double` с символами (CRLF) на конце
`void`	`println(float f)` Посылает значение типа `float` с символами (CRLF) на конце
`void`	`println(int i)` Посылает значение типа `int` с символами (CRLF) на конце
`void`	`println(long l)` Посылает значение типа `long` с символами (CRLF) на конце
`void`	`println(java.lang.String s)` Посылает значение типа `String` с символами (CRLF) на конце

Класс `Cookie` содержит методы, используемые для работ с HTTP Cookikes (табл. П2.25).

Таблица П2.25. *Методы класса* `Cookie`

Тип	Метод
`Java.lang.Object`	`clone()` Переопределяет стандартный метод `java.lang.Object.clone` для возвращения копии `Cookie`

Таблица П2.25 *(окончание)*

Тип	Метод
`Java.lang.String`	`getComment()` Возвращает комментарий, описывающий `Cookie` или `null`
`Java.lang.String`	`getDomain()` Возвращает доменное имя для `Cookie`
`Int`	`getMaxAge()` Возвращает максимальный возраст в секундах, допустимый для `Cookie`, по умолчанию `-1` — существует до закрытия браузера
`Java.lang.String`	`getName()` Имя `Cookie`
`Java.lang.String`	`getPath()` Путь на сервере, куда возвращается `Cookie`
`Boolean`	`getSecure()` Возвращает `true`, если браузер возвращает `Cookies` только с использованием безопасного протокола, `false` — `Cookies` могут быть посланы с использованием любого протокола
`Java.lang.String`	`getValue()` Значение `Cookie`
`Int`	`getVersion()` Версия протокола
`Void`	`setComment(java.lang.String purpose)` Комментарий
`Void`	`setDomain(java.lang.String pattern)` Домен, с которым связаны `Cookie`
`Void`	`setMaxAge(int expiry)`
`Void`	`setPath(java.lang.String uri)` Путь, по которому клиент возвращает `Cookie`
`void`	`setSecure(boolean flag)` Сообщает браузеру, должен ли браузер посылать `Cookie` по безопасному протоколу HTTPS или SSL
`void`	`setValue(java.lang.String newValue)` Задает новое значение для `Cookie`
`void`	`setVersion(int v)` Задает протокол

Интерфейс `HttpSession` содержит описания методов, позволяющих работать с сессиями (табл. П2.26).

Таблица П2.26. *Методы интерфейса* `HttpSession`

Тип	Метод
`java.lang.Object`	`GetAttribute(java.lang.String name)` Возвращает объект по указанному имени в пределах сессии или `null`, если объекта нет
`Java.util.Enumeration`	`getAttributeNames()` Возвращает `Enumeration` объектов типа `String` с именами объектов, связанных с сессией
`long`	`getCreationTime()` Возвращает время создания сессии в миллисекундах начиная с 1 января 1970 года
`java.lang.String`	`getId()` Возвращает строку с уникальным идентификатором сессии
`long`	`getLastAccessedTime()` Возвращает время последнего обращения клиента в миллисекундах с 1 января 1970 года
`int`	`getMaxInactiveInterval()` Максимальный интервал между обращениями клиента в секундах, в течение которого остается действительной текущая сессия
`ServletContext`	`GetServletContext()` Возвращает `ServletContext`, к которой относится текущая сессия
`HttpSessionContext`	`GetSessionContext()` Устарел
`java.lang.Object`	`getValue(java.lang.String name)` Устарел. Используется `getAttribute(java.lang.String)`
`java.lang.String[]`	`getValueNames()` Устарел. Используется `getAttributeNames()`
`void`	`invalidate()` Удаляет сессию
`boolean`	`isNew()` Возвращает `true`, если клиент не работает с сессией
`void`	`putValue(java.lang.String name, java.lang.Object value)` Устарел. Используется `setAttribute(java.lang.String, java.lang.Object)`
`void`	`removeAttribute(java.lang.String name)` Удаляет из сессии объект с указанным именем

Таблица П2.26 *(окончание)*

Тип	Метод
`void`	`RemoveValue(java.lang.String name)` Устарел. Используется `removeAttribute(java.lang.String)`
`void`	`setAttribute(java.lang.String name, java.lang.Object value)` Объект с указанным именем включается в состав сессии
`void`	`setMaxInactiveInterval(int interval)` Время в секундах между клиентскими запросами, по истечении которого контейнер прекращает сессию

Методы класса `PageContext` приводятся в табл. П2.27.

Таблица П2.27. *Методы класса* `PageContext`

Тип	Метод
`abstract java.lang.Object`	`findAttribute(java.lang.String name)` Производит поиск атрибутов на странице, сессии, приложении и возвращает соответствующее значение (или `null`)
`abstract void`	`forward(java.lang.String relativeUrlPath)` Перенаправляет текущие объекты типов `ServletRequest` и `ServletResponse` к другому активному компоненту приложения
`abstract java.lang.Object`	`getAttribute(java.lang.String name)` Возвращает объект по имени, или `null`
`abstract java.lang.Object`	`getAttribute(java.lang.String name, int scope)` Возвращает объект по имени, расположенный в указанной зоне доступности, или возвращает `null`, если объект не найден
`Abstract java.util.Enumeration`	`getAttributeNamesInScope(int scope)` Все атрибуты в указанной зоне видимости
`abstract int`	`getAttributesScope(java.lang.String name)` Возвращает область доступа, из которой виден атрибут с указанным именем
`Abstract java.lang.Exception`	`getException()` Текущее значение объекта исключения

Таблица П2.27 *(продолжение)*

Тип		Метод
abstract	JspWriter	`getOut()` Текущее значение объекта потока вывода (`JspWriter`)
abstract	java.lang.Object	`getPage()` Текущее значение для объекта страницы (`Servlet`)
abstract	ServletRequest	`getRequest()` Текущее значение для объекта запроса (`ServletRequest`)
abstract	ServletResponse	`getResponse()` Текущее значение для объекта ответа (`ServletResponse`)
abstract	ServletConfig	`getServletConfig()` Получает объект типа `ServletConfig`
abstract	ServletContext	`getServletContext()` Получает объект типа `ServletContext`
abstract	HttpSession	`getSession()` Текущее значение объекта сессии (`HttpSession`)
abstract	void	`handlePageException(java.lang.Exception e)` Метод используется для обработки исключительной ситуации на странице, для этого производится перенаправление исключения на определенную страницу ошибки или для выполнения каких-либо действий
abstract	void	`handlePageException(java.lang.Throwable t)` Идентичен методу `handlePageException(Exception)`, но в качестве аргумента выступает `Throwable`
abstract	void	`include(java.lang.String relativeUrlPath)` Приводит к тому, что указанный ресурс будет выполняться совместно для обработки текущих объектов `ServletRequest` и `ServletResponse` в составе текущего потока `Thread`
abstract	void	`initialize(Servlet servlet, ServletRequest request, ServletResponse response, java.lang.String errorPageURL, boolean needsSession, int bufferSize, boolean autoFlush)` Используется для инициализации `PageContext`, после чего контекст может быть использован классом имплементации JSP для обработки входящих запросов с помощью метода `_jspService()`

Таблица П2.27 *(окончание)*

Тип	Метод
`JspWriter`	`popBody()` Возвращает предыдущий фрагмент потока вывода `JspWriter`, сохраненный при вызове `pushBody()`
`BodyContent`	`pushBody()` Возвращает новый объект `BodyContent`
`abstract void`	`release()` Освобождает состояние `PageContext`, снимает все внутренние ссылки, подготавливает `PageContext` для нового использования с применением `initialize()`
`abstract void`	`removeAttribute(java.lang.String name)` Удаляет ссылку на объект с указанным именем

`abstract void`	`removeAttribute(`
	`java.lang.String name, int scope)`
	Удаляет ссылку на объект с указанным именем в пределах заданной области видимости

`abstract void`	`setAttribute(java.lang.String name,`
	`java.lang.Object attribute)`
	Регистрирует объект с заданным именем

`abstract void`	`setAttribute(java.lang.String name,`
	`java.lang.Object o, int scope)`
	Регистрирует объект по указанному имени в пределах указанной области видимости

Приложение 3

Сервер Blazix

Сервер Blazix является полнофункциональным Web-сервером, поддерживающим J2EE.

Утилиты и команды сервера

Команды сервера Blazix представляют собой обращения интерпретатора Java к тем или иным классам сервера Blazix. Классы могут быть вызваны непосредственно из командной строки вручную, в состав сервера входят готовые командные файлы вызова соответствующих классов, что упрощает работу с сервером.

Команды сервера

Перечень команд и утилит приведен ниже.

- blxWeb — Web-сервер Blazix
- blxejbc — компилятор EJB
- blxejbs — сервер EJB
- blxui — утилита удаленного администрирования
- blizzard — проводник создания компонентов EJB
- jspDebug — отладчик JSP
- blxcls — сервер кластеров
- blxsvrmgr — менеджер сервера
- blxPacker — упаковщик Web-приложений
- blxionreg — утилита регистрации ION
- blxI18nTagExtract — утилита интернационализации (основана на библиотеке ярлыков `blx`)
- blxjmsmgr — менеджер тем и очередей JMS
- blxjmss — сервер JMS
- SetAutoEjbKey — генератор первичных ключей EJB

Web-сервер blxWeb

Команда:

```
java desisoft.server.ServerMain
```

Запускает Web-сервер.

Флаги:

- `-c` или `/c` — (имя файла конфигурации);
- `-t` или `/t` — имя файла конфигурации кластера;
- `-noconsole` или `/noconsole` — выключает вывод в стандартный поток вывода, вывод в файл логов продолжается;
- `-log` или `/log` — имя файла логов, по умолчанию `Weblog`.

Команда *Blxejbc*

Компилятор компонентов EJB.

Команда:

```
java desisoft.ejbtools.EjbCompiler
```

В качестве источника компилятор использует файл архива компонентов EJB. Исходный архив должен содержать в себе дескриптор ejb-jar.xml и классы компонентов EJB. На выходе получаем файл архива, в котором содержатся все исходные классы компонентов EJB, а также классы, сгенерированные контейнером. Полученный файл архива может быть размещен на сервере компонентов EJB.

В качестве аргументов указывается имя входного файла и имя результирующего файла.

Флаги:

- `-c` или `/c` — местоположение файла конфигурации EJB;
- `-keepgenerated` или `/keepgenerated` — если указан этот параметр, то промежуточные файлы не будут удалены;
- `-t` или `/t` — имя папки, используемой для размещения временных файлов.

Пример

```
blxejbc -c c:\Blazix\ejb.ini EjbInput.jar EjbOutput.jar
```

Команда *blxejbs*

EJB-сервер.

Команда:

```
java desisoft.ejb.server.EjbServer
```

Команда запуска сервера EJB.

Флаги:

- `-c` или `/c` — имя файла конфигурации EJB-сервера;
- `-noconsole` или `/noconsole` — останавливает текущий вывод в стандартный поток вывода, вывод в файл логов продолжается;
- `-log` или `/log` — файл статистики обращений к серверу EJB, по умолчанию `ejblog`.

Команда *blxui*

Утилита удаленного администрирования.

Команда:

```
java desisoft.blazixui.BlazixUi
```

Чтобы иметь доступ к удаленным серверам, необходимо задать файл конфигурации кластера. Если файл конфигурации кластера не будет найден, то утилита предоставит возможность администрирования только локальных серверов.

Утилита может использоваться для решения следующих задач:

- загрузки файлов архивов компонентов EJB на сервер в папку компонентов EJB (по умолчанию папка `ejbDir`);
- осуществления нового просмотра сервером папки компонентов EJB (для размещения на сервере новых компонентов EJB);
- разгрузки на сервер Web-архивов WAR в папку Web-архивов (в соответствии с файлом конфигурации Web-сервера, по умолчанию это warDir);
- осуществления нового просмотра папки Web-архивов для размещения на сервер новых Web-приложений;
- просмотра файлов логов;
- остановки сервера;
- просмотра файлов Web.ini и ejb.ini.

Флаги:

- `-t` или `/t` — файл конфигурации кластера;
- `-w` или `/w` — файл конфигурации отдельного Web-сервера;
- `-e` или `/e` — файл конфигурации отдельного EJB-сервера;
- `-j` или`/j` — файл конфигурации отдельного JMS-сервера.

Команда *blizzard*

Проводник создания компонентов EJB.

Команда:

```
java desisoft.ejbtools.blizzard.Blizzard
```

Эта утилита используется для создания компонентов EJB-сессии и компонентов EJB-сущности. Для создания компонентов EJB-сущности необходим по крайней мере один источник данных, который должен быть указан в файле инициализации сервера EJB. База данных должна содержать таблицу, которая будет использоваться с компонентом EJB.

Флаги:

- `-c` или `/c` — файл конфигурации сервера EJB.

Команда *blxionreg*

Утилита регистрации ION.

Команда:

```
java desisoft.dsap.IonRegister
```

Утилита регистрирует интерфейсы Java для использования в соответствии с правилами технологии ION. Все интерфейсы ION должны быть зарегистрированы на Web-сервере.

Флаги:

- `-c` или `/c` — файл конфигурации Web-сервера.

Команда *blxsvrmgr*

Менеджер сервера Blazix.

Команда:

```
java desisoft.tools.ServerManager
```

Команда вызывает менеджер сервера. Менеджер может управлять одним или несколькими серверами на машине.

Флаги:

- `-t` или `/t` — файл конфигурации кластера.

Команда *blxcls*

Кластерный сервер подмены.

Команда:

```
java desisoft.cluster.SessionBackupServer
```

Флаги:

- `-t` или `/t` — файл конфигурации кластера.

Команда *blxPacker*

Утилита упаковки приложений.

Команда:

```
java desisoft.tools.blxpack.BlxPacker
```

Создает WAR-архив Web-приложения из файлов в папке приложения.

Флаги:

- `-c` или `/c` — файл конфигурации Web-сервера.

Команда *jspDebug*

Утилита отладки JSP.

Команда:

```
java desisoft.tools.JspDebug
```

Утилита в качестве аргументов получает имена одного или нескольких файлов JSP, создает соответствующие им файлы Java, которые могут быть подвергнуты отладке. Файлы должны быть указаны в формате URL, начиная с косой черты "/".

Флаги:

- `-d` или `/d` — папка, где должны быть размещены файлы java. Обязательный аргумент;
- `-c` или `/c` — файл конфигурации Web-сервера.

Команда *blxI18nTagExtract*

Утилита для работы с ярлыками интернационализации библиотеки ярлыков JSP сервера Blazix.

Команда:

```
java desisoft.tools.i18n.Xltagext
```

Флаги:

- `-c` или `/c` — файл конфигурации Web-севрера.

Команда *blxjmsmgr*

Менеджер тем и очередей JMS.

Команда:

```
java desisoft.jms.manager.JmsManager
```

Флаги:

- `-c` или `/c` — файл конфигурации JMS-сервера.

Команда *blxjmss*

Конфигурация сервера JMS.

Команда:

```
java desisoft.jms.main.JmsMain
```

Флаги:

- `-c` или `/c` — имя файла конфигурации JMS-сервера.
- `-log` или `/log` — каталог расположения файла логов, по умолчанию jmslog.

Команда *SetAutoEjbKey*

Генерация ключей компонентов EJB-сущности.

Команда:

```
java desisoft.ejb.server.EnableAutoPkeys
```

Команда должна содержать два аргумента:

- имя компонента EJB;
- имя источника данных. (Если в файле конфигурации задано соответствующее источнику данных имя JNDI, то следует указать именно его. Если заданы пароль и имя пользователя, то после названия источника данных следует указать имя и пароль.)

Флаги:

- `-c` или `/c` — имя файла конфигурации EJB.

Эта команда создает таблицу в базе данных, связанной с источником данных, если таблица там еще не создана. Таблица будет названа `BLAZIX_AUTO_EJB_KEYTABLE`. Она состоит из двух столбцов `EJBNAME` и `NEXTID`. `NEXTID` — возрастающие идентификаторы ключей. Имена `EJB` не чувствительны к регистру, источник данных должен существовать и быть должным образом сконфигурирован.

Конфигурирование сервера Blazix для Windows

Сервер Blazix можно запустить как сервис в операционной системе Windows. Программа Server Manager, входящая в состав сервера Blazix, предоставляет возможность автоматического запуска серверов Web, EJB, JMS

и серверов, входящих в состав кластеров, а также возможность удаленного управления этими серверами. Менеджер Server Manager может быть установлен в качестве сервиса в Windows NT, при этом он будет запускаться автоматически. Автоматический запуск сервера, однако, требует наличия у пользователя определенных привилегий, например, пользователь должен быть администратором.

С помощью BlazixWinService.exe можно стартовать менеджер Server Manager как сервис Windows. Необходимо создать файл инициализации, путь к которому должен быть такой же, как путь к файлу exe, например, C:\BlazixServer\BlazixWinService.exe и C:\BlazixServer\BlazixWinservice.ini.

Параметры файла инициализации

Параметры файла инициализации представлены ниже.

- `javaPath`

  Путь к установочной папке Java, должен быть задан до того, как будет произведено обращение к Java.

- `JavaExe`

  Команда должна соответствовать exe-файлу Java. Если указано значение Java (как по умолчанию), то необходимо, чтобы путь к установочному каталогу Java был указан в качестве значения переменной окружения `path`.

- `homeDir`

  Домашний директорий сервера, откуда он запускается.

- `classPath`

  Переменная окружения `CLASSPATH`, должна включать файл blazix.jar и папку с классами.

- `server.port`

  Порт, на котором менеджер Server Manager будет слушать утилиту администрирования сервера, утилиту Server Administration. При наличии proxy-сервера, этот порт не должен быть доступен для внешнего доступа.

- `server.address`

  Важно указать адрес сервера для машин с несколькими IP-адресами.

- `erver.password`

  Пароль для Server Manager.

- `WebServer.name`

  Имя Web-сервера (для внутреннего использования в файле конфигурации).

- `WebServer.<name>.config`

  Полный путь к файлу конфигурации Web-сервера.

- `WebServer.<name>.classpath`

  Переменная `Classpath` для Web-сервера.
- `WebServer.<name>.clusterFile`

  Полный путь к файлу конфигурации кластера.
- `ejbServer.name`

  Имя сервера EJB.
- `ejbServer.<name>.config`

  Полный путь к файлу конфигурации EJB-сервера.
- `ejbServer.<name>.classpath`

  Переменная `Classpath` для сервера EJB.
- `ejbServer.<name>.clusterFile`

  Полный путь к файлу конфигурации кластера.
- `jmsServer.name`

  Имя сервера JMS.
- `jmsServer.<name>.config`

  Полный путь к файлу конфигурации сервера JMS.
- `clusterServer.name`

  Имя сервера кластера Web-серверов, ответственного за подмену.
- `clusterServer.<name>.config`

  Полный путь к файлу конфигурации кластера.

Пример

```
# Путь и домашний каталог
javapath: c:\jdk1.3\bin
classpath: c:\Blazix\blazix.jar;C:\Blazix\classes;c:\NashuClasses
homeDir: c:\Blazix

# Порт и пароль менеджера
server.port: 4400
server.password: myServerPassword

# Сервер
WebServer.name: Web1
WebServer.Web1.config: c:\Blazix\Web.ini
```

```
ejbServer.name: ejb1
ejbServer.ejb1.config: c:\Blazix\ejb.ini

jmsServer.name: jms1
jmsServer.jms1.config: c:\Blazix\jms.ini
```

Разработчики сервера приводят следующие инструкции, которые полезно учитывать, если файл конфигурации уже сконфигурирован и готов к использованию. Исполняемый файл сервера следует запускать из консоли, командного окна сеанса DOS, с использованием аргумента "`/interactive`". Если аргумент `/interactive` не будет указан, то могут возникнуть проблемы при остановке сервера. Если сервер благополучно стартовал, то остановить его можно, нажав комбинацию клавиш <Ctrl>+<C> и использованием аргумента "`/interactiveShutdown`". При этом стартует менеджер сервера и попытается остановить сервер по истечении двух минут.

После того как файл конфигурации успешно протестирован (см. предыдущий абзац), сервис может быть запущен с использованием аргумента "`/install`". Работающий сервис может быть остановлен с использованием аргумента "`/uninstall`".

В системе Windows 2000 требуется наличие файла BlazixServiceConsoleHandler.dll в папке, в которой расположен файл BlazixWinService.exe.

Конфигурирование менеджера сервера

Местоположение файла конфигурации

При запуске сервера в качестве сервиса Windows NT или Windows 2000, файл конфигурации менеджера сервера будет таким же, как и файл инициализации сервиса. При самостоятельной работе сервера файл конфигурации будет расположен там, куда укажет следующая команда.

```
java desisoft.tools.ServerManager -c c:\Mylocation\servermgr.ini
```

Если местоположение файла не задано, то сервер будет искать этот файл, просматривая каталоги в следующей последовательности:

1. В случае, если задано `desisoft.ServerManager.config`, то это значение будет использовано для поиска файла конфигурации.
2. Файл c:\BlazixServerManager.ini (Windows) или /usr/blazix/BlazixServerManager.ini (Unix).
3. Текущий каталог (на предмет наличия файла ServerManager.ini).

Параметры конфигурации

Используются следующие параметры конфигурации (табл. П3.1).

Таблица П3.1. *Параметры конфигурации*

Параметр	Назначение
`JavaExe`	Команда, стартующая Java. Если используется команда по умолчанию (*"java"*), то следует особо позаботиться о задании переменной `path` (чтобы найти java)
`HomeDir`	Домашний каталог сервера
`ClassPath`	Полный набор папок для переменной *classpath*, включая `blazix.jar`. Для отдельных серверов может быть задан индивидуально
`server.port`	Номер порта, на котором менеджер сервера прослушивает утилиту администрирования. При использовании proxy-серверов этот порт следует делать недоступным для доступа извне
`server.address`	Адрес сервера. Задает на машинах с несколькими IP
`server.password`	Пароль менеджера сервера
`WebServer.name`	Имя Web-сервера (для внутреннего использования в файл конфигурации)
`WebServer.<name>.config`	Полный путь к файлу конфигурации Web-сервера
`WebServer.<name>.classpath`	`Classpath` для Web-сервера
`WebServer.<name>.clusterFile`	Путь к файлу конфигурации кластера. Если сервер в кластере, то этот параметр обязателен
`WebServer.<name>.logBase`	Каталог файла логов ("Weblog")
`ejbServer.name`	Имя сервера EJB
`ejbServer.<name>.config`	Полный путь к файлу конфигурации сервера EJB
`ejbServer.<name>.classpath`	`Classpath` для сервера EJB
`ejbServer.<name>.clusterFile`	Полный путь к файлу конфигурации кластера
`ejbServer.<name>.logBase`	Каталог файла логов ("ejblog")
`jmsServer.name`	Имя сервера JMS
`jmsServer.<name>.config`	Полный путь к файлу конфигурации сервера JMS

Таблица П3.1 *(окончание)*

Параметр	Назначение
`jmsServer.<name>.logBase`	Имя файла, используемого для ведения логов (по умолчанию это "jmslog")
`clusterServer.name`	Имя сервера, ответственного за подмену в кластере Web-серверов
`clusterServer.<name>.config`	Полный путь к файлу конфигурации кластера

Пример

```
# Set up paths and home directory
javapath: c:\jdk1.3\bin classpath: c:\Blazix\blazix.jar;C:\
Blazix\classes;c:\MyClasses homeDir: c:\Blazix

# allocate a port and a password for the ServerManager itself.
server.port: 4400
server.password: myServerPassword

# specify the servers to be managed
WebServer.name: Web1
WebServer.Web1.config: c:\Blazix\Web.ini

ejbServer.name: ejb1
ejbServer.ejb1.config: c:\Blazix\ejb.ini

jmsServer.name: jms1
jmsServer.jms1.config: c:\Blazix\jms.ini
```

Библиотека JSP-ярлыков сервера Blazix

Сервер Blazix содержит в своем составе библиотеку ярлыков JSP (спецификация JSP 1.1 по состоянию сервера на конец лета 2002 г.). Для доступа к библиотеке ярлыков сервера Blazix в JSP-страницу необходимо вставить следующую инструкцию:

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
```

Это не является обязательным, но рекомендуется. Заданный префикс может быть и изменен. После того как в файл JSP вставлена указанная строка, становятся доступны следующие ярлыки библиотеки сервера.

Ярлыки обработки форм

- `blx:setProperty`
- `blx:getProperty`

Ярлыки для работы с почтой

- `blx:email`
- `blx:emailTo`

- `blx:emailCc`
- `blx:emailBcc`
- `blx:emailTo`
- `blx:emailSubject`
- `blx:emailAttach`

Ярлыки для обработки SQL-запросов

- `blx:sqlConnection`
- `blx:sqlQuery`
- `blx:sqlExecuteQuery`
- `blx:sqlGet`
- `blx:sqlExecuteUpdate`

Ярлыки для поддержки языков

- `blx:xlt`
- `blx:xltFile`

Ярлыки обработки форм

Ярлык *blx:setProperty*

Аналогично ярлыку `jsp:setProperty`, улучшена обработка пустых строк. Атрибуты перечислены в табл. П3.2.

Таблица П3.2. *Атрибуты ярлыка* `blx:setProperty`

Имя атрибута	Описание атрибута	Обязателен или нет
`Name`	Имя компонента EJB	Обязателен
`property`	Свойство компонента EJB, для задания всех свойств указывается *	Обязателен
`onError`	Имя метода, вызываемого при возникновении исключительной ситуации: `public void onError(String fieldname, String value, Exception exception);`	Необязателен
`stringNull`	Если форма посылает пустую строку, то устанавливать ее значение равным `null` или пустой строке, соответственно задается значение для атрибута `true` или `false`	Необязателен

Таблица П3.2 *(окончание)*

Имя атрибута	Описание атрибута	Обязателен или нет
checkbox	Список имен всех полей для выбора в форме, разделенных запятыми	Необязателен
emptyInt	Целое значение, которое будет установлено целой величине, если форма посылает пустое поле для целого значения. По умолчанию 0	Необязателен

Ярлык *blx:getProperty*

Ярлык получения значения свойства, противоположен ярлыкам jsp:setProperty и blx:setProperty. Ярлык не получает свойство из формы, а задает значение свойства формы, получая это значение от компонента EJB, т. е. из Java-программы. Ярлык работает с полями INPUT, SELECT и TEXTAREA. Атрибуты ярлыка описаны в табл. П3.3.

Таблица П3.3. *Атрибуты ярлыка* blx:getProperty

Имя атрибута	Описание	Обязателен или нет
name	Имя компонента EJB	Обязателен
property	Свойство (для всех свойств используется)	Обязателен
dateFormat	Способ форматирования исходящего объекта даты: short, medium, long или full	Необязателен
timeFormat	Форматирование времени в объекте даты: short, medium, long или full	Необязателен
emptyInt	Целое значение, используемое для подмены пустых значений целых переменных в форме	

Пример

```
<blx:getProperty name=testBean property=* dateFormat="short">
<FORM METHOD=POST ACTION=target.jsp>
Value <INPUT NAME=value TYPE=TEXT><BR>
<INPUT TYPE=SUBMIT>
</FORM>
</blx:getProperty>
```

Ярлыки для работы с почтой

Ярлыки для работы с почтой облегчают задачи отсылки почты из JSP-страниц. В них включена возможность работы с простой почтой, а также

поддерживается посылка почты с прикрепленными файлами. Почтовое отправление помещается между открывающим и закрывающим ярлыками blx:email. Между этими ярлыками указываются получатель, текст сообщения, прилагаемые файлы и прочая необходимая информация.

Ярлык требует задания некоторых обязательных атрибутов, а именно, host и from. Host — это имя SMTP-сервера, from — это e-mail-адрес отправителя. Например:

```
<blx:email host="mail.myhost.com" from="me@myhost.com"
        to="somebody@somewhere.com">
Hello.  Just testing this email tag.
</blx:email>
```

Поле TO может быть задано с применением ярлыка <blx:emailTo> (вместо указания атрибута в ярлыке blx:email):

```
<% String target = "somebody@somewhere.com"; %>
<blx:email host="myhost.com" from="me@myhost.com">
<blx:emailTo><%= target %></blx:emailTo>
Hello.  Just testing this email tag.
</blx:email>
```

Полям "CC" и "BCC" аналогично соответствуют ярлыки <blx:emailCc> и <blx:emailBcc>.

Наконец, ярлык <blx:emailAttach> используется для вставки файлов приложений. Этот ярлык можно использовать для посылки почты, форматированной в виде HTML, поскольку такая почта представляет собой прикрепленный к сообщению HTML-файл.

Пример отправки почты

```
<blx:email host="myhost.com" from="me@myhost.com"
         to="somebody@somewhere.com">
Hello. Проверка, это почта с приложенным файлом.
<blx:emailAttach file="C:\file.txt"
      contentType="text/plain" name="file.txt"/>
</blx:email>
```

Если в ярлыке blx:emailAttach не указан атрибут file, то приложение к почтовому сообщению начинается сразу с началом тела этого ярлыка, например:

```
<blx:email host="myhost.com" from="me@myhost.com"
           to="somebody@somewhere.com">
Hello.  Проверка. Это почта с приложением. <blx:emailAttach
name="test.html" contentType="text/html">
```

```
<jsp:include page="/myjsp.jsp"/>
</blx:emailAttach>
</blx:email>
```

С применением `jsp:include` можно вставлять только JSP-файлы. Для прочих файлов следует применять `@include`.

Для отправки HTML-почты с приложениями (при этом получится более одного приложения), HTML-файл сообщения должен быть первым приложением, кроме того, для него должны быть заданы значения `inline=true`, `contentType=text/html`, `plainText=true` в ярлыке `emailAttach`. В ярлыке email при этом следует указать `noText=true`. Подробнее ярлыки описаны далее по тексту.

Ярлык *blx:email*

Ярлык посылает почту, направляя ее указанному почтовому серверу по указанному адресу получателя. Адрес получателя (поле `TO`) может быть задан также в ярлыке `blx:emailTo`. Необходимо задать хотя бы один адрес получателя. Значения атрибутов приведены в табл. П3.4.

Таблица П3.4. *Атрибуты ярлыка* `blx:email`

Имя атрибута	Описание	Обязателен или нет
`host`	SMTP-сервер	Обязателен
`from`	Адрес отправителя	Обязателен
`to`	Адрес получателя	Необязателен
`subject`	Поле `subject`	Необязателен
`noText`	Используется при наличии приложенных файлов. Устанавливается равным `true`, если нет простого текста сообщения, тогда первое приложение становится текстом сообщения	Необязателен

Ярлык *blx:emailTo*

Ярлык для задания одного или нескольких адресов получателей. Атрибутов не имеет. Адрес получателя указывается в виде тела ярлыка. Ярлык должен быть вложен в тело элемента `blx:email`.

Ярлык *blx:emailCc*

Ярлык используется для задания значения поля `Cc` заголовка сообщения. Адрес получателя копии указывается в теле элемента, описываемого ярлыком. Ярлык используется в теле элемента `blx:email`. Не имеет атрибутов.

Ярлык *blx:emailBcc*

Ярлык используется для задания значения поля `Bcc` заголовка сообщения. Адрес получателя копии указывается в теле элемента, описываемого ярлыком. Ярлык используется в теле элемента `blx:email`. Не имеет атрибутов.

Ярлык *blx:emailSubject*

Ярлык используется для задания значения поля `Subject` заголовка сообщения. Адрес получателя копии указывается в теле элемента, описываемого ярлыком. Ярлык используется в теле элемента `blx:email`. Не имеет атрибутов.

Ярлык *blx:emailAttach*

Ярлык используется для вставки приложений в почтовое отправление. Используется в теле элемента `blx:email`. Прилагаемый файл задается в качестве значения атрибута `file`. Если это значение не указано, то приложение начинается с начала тела элемента, описываемого этим атрибутом. Все атрибуты ярлыка приведены в табл. П3.5.

Таблица П3.5. *Атрибуты ярлыка* `blx:emailAttach`

Имя атрибута	Описание	Обязателен или нет
`File`	Имя прилагаемого файла	Необязателен
`Name`	Имя приложения	Необязателен
`contentType`	Тип приложения, например, `text/plain` — текстовые файлы, `text/html` — файлы HTML, `image/gif` и `image/jpeg` — рисунки, `application/octet-stream` — двоичные файлы, `application/x-unknown-content-type`	Необязателен
`Inline`	Значение задается равным `true` для того, чтобы указать необходимость отображения содержания первого приложения в качестве тела сообщения (если это HTML-файл). При этом полезно также задать значение `true` для атрибута `plainText`	Необязателен
`plainText`	Задается значение `true`, чтобы указать, что приложение представлено в виде простого текста без кодировки. Это нужно тогда, когда используются текстовые приложения, например, приложение HTML	Необязателен
`targetFilename`	Это имя файла, которое будет предложено для запоминания полученного приложения	Необязателен
`Charset`	Если набор символов не US-ASCII, то этот набор указывается в качестве значения атрибута	Необязателен

Ярлыки для работы с базами данных

Ярлыки для работы с SQL позволяют создавать SQL-запросы из JSP-страницы и выполнять их. Ярлыки выполнения SQL-запроса — это ярлыки `<blx:sqlExecuteQuery>` и `<blx:sqlExecuteUpdate>`. Если запрос не возвращает никакого значения, то используется ярлык `<blx:sqlExecuteUpdate>`; для запросов, которые возвращают значения, применяется ярлык `<blx:sqlExecuteQuery>`. Возвращаемые значения затем можно просмотреть. Пример использования этих ярлыков приведен ниже.

```
<TABLE>
  <TR><TD>Item</TD><TD>Price</TD></TR>
    <blx:sqlExecuteQuery resultSet="resultat" queryRef="mojZaprosa">
  <TR>
    <TD><%= resultat.getString("Item") %></TD>
    <TD><%= resultat.getFloat("Price") %></TD>
  </TR>
  </blx:sqlExecuteQuery>
</TABLE>
```

В этом примере результат выполнения запроса помещается в переменную `resultat`, доступ к которой осуществляется из тела элемента, описанного ярлыком. Запрос определяется в элементе `blx:sqlQuery`:

```
<blx:sqlQuery  id="mojZaprosa">
SELECT Item, Price FROM PriceTable
</blx:sqlQuery>
```

Строка запроса может содержать выражения JSP, например,

```
<blx:sqlQuery  id="mojZaprosa">
SELECT Item, Price FROM <%= tableName %>
</blx:sqlQuery>
```

Необходимо указать базу данных, к которой направляется запрос. Команда выполнения запроса `<blx:sqlExecuteQuery>` должна быть размещена в теле элемента `<blx:sqlConnection>`. Другой способ указания базы данных — задание ее имени в ярлыке запроса. Имя базы данных (источник данных, имя связи) должно быть указано в файле конфигурации Web-сервера.

Для получения единственного значения можно воспользоваться ярлыком `<blx:sqlGet>`, например,

```
<P>Nubmer of Total Items =
<blx:sqlGet query="SELECT COUNT(*) FROM Items"/>
```

Ярлык *blx:sqlConnection*

Имя данных JDBC как имя JNDI.

Атрибуты:

- `jndiName` — имя JNDI для связи с базой данных. Обязателен.

Ярлык *blx:sqlQuery*

Ярлык используется для записи SQL-запроса (который может включать в себя выражения JSP). Запросу сопоставляется `id`, по которому запрос затем может быть вызван и использован.

Атрибуты:

- `id` — идентификатор запроса, по которому запрос может быть доступен из других ярлыков. Обязателен.

Ярлык *blx:sqlExecuteQuery*

Ярлык выполнения запроса. Имеет один обязательный атрибут `resultSet`. Запрос может быть указан либо в виде значения атрибута `query`, либо путем указания идентификатора `id` отдельного элемента запроса `sqlQuery` в атрибуте `queryRef`. Если этот ярлык используется вне элемента `sqlConnection`, то необходимо задание значения атрибута `connection`. Атрибуты описаны в табл. П3.6.

Таблица П3.6. *Атрибуты ярлыка* `blx:sqlExecuteQuery`

Имя атрибута	Описание	Обязателен или нет
`resultSet`	Имя переменной, которая будет создана и доступна внутри тела элемента, описанного ярлыком. Ссылка на объект типа `java.sql.ResultSet`	Обязателен
`Query`	Строка запроса	Необязателен
`queryRef`	Ссылка на идентификатор `id` запроса в ярлыке `<blx:sqlQuery>`	Необязателен
`connection`	Имя JNDI источника данных	Необязателен

Ярлык *blx:sqlGet*

Ярлык используется для выполнения запроса, который возвращает единственное значение. Результат выводится в поток JSP-вывода. Строка запроса задается либо в виде значение атрибута `query`, либо используется идентификатор существующего запроса в элементе `sqlQuery`, идентификатор указывается как значение атрибута `queryRef`. Атрибуты описаны ниже (табл. П3.7).

Таблица П3.7. *Атрибуты ярлыка* `blx:sqlGet`

Имя атрибута	Описание	Обязателен или нет
`Query`	Строка запроса	Необязателен

Таблица П3.7 *(окончание)*

Имя атрибута	Описание	Обязателен или нет
`QueryRef`	Ссылка на элемент `blx:sqlQuery` по идентификатору `id`, содержащий строку запроса	Необязателен
`connection`	Имя JNDI источника данных	Необязателен

Ярлык *blx:sqlExecuteUpdate*

Этот ярлык используется для выполнения запросов, которые содержат инструкции `INSERT` или `UPDATE`. Запрос не имеет возвращаемых результатов. Строка запроса может быть указана в качестве значения параметра `query` или путем ссылки на существующий элемент запроса `sqlQuery` путем указания идентификатора `id` этого элемента в качестве значения атрибута `queryRef`. Если элемент запроса не вложен в тело элемента `sqlConnection`, то необходимо указать имя JNDI источника данных в качестве значения атрибута `connection`. Список атрибутов приведен в табл. П3.8.

Таблица П3.8. *Атрибуты ярлыка* `blx:sqlExecuteUpdate`

Имя атрибута	Описание	Обязателен или нет
`Query`	Строка запроса	Необязателен
`QueryRef`	Ссылка на запрос в элементе `<blx:sqlQuery>`	Необязателен
`Connection`	Имя JNDI источника данных	Необязателен

Ярлыки для работы с естественными языками

Сервер Blazix предлагает использовать ярлыки работы с естественными языками для выделения фрагментов JSP-страниц, использующих естественные языки, и отображения различных вариантов (различных переводов) во время просмотра страниц. Существует два ярлыка: `<blx:xltFile>` и `<blx:xlt>`.

Текст, который должен быть переведен, помещается между ярлыками `<blx:xltFile>` и `</blx:xltFile>`. Этот элемент содержит имя (базовое) файла, имя файла должно содержать суффикс, соответствующий языку. Так, например, если базовое имя файла filename, а язык пользователя немецкий, то полное имя файла будет `filename_de`, поскольку код немецкого языка de. Если базовое имя файла с расширением `translate.txt`, то полное имя для французского языка будет `translate_fr.txt`. Ярлык `<blx:xlt>` используется для задания текста, который следует перевести.

Пример

```
<blx:xltFile file="translate.txt">
<P><blx:xlt ref="intro">
Introduction to
this site..
</blx:xlt><br>
<blx:xlt ref="getname">Enter your name here</blx:xlt>
<FORM METHOD=POST ACTION=savename.jsp>
<INPUT NAME=name TYPE=TEXT>
<INPUT TYPE=SUBMIT VALUE=<blx:xlt ref="submit">Submit</blx:xlt>>
</FORM>
</blx:xltFile>
```

Файл перевода имеет такой формат:

```
tag=value
```

На одну строку приходится один фрагмент перевода. Строки, начинающиеся с "-" — это продолжение предыдущих строк (черта будет заменена знаком конца строки). В качестве иллюстрации можно привести файл перевода для примера, описанного выше, файл `translate_fr.txt`. Он может содержать следующий текст.

Пример translate_fr.txt

```
intro=Introduction a
-ce site
getname=ecrivez votre nom ici
submit=Soumettent
```

Ярлык *blx:xltFile*

Ярлык используется для создания интернационализованных страниц JSP.

Атрибуты:

- `File` — имя файла перевода. Обязательный атрибут.

Ярлык *blx:xlt*

В элементе помещается текст для перевода. Ярлык располагается в элементе `xltFile`.

Атрибуты:

- `Ref` — имя ссылки. Обязательный атрибут.

Приложение 4

Основы Java

Вводная часть

Компьютер — сложная система, состоящая из многих частей. Одна из главнейших составляющих компьютера — это центральный процессор (ЦП). Именно процессор производит вычисления. В современном компьютере процессор — это одна отдельная микросхема, один чип, квадрат, сторона которого составляет несколько сантиметров. Задача процессора — выполнять программы. Программа — это набор инструкций, которые компьютер выполняет автоматически. Компьютер реализован таким образом, что он понимает инструкции, которые написаны на специальном языке — машинном языке. Каждый тип компьютеров имеет свой машинный язык. Компьютер может выполнять программу, если она написана на его машинном языке. Он может выполнять также программы, созданные на других языках, если первоначально эти программы будут переведены на машинный язык.

Во время выполнения программы компьютером, программа хранится в оперативной памяти компьютера (память прямого доступа, RAM, Random Access Memory, ОЗУ, оперативное запоминающее устройство). Кроме команд программы память может содержать данные, которые необходимы для работы программы. Доступ к памяти осуществляется по адресам. Когда процессор требует выполнения инструкции программы или ему необходимо получить данные из памяти, он посылает адрес в виде сигнала к памяти, а память в ответ посылает содержимое, хранимое по этому адресу. Процессор подбирает инструкции из памяти машины одну за другой по порядку и выполняет их.

Работа процессора состоит в том, что он получает инструкции из памяти компьютера и выполняет их. Помимо процессора компьютер включает в свой состав жесткий диск. Жесткий диск используется для хранения программ и данных в виде файлов. Для работы с компьютером к нему подключают различные периферийные устройства: клавиатуру, мышь, монитор, принтер, модем, сетевую карту, сканер и т. д. Клавиатура и мышь нужны для того, чтобы можно было осуществлять ввод информации в компьютер. Монитор и принтер служат для вывода информации пользователю. Модем

позволяет компьютерам общаться друг с другом по телефонной линии. Сетевая карта используется для объединения двух или нескольких компьютеров в сеть. При помощи сканера можно осуществлять перевод изображений в двоичные данные и хранить их на компьютере. Для каждого их этих устройств используются драйверы устройств, которые представляют собой программное обеспечение, при помощи которого происходит взаимодействие процессора с тем или иным устройством. Устройства присоединяются к компьютеру, образуя с ним единое целое. Присоединение устройств осуществляется при помощи одной или нескольких шин. Шина — это набор проводников, которые несут на себе различную нагрузку, исполняя те или иные функции, передавая от одного устройства к другому информацию разного назначения: данные, адреса, управляющую информацию (рис. П4.1). Адреса направляют данные к конкретному устройству и в конкретное место этого устройства. Сигналы управления могут, например, служит для передачи служебных сообщений от одного устройства к другому.

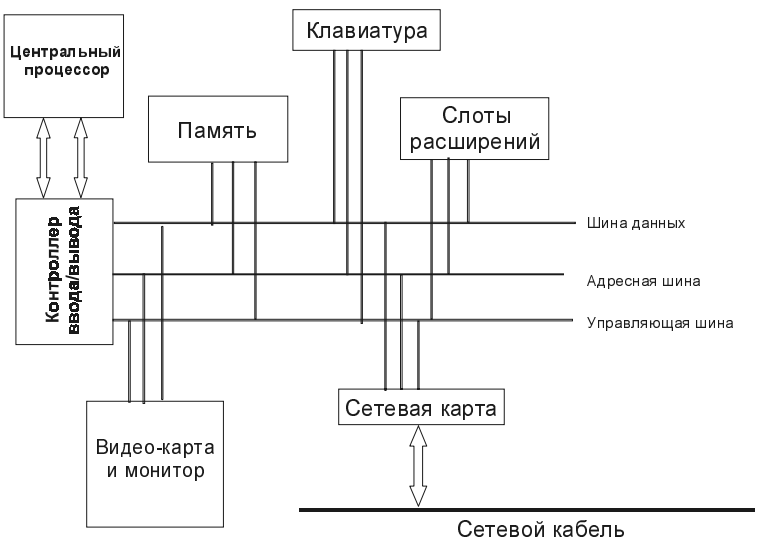


Рис. П4.1. Ввод-вывод происходит через шину

Процессор получает информацию от устройств, используя механизм постоянной проверки состояния присоединенных устройств. Проверка повторяется постоянно раз за разом, как только будут обнаружены данные на устройстве, процессор тут же их обработает. Такой процесс не очень эффективен, процессор тратит все время на ожидание данных впустую. Вместо прямого прослушивания устройств в ожидании вводимой информации используются прерывания. Прерывание — это сигнал, создаваемый устройством и посылаемый процессору (прерывание как бы говорит, что на устройстве появились данные). Полученное прерывание приводит к тому, что процессор на

время откладывает выполняемые задачи и обрабатывает информацию, которую получает от устройства. После того как процессор обработает полученную от устройства информацию, он возвращается к выполнению прежней работы. Например, при нажатии клавиши на клавиатуре, она посылает сигнал прерывания процессору, процессор в ответ на полученное прерывание прекращает исполнение текущих команд, читает данные о том, какая клавиша была нажата, обрабатывает полученное значение и затем возвращается к выполнению отложенного задания. Весь этот процесс происходит автоматически и незаметен для пользователя за счет обработчика прерываний, который является составной частью драйвера устройства.

Прерывания позволяют процессору работать с асинхронными сообщениями. При обычном ходе выполнения программы все инструкции выполняются синхронно, т. е. в заранее установленной последовательности, прерывания позволяют обрабатывать асинхронные сообщения, которые процессор получает непредсказуемым образом в произвольный момент. Еще один пример работы с прерываниями — получение данных с жесткого диска. Процессор может читать данные напрямую только из оперативной памяти. Перед тем как данные, хранимые на жестком диске, станут доступны, их необходимо скопировать в оперативную память компьютера. По сравнению со скоростью работы процессора и оперативной памяти жесткий диск работает чрезвычайно медленно. Если процессору требуются данные с жесткого диска, то он посылает сообщение драйверу диска о том, чтобы требуемые данные были найдены и приготовлены для чтения. Этот запрос посылается синхронно, в составе обычного потока выполнения программы. После этого процессор продолжает работу, не выделяя ресурсов для ожидания данных с жесткого диска. Когда данные становятся готовы, процессору посылается сигнал прерывания, после чего обработчик прерывания может прочитать полученные данные.

Все описанные процессы могут происходить только в том случае, если процессор может одновременно выполнять несколько заданий. Все современные компьютеры многозадачны, т. е. могут выполнять несколько задач одновременно. Многозадачность тесно связана с разделением времени. Весь цикл работы процессора разделяется между несколькими выполняемыми задачами. Например, пользователь может набирать текст одновременно с процессом получения информации по сети, при этом система может выводить на экран изображения часов и состояние процесса скачивания файла. Каждая отдельная задача, выполняемая процессором, называется процессом, или потоком. Существуют некоторые технические детали, которые отличают процессы от потоков, сейчас мы не будем на них останавливаться. Процессор продолжает работать с отдельным процессом до тех пор, пока не наступит одно из следующих событий:

- процесс отдает управление другому процессу;
- процесс должен подождать наступления тех или иных асинхронных событий, до наступления которых процесс блокируется. Когда наступает ожидаемое асинхронное событие, процесс возобновляет работу;

❒ процесс превышает лимиты времени и останавливается.

Работа с потоками и событиями является важной частью при программировании на языке Java. Программисту нет необходимости работать с прерываниями, программист сосредоточен на работе с обработчиками событий.

Программное обеспечение, которое занимается работой, связанной с обработкой прерываний от устройств, называется операционной системой. Помимо операционной системы существует богатый выбор программного обеспечения, включая программы для работы с текстами (текстовые процессоры), программы для просмотра ресурсов Интернета (браузеры) и т. п. Наиболее распространенными операционными системами в настоящее время являются системы FreeBSD, Linux, DOS, Windows 98, Windows 2000, Windows XP, Macintosh OS.

Машинный язык состоит из набора простых инструкций, которые выполняются центральным процессором. Большинство программ создаются на языках высокого уровня, например, на языках Java или C++. Программа, написанная на языке высого уровня, не может быть непосредственно выполнена на машине. Для трансляции такой программы, перевода ее на машинный язык требуется компилятор. Компилятор читает высокоуровневую программу и переводит ее на язык машинных команд. После завершения компиляции программа на машинном языке может быть выполнена. Для выполнения программы на другом типе компьютера требуется скомпилировать программу заново, переведя ее на другой машинный язык.

Виртуальная машина Java

Существует альтернативный вариант. Компилятор переводит всю программу целиком, преобразуя ее в машинные команды. Затем целая программа на машинном языке выполняется процессором. Можно также воспользоваться интерпретатором. Интерпретатор переводит команды высокоуровнего языка последовательно одну за другой в процессе выполнения программы. При этом не происходит перевод всей исходной программы целиком сразу.

Создатели языка Java объединили свойства компилятора и интерпретатора. Команды языка Java компилируются в машинный язык, но сама эта машина (в физическом смысле, процессор, память и т. п.) не существует. Она назвается "Виртуальная машина Ява". Машинным языком для виртуальной машины Ява является Ява-код. Компания Sun Microsystems (разработчик языка Java) реализовала процессор, машинным языком которого является Ява-код. Язык Java может работать на любом компьютере (рис. П4.2).

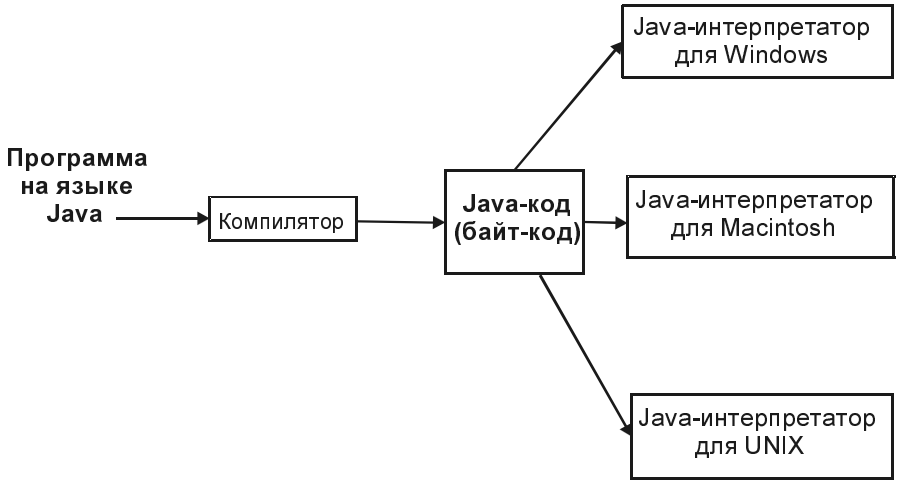


Рис. П.4.2. Программа, написанная на языке Java, может работать на любой платформе

Многие программы, написанные на языке Java, передаются по сети Интернет. Это вызывает некоторые проблемы, связанные с безопасностью. Ява-код служит своего рода буфером. Интерпретатор Ява-кода служит своего рода защитой, так как Ява-код при этом выполняется не напрямую, а при помощи интерпретатора.

Основные блоки программы

В программировании существует два важных элемента, составляющих программу, — это данные и инструкции. При работе с данными требуется знание того, что такое переменные и типы, а при работе с инструкциями — понимание того, как происходит передача управления от одной инструкции к другой и что такое процедуры (функции, методы). Переменные — это элементы памяти, которые имеют имя. При помощи имени доступ к памяти в программах осуществляется в наиболее наглядной форме. Программист работает только с именем, но не с самой памятью. В большинстве языков программирования переменные обладают типом, который указывает, какие данные хранятся в переменной. Например, это могут быть целые числа, такие как 3, −7, 0, другой тип переменных используется для хранения вещественных чисел 3.14, −2.7, 17.0 и т. п. Например, для компьютера существует разница между числом 17 и числом 17.0. В первом случае мы имеем целое число, а во втором — вещественное число. Также существует тип, который представляет отдельный символ, например, `A`, `;`. Есть свой тип для представления строк, например "Hello".

Программные языки всегда содержат такие команды, которые помогают получить данные, хранящиеся в переменных. Вот, например, инструкция присваивания, которая говорит, что следует взять данные из переменной `principal`, умножить их на число 0.07 и записать результат в переменную `interest`:

```
interest = principal * 0.07;
```

Существуют также команды ввода, с помощью которых программа читает данные, вводимые пользователем, или получает их с жесткого диска. Команды вывода служат для выполнения противоположных действий.

Программа — это последовательность инструкций. Инструкции выполняются одна за одной последовательно. Существуют также специальные инструкции, которые управляют ходом выполнения программы, они могут менять последовательность выполнения инструкций. Инструкции, осуществляющие управление ходом выполнения программы, подразделяются на два типа: циклы и ветвления. Циклы позволяют выполнять одну и ту же последовательность инструкций снова и снова много раз. Ветвления используются для того, чтобы произвести выбор направления, по которому будет продолжаться дальнейшее выполнение программы.

Например, есть задача: если значение переменной `principal` будет больше, чем 10 000, то переменная `interest` вычисляется путем умножения `principal` на 0.05, а если значение `principal` будет меньше, чем 10 000, то умножать надо на 0.04. Для решения такой задачи в языке Java используется инструкция `if`, например:

```
if (principal > 10000)
    interest = principal * 0.05;
else
    interest = principal * 0.04;
```

Большие программы оказываются весьма сложными. Их разбивают на более мелкие "кусочки", создавая подпрограммы. Подпрограммы состоят из инструкций, собранных вместе, образующих как бы одно целое, у которого есть имя. Это имя используется для того, чтобы выполнить весь набор инструкций, записанных в подпрограмме. Например, если мы хотим нарисовать дом, назовем подпрограмму рисования дома `risovat()`. Вызов этой подпрограммы будет осуществляться при помощи следующей инструкции:

```
risovat();
```

Переменные, типы, циклы, ветвления — все эти понятия относятся к традиционным методам программирования. Традиционным оно названо постольку, поскольку такой метод программирования использовался до середины 80-х годов XX-го века. Сейчас ему наиболее походит другой термин — процедурное программирование. По мере усложнения программ появились альтернативные подходы, возникло объектно-ориентированное программирование.

Объектно-ориентированное программирование

Конкретная семантика прикладных систем может быть выражена с помощью различных абстрактных синтаксисов ($АС_1$, ..., $АС_n$), каждый из которых использует одинаковые или различные синтаксисы передачи ($СП_1$, ..., $СП_m$).

ЗКР. "Сервис открытых ИВС"

Программу следует сначала сконструировать, разработать. Никто не сможет сесть за компьютер и создать программу, какой бы простой или сложной она ни была, сразу, без предварительной подготовки. Вначале появляется идея, затем создается подход, конструируется модель, затем разрабатывается программа. Структурное программирование, распространенное в 80-е годы XX-го века, для решения больших и сложных задач использовало метод, когда задача разбивалась на части и каждая часть решалась самостоятельно. Часть в свою очередь могла быть разбита на набор, состоящий из еще более мелких частей.

В объектно-ориентированном программировании (ООП) основное место уделяется понятию объекта. *Объект* — это своеобразный программный модуль, который содержит в себе данные и функции (другой термин для подпрограмм). Такой модуль представляет собой самодостаточный самостоятельный элемент, имеющий свое состояние (входящие в него данные) и который может реагировать на сообщения. Объект можно представлять себе в виде самостоятельного модуля в памяти компьютера, который может реагировать на сообщения. Объект должен "знать", как реагировать на определенные типы сообщений. Различные объекты могут реагировать на одно и то же сообщение совершенно разными способами. Это свойство называется *полиморфизмом*.

Объекты объединяются в классы. Если объекты содержат одинаковые типы данных и отвечают на одинаковые сообщения (пусть по-разному), то они принадлежат одному и тому же классу. Класс — это своего рода шаблон, на основе которого образуются объекты. Объекты — это разные экземпляры одного и того же класса.

Рассмотрим гипотетическую программу для рисования линий, прямоугольников, овалов, кривых. Пусть у нас будет пять классов объектов, по одному классу для каждого типа объектов. Все пять классов имеют тесную взаимосвязь, они могут быть нарисованы. Все рисуемые объекты распадаются на объекты, определяемые двумя точками, и объекты, задаваемые по нескольким точкам (необязательно по двум) — рис. П4.3.

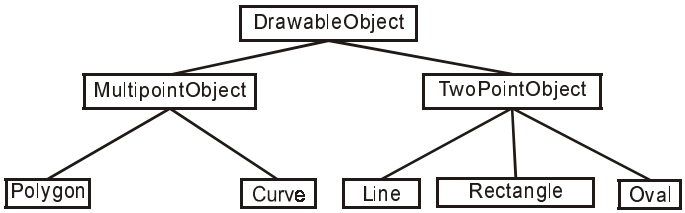


Рис. П4.3. Объекты для рисования

Объекты `DrawableObject`, `MultipointObject`, `TwoPointObject` образуют классы в программе. Объекты `MultipointObject` и `TwoPointObject` — это подклассы класса `DrawableObject`. Класс `Line` — подкласс класса `TwoPointObject` и класса `DrawableObject`.

Подкласс наследует свойства родительского класса. Подкласс может иметь свойства, которые не определены в суперклассе (родительском классе), подкласс может переопределять свойства и методы суперкласса (изменять методы). Наследование предоставляет мощные средства при создании программ. Класс можно использовать много раз. Класс можно использовать для создания подкласса на основе исходного класса.

Современный интерфейс пользователя

Первые персональные компьютеры использовали режим командной строки для ввода выполняемых инструкций и последующего их исполнения процессором. Таков интерфейс командной строки. Современные компьютеры используют графический интерфейс пользователя (GUI, Graphical User Interface). Пользователь имеет возможность работать с клавиатурой и мышью на полноценном графическом экране компьютера. В языке Java для создания элементов графического интерфейса пользователя используется набор классов пакета AWT (рис. П4.4).

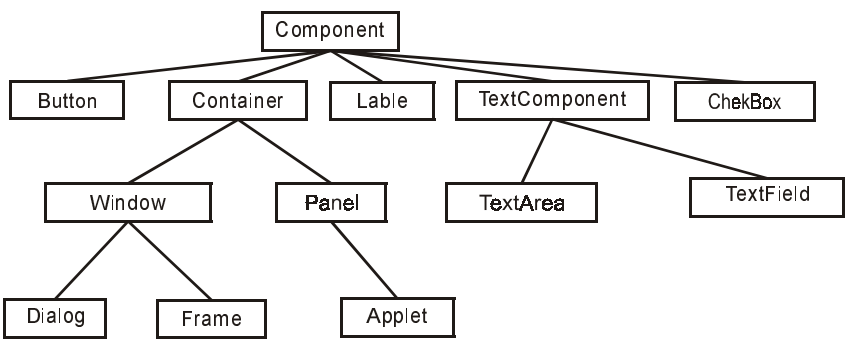


Рис. П4.4. Классы для работы с графическими компонентами

Интернет и сетевые протоколы

Компьютеры могут быть соединены друг с другом, образуя сети. Компьютеры одной сети могут общаться друг с другом, обмениваясь данными и файлами, посылая и получая сообщения. Компьютеры, объединенные в одну сеть, могут работать над одним большим заданием совместно, распределяя собственные ресурсы. Миллионы компьютеров объединены в одну большую сеть, называемую Интернет. Чтобы позволить компьютерам общаться друг с другом, разработано множество протоколов, т. е. правил, по которым компьютеры общаются друг с другом. Основным протоколом, используемым в Интернете, является протокол IP (интернет-протокол, Internet Protocol). Он определяет то, как данные передаются физически от одного компьютера к другому. Кроме этого используется протокол TCP (протокол управления передачей, Transmission Control Protocol), при помощи этого протокола происходит контроль за передаваемыми данными так, чтобы мы могли быть уверены, что переданные данные получены по назначению. Эти два протокола, обозначаемые как TCP/IP, служат основой для установления соединений, передачи информации, файлов, электронной почты и т. п. Вся связь через Интернет осуществляется путем передачи пакетов. Пакет состоит из данных и пересылается от одного компьютера к другому, в пакете помимо данных находится информация об IP-адресах отправителя и получателя. Можно представить себе IP-пакет в виде конверта, на котором написан адрес. Конверт также содержит обратный адрес, т. е. адрес отправителя. Пакет может содержать в себе ограниченный объем данных. Большие сообщения разбиваются и упаковываются в несколько пакетов. Затем пакеты посылаются самостоятельно по сети (при этом физически они могут пройти разные пути от отправителя к получателю), а получатель собирает из пакетов исходное сообщение.

Каждый компьютер в сети Интернет имеет собственный IP-адрес, который однозначно идентифицирует этот компьютер во всемирной сети. Компьютер может послать данные (пакеты) другому компьютеру в сети только при том условии, что он знает IP-адрес компьютера-партнера. IP-адрес служит в качестве адреса местоназначения IP-пакета. Поскольку людям удобнее использовать имена, а интернет-адрес IP — это набор чисел, то существует способ идентифицировать комьютеры при помощи доменных имен. Например, доменное имя может быть **www.yahoo.com**.

Протокол TCP/IP используется для передачи данных различных типов. Среди них наиболее известные службы такие, как удаленный терминальный доступ, электронная почта, FTP (File Transfer Protocol, протокол передачи файлов), WWW.

Удаленный терминальный доступ позволяет пользователю с одного компьютера войти в другой компьютер. Для осуществления удаленного доступа используется несколько различных протоколов, например, протокол telnet или

безопасный протокол ssh (безопасный терминал). Оба протокола предоставляют интерфейс командной строки. Первый компьютер, за которым находится пользователь, является терминалом для удаленного компьютера, к которому подключается этот пользователь. Telnet как правило используется для подключения к домашнему компьютеру, когда пользователь находится на расстоянии.

Электронная почта предоставляет возможности обмениваться сообщениями. Почтовое сообщение посылается отдельным пользователем другому пользователю. Каждый пользователь электронной почты идентифицируется по электронному адресу в формате "username@domain.name". Почта передается, как правило, с использованием протокола SMTP (Simple Mail Transfer Protocol — простой протокол передачи почты).

World Wide Web (WWW) — всемирная паутина, которая главным образом состоит из Web-станичек, содержащих ссылки. Эти странички просматриваются с использованием специальных программ-браузеров. Для передачи страничек используется протокол HTTP (HyperText Transfer Protocol) — протокол передачи гипертекста. При помощи этого протокола общаются Web-серверы с браузерами.

Контрольные вопросы

1. Что такое процессор в компьютере, каково его назначение?

   Ответ. Центральный процессор выполняет программы, закодированные на машинном языке, которые хранятся в оперативной памяти компьютера (память прямого доступа). Процессор раз за разом выбирает из памяти команды и выполняет их.

2. Что такое асинхронные события? Приведите примеры.

   Ответ. Асинхронные события — это такие события, которые наступают непредсказуемо и вне контроля из выполняемой процессором программы, они не синхронизированы с ходом выполнения этой программы. Примеры — нажатие клавиши на клавиатуре, чтение данных с диска.

3. Каковы различия между компилятором и транслятором?

   Ответ. И компилятор, и транслятор выполняют похожие функции. Они переводят программу, написанную на языке программирования, на машинный язык. Компилятор производит весь перевод сразу, создавая законченный самостоятельный фрагмент команд на машинном языке. Эта программа, написанная на машинном языке, может быть выполнена компьютером. Транслятор переводит последовательно по одной инструкции программы, переходя от одной инструкции к другой по мере выполнения программы. После того как очередная инструкция будет переведена на машинный язык, она будет немедленно выполнена и интерпертатор перейдет к другой команде языка программирования. В языке Java ис-

пользуется компилятор для трансляции программы в Java-код, который является машинным языком для воображаемой виртуальной машины Java.

4. В чем различия между языками высокого уровня и машинным языком?

   Ответ. Программы, написанные на машинном языке определенного типа, могут быть выполнены на компьютере непосредственно, этот язык понятен процессору. Перед выполнением программ, написанных на языках высокого уровня, программы необходимо перевести в машинный язык. Машинный язык содержит инструкции, которые представляют собой набор двоичных состояний, т. е. набор нулей и единиц, он не предназначен для чтения человеком. Язык высокого уровня использует синтаксис, который более понятен и привычен человеку.

5. Когда у вас есть исходный код на языке Java и вы хотите выполнить эту программу, то вам понадобятся и компилятор, и интерпретатор. Что делает компилятор Java? Зачем нужен интерпретатор Java?

   Ответ. Компилятор Java переводит Java-программу в Java-код. Несмотря на то, что Java-код является машинным языком, в действительности не существует компьютера, который бы использовал этот язык в качестве своего машинного языка. Интерпретатор Java используется для выполнения Java-кода на конкретном компьютере.

6. Что такое функция (подпрограмма)?

   Ответ. Подпрограмма — это набор инструкций для выполнения того или иного задания, эти инструкции объединены вместе. Функции присваивается имя. Про вызове функции по этому имени все инструкции, составляющие функцию, будут выполнены в том виде, как они написаны. Функции позволяют выполнять один и тот же набор инструкций несколько раз, обращаясь к нему по имени функции.

7. Java — это объектно-ориентированный язык программирования. Что такое объект?

   Ответ. Объект состоит из данных и функций, которые работают с данными. Объект — это самостоятельный модуль, который взаимодействует с окружением, с другими объектами. Программный объект — это аналог объекта из реальной жизни.

8. Что такое переменная? (Можно выделить четыре идеи, имеющие отношение к понятию переменной. Постарайтесь вспомнить все четыре аспекта. Подсказка — один из аспектов связан с именем переменной.)

   Ответ. Переменная — это область памяти, с которой сопоставлено имя. С помощью имени легко осуществляется доступ к памяти. Переменная содержит значение определенного типа, которое может быть изменено в ходе выполнения программы. Четыре аспекта: область памяти, имя, значение, тип.

9. Java — платформенно-независимый язык. Что это значит?

   Ответ. Программа Java может быть скомпилирована в Java-код. Этот Java-код может быть выполнен на любом компьютере, если на нем установлена виртуальная машина Java. Программы, созданные на других языках, должны быть скомпилированы вновь для каждого нового типа машины.

10. Что такое Интернет? Приведите пример, как его можно использовать. (Какие службы существуют в Интернете?)

    Ответ. Интернет — это сеть, объединяющая в себе миллионы компьютеров по всему миру. Компьютеры, объединенные этой сетью, могут общаться друг с другом. Для общения используется ряд служб, например, Telnet, FTP, WWW.

Основные понятия

Программа — это последовательность инструкций, созданных на языке программирования. Язык программирования отличается от обычных языков людей, используемых в повседневном общении. Язык программирования жестко регламентирован, от его правил нельзя отступать. Правила, которые определяет, как устроен язык, составляют синтаксис языка. Правила синтаксиса определяют словарь языка, как конструируется программа с использованием таких структур, как, например, циклы, ветвления, функции. Только синтаксически правильная программа может быть скомпилирована. Программы с ошибками будут отвергнуты. Программист должен создать такую программу, которая не будет содержать ни одной синтаксической ошибки. Для этого программисту необходимы детальные знания правил синтаксиса. Помимо синтаксической корректности программа должна быть осмысленна, т. е. выполнять логически обоснованную последовательность действий. Смысл программы — это семантика программы. Семантически верная программа считает именно то, что программист подразумевал при ее создании.

Создадим простейшую программу. Процесс создания состоит из трех основных частей:

- ввод текста программы в компьютер;
- компиляция введенного кода;
- выполнение скомпилированного кода.

Простейший код программы будет выглядеть так:

```
public class HelloWorld {
  // A program to display the message
  // "Hello, World!" on standard output
```

```
    public static void main(String[] args) {
       System.out.println("Hello, World!");
}
  }
```

Эта программа выводит сообщение "Hello, World!" Вывод этого сообщения осуществляется при помощи команды

```
System.out.println("Hello, World!");
```

Это пример вызова функции. Здесь вызвана функция `System.out.println`. *Функция* — это набор нескольких инструкций, объединенных вместе и образующих единое целое с заданным именем. В данном случае мы используем встроенную функцию. Встроенная функция — это заранее определенная функция, которая является неотъемлемой частью языка. Программа содержит в себе комментарии. Комментарии при работе программы полностью игнорируются. В данном случае строки комментария начинаются с символов `//` и заканчиваются с концом строки. Многострочные комментарии заключаются между символами `/*` и `*/`. Все, кроме комментариев, подчинено правилам синтаксиса языка. В первой строке нашей программы мы определяем класс с именем `HelloWorld`. Отметим, что не всякий класс является программой. Для того чтобы класс стал самостоятельной программой, в нем должна содержаться функция `main()`, которая определяется таким образом:

```
public static void main(String[] args) {
   statements
   }
```

При обращении к интерпретатору Java при запуске программы, интерпретатор обращается к функции `main()`. Слово `public` означает, что эта функция может быть вызвана извне, т. е. не из самой программы. Это важно, поскольку функция вызывается интерпретатором. Значения остальных слов станут понятны позже. Между фигурными скобками располагается набор инструкций, которые будут вызваны и выполнены при обращении к функции.

Апплеты создаются по-другому:

```
public class Interest1Console extends ConsoleApplet {

   protected String getTitle() {
      return "Sample program \"Interest1\"";
   }

   protected void program() {
      double principal;     // вложенная сумма
```

```
      double rate;          // годовой процент
      double interest;      // процент за год

      /* вычисления */
      principal = 17000;
      rate = 0.07;
      interest = principal * rate;   // процент
          principal = principal + interest;
          /* вывод результата */
      console.put("The interest earned is $");
      console.putln(interest);
      console.put("The value of the investment after one year is $");
      console.putln(principal);
   }
```

При этом используется следующая вспомогательная программа, содержащаяся в файле ConsoleApplet.java:

```
import java.awt.*;
import java.awt.event.*;
public class ConsoleApplet extends java.applet.Applet
                           implements Runnable, ActionListener {
   protected String title = "Java Console I/O";
   protected String getTitle() {
      return title;
   }
   protected  ConsolePanel console;
   protected void program() {
      console.putln("Hello, World!");
   }
   private Button runButton;
   private Thread programThread = null;
     // поток run()
   private boolean programRunning = false;
   private boolean firstTime = true;
//     false — если программа считает первый раз

   public void run() {   // запуск
      programRunning = true;
```

```
      program();
      programRunning = false;
      stopProgram();
   }

   synchronized private void startProgram() {
      runButton.setLabel("Abort Program");
      if (!firstTime) {
         console.clear();
         try { Thread.sleep(300); }
// задержка перед перезапуском программы
         catch (InterruptedException e) { }
      }
      firstTime = false;
      programThread = new Thread(this);
      programThread.start();
   }

   synchronized private void stopProgram() {
      if (programRunning) {
         programThread.stop();
         try { programThread.join(1000); }
         catch (InterruptedException e) { }
      }
      console.clearBuffers();
      programThread = null;
      programRunning = false;
      runButton.setLabel("Run Again");
      runButton.requestFocus();
   }

   public void init() {

      setBackground(Color.black);

      setLayout(new BorderLayout(2,2));
      console = new ConsolePanel();
      add("Center",console);

      Panel temp = new Panel();
```

```
      temp.setBackground(Color.white);
      Label lab = new Label(getTitle());
      temp.add(lab);
      lab.setForeground(new Color(180,0,0));
      add("North", temp);

      runButton = new Button("Run the Program");
      temp = new Panel();
      temp.setBackground(Color.white);
      temp.add(runButton);
      runButton.addActionListener(this);
      add("South",temp);
         }
   public Insets getInsets() {
      return new Insets(2,2,2,2);
   }
  public void stop() {
      if (programRunning) {
         stopProgram();
         console.putln();
         console.putln("*** PROGRAM HALTED");
      }
   }
   synchronized public void actionPerformed(ActionEvent evt) {
      if (programThread != null) {
         stopProgram();
         console.putln();
         console.putln("*** PROGRAM ABORTED BY USER");
      }
      else
         startProgram();
   }
}
```

Переменные и примитивные типы

Имена играют фундаментальную роль в программировании. При помощи имен называются разные вещи. Важно знать синтаксис и семантику при использовании имен. *Имя* — это последовательность одного или нескольких

символов. Имя должно начинаться с буквы и состоять из букв, цифр, символов подчеркивания (_). Например:

- `N`
- `n`
- `rate`
- `x15`
- `quite_a_long_name`
- `HelloWorld`

Заглавные буквы и строчные буквы считаются различными. Имена `HelloWorld`, `helloworld`, `HELLOWORLD`, `hElloWorLD` — это разные имена. Определенные имена являются зарезервированными и имеют специальное назначение и не могут быть использованы программистом для других целей, например:

- `class`
- `public`
- `static`
- `if`
- `else`
- `while`

и еще несколько десятков других имен.

Значение присваивается переменной при помощи оператора присваивания в виде:

```
variable = expression;
```

Например:

```
rate = 0.07;
interest = rate * principal;
```

Переменная служит для хранения только одного типа данных. Существуют типы, встроенные в Java, это примитивные типы:

- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `char`
- `boolean`

Тип `short` соответствует двум байтам, интервал значений от −32 768 до 32 767. Тип `int` соответствует 32 битам, интервал значений от −2 147 483 648 до 2 147 483 647. Тип `long` соответствует 64 битам, интервал значений от −9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Тип `float` занимает 4 байта памяти, максимальное значение примерно 10^{38}. Тип `char` занимает два байта памяти. Имя константы называется литералом. Для логического литерала существует два значения `true` и `false`.

Пример программы, в которой используются переменные и операторы присваивания:

```
public class Interest {
   public static void main(String[] args) {
                /* объявление переменных */
   double principal;     // вложенная сумма
   double rate;          // годовой процент
   double interest;      // начисление за год

                /* вычисления */
   principal = 17000;
   rate = 0.07;
   interest = principal * rate;  .
   principal = principal + interest;

                /* вывод результата */
   System.out.print("The interest earned is $");
   System.out.println(interest);
   System.out.print("The value of the investment after one year is $");
   System.out.println(principal);
        }
}
```

Строки, объекты, функции

В предыдущем разделе мы рассмотрели восемь примитивных типов и тип строк `String`. Существует принципиальное различие между примитивными типами и типом `String`. Дело в том, что `String` — это объект. В наших примерах будет использоваться класс `TextIO`. Код Java-программы (TextIO.java) приведен в конце настоящего приложения.

Классы в Java выполняют две важные функции. Первая состоит в том, что классы объединяют в себе переменные и функции, содержащиеся в этом классе. Переменные и функции становятся статическими членами класса,

так, например, функция `main()` — статическая функция класса. В определении таких функций используется слово `static`. Второе назначение классов — они описывают объекты. Класс — это тип, а объект — это значение этого типа. `String` — это название класса, который является частью языка Java. И строка, например, "Hello, World" — это значение типа `String`.

Всем известны математические функции, такие, как, например, квадратный корень. В языке Java существует аналогичная функция, `Math.sqrt`. Эта функция — статический член класса, называемого `Math`. Если `x` — это какое-либо числовое значение, то `Math.sqrt(x)` — это тоже значение, равное квадратному корню от `x`. Чтобы вывести это значение на печать, мы используем следующую команду:

```
System.out.print(Math.sqrt(x));
```

Можно использовать оператор присваивания для инициализирования переменной:

```
lengthOfSide = Math.sqrt(x);
```

Функция представляет переменную типа `double` и может применяться там, где используются переменные типа `double`.

Еще один пример математической функции:

```
Math.abs(x)
```

Эта функция вычисляет модуль (абсолютное значение). Среди математических функций есть и другие, например, `Math.sin(x)`, `Math.cos(x)`, `Math.tan(x)`, `Math.asin(x)`, `Math.acos(x)`, `Math.atan(x)`, `Math.exp(x)`, `Math.log(x)`, `Math.pow(x,y)`, `Math.floor(x)`, `Math.random()`.

Приведем пример программы, использующей эти функции:

```
public class TimedComputation {
  public static void main(String[] args) {
    long startTime; // стартовое время в миллисекундах
    long endTime;   // время при завершении вычислений, в миллисекундах
    double time;    // Разница времен, в секундах.
    startTime = System.currentTimeMillis();
    double width, height, hypotenuse;  // стороны треугольника
    width = 42.0;
    height = 17.0;
    hypotenuse = Math.sqrt(width*width + height*height);
    System.out.print("A triangle with sides 42 and 17 has hypotenuse ");
    System.out.println(hypotenuse);
    System.out.println("\nMathematically, sin(x)*sin(x) + "
                       + "cos(x)*cos(x) - 1 should be 0.");
    System.out.println("Let's check this for x = 1:");
```

```
    System.out.print("sin(1)*sin(1) + cos(1)*cos(1) - 1 is ");
    System.out.println(Math.sin(1)*Math.sin(1)
                       + Math.cos(1)*Math.cos(1) - 1);
    System.out.println("(There can be round-off errors when "
                         + " computing with real numbers!)");

    System.out.print("\nHere is a random number:  ");
    System.out.println(Math.random());
    endTime = System.currentTimeMillis();
    time = (endTime - startTime) / 1000.0;
    System.out.print("\nRun time in seconds was:  ");
    System.out.println(time);
 }
}
```

Величина типа `String` — это объект. Этот объект содержит данные, а именно, последовательность символов, составляющих строку. В нем есть также функции. Например, функция `length()` вычисляет длину строки. Строку можно создать при помощи следующей декларации:

```
String str;
str = "Seize the day!";
```

Еще один пример:

```
System.out.print("The number of characters in ");
System.out.println("the string \"Hello World\" is ");
System.out.println("Hello World".length());
```

Объект `String` содержит множество функций. Вот некоторые из них.

- `s1.equals(s2)` — возвращает логическое значение. `true` — если строка `s1` в точности такая же, как строка `s2`.
- `s1.equalsIgnoreCase(s2)` — то же самое, что и `s1.equals()`, но заглавные и строчные буквы считаются одинаковыми.
- `s1.length()` — целое значение, равное количеству символов в строке.
- `s1.charAt(N)` — значение типа `char`, символ, расположенный на позиции с номером N в строке, начиная с нулевой позиции.
- `s1.substring(N,M)` — тип `String`, строка с символами в позициях N, $N + 1$, ..., $M - 1$.
- `s1.indexOf(s2)` — целое, если `s2` является фрагментом `s1`, то возвращается номер позиции, с которого начинается `s2` в `s1`.
- `s1.compareTo(s2)` — целое, если строки равны, то 0.

- `s1.toUpperCase()` — строка, записанная заглавными буквами.
- `s1.trim()` — строка с удаленными непечатаемыми символами, такими как пробелы, табуляции и т. п.

Строки можно складывать:

```
System.out.println("Hello, " + name + ". Pleased to meet you!");
```

Примитивные типы можно прибавлять к строке, тогда они приводятся к строчному типу:

```
System.out.print("After ");
System.out.print(years);
System.out.print(" years, the value is ");
System.out.print(principal);
```

Это можно записать в виде:

```
System.out.print("After " + years + " years, the value is " + principal);
```

Выражения

Литералы, переменные, функции — это примеры простых выражений. Более сложные выражения состоят из простых выражений и операторов, например `A+B*C`, `B*C`.

Арифметические операторы

Арифметические операторы обозначаются знаками `+`, `-`, `*`, `/`.

Операторы увеличения и уменьшения на единицу

Эти операторы требуют только одного операнда, обозначаются `++` и `--`.

Пример

```
counter = counter + 1;
goalsScored = goalsScored + 1;
```

Эти выражения с помощью операторов увеличения и уменьшения можно записать следующим образом:

```
counter++;
goalsScored++;
```

Пример

```
y = x++;
y = ++x;
```

```
TextIO.putln(--x);
z = (++x) * (y--);
```

Если x равно 6, то после выполнения инструкции y = x++ y будет равно 6, а после выполнения инструкции y = ++x y будет равно 7. В обоих случаях новое значение для x будет 7.

Операторы сравнения

Операторы сравнения используются при работе с логическими величинами, значение их булево, т. е. true или false:

- A == B — A "равняется" B;
- A != B — A "не равняется" B;
- A < B — A "меньше чем" B;
- A > B — A "больше чем" B;
- A <= B — A "меньше или равно" B;
- A >= B — A "больше или равно" B.

Эти операторы можно использовать при работе с числовыми значениями в качестве операндов.

Пример

```
boolean sameSign;
sameSign = ((x > 0) == (y > 0));
```

Логические операторы

Логические операторы работают с логическими значениями. Это операторы and (И) (&&), or (ИЛИ) (||), not (НЕ) (!). Например:

```
(x != 0) && (y/x > 1)
test = ! test;
```

Условные операторы

Общий вид простого условного оператора:

```
boolean-expression ? expression-1 : expression-2
```

Если булево выражение имеет значение true, то выполняется expression-1, в противном случае выполняется expression-2. Например:

```
next = (N % 2 == 0) ? (N/2) : (3*N+1);
```

Операторы присваивания и приведения типа

Примеры

```
int A;
double X;
short B;
A = 17;
X = A;    // A приводится к типу double
B = A;    // нельзя, тип не приводится автоматически
Исправление:
int A;
short B;
A = 17;
B = (short)A;  // тип приводится в явном виде
```

При приведении некоторых типов могут произойти смысловые изменения значений, так (`short`) 100 000 равно 34 464. Тип `short` образуется путем взятия 4 битов от исходного значения, часть информации теряется.

Пример использования операторов приведения типа `(int)(6*Math.random())` — целое случайное число из набора 0, 1, 2, 3, 4, 5.

Можно приводить типы из целых к `char`, при этом берется значение соответствующего символа в соответствии с кодировкой Unicode. Например, `(char)97` — это 'a', а `(int)'+'` равно 43.

Примеры

```
x -= y;     //    x = x - y;
x *= y;     //    x = x * y;
x /= y;     //    x = x / y;
x %= y;     //    x = x % y;    (для целых x и y)
q &&= p;    //    q = q && p;   (для логических booleans q и p)
```

Комбинированные операторы присваивания можно использовать и со строками:

```
str += x
str = str + x
```

Иерархия операторов

Между операторами установлено старшинство. В последовательности операторов они выполняются не один за другим, как написано, а в соответствии

с правилами иерархии операторов. Приведем список операторов в соответствии с их положением в иерархии операторов. Чем выше в этом списке находится оператор, тем большим приоритетом он обладает. Из двух операторов с разным приоритетом в первую очереди выполняется такой оператор, приоритет которого выше.

Унарными операторами называются такие операторы, для выполнения которых необходим один операнд. Бинарные операторы требуют наличия двух операндов. Тернарные операторы работают с тремя операндами.

Список операторов в соответствии с иерархией:

1. Унарные операторы: ++, --, !, унарные — и +, оператор приведения типа.
2. Умножение и деление: *, /, %.
3. Сложение и вычитание: +, -.
4. Операторы отношений: <, >, <=, >=.
5. Операторы равенства и неравенства: ==, !=.
6. Логическое сложение "И": &&.
7. Логическое "ИЛИ": ||.
8. Условный оператор: ?:
9. Операторы присваивания: =, +=, -=, *=, /=, %=.

Операторы, находящиеся в одной строчке в списке, который приводится выше, имеют одинаковый приоритет. Операторы с одинаковым приоритетом выполняются в том порядке, в котором они появляются в тексте программы. При этом необходимо учитывать, что для унарных операторов и операторов присваивания этот порядок таков, что операторы выполняются по порядку справа налево. Остальные операторы выполняются по порядку слева направо. Порядок выполнения операторов можно изменять. Для этого используются круглые скобки. В первую очередь будут выполняться те операторы, которые заключены в скобки.

Управление ходом выполнения программы

В этом разделе мы рассмотрим структуры, которые используются для управления ходом выполнения программы.

Для того чтобы иметь возможность выполнять сложные задачи с помощью языка программирования, используются структуры, управляющие ходом выполнения программы. Существует шесть типов таких структур:

- блок;
- цикл `while`;

- цикл `do..while`;
- цикл `for`;
- инструкция `if`;
- инструкция `switch`.

Блок — простейшая конструкция. Это набор нескольких инструкций, помещенных в фигурные скобки. Блок используется для того, чтобы объединить несколько инструкций в одно целое, которое само по себе является одной самостоятельной инструкцией. Из нескольких инструкций мы получаем всего одну. Таким образом:

```
{
  инструкции
}
```

Блок может быть пустым, т. е. не содержать в себе ни одной инструкции. Мы уже встречались с функцией `main()`. Как и все прочие функции, эта функция является не чем иным, как блоком. Функция — это блок, все инструкции функции помещены между фигурных скобок, между `{` и `}`. Функция — не простой блок. Функция имеет имя, которому присваивается результат, получающийся при выполнении инструкций блока. Функция также имеет тип. Иными словами, функция — это блок с присвоенным именем, типом и, может быть, другими модификаторами. Переменная, определенная в блоке, доступна в пределах этого блока. Более точно, идентификатор (имя), описанный в блоке, доступен в пределах этого блока, начиная с того места, где расположено описание идентификатора (имени).

Приведем два примера блоков:

Примеры

```
{
    System.out.print("The answer is ");
    System.out.println(ans);
}
{  // обмен значениями между x и y
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Блок как таковой не влияет на ход выполнения программы, но блоки удобно использовать в других структурах. Пять оставшихся типов структур могут оказывать влияние на последовательность выполнения программы.

Циклы — это такие структуры, выполнение которых осуществляется большое число раз снова и снова до наступления определенных условий. Так, цикл `while` приведет к тому, что инструкция будет выполняться до тех пор, пока условие цикла будет принимать значение `true`. Как только условие примет значение `false`, выполнение инструкции прекратится.

```
while (логическое выражение-условие) инструкция
```

Поскольку блок — это инструкция, то цикл `while` может быть записан в виде

```
while (логическое выражение-условие)  {
  инструкции
}
```

Вот пример цикла, с помощью которого можно напечатать числа 1, 2, 3, 4, 5:

```
int number;
number = 1;
while (number < 6) {
  System.out.println(number);
  number = number + 1;
}
System.out.println("Gotovo!");
```

В программе цикл никогда не бывает расположен сам по себе, он всегда находится внутри функции. В свою очередь функция всегда располагается внутри того или иного класса. Приведем пример полной программы:

```
public class Interest3 {
   public static void main(String[] args) {
      double principal;  // Вносимая сумма.
      double rate;       // Годовой процент.
      /* получение значений вносимой суммы и процента от пользователя */
     TextIO.put("Enter the initial investment: ");
      principal = TextIO.getlnDouble();
      TextIO.put("Enter the annual interest rate: ");
      rate = TextIO.getlnDouble();
      /* На 5 лет. */
     int years;
     years = 0;
     while (years < 5) {
         double interest;  /
         interest = principal * rate;
         principal = principal + interest;
         years = years + 1;
         System.out.print("The value of the investment after ");
```

```
            System.out.print(years);
            System.out.print(" years is $");
            System.out.println(principal);
        }
    }
}
```

В этой программе используется специально созданный пользовательский класс `Text.IO`. Вместо этого класса можно воспользоваться стандартными классами, с которыми мы познакомимся далее.

Инструкция `if` заставляет компьютер проверить логическое значение условия. Если условие принимает значение `true`, то будет выполняться `инструкция1`, если значение `false`, то выполняется `инструкция2`:

```
if (выражение-условие)
      инструкция1
      else
          инструкция2
```

Можно использовать упрощенный вариант, при этом вторая инструкция отсутствует. Компьютер выполнит инструкцию `инструкция1` только в том случае, если значение выражения-условия будет `true`.

```
if (выражение-условие)
    инструкция1
```

С использованием блоков условие `if` можно записать в следующем виде:

```
if (выражение-условие) {
      инструкции
      }
    else {
        инструкции
  }
```

Его упрощенный аналог выглядит так:

```
if (выражение-условие) {
     инструкции
  }
```

В качестве примера приведем фрагмент программы, который меняет значения `x` и `y` только в том случае, если первоначально `x` больше, чем `y`.

```
if (x > y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Пример полного условия (включая `else`):

```
if (years > 1) {
    System.out.print("The value of the investment after ");
    System.out.print(years);
     System.out.print(" years is $");
 }
     else {
    System.out.print("The value of the investment after 1 year is $");
}
System.out.println(principal);
```

Разработка алгоритмов

Программировать не так уж и сложно. При написании программы мы сообщаем компьютеру различные детали о том, что необходимо сделать в целом. При этом мы ошибаться, все должно быть сделано абсолютно верно, нельзя допустить ни одной синтаксической ошибки.

Программа выражает идею. Программист начинает с того, что формулирует идею, которая должна затем превратиться в задание, выполняемое компьютером. Вероятно, программист имеет представление о том, как в принципе можно было бы решить поставленную задачу, например, вручную, какие действия шаг за шагом должны были бы при этом быть предприняты. Последовательность таких действий, общая схема выполнения задачи — это алгоритм. Он скрыт внутри логической структуры пока не существующей программы. Алгоритм не должен содержать все детальные шаги, которые будут описаны в программе. Он всего лишь вспомогательное средство. Алгоритм в случае необходимости может быть создан и записан на любом языке, например, на русском, или с использованием графических построений.

Зачем нужен алгоритм? Часто задача оказывается достаточно трудной и требует большой концентрации и большого количества работы. Составление алгоритма позволяет взглянуть на задачу в целом, понять ее более глубоко, найти и использовать более эффективные подходы.

Допустим, что перед нами стоит задача вывести баланс на счету на конец каждого года в течение 5 лет, который считается на основе данных, вводимых пользователем. Для этого мы используем следующий алгоритм.

1. Получить данные от пользователя.
2. Вычислить баланс по истечении первого года.
3. Показать баланс.

4. Вычислить баланс после 2 лет.
5. Показать баланс.
6. Вычислить баланс после 3 лет.
7. Показать баланс.
8. Вычислить баланс после 4 лет.
9. Показать баланс.
10. Вычислить баланс после 5 лет.

Такой алгоритм верен, но в нем очень много повторов. Его можно упростить:

1. Получить данные от пользователя.
2. До тех пор, пока необходимо считать:
   - вычислить баланс на следующий год;
   - показать баланс.

Этот алгоритм достаточно схематичен. Для компьютера следует указывать более детальные инструкции. Такие инструкции не обязательно записывать при составлении алгоритма. Степень детализации алгоритма зависит от конкретных потребностей разработчика. Вот, например, как можно записать алгоритм, детализируя пункт 1 ("Получить данные от пользователя"), сделав его более приближенным к языку программирования, т. е. тем инструкциям, которые должны быть указаны для машины:

1. Запросить у пользователя величину начального вложения.
2. Прочитать ответ пользователя.
3. Запросить у пользователя величину годового процента.
4. Прочитать ответ пользователя.

Оставшиеся три пункта алгоритма можно расписать следующим образом:

❒ до тех пор, пока следует вычислять:
   - вычислить годовой процент;
   - прибавить годовой процент к балансу;
   - показать баланс.

Для того чтобы проверить, следует ли продолжать процесс вычисления (пункт 5 в последнем примере), можно записать алгоритм:

1. `years = 0`.
2. `while years < 5`.
3. `years = years + 1`.
4. Вычислить процент.

5. Прибавить процент к балансу.
6. Показать баланс.

Итак, весь алгоритм будет выглядеть следующим образом:

1. Запросить у пользователя величину начального вложения.
2. Прочитать ответ пользователя.
3. Запросить у пользователя величину годового процента.
4. Прочитать ответ пользователя.
5. `years = 0.`
6. `while years < 5.`
7. `years = years + 1.`
8. Вычислить процент.
9. Прибавить процент к балансу.
10. Показать баланс.

Такой алгоритм (он весьма детален) можно перевести в программу, написанную с соблюдением всех правил синтаксиса:

```
double principal, rate, interest;  // объявление  переменных
int years;
System.out.print("Pervonachalnoe vlozhenie: ");
principal = TextIO.getlnDouble();
System.out.print("Godovoj protsent: ");
rate = TextIO.getlnDouble();
years = 0;
while (years < 5) {
years = years + 1;
interest = principal * rate;
principal = principal + interest;
System.out.println(principal);
          }
```

Инструкция *while do..while*

Инструкция `do..while` выполняет инструкция до тех пор, пока условие принимает значение `true`:

```
do
  инструкция
while (условие);
```

Или в виде

```
do {
  инстуркция
} while (условие);
```

В конце мы пишем точку с запятой. Точка с запятой — это часть инструкции. Каждая инструкция в Java заканчивается либо точкой с запятой, либо правой фигурной скобкой "}".

Пример

```
boolean wantsToContinue;
  // True — если пользователь желает повторить игру.
do {
   Checkers.playGame();
   TextIO.put("Do you want to play again? ");
   wantsToContinue = TextIO.getlnBoolean();
} while (wantsToContinue == true);
```

Слово проверяется уже после того, как инструкция будет выполнена (хотя бы один раз). Иными словами, две ниже приведенные конструкции выполняют одно и то же действие:

```
do {
    doSomething
    } while (boolean-expression);
```

и

```
doSomething
while (boolean-expression) {
    doSomething
}
```

Конструкцию

```
while (boolean-expression) {
    doSomething
}
```

можно заменить следующей конструкцией:

```
if (boolean-expression) {
   do {
       doSomething
   } while (boolean-expression);
}
```

такие замены не изменят смысл программы.

Внутри циклов `while` и `do..while` можно использовать инструкцию `break;`. Эта инструкция приводит к прекращению выполнения цикла. При этом программа переходит к следующей после цикла инструкции, например:

```
while (true) {  // походит на бесконечный цикл!
TextIO.put("Enter a positive number: ");
N = TextIO.getlnlnt();
if (N > 0)   // ввод верен, выходим из цикла
  break;
  TextIO.putln("Your answer must be > 0.");
}
// продолжение программы
```

Можно использовать инструкцию `continue`. Эта инструкция прекращает выполнение текущего шага цикла и переходит к следующему шагу цикла.

Инструкция *for*

Многие циклы имеют следующий вид:

```
инициализация
while (уловие) {
инструкции
обновление
}
```

В качестве примера можно привести следующий код:

```
years = 0;
while (years < 5) {
      interest = principal * rate;
      principal += interest;
         System.out.println(principal);
         years++;
}
```

Этот код можно записать более компактно с использованием цикла `for`:

```
for (years = 0;  years < 5;  years++) {
   interest = principal * rate;
   principal += interest;
   System.out.println(principal);
}
```

Общий вид цикла `for` следующий:

```
for (инициализация переменной; условие; обновление переменной)
инструкция
```

Или, что то же самое, но с блоком:

```
for (инициализация переменной; условие; обновление переменной) {
инструкции
}
```

Вывод 10 чисел натурального ряда может быть осуществлен следующим образом:

```
for (N = 1; N <= 10; N++) {
System.out.println(N);
}
```

Вложенные циклы

Циклы можно вкладывать друг в друга. Для решения задачи вывода, например, такой таблицы, как показана ниже, можно использовать два цикла — при этом один цикл будет вложен в другой.

Задача: вывести таблицу

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108
10	20	30	40	50	60	70	80	90	100	110	120
11	22	33	44	55	66	77	88	99	110	121	132
12	24	36	48	60	72	84	96	108	120	132	144

Решение

```
for (rowNumber = 1;  rowNumber <= 12;  rowNumber++) {
     for (N = 1;  N <= 12;  N++) {
         // print in 4-character columns
    TextIO.put(N * rowNumber,  4);
}
  TextIO.putln();
}
```

Здесь же мы приведем еще один пример, в котором используется условие `if`.

Пример с if

```
public class LengthConverter {
   public static void main(String[] args) {

      double measurement;  // Численое значение, вводится пользователем.
         String units;     // Единицы измерения,
                           // также задаются пользователем.
      double inches, feet, yards, miles;  // Значения во всех допустимых
                                          // единицах измерений.
      TextIO.putln("Enter measurements in inches, feet,
                    yards, or miles.");
      TextIO.putln("For example:  1 inch    17 feet    2.73 miles");
      TextIO.putln("You can use abbreviations:   in   ft  yd   mi");
      TextIO.putln("I will convert your input into the other units");
      TextIO.putln("of measure.");
      TextIO.putln();

      while (true) {

       /* Информация от пользователя переводится в нижний регистр. */

   TextIO.put("Enter your measurement, or 0 to end:  ");
   measurement = TextIO.getDouble();
   if (measurement == 0)
      break;  // прекращение цикла while
   units = TextIO.getlnWord();
   units = units.toLowerCase();

   /* преобразование введенных значений в дюймы */

  if (units.equals("inch") || units.equals("inches")
                           || units.equals("in")) {
  inches = measurement;
  }
   else if (units.equals("foot") || units.equals("feet")
                                 || units.equals("ft")) {
```

```
      inches = measurement * 12;
 }
   else if (units.equals("yard") || units.equals("yards")
                                 || units.equals("yd")) {
      inches = measurement * 36;
   }
  else if (units.equals("mile") || units.equals("miles")
                                || units.equals("mi")) {
     inches = measurement * 12 * 5280;
   }
     else {
    TextIO.putln("Sorry, but I don't understand \""
                                    + units + "\".");
    continue;
 }

   /* Преобразование из дюймов в футы, ярды и мили. */

  feet = inches / 12;
  yards = inches / 36;
  miles = inches / (12*5280);

  /* Вывод значений. */

  TextIO.putln();
  TextIO.putln("That's equivalent to:");
  TextIO.put(inches, 15);
  TextIO.putln(" inches");
  TextIO.put(feet, 15);
  TextIO.putln(" feet");
  TextIO.put(yards, 15);
  TextIO.putln(" yards");
  TextIO.put(miles, 15);
  TextIO.putln(" miles");
  TextIO.putln();

}

TextIO.putln();
```

```
TextIO.putln("OK!  Bye for now.");

}

}
```

Эта программа запрашивает у пользователя ввод длины с указанием единиц, переводит введенную длину в другие единицы и выводит информацию пользователю.

Переключатель *switch*

Условие `if` направляет поток программы по одному из двух направлений. В том случае, когда следует сделать выбор, состоящий из нескольких возможностей, можно использовать переключатель `switch`.

Инструкция `switch` используется следующим образом:

```
switch (выражение) {
  case constant-1:
    statements-1
    break;
  case constant-2:
    statements-2
    break;
  . . .
  // (еще варианты)
  case constant-N:
    statements-N
    break;
  default:  // необязательные инструкции
    statements-(N+1)
}
```

Если выражение — целое число, то переключатель можно записать так:

```
switch (N) {   //  N  — целое
   case 1:
     System.out.println("The number is 1.");
     break;
   case 2:
   case 4:
   case 8:
```

```
      System.out.println("The number is 2, 4, or 8.");
      System.out.println("(That's a power of 2!)");
      break;
    case 3:
    case 6:
    case 9:
      System.out.println("The number is 3, 6, or 9.");
      System.out.println("(That's a multiple of 3!)");
      break;
    case 5:
      System.out.println("The number is 5.");
      break;
    default:
      System.out.println("The number is 7,");
      System.out.println("   or is outside the range 1 to 9.");
}
```

Типы инструкций в Java

В языке Java используются следующие типы инструкций:

- декларация (объявление) переменных
- инструкции присваивания
- вызовы функций (включая input/output)
- другие инструкции, например `x++;`
- пустая инструкция
- блок
- инструкция `while`
- инструкция `do..while`
- инструкция `if`
- инструкция `for`
- инструкция `switch`
- инструкция `break`
- инструкция `continue`
- инструкция `return` (только в теле функции)
- инструкция `try..catch`
- инструкция `throw`
- инструкция `synchronized`

Графика и апплеты

Java позволяет создавать самостоятельные программы. Самостоятельные программы должны иметь функцию `main()`, доступную извне. Апплеты — это не самостоятельные программы, апплет не имеет функции `main()`. Как правило, апплет помещается на Web-страницу, где занимает прямоугольную область. В задачу апплета входит прорисовка того, что будет отображено в этой области. Когда необходимо вывести область, занимаемую апплетом, из браузера вызывается функция `paint()`, которая находится в апплете.

Простейший апплет имеет следующую структуру:

```
import java.awt.*;
import java.applet.*;
  public class ImyaAppleta Applet {
    public void paint(Graphics g) {
выражения
  }
}
```

Имя апплета — идентификатор, одновременно являющийся именем класса. Выражения, применяемые в апплете, служат для отображения его содержимого. При создании программ мы имеем возможность пользоваться заранее созданными классами. Это упрощает процесс программирования. В частности, к числу встроенных классов относятся классы `Math` и `System`. Помимо непосредственно встроенных классов существуют стандартные классы, которые не являются автоматически доступными для любой программы. Классы эти сгруппированы в пакеты. В нашем примере используются два таких пакета, а именно, пакеты `java.awt` и `java.applet`. Директива `import java.awt.*;` делает доступными для использования в программе все классы пакета `java.awt`. Пакет `java.applet` содержит классы, предназначенные для работы с апплетами. Помимо пакета `java.applet` для создания апплетов может быть использован класс `javax.Japplet`, в котором используются возможности графического пакета `Swing`.

Приведем пример простого апплета.

Апплет

```
import java.awt.*;
import java.applet.Applet;
   public class StaticRects extends Applet {
      public void paint(Graphics g) {
```

```
         // Отображает набор вложенных прямоугольников на красном фоне.
         // Каждый прямоугольник отделен пространством в 15 пикселов
         // от содержащего его прямоугольника со всех сторон.
      int inset;    // отступ от краев апплета до прямоугольников
      int rectWidth, rectHeight;   // размер прямоугольника
   g.setColor(Color.red);
   g.fillRect(0,0,300,160); //заполнение всего апплета красным цветом
   g.setColor(Color.black); // черный прямоугольник
      inset = 0;
      rectWidth = 299;  // размер первого прямоугольника
                        // соответствует размеру апплета
      rectHeight = 159;
      while (rectWidth >= 0 && rectHeight >= 0) {
         g.drawRect(inset, inset, rectWidth, rectHeight);
         inset += 15;
         // прямоугольники отделены расстоянием 15 пикселов
         rectWidth -= 30;   // ширина уменьшает на 15 пикселов
                            // как справа, так и слева
         rectHeight -= 30;  // высота уменьшается на 15 пикселов
                            // как сверху, так и снизу
      }
          }
}
```

Контрольные вопросы

1. Что такое блок? Как используется блок?

   Ответ. Блок — это инструкция, включающая в себя другие инструкции, которые должны быть помещены между фигурными скобками `{` и `}`. Тело функции — это блок. Блоки используются тогда, когда из нескольких инструкций необходимо сделать одну инструкцию.

2. Каково основное отличие между циклами `while` и `do..while`?

   Ответ. Обе инструкции выполняют блок до тех пор, пока условие не примет значение `false`. Основное отличие состоит в том, что цикл `while` производит проверку условия перед выполнением блока, а цикл `do..while` проверяет условие после того, как блок будет выполнен.

3. Напишите цикл, который выводит числа, кратные 3, начиная с числа 3 и заканчивая числом 36, т. е. числа 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36.

Ответ

```
for (N = 3;  N <= 36;  N = N + 3) {
     System.out.println(N);
}
```

Или

```
for (N = 3;  N <= 36;  N++) {
    if (N % 3 == 0)
      System.out.println(N);
}
```

4. Создайте функцию `main()`, которая запрашивает у пользователя целое число, читает его и сообщает пользователю о том, какое число введено: четное или нечетное. Используйте следующую конструкцию

```
public static void main(String[] args) {
            // здесь запишите свои функции
}
```

Ответ

```
public static void main (String[] args) {
  int n;  // число, введенное пользователем
TextIO.put("Type an integer: ");  // просим ввести целое число
n = TextIO.getInt();   // читаем то, что было введено
  if (n % 2 == 0)      // сообщаем, четное число или нет
System.out.println("chetnoe chislo.");
   else
  System.out.println("Nechetnoe chislo.");
   }
```

5. Что создает на выходе следующая программа?

```
public static void main(String[] args) {
     int N;
     N = 1;
 while (N <= 32) {
     N = 2 * N;
      System.out.println(N);
        }
  }
```

Ответ

```
2
4
8
```

```
16
32
64
```

6. Что получается на выходе этой программы:

```
public static void main(String[] args) {
    int x,y;
    x = 5;
    y = 1;
while (x > 0) {
    x = x - 1;
    y = y * x;
System.out.println(y);
     }
 }
```

Ответ. Запишем ход вычислений в виде таблицы (табл. П4.1).

Таблица П4.1. *Вывод программы*

Значение x	Значение y	Вывод программы
5	1 (до цикла)	
4	4 (= 4×1)	4
3	12 (= 4×3)	12
2	24 (=12×2)	24
1	24 (=24×1)	24
0	0 (=24×0)	0

7. Что получается на выходе следующей программы? (Здесь `name.charAt(i)` — это символ в позиции `i` в строке с именем `name`.)

```
 String name;
 int i;
 boolean startWord;
 name = "Richard M. Nixon";
startWord = true;
for (i = 0; i < name.length(); i++) {
 if (startWord)
 System.out.println(name.charAt(i));
 if (name.charAt(i) == ' ')
 startWord = true;
```

```
    else
    startWord = false;
  }
```

Ответ. Три строки:

```
R
M
N
```

Статические функции и статические переменные

Каждая функция в языке Java должна быть определена внутри какого-либо класса. Функции, которые являются членами класса, называются *методами*. Все функции в Java являются методами. Иногда термин "функция" распространяется на статические функции, все нестатические функции при этом называются методами. В этой части мы будем иметь дело со статическими функциями.

Определение (или описание) функции в языке Java выглядит следующим образом:

```
модификаторы возвращаемое_значение имя_функции (список_параметров) {
    инструкции
}
```

Инструкции, размещенные между скобками `{` и `}`, составляют тело функции. Эти инструкции выполняются тогда, когда происходит обращение к функции (методу), в которой они содержатся. Функции могут содержать практически все типы инструкций. Начало описания функции содержит слова, которые задают некоторые характеристики функции, например, является ли она статической. Для этого используются модификаторы `static` и `public`.

Пример 1

```
public void changeChannel(int channelNum) {...}
```

Пример 2

```
public static void playGame() {
     // "public" и "static" — модификаторы
     // void — возвращаемый тип;
     //  "playGame" — имя функции
     // список параметров пуст
    // выражения (инструкции)
}
```

Пример 3

```
int getNextN(int N) {
  // модификаторов нет
  // int — возвращаемый тип
  // "getNextN" — имя функции
  // один параметр  "N"
  // тип параметра "int"
  // инструкции
}
```

Пример 4

```
static boolean lessThan(double x, double y) {
     // "static" — модификатор
     // "boolean" — возвращаемый тип
     // "lessThan" — имя функции
     // список параметров состоит из двух параметров "x" и "y"
     // тип и того и другого параметра "double" инструкции
}
```

Каждая самостоятельная выполняемая программа должна содержать в себе функцию `main()`. Эта функция, как правило, определяется так:

```
public static void main(String[] args) { .... }
```

Здесь аргументами функции (ее параметрами) служит массив строк `String[] args`. Фактически значения аргументов могут быть введены в момент запуска программы в командной строке.

Вызов функции из программы производится при помощи инструкции, которая состоит из имени подпрограммы (с указанием фактических параметров). Например, для функции, описанной выше, это может быть такая инструкция:

```
playGame();
```

Поскольку метод `playGame()` описан как `public`, он доступен и из других классов, а не только из того класса, в котором он описан. Однако при обращении к этому методу из других классов необходимо указать имя класса, в котором описан этот метод. Если, например, этот метод описан в классе `Poker`, то вызов метода из класса, отличающегося от класса `Poker`, производится при помощи инструкции

```
Poker.playGame();
```

Метод с именем `playGame()`, определенный в классе `Poker`, может быть не единственным методом с таким именем. Функции с такими же именами

могут быть расположены в других классах. При этом одноименные функции из разных классов могут не иметь ничего общего друг с другом, например `Roulette.playGame()` и `Blackjack.playGame()`.

В общем виде инструкция, содержащая вызов функции, выглядит следующим образом:

```
Имя-функции(параметры);
```

или

```
имя_класса.имя-Функции(параметры);
```

Пример описания функции

```
static void playGame() {
     int computersNumber; // случайное число, выбираемое компьютером
     int usersGuess;      // число по выбору пользователя
     int guessCount;      // число попыток
     computersNumber = (int)(100 * Math.random()) + 1;
 // присвоение случайного числа в промежутке от 1 до 100
     guessCount = 0;
     TextIO.putln();
     TextIO.put("kakova pervaya popytka? ");
     while (true) {
 usersGuess = TextIO.getInt();  // получили пользовательский ввод
      guessCount++;
      if (usersGuess == computersNumber) {
          TextIO.putln("You got it in " + guessCount
                 + " guesses!  My number was " + computersNumber);
          break;  // игра закончена, пользователь победил
    }
        if (guessCount == 6) {
          TextIO.putln("Vy ne ugadali za 6 popytok.");
    TextIO.putln("Vy projgrali. Bylo chislo " + computersNumber);
           break;  // Игра закончена, пользователь проиграл
      }
       // Игра продолжается
       if (usersGuess < computersNumber)
          TextIO.put("slishkom malo.  Esche popytka: ");
       else if (usersGuess > computersNumber)
           TextIO.put("slishkom veliko.  Esche popytka: ");
          }
```

```
         TextIO.putln();
}
```

Чтобы программа была полной, необходимо создать функцию `main()`. Это несложно.

Фукнция main() (программа целиком)

```
public class GuessingGame {
   public static void main(String[] args) {
        TextIO.putln("Budem igrat. Ya vyberu chislo mezhdu");
           TextIO.putln("1 и 100, a vy ugadajte ego.");
           boolean playAgain;
           do {
              playGame();  // вызов функции
              TextIO.put("Sygraem esche raz? ");
              playAgain = TextIO.getlnBoolean();
           } while (playAgain);
           TextIO.putln("Spasibo za igru. Do svidaniya.");
        }
      static void playGame() {
           int computersNumber; // случайное число, выбираемое компьютером
           int usersGuess;      // число по выбору пользователя
           int guessCount;      // число попыток
           computersNumber = (int)(100 * Math.random()) + 1;
              // присвоение случайного числа в промежутке от 1 до 100
           guessCount = 0;
           TextIO.putln();
           TextIO.put("kakova pervaya popytka? ");
           while (true) {
       usersGuess = TextIO.getInt();  // получили пользовательский ввод
             guessCount++;
    if (usersGuess == computersNumber) {
         TextIO.putln("You got it in " + guessCount
               + " guesses!  My number was " + computersNumber);
    break;  // игра закончена, пользователь победил
}
   if (guessCount == 6) {
       TextIO.putln("Vy ne ugadali za 6 popytok.");
```

```
            TextIO.putln("Vy projgrali. Bylo chislo " + computersNumber);
                  break;  // Игра окончена, пользователь проиграл
}
// Игра продолжается
               if (usersGuess < computersNumber)
                  TextIO.put("slishkom malo.  Esche popytka: ");
               else if (usersGuess > computersNumber)
                  TextIO.put("slishkom veliko.  Esche popytka: ");
            }
            TextIO.putln();
    }
}
```

В рассмотренном примере функция не возвращает никакого значения, так как она имеет возвращаемый тип — `void`. Также функция может принимать значение. Тип этого значения указывается похожим образом: так, как это делается с объявлением типа переменных.

Следующий пример во многом походит на предыдущий, все отличия выделены полужирным шрифтом.

Пример 2

```
public class GuessingGame2 {

  static int gamesWon;  // количество выигранных пользователем игр

  public static void main(String[] args) {
    gamesWon = 0;  // это необязательно
                   // 0 — начальное значение.

    TextIO.putln("Budem igrat. Ya vyberu chislo mezhdu");
    TextIO.putln("1 и 100, a vy ugadajte ego.");
    boolean playAgain;
    do {
      playGame();  // вызов функции
      TextIO.put("Sygraem esche raz? ");
      playAgain = TextIO.getlnBoolean();
    } while (playAgain);
        TextIO.putln();
```

```
      TextIO.putln("Vy vyigrali " + gamesWon + " igr.");

      TextIO.putln("Spasibo za igru. Do svidaniya.");
    }

  static void playGame() {
    int computersNumber; // случайное число, выбираемое компьютером
    int usersGuess;      // число по выбору пользователя
    int guessCount;      // число попыток
    computersNumber = (int)(100 * Math.random()) + 1;
    // присвоение случайного числа в промежутке от 1 до 100
    guessCount = 0;
    TextIO.putln();
    TextIO.put("kakova pervaya popytka? ");
    while (true) {
      usersGuess = TextIO.getInt();  // получили пользовательский ввод
      guessCount++;
      if (usersGuess == computersNumber) {
        TextIO.putln("Ugadali " + guessCount +
                     " !  moe chislo " + computersNumber);
        gamesWon++;  // Учитываем выигрыш, увеличиваем gamesWon.
        break;  // игра закончена, пользователь победил
      }
      if (guessCount == 6) {
        TextIO.putln("Vy ne ugadali za 6 popytok.");
        TextIO.putln("Vy projgrali. Bylo chislo " + computersNumber);
        break;  // игра закончена, пользователь проиграл
      }
      // Игра продолжается
      if (usersGuess < computersNumber)
        TextIO.put("slishkom malo.  Esche popytka: ");
      else if (usersGuess > computersNumber)
        TextIO.put("slishkom veliko.  Esche popytka: ");
      }
      TextIO.putln();
    }
  }
```

Функция может возвращать значение. Для этого используется инструкция `return`.

```
static double pythagorus(double x, double y) {
  // Вычисление длины гипотенузы
  return  Math.sqrt(x*x + y*y);
}
```

Пакеты и API

Язык Java содержит большое количество заранее разработанных классов, при помощи которых создание приложений становится простым и удобным делом. Все стандартные пакеты являются частью программного интерфейса API (Application Program Interface). Некоторые части упомянутого API нам уже знакомы. Например, это функции `Math.sqrt()`, `System.out.print()`. В состав стандартного API Java входят стандартные классы, которые используются для создания графического интерфейса пользователя, для работы с сетью, для записи и чтения файлов и т. п. Можно считать, что эти возможности являются встроенными свойствами языка Java.

Язык Java платформенно-независимый язык. Это значит, что одни и те же программы могут работать на различных системах, включая Macintosh, Windows, UNIX и другие. Мы имеем возможность использовать один и тот же интерфейс API для создания программ, которые будут работать на разных платформах.

Все функции в Java описываются внутри классов. Все функции API тоже входят в состав того или иного класса. Классы могут быть сгруппированы в пакеты. В свою очередь, эта структура может быть еще более усложнена. Одни пакеты могут входить в состав других пакетов. Стандартный Java API состоит из нескольких пакетов. Один из пакетов, носящий имя `Java`, включает в себя неграфические пакеты, а также основные графические средства работы с Java — AWT. Это не единственный стандартный пакет в Java. Например, пакет `javax` существует, начиная с версии Java 1.2. Этот пакет содержит графический интерфейс пользователя, носящий имя `Swing`. Пакет состоит из классов и других пакетов (некоторые файлы классов могут описывать интерфейсы, но здесь мы не будем останавливаться на этом). Пакеты, входящие в состав другого пакета, называются по имени родительского пакета и своему собственному имени, например:

```
java.awt
javax.Swing
```

Классы, входящие в пакеты, называются именем пакета, в состав которого они входят, с указанием имени класса, например:

```
java.awt.Button
java.awt.Graphics
```

```
javax.swing.JButton
javax.swing.JApplet
```

Пакет `java` содержит в себе несколько других подпакетов, среди них `java.io` (пакет для работы с вводом и выводом), `java.applet` (пакет для работы с апплетами), `java.lang` (базовый пакет). Последний пакет содержит наиболее фундаментальные классы, такие, как `String` и `Math` (рис. П4.5).

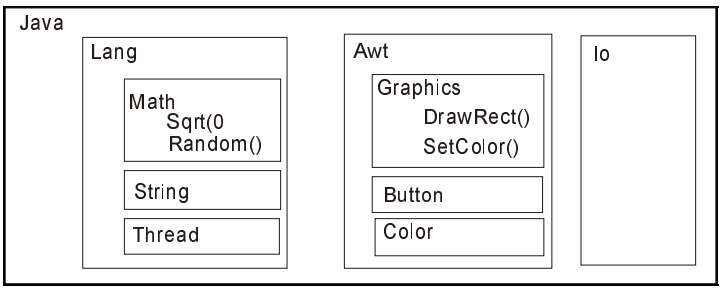


Рис. П4.5. Структура пакета `Java`

Если мы хотим использовать класс `java.awt.Color`, то можно указать полное имя класса при объявлении переменной:

```
java.awt.Color rectColor;
```

Здесь объявлена переменная `rectColor`, которая имеет тип `java.awt.Color`.

Существует способ избежать написания полных имен классов. Для этого в начале программы Java следует вставить инструкцию

```
import java.awt.Color;
```

Затем нашу переменную можно будет объявить в следующем виде:

```
Color rectColor;
```

Также можно импортировать все пакеты из `java.awt`. Для этого следует написать:

```
import java.awt.*;
```

Все пакеты из `javax.swing` импортируются при помощи

```
import javax.swing.*;
```

Так, если мы предварительно напишем

```
import java.lang.*;
```

то в дальнейшем к математическим функциям можно будет обращаться при помощи таких имен, как `Math.sqrt()`, вместо указания полного имени `java.lang.Math.sqrt()`.

Программист имеет возможность создавать свои собственные пакеты.

Контрольные вопросы

1. Что такое API?
2. Напишите функцию, которая называется `stars`, которая выводит в поток стандартного вывода строку, состоящую из звездочек:

```
**********
```

Ответ. Функция может быть написана следующим образом.

```
static void stars(int numberOfStars) {
       // выводим строку из указанного количества звездочек
       for (int i = 0; i < numberOfStars; i++) {
       System.out.print('*');
    }
System.out.println();
  // после звездочек выводим символ возврата каретки
}
```

3. Напишите функцию `main()`, в которой используется функция из вопроса 2 и которая служит для вывода десяти строк звездочек. Первая строка содержит 1 звездочку, последняя — 10 звездочек:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Ответ

```
public static void main(String[] args) {
     int line;  // номер строки.
     for (line = 1;  line <= 10;  line++) {
      stars(line);
  }
}
```

4. Напишите функцию с именем `countChars`, которая содержит переменную `String` и переменную `char` в качестве параметров. Функция должна сосчитать, сколько раз указанный символ встречается в строке.

Ответ

```
static int countChars(String str, char searchChar) {
 // сколько раз встречается searchChar
 // в строке str
int i;     // положение в строке str
char ch;   // символ в строке
int count; // сколько раз встретился searchChar
count = 0;
for (i = 0;  i < str.length();  i++) {
ch = str.charAt(i);  // получаем символ в позиции i в строке str.
if (ch == searchChar)
count++;
}
return count;
}
```

5. Напишите функцию, которая содержит три параметра типа `int`. Функция определяет наименьший параметр и возвращает это значение.

Ответ

```
static int smallest(int x, int y, int z) {
if (x <= y && x <= z) {
return x;
}
else if (y <= x && y <= z) {
return y;
}
else
return z;
}
Можно решить задачу, не используя else:
static int smallest(int x, int y, int z) {
if (x <= y && x <= z) {
return x;
}
if (y <= x && y <= z) {
return y;
}
return z;
}
```

Классы и объекты

Класс описывается с использованием слова `class`. Простейший пример описания класса:

```
class UserData {
    static String name;
    static int age;
}
```

Класс — это тип, наподобие таких типов, как `int` и `boolean`. Имя класса используется для объявления типов переменных. Например, можно описать переменную `std`, указав ее тип как тип `Student`:

```
Student std;
```

При этом мы имеем в виду, что класс `Student` описан в следующем виде:

```
class Student {
  String name;                  // имя студента
  double test1, test2, test3;   // оценки за три теста
  double getAverage() {         // средняя оценка
    return (test1 + test2 + test3)/3;
  }
}
```

Отметим важный факт. В языке Java переменные никогда не содержат объект. Переменная может хранить только ссылку на объект.

Можно представить объект свободно "плавающим" в памяти компьютера. В действительности в памяти выделен специальный "накопитель", в котором хранятся объекты. Переменные же содержат информацию, на основании которой объект может быть найден.

Сам объект может быть создан при помощи следующей инструкции:

```
std = new Student();
```

Эта инструкция создает объект класса `Student` и помещает ссылку на этот объект в переменную `std`. Значение переменной — это ссылка на объект, но не сам объект. К свойствам объекта можно обратиться, указав их имена следующим образом: `std.name`, `std.test1`, `std.test2`, `std.test3`.

Пример

```
System.out.println("Hello, "  +  std.name + ".  Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
System.out.println("Srednyaya otsenka " + std.getAverage());
```

Переменная, тип которой задается с указанием имени класса, может содержать нулевую ссылку, которая записывается как "`null`":

```
std = null;
```

Проверка того, является ли ссылка на объект нулевой, осуществляется при помощи такой конструкции

```
if (std == null) ...
```

Рассмотрим еще один пример.

Пример

```
Student std, std1,    // объявляем четыре переменных
       std2, std3;    // типа Student
std = new Student(); // создаем новый объект,
                     // принадлежащий классу Student
                     // сохраняем ссылку на объект в переменной std
std1 = new Student();     // создаем второй объект класса Student
                          // сохраняем ссылку на этот объект
                          // в переменной std1
std2 = std1;              // Копируем ссылку на объект std1
                          // в переменную std2
std3 = null;              // сохраняем нулевую ссылку на объект
                          //   в переменной std3
std.name = "John Smith";  // задаем значения некоторых переменных
std1.name = "Mary Jones";
```

После выполнения этих инструкций память компьютера можно изобразить с помощью рисунка (рис. П4.6).

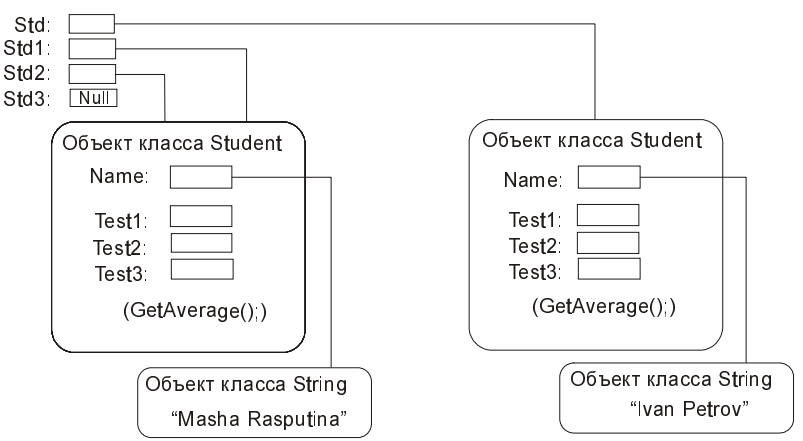


Рис. П4.6. Схема памяти компьютера

Важно помнить, что при копировании переменной объекта в другую переменную, копируется только ссылка на объект, но сам объект не копируется.

Тот факт, что переменные содержат ссылки на объекты, а не сами объекты, имеет ряд важных последствий, которые необходимо иметь в виду. Их легко понять, если иметь в виду, что переменная хранит лишь ссылку, но не сам объект.

Если переменная содержит ссылку на объект, который описан при помощи `final` (то есть, после инициализации переменные никогда не могут быть изменены), то это вовсе не значит, что данные, которые хранятся в объекте, не могут быть изменены. Неизменна только лишь ссылка на объект. Например, следующая конструкция будет вполне верной:

```
final Student stu = new Student();
stu.name = "John Doe";  // изменяем данные в объекте
                        // значение в  stu не изменяется
```

При этом интересно сравнить результат работы следующих двух фрагментов кода (табл. П4.2).

Таблица П4.2. *Сравниваем код*

Первый фрагмент	Второй фрагмент
`void dontChange(int z) {`	`void change(Student s) {`
`  z = 42;`	`  s.name = "Fred";`
`}`	`}`
`x = 17;`	`stu.name = "Jane";`
`dontChange(x);`	`change(stu);`
`System.out.println(x);`	`System.out.println(stu.name);`
На входе получаем 17	На выходе получаем "Fred"

Инициализация объектов. Конструкторы

Типы объектов сильно отличаются от примитивных типов. Простое объявление переменной, тип которой соответствует имени класса, еще не создает объекта этого класса. Объекты должны быть созданы в явном виде. Процесс создания объекта на компьютере состоит из двух частей. Сначала должна быть выделена память для размещения объекта. Затем компьютер должен заполнить переменные объекта, поместив в них значения. Выделение памяти не является задачей программиста. Зато инициализация переменных и присвоение им начальных значений должны быть решены программистом.

Переменная экземпляра класса может быть инициирована во время своего объявления. Например, рассмотрим класс `PairOfDice`. Объект этого класса содержит две переменные:

```
public class PairOfDice {
  public int die1 = 3;   // первая кость
  public int die2 = 4;   // вторая кость
  public void roll() {
    // выбрасываем кости и выбираем случайное чило от 1 до 6
    die1 = (int)(Math.random()*6) + 1;
    die2 = (int)(Math.random()*6) + 1;
  }
}
```

Здесь переменные `die1` и `die2` инициализированы значениями 3 и 4 соответственно. Эта инициализация производится в момент создания объекта `PairOfDice`. Может существовать много объектов типа `PairOfDice`. Каждый раз, когда создается объект такого типа, в нем инициализируются указанные переменные с заданными начальными значениями. Вместо наперед заданных фиксированных значений переменные объекта могут быть инициализированы случайными числами:

```
public class PairOfDice {
  public int die1 = (int)(Math.random()*6) + 1;
  public int die2 = (int)(Math.random()*6) + 1;
  public void roll() {
  die1 = (int)(Math.random()*6) + 1;
  die2 = (int)(Math.random()*6) + 1;
}
}
```

Объект создается при помощи оператора `new`. Если в программе создается объект `PairOfDice`, то мы пишем:

```
PairOfDice dice;          // Объявление переменной с типом PairOfDice.
dice = new PairOfDice();  // создание нового объекта
```

Инструкция `new PairOfDice()` выделяет память, в которой будет храниться объект, в переменной `dice` будет храниться ссылка на этот объект. Часть инструкции, используемой для создания объекта, выглядит как вызов функции `PairOfDice()`. Это обращение к специальному типу функций, который представляет собой конструктор. Может показаться странным то, что в классе мы нигде не создавали такой функции, так откуда она взялась? Дело в том, что каждый класс имеет конструктор. Если программист не создаст конструктор сам, то автоматически будет создан конструктор по умолчанию. Такой конструктор производит самые необходимые действия.

Он выделяет память и инициализирует переменные объекта. Если нам необходимо произвести еще какие-то действия в момент создания объекта, то необходимо вставить в описание класса по крайней мере один конструктор.

Тело конструктора представляет собой обычное тело функции, которое состоит из блока инструкций. На используемые инструкции не накладывается никаких ограничений. Одной из основных причин, по которой используются конструкторы, является работа с параметрами. Если конструктор использует параметры, то определение класса можно написать в следующем виде.

Пример класса с конструктором с заданными параметрами

```
public class PairOfDice {
  public int die1;   // первая кость
  public int die2;   // вторая кость
  public PairOfDice(int val1, int val2) {
    // создает две кости
    // устанавливая их значения равными val1 и val2
    die1 = val1;  //  присваивает значения
    die2 = val2;  //  переменным
  }
  public void roll() {
    // бросаем кости и выбираем случайные числа от 1 до 6
    die1 = (int)(Math.random()*6) + 1;
    die2 = (int)(Math.random()*6) + 1;
  }
}
```

После того как мы вставили такой конструктор в описание класса `PairOfDice`, мы больше не можем создать объект, написав инструкцию `new PairOfDice()`. Система использует конструктор по умолчанию только в том случае, если в классе не описан конструктор в явном виде. Наш конструктор имеет два параметра. Однако эта проблема легко решается. Мы имеем возможность создать еще один конструктор в этом же классе. Этот конструктор будет отличаться тем, что в нем будет присутствовать иное число параметров.

Пример с двумя конструкторами

```
public class PairOfDice {
  public int die1;   // первая кость
  public int die2;   // вторая кость
  public PairOfDice() {
```

```
    roll();
  }
  public PairOfDice(int val1, int val2) {
    // Конструктор.  Создает пару костей по заданным значениям.
    die1 = val1;  // присваивает конкретные значения переменным
    die2 = val2;
  }
  public void roll() {
  // Бросает кости
  // случайные числа от 1 до 6
  die1 = (int)(Math.random()*6) + 1;
  die2 = (int)(Math.random()*6) + 1;
  }
}
```

Сейчас мы имеем выбор: мы можем создавать новый объект, используя либо `new PairOfDice()`, либо `new PairOfDice(x,y)`, где `x` и `y` имеют тип `int`.

Приведем пример полной программы, которая содержит фукнцию `main()` и использует созданный нами класс. Класс может быть расположен в отдельном самостоятельном файле.

Программа полностью

```
public class RollTwoPairs {
  public static void main(String[] args) {
     PairOfDice firstDice;  // первая пара костей
     firstDice = new PairOfDice();
     PairOfDice secondDice; // вторая пара костей
     secondDice = new PairOfDice();
     int countRolls;  // сколько раз бросали
                      // первую и вторую пару костей
     int total1;      // для первой пары костей
     int total2;      // для второй пары костей
     countRolls = 0;
     do {  // Бросаем две пары костей, пока сумма не совпадет
         firstDice.roll();    // бросаем первую пару
         total1 = firstDice.die1 + firstDice.die2;   // сумма
```

```
        System.out.println("First pair comes up  " + total1);
        secondDice.roll();    // бросаем вторую пару
        total2 = secondDice.die1 + secondDice.die2;   // сумма
        System.out.println("Second pair comes up " + total2);
        countRolls++;   // считаем броски
        System.out.println();  // пустая строка
        } while (total1 != total2);
      System.out.println("Bylo " + countRolls +
                         " broskov do sovpadeniya cummy.");
    }
}
```

В отличие от обычной функции, конструктор может быть вызван только с использованием оператора `new` в такой форме:

```
new class-name (parameter-list)
```

Список параметров может быть пустым.

Вызов конструктора осуществляется несколько более сложным образом, чем вызов обычной функции. Вначале компьютер выделяет память, достаточную для размещения объекта заданного типа. Затем происходит инициализация переменных. Если в описании заданы начальные значения, то переменным присваиваются эти значения, если начальные значения не указаны, то переменные инициализируется значениями, используемыми по умолчанию. Если присутствуют параметры, то вычисляются значения фактических параметров конструктора. Если конструктор содержит инструкции, то выполняются эти инструкции. В качестве своего значения конструктор возвращает ссылку на объект.

В результате мы получаем ссылку на объект, при помощи которой можем обращаться к переменным и методам этого объекта.

Перепишем класс `Student`, используя в нем конструктор, а переменную `name` определим как `private`.

```
public class Student {
    private String name;                  // имя студента
    public double test1, test2, test3;   // оценки за три теста
    Student(String theName) {
        // конструктор объектов Student;
        // с заданием имени студента
    name = theName;
     }
public String getName() {
        // метод для чтения значения защищенного
```

```
        // свойства — переменной name
    return name;
   }
public double getAverage() {
        // сосчитать среднюю оценку
    return (test1 + test2 + test3) / 3;
    }
}
```

Такое описание класса позволяет создавать новые объекты студентов с использованием следующих инструкций:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

Поскольку имя `name` было объявлено защищенным, т. е. `private`, мы не сможем изменить это имя и даже прочитать его напрямую извне класса. Чтобы найти и прочитать имя, используется метод `getName()`, его можно вызывать и извне класса. Это позволяет быть уверенным в том, что после создания объекта `Student` в нем будет хранится одно и то же имя, которое не может быть изменено.

Сборщик мусора

Ранее мы познакомились с тем, как создаются объекты. А как можно уничтожить объект? В языке Java уничтожение объектов происходит автоматически. Объект существует в памяти, доступ к объекту может быть осуществлен с использованием переменных, в которых хранится ссылка на объект. Что делать с объектом, если не существует переменных, содержащих ссылку на этот объект? Рассмотрим две инструкции:

```
Student std = new Student("John Smith");
std = null;
```

В первой строке мы создаем новый объект и помещаем ссылку на этот объект в переменную `std`. В следующей строке мы изменяем значение переменной и теряем ссылку на только что созданный объект. Сейчас ссылки на этот объект нигде не существует. Программа не имеет возможности использовать этот объект. Память, в которой размещается объект, могла бы быть использована для других целей.

В Java используется сборщик мусора для того, чтобы очистить память от объектов, которые более не доступны для программы. Ответственность за то, чтобы определить, что является "мусором", т. е. какие объекты не доступны для программы, возлагается на систему. Это не является задачей программиста. В приведенном только что примере предельно ясно, какой объект становится мусором. В реальности дело обстоит значительно сложнее.

Если объект использовался, то программа может содержать несколько ссылок на этот объект. Объект не превратится в мусор до тех пор, пока не будут потеряны все эти ссылки.

Во многих языках программирования ответственность за удаление мусора возлагается на программиста. Эта задача не представляется очень легкой, поскольку отслеживание того, как используется память, — занятие, которое приводит к появлению множества ошибок. Можно по ошибке удалить тот объект, на который есть еще ссылки. Другая проблема — утечка памяти. Она возникает тогда, когда программист пренебрегает необходимостью убирать ненужный мусор, т. е. те объекты, которые уже не используются. Это может приводить к тому, что память будет заполняться объектами, которые недоступны из программы.

Процесс уборки мусора в Java избавляет от всех этих неудобств.

Работа с объектами

При программировании концепция объектной ориентированности может быть использована на различных уровнях. Наиболее распространенный подход состоит в том, что объектно-ориентированный метод используется на самых ранних этапах разработки программы, при этом применяется идея, состоящая в том, что все, что описывает программа, представляет собой объекты. Может быть использован и другой уровень применения объектно-ориентированной модели программирования. При этом ставится такая задача, решая которую, программист создает программное обеспечение, которое в дальнейшем может быть использовано во многих других проектах.

Встроенные классы

Объектно-ориентированное программирование предполагает, что будут конструироваться, разрабатываться и имплементироваться новые классы. При этом полезно использовать уже существующие классы. Разработчики Java предлагают богатый выбор классов, которые могут быть многократно использованы. Часть из этих классов используется как база для создания других классов на их основе. Другие классы используются для создания полезных объектов. Далее мы познакомимся с некоторыми из них.

Наследование. Полиморфизм. Абстрактные классы

Класс представляет набор объектов, которые имеют одинаковую структуру и поведение. Класс определяет структуру объекта, задавая переменные, которые содержатся в каждом экземпляре этого класса, и определяет поведение каждого экземпляра класса. Центральная идея объектно-ориентированного

программирования состоит в том, что классы могут использоваться для того, чтобы отражать сходства объектов, похожесть разных объектов друг на друга. Объекты могут иметь частично одинаковую структуру и поведение. Такие сходства могут быть выражены при помощи инструментов, описываемых понятиями наследования и полиморфизма. Каждый разработчик программ должен знать, что такое подкласс, наследование и полиморфизм.

Под наследованием понимается тот факт, что один класс может полностью или частично наследовать структуру и поведение другого класса. Тот класс, который наследует, называется подклассом по отношению к классу, от которого он наследует. Если класс `B` является подклассом класса `A`, то говорят, что `A` — это суперкласс для класса `B`. Можно также сказать, что класс `A` — это базовый класс или родительский класс. Подкласс может вносить изменения в наследуемые структуру и поведение.

При создании нового класса на Java мы имеем возможность использовать уже существующий класс (рис. П4.7). Если мы создаем класс `B`, который будет подклассом для класса `A`, то мы пишем

```
class B extends A {
  // добавления и исправления
  // для класса, унаследованного от класса A
}
```

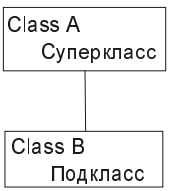


Рис. П4.7. Класс B создается на основе класса A

У одного суперкласса может быть несколько подклассов. Такие подклассы могут быть названы смежными. Наследование может распространяться на несколько "поколений" классов (рис. П4.8).

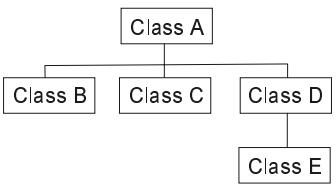


Рис. П4.8. Один класс может быть суперклассом для нескольких подклассов

При помощи классов можно описать предметы из реальной жизни (рис. П4.9).

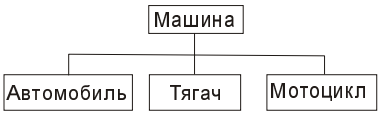


Рис. П4.9. Классы могут описывать реальные предметы

Пример

```
class Machina {
   int registrationNumber;
   Person owner;  // владелец
      // предполагается, что существует класс Person
   void transferOwnership(Person newOwner) {

  }
}
class Avtomobil extends Machina {
   int numberOfDoors;

}
class Tyagach extends Machina {
    int numberOfAxels;

}
class Motocikl extends Machina {
    boolean hasSidecar;

}
```

Существует правило, состоящее в том, что если переменная может хранить ссылку на объект класса `A`, то эта же переменная может хранить ссылку на любой объект, принадлежащий подклассу `A`. В следующем примере объект типа `Avtomobil` может быть присвоен переменной типа `Machina`.

```
Machina mojaMachina = mojAvtomobil;
```

или

```
Machina mojaMachina = new Machina();
```

Можно осуществить проверку того, принадлежит ли тот или иной объект определенному классу. Такая проверка может выглядеть следующим образом:

```
if (mojAvtomobil instanceof Machina) ...
```

Присваивание такого рода, как, например

```
mojAvtomobil = mojaMachina;
```

будет неправомерным, поскольку переменная `mojaMachina` потенциально может ссылаться на объект, тип которого отличен от типа объекта, ссылка на который содержится в переменной `mojAvtomobil`. Нельзя также присваивать значение типа `Machina` переменной, которая должна содержать значение типа `Avtomobil`, поскольку не всякая машина является автомобилем. Вот же решение существует. Можно применить приведение типов. Если по каким-либо причинам нам известно, что переменная `mojaMachina` в действительности содержит ссылку на объект класса `Avtomobil`, то мы можем произвести приведение типа, используя `(Avtomobil)` `MojaMachina`, тем самым мы сообщим компьютеру, что переменная `mojaMachina` содержит ссылку на объект типа `Avtomobil`. Мы можем написать так:

```
mojAvtomobil = (Avtomobil)moyaMachina;
```

Затем мы можем обращаться к свойствам автомобиля, или же мы могли бы сразу указать так: `((Avtomobil)mojaMachina).numberOfDoors`.

Все сказанное ранее суммируется в следующем примере:

```
System.out.println("Dannya Machiny:");
System.out.println("Registratsionnya Dannye: "
                                + myVehicle.registrationNumber);
    if (myVehicle instanceof Car) {
     System.out.println("Tip machiny:  Avtomobil");
     Car c;
     c = (Car)myVehicle;
     System.out.println("Kolichestvo dverej:  " + c.numberOfDoors);
}
    else if (myVehicle instanceof Tyagach) {
       System.out.println("Tip machiny:  Tyagach");
       Truck t;
       t = (Truck)myVehicle;
       System.out.println("Kolichestvo osej:  " + t.numberOfAxels);
}
    else if (myVehicle instanceof Motorcycle) {
       System.out.println("Tip machiny:  Motocikl");
       Motorcycle m;
       m = (Motorcycle)myVehicle;
       System.out.println("Imeet Kolyasku:    " + m.hasSidecar);
}
```

Создание классов на основе существующих классов

Мы говорили о подклассах, но делали это только теоретически. Сейчас мы коснемся практической стороны работы с подклассами. В повседневном программировании работа с подклассами главным образом сводится к тому, что разработчик создает новые классы на основе уже существующих классов. Разработка новых классов с нуля, создание классов на основе собственных классов — весь этот процесс создания больших пакетов с нуля используется не всегда. Существующий класс может быть использован для создания подкласса, при этом подкласс расширяет базовый суперкласс, на что указывает модификатор `extends`:

```
class имя-подкласса extends имя-существующего-класса {
   // изменения и дополнения
}
```

(Конечно, класс может быть объявлен как `public`.)

В качестве примера можно рассмотреть карточную игру Blackjack. При создании программы мы можем воспользоваться существующими классами `Card` (Карта), `Hand` (Сдача), `Deck` (Набор карт). Сдача при игре в Blackjack отличается от сдачи в других играх. Мы должны иметь возможность посчитать значение сдачи в игре Blackjack. Правило вычисления значения следующее. Значение сдачи вычисляется путем сложения значений всех карт, входящих в сдачу. Значение для валета, дамы и короля равно 10, значение туза равно либо 1, либо 11, в зависимости от того, превысит сумма (значение сдачи) величину 21 или нет. Если сумма становится больше 21, то значение туза 1, в противном случае туз дает 11 очков. Это означает, что второй, третий и четвертый туз в сдаче дает по одному очку.

Чтобы учесть эти особенности, можно создать класс на основе класса `Hand`, в который можно вставить метод, вычисляющий значение сдачи:

```
public class BlackjackHand extends Hand {
      public int getBlackjackValue() {
      // возвращает значение сдачи
      // для игры Blackjack
      int val;      // значение сдачи
      boolean ace;  // значение true, если в сдаче есть туз
      int cards;    // количество карт в сдаче
      val = 0;
      ace = false;
      cards = getCardCount();
      for (int i = 0;  i < cards;  i++) {
```

```
      // прибавляем к значению сдачи значение I-ой карты
    Card card;    // I-ая карта
      int cardVal;  // значение I-ой карты
      card = getCard(i);
      cardVal = card.getValue();  // обычное значение от 1 до 13
            if (cardVal > 10) {
            cardVal = 10;   // Для валета, дамы, короля.
    }
            if (cardVal == 1) {
            ace = true;     // Есть хотя бы один туз.
  }
      val = val + cardVal;
 }
            if (ace == true  &&  val + 10 <= 21)
            val = val + 10;
      return val;
}
}
```

Указатели *this* и *super*

В языке Java существуют специальные переменные `this` и `super`. Статический член класса обладает простым именем, которое доступно только внутри класса. Чтобы использовать это имя за пределами класса, его необходимо указать в составе полного имени в форме `имя-класса.простое-имя`, например `System.out`. Полное имя для статических имен можно использовать всегда, в том числе и внутри класса, которому они принадлежат. Иногда это просто необходимо, например, в том случае, если статическое имя члена класса скрыто за таким же локальным именем.

В объектах полное имя должно включать в себя часть, которая является именем объекта. Но важно помнить, что переменные и методы объектов принадлежат объектам, но не классам. Полное имя в объекте должно включать в себя имя объекта (ссылку на объект), в котором содержится простое имя переменной или метода. Это имя имеет вид `имя-переменной (т. е. имя ссылки на объект) .простое-имя`. Теперь предположим, что мы создаем описание метода в некотором классе. Как при этом мы можем получить ссылку на объект, который является экземпляром того класса, где расположен этот метод? Эта ссылка нужна тогда, когда необходимо использовать полное имя, например, для переменной экземпляра класса, когда имя скрыто за локальной переменной или параметром.

Java предлагает специальную, заранее определенную переменную `this`. Именно эта переменная помогает в данной ситуации. Эта переменная указывает на текущий объект. Если `x` — это переменная экземпляра класса, то полное имя этой переменной будет `this.x`. Если метод `otherMethod()` — метод экземпляра класса, то его полное имя `this.otherMethod()`.

Пример

```
public class Student {
  private String name;  // имя студента
  public Student(String name) {
    // Конструктор. Создаем учётную запись студента с указанным именем.
    this.name = name;
  }
  .
  .   //Прочие переменные и методы.
   .
}
```

В экземпляре объекта имя `name` скрыто за именем, которое используется в списке параметров. Чтобы использовать переменную экземпляра класса с таким же именем, как и имя параметра, мы пользуемся полным именем, а именно `this.name`. Это упрощает жизнь — нет необходимости придумывать новые имена для параметров. Еще один пример применения `this` — использование методов, которые в качестве параметра получают ссылку на объект. Для вывода строкового представления объекта мы можем написать `System.out.println(this);`.

В Java существует еще одна специальная переменная, имя которой `super`. Эта переменная также используется для задания полных имен, используемых в подклассах, `super` указывает на объект суперкласса. При этом необходимо всегда иметь в виду, что суперкласс не содержит в себе дополнений и изменений, которые могут присутствовать в подклассе.

Пример

```
public class GraphicalDice extends PairOfDice {
  public void roll() {
    // бросает и прорисовывает кости.
    super.roll();  // вызывает метод roll класса.
    redraw();      // рисует кости.
  }
// прочие методы, в том числе определение метода redraw()
}
```

Конструкторы в подклассах

Конструкторы не наследуются. Если мы создаем подкласс, то конструкторы суперкласса не становятся конструкторами подкласса. Это порой может создавать определенные неудобства. Кажется, что всю работу по созданию конструкторов необходимо проводить заново. Особенно неприятной становится задача в том случае, если мы не имеем исходного кода суперкласса, а к тому же не знаем, как он работает. Еще может возникнуть неудобная ситуация в том случае, если конструкторы суперкласса инициализируют статические переменные, к которым нет возможности обратиться из подкласса.

Проблема решается с использованием специальной переменной `super`. В качестве первой инструкции конструктора мы можем использовать `super`, эта инструкция вызовет конструктор суперкласса. Если конструктор суперкласса в качестве параметров требует задания двух целых чисел (как в примере ниже), то мы можем написать так:

```
public class GraphicalDice extends PairOfDice {
  public GraphicalDice() {  // конструктор этого класса
  super(3,4); // вызов конструктора класса PairOfDice с параметрами 3, 4
  initializeGraphics();  // инициализация для класса GraphicalDice
}
//прочие конструкторы, методы, переменные...
}
```

Интерфейсы. Вложенные классы

В этой части дан краткий обзор понятия интерфейсов и вложенных классов.

Интерфейсы

В некоторых языках программирования, например, в языке C++, допускается иметь два и более суперкласса. Это есть множественное наследование. На рис. П4.10 класс `E` имеет два класса (`A` и `B`), которые являются прямыми суперклассами для этого класса, а класс `F` имеет три суперкласса.

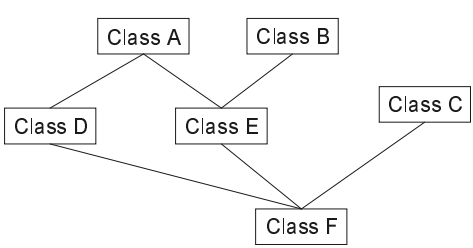


Рис. П4.10. Множественное наследование

Такое множественное наследование не допускается в языке Java. Однако в языке Java существует механизм, позволяющий решать задачи при помощи множественного наследования. Для этого используются интерфейсы. Один класс может имплементировать сразу несколько интерфейсов.

В Java интерфейс (`interface`) — это зарезервированное слово, оно несет определенный смысл. Интерфейс, определенный с помощью специального слова `interface`, состоит из набора функций, с которыми не связывается никакая конкретная имплементация. Класс может имплементировать интерфейс, для этого он должен имплементировать каждую функцию, описанную в интерфейсе. Приведем пример простого интерфейса Java:

```
public interface Drawable {
   public void draw(Graphics g);
}
```

Все выглядит похожим на описание класса. Однако имплементация метода `draw()` отсутствует. Пример имплементации этого интерфейса приведен далее:

```
class Line implements Drawable {
    public void draw(Graphics g) {
    // функции (рисование линии)
  }
// прочие методы и переменные
}
```

Каждый класс, имплементирующий интерфейс `Drawable`, описывает метод `draw()`. Каждый объект, созданный на основе этого класса, содержит метод `draw()`. Говорят, что объект имплементирует интерфейс, если этот объект принадлежит классу, который ипмлементирует интерфейс. Помимо того, что класс должен имплементировать все методы интерфейса, необходимо явно указать, что класс имплементирует данный интерфейс (например, написать `implements Drawable`). Класс может иметь только один суперкласс, но он может имплементировать несколько интерфейсов.

Пример

```
class FilledCircle extends Circle implements Drawable, Fillable {
}
```

Интерфейс — это что-то вроде абстрактного класса, который никогда не может быть использован для создания объектов, но может быть использован только для создания подклассов. Функции интерфейса — это абстрактные методы, которые должны быть имплементированы в каждом конкретном классе, имплементирующем интерфейс. Как и в случае абстрактных классов,

когда мы не можем создать объект на его основе, мы все же имеем возможность описать переменную, задав ее тип при помощи имени интерфейса или абстрактного класса.

Пример (Drawable — интерфейс)

```
Drawable figure;  // Объявляет переменную типа Drawable.
                  // Она может содержать ссылку на любой объект,
                  // имплементирующий интерфейс Drawable.
figure = new Line();  // figure ссылается на объект класса Line
figure.draw(g);       // вызывает draw() класса Line
figure = new FilledCircle();   // сейчас figure ссылается на объект
                               //   класса FilledCircle
figure.draw(g);   // вызывает метод draw() из класса FilledCircle
```

Тип используется при объявлении переменных. Возвращаемый функцией тип может быть задан внутри функции. В языке Java тип может быть классом, интерфейсом или одним из восьми встроенных примитивных типов.

Вложенные классы

Вложенный класс — это класс, описание которого находится внутри описания другого класса. Внутренние вложенные классы могут иметь собственное имя или быть анонимными. Именованный внутренний класс выглядит точно так же, как и любой другой обычный класс, единственное его отличие состоит в том, что он определен внутри другого класса. Как и прочие элементы класса, внутренний класс может быть как статическим, так и нет. Статический вложенный класс является частью статической структуры основного класса. Этот класс может использоваться внутри основного класса для создания объектов этого класса. Если внутренний класс не был объявлен как `private`, то этот класс также может использоваться и извне. При обращении к классу извне имя этого класса должно содержать часть, указывающую на класс, частью которого он является. Если, предположим, класс `WireFrameModel` представляет набор линий в трехмерном пространстве и содержит статический вложенный класс `Line`, тогда доступ к классу `Line` может быть осуществлен при помощи `WireFrameModel.Line`. В этом нет ничего особенного или нового. Описание класса `WireFrameModel` может иметь следующий вид:

```
public class WireFrameModel {
  // прочие члены класса WireFrameModel
  static public class Line {
```

```
  // представляет линию от точки (x1,y1,z1)
  // к точке (x2,y2,z2) в трехмерном пространстве
  double x1, y1, z1;
  double x2, y2, z2;
  }
  ... // прочие члены класса WireFrameModel
}
```

Внутри класса `WireframeModel` объект типа `Line` может быть создан с применением конструктора `new Line()`. Вне класса используется `new WireFrameModel.Line()`. Статический вложенный класс имеет полный доступ ко всем членам содержащего его класса, в том числе к членам, объявленным как `privat`. Этот факт, в частности, может служить причиной для использования вложенных классов. С помощью вложеных классов можно организовать доступ к защищенным членам класса, т. е. к тем членам, которые непосредственно не доступны для внешних классов.

Если мы используем написанный выше код и скомпилируем его, то получим два класса. Это произойдет несмотря на то, что класс `Line` определен внутри другого класса. Класс `Line` будет сохранен в виде отдельного файла WireFrameModel$Line.class.

Нестатические вложенные классы не сильно отличаются от статических вложенных классов. Нестатический класс имеет большее отношение к объекту, чем к содержащему его классу. Всякий нестатический член класса не является как таковым членом этого класса, даже несмотря на то, что код включен в описание класса. Это справедливо и в отношении вложенных нестатических классов. Нестатические члены описывают то, из чего будет состоять объект, созданный на основе класса. Каждый объект основного класса содержит в себе собственную копию вложенного класса. Этот класс имеет доступ ко всем переменным и методам экземпляра класса. Две копии вложенного класса, относящиеся к разным объектам, отличаются друг от друга. Принятие решения о том, будет вложенный класс статическим или нестатическим, — это не очень сложная задача. Если класс будет использовать переменные и методы объекта (экземпляра класса), то следует создавать нестатический класс. В противном случае создается статический класс.

Извне класса, содержащего другой вложенный класс, вложенный класс доступен по имени `имяПеременной.ИмяВложенногоКласса`, здесь `имяПеременной` — это переменная, которая содержит ссылку на основной класс, в который вложен второй, вложенный класс. Однако такая ситуация возникает довольно редко, так как нестатический вложенный класс как правило используется внутри основного класса, и здесь обращение к вложенному классу происходит с указанием простого имени вложенного класса.

Чтобы создать объект, относящийся к классу, который вложен в другой класс, необходимо сначала создать объект, относящийся к основному классу,

в который вложен второй класс. Объект вложенного класса навсегда связан с объектом основного класса, он имеет полный доступ к членам содержащего его класса. Рассмотрим класс `PokerGame`, сордержащий вложенные классы:

```
class PokerGame {  // Игра в покер.
  class Player {  // Представляет одного игрока.
. . .
}
private Deck deck;       // Карты.
private int pot;         // Количество денег.
 . . .
}
```

Каждый объект игры содержит свой набор карт и пот в качестве переменных экземпляра этого класса. В классе можно объявить нового игрока с помощью `new Player()`. Каждый экземпляр класса будет иметь свой собственный набор игроков. Именно в этом и состоит эффект объявления нестатических вложенных классов. Игроки принадлежат конкретной игре. Если бы класс игрока был статическим, то он бы выражал общую идею, общее представление об игроке в покер, который бы не был связан с конкретной игрой.

В некоторых ситуациях можно было бы создать вложенный класс, который затем используется лишь однажды и только в одной единственной строке программы. Возникает вопрос, а стоит ли создавать такой класс? Конечно, его можно было бы создать, но для таких ситуаций мы имеем возможность воспользоваться анонимными классами. Анонимный класс создается с помощью разновидности оператора `new` по следующей схеме:

```
new  superclass-or-interface () {
  methods-and-variables
}
```

Конструктор определяет новый класс без указания имени, одновременно создавая объект этого класса. Такая конструкция может быть использована в составе любой инструкции, где можно использовать оператор `new`. Эта инструкция сообщает, что необходимо создать новый объект класса (или интерфейса) `superclass-or-interface` с дополнительными методами и переменными `methods-and-variables`. При помощи такого способа можно создавать новый объект в любом месте программы.

Анонимные методы часто используются для обработки событий при создании графического интерфейса пользователя. Рассмотрим интерфейс `Drawable` (ранее мы о нем уже говорили). Предположим, что нам нужен красный квадрат со стороной 100 пикселов. Вместо того чтобы создавать отдельный самостоятельный класс, который в дальнейшем будет использован

для создания объекта квадарта, мы используем анонимный класс и создаем объект одной строкой:

```
Drawable redSquare = new Drawable() {
  void draw(Graphics g) {
  g.setColor(Color.red);
  g.fillRect(10,10,100,100);
  }
};
```

Точка с запятой в конце не является частью определения класса, она является частью инструкции, с помощью которой мы объявляем переменную. При компиляции анонимные классы сохраняются в виде самостоятельных файлов. Если имя основного класса `MainClass`, то анонимные классы будут сохранены в файлах с именами `MainClass$1.class`, `MainClass$2.class`, `MainClass$3.class` и т. д.

Модификаторы доступа

Класс может быть объявлен общедоступным при помощи модификатора доступа `public`. Такой класс можно использовать из любого места. Некоторые классы просто обязаны иметь открытый доступ. Класс, определяющий самостоятельное приложение, просто обязан быть описан как `public`, чтобы система могла обратиться к функции `main()`. Класс, который определяет апплет, также должен быть общедоступным, чтобы браузер смог к нему обратиться. Если класс не будет объявлен как `public`, то к нему можно обратиться только из других классов того же самого пакета, в который входит этот класс. Если для класса не указано, к какому пакету он должен относиться, то такой класс помещается в пакет по умолчанию. Все примеры, рассмотренные ранее, образуют классы, относящиеся к пакету по умолчанию. Следовательно, все созданные классы будут доступны друг для друга вне зависимости от того, были они объявлены как `public` или нет.

При создании пакетов все будет несколько иначе. Пакет содержит набор классов, которые имеют друг к другу какое-либо отношение. Некоторые из этих классов могут быть общедоступны (`public`). Другие классы используются только внутри пакета и не должны быть объявлены как `public`. Любая переменная или функция, которая является членом класса, может быть также объявлена общедоступной (`public`), что означает, что она будет доступна из любого места. Ее можно объявить закрытой (`private`), тогда она будет доступна только из кода содержащего ее класса. Если у переменной или функции не указан явно модификатор доступа, то переменная будет доступна из любого класса, входящего в тот же самый пакет, что и класс, в котором расположена переменная или функция. Существует еще один модификатор доступа — `protected`. При помощи этого модификатора описывается функция или переменная, которая будет доступна только из таких классов,

которые являются подклассами для того класса, где описана эта переменная или функция. Этот модификатор накладывает больше ограничений, чем модификатор `public`, но эти ограничения менее строги, чем те, что будут наложены в случае использования модификатора `private`. Те классы, которые разрабатываются специально для того, чтобы на их основе создавались подклассы, часто имеют в качестве своих членов имена, описанные при помощи модификатора `protected`.

Объединение статических и нестатических членов в одном классе

Классы имеют два основных назначения. С одной стороны, класс используется для объединения статических переменных и статических функций в одном блоке. С другой стороны, класс используется для создания объектов — экземпляров класса. Нестатические методы и переменные класса определяют свойства и методы объектов, созданных на основе этого класса. Как правило, класс используется для решения одной из этих двух задач, но не обеих задач сразу. Тем не менее, иногда возникают ситуации, когда в классе присутствуют как статические, так и нестатические члены. В такой ситуации класс выполняет двоякую роль. Возможно также организовать взаимодействие между нестатическими и статическими методами и переменными класса. Это происходит в том случае, если методы экземпляра класса используют статические методы и статические переменные класса. Методы экземпляра класса принадлежат объекту, а не самому классу, и может существовать множество объектов с различными методами и переменными. Но существует лишь единственный экземпляр для каждого статического метода и каждой статической переменной.

Предположим, что мы хотим создать класс, описывающий игральные кости, класс `PairOfDice`, и этот класс использует другой класс `Random`, который бросает кости, к которому обращается объект `PairOfDice`. Может существовать несколько объектов костей. Для каждого такого объекта вовсе не обязательно иметь свой собственный объект `Random`. Более того, было бы даже неправильным, если бы у каждой пары костей был свой объект `Random`. Наиболее удобным будет такое решение, когда один и тот же статический класс `Random` будет использоваться всеми объектами типа `PairOfDice`.

Пример

```
class PairOfDice {
   private static Random randGen = new Random();
   // предполагается, что импортирован java.util.Random
  public int die1;   // первая кость
```

```
  public int die2;   // вторая кость
  public PairOfDice() {
  // Конструктор. Создает две кости
   // со случайными значениями.
        roll();
}
   public void roll() {
   // бросаем кости
   // случайные значения от 1 до 6
   die1 = randGen.nextInt(6) + 1;
   die2 = randGen.nextInt(6) + 1;
  }
}
```

В качестве второго примера перепишем класс `Student`. Для каждого студента укажем идентификатор `ID` и создадим статическую переменную `nextUniqueID`. Для каждого экземпляра класса `Stydent` создается собственный идентификатор `ID`, но в то же время существует лишь одна общая переменная `nextUniqueID`.

Класс `Student`

```
public class Student {
  private String name;  // имя студента
  private int ID;  // идентификатор студента
  public double test1, test2, test3;   // оценки за три теста
     private static int nextUniqueID = 0;
       // запоминаем следующий идентификатор
       Student(String theName) {
         // конструктор объекта студента;
         // задается имя студента,
         // присваивается уникальный
         // идентификатор студента
         name = theName;
         nextUniqueID++;
         ID = nextUniqueID;
       }
     public String getName() {
       // Метод доступа для чтения значения private
```

```
        // переменной экземпляра класса.
        return name;
      }
      public int getID() {
        // Метод для чтения значения переменной ID.
        return ID;
      }
      public double getAverage() {
        // вычисление средней оценки
        return (test1 + test2 + test3) / 3;
       }
}
```

Контрольные вопросы

1. В объектно-ориентированном программировании используются классы и объекты. Что такое классы и что такое объекты?

   Ответ. В программно-ориентированном подходе класс представляет собой шаблон, по которому создается объект (мы имеем в виду нестатические элементы класса). Объект — это набор данных и способов поведения, который представляет ту или иную сущность (абстрактную или конкретную). Класс определяет строение и поведение всех объектов определенного типа.

2. Что обозначает специальное значение `null` и зачем оно нужно?

   Ответ. Если переменная имеет тип объекта, описываемого тем или иным классом, то в ней хранится не сам объект, а ссылка на этот объект. `Null` используется в том случае, если переменная типа объекта пока не содержит никакой ссылки на объект или ссылка на объект была уничтожена.

3. Что такое конструктор? Каково назначение конструктора в классе?

   Ответ. Конструктор — это специальная функция в классе. Имя конструктора должно совпадать с именем класса. Конструктор не возвращает никакого значения, в том числе значения `void`. Конструктор вызывается с применением оператора `new`, который создает новый объект. Назначение конструктора — инициализация новых объектов.

4. В этом задании создайте простой класс, который будет содержать счетчик (проходит ряд 0, 1, 2, 3, 4, ...). Класс назовем `Counter`. В классе будет содержаться одна переменная `private` для счетчика. Класс будет иметь два метода: метод `increment()` (увеличивает значение счетчика на 1) и метод `getValue()` (возвращает текущее значение счетчика). Создайте описание класса `Counter`.

Ответ. Один из вариантов решения поставленной задачи может выглядеть следующим образом.

```
public class Counter
{
    // Объект этого класса представляет собой счетчик
    // который считает, начиная с 0.
  private int value = 0;  // текущее значение счетчика
  public void increment()
    {
    // увеличение значения счетчика
    value++;
     }
   public int getValue()
    {
     // возвращает текущее значение счетчика
     return value;
     }
}
```

5. В этом задании используется класс из предыдущего вопроса. Фрагмент программы производит бросание монетки 100 раз. Используйте два объекта типа `Counter`, объекты `headCount` и `tailCount`. Эти объекты производят подсчет числа выпадений решки и орла, соответственно. Заполните пропущенные места в программе.

```
Counter headCount, tailCount;
  tailCount = new Counter();
  headCount = new Counter();
    for (int flip = 0;  flip < 100;  flip++) {
      if (Math.random() < 0.5)    // шанс выпадения 50 на 50
          _____________________ ;   // считаем решки

          else

          _____________________ ;   // считаем орлы
  }
System.out.println("Vypalo " + _____________________ + " reshek.");
System.out.println("Vypalo " + _____________________ + " orlov.");
```

Ответ. Переменная `headCount` — это переменная типа `Counter`. Все, что нам остается сделать, это обратиться к методам `headCount.increment()` и `headCount.getValue()`. Вызов метода `headCount.increment()` увеличивает

значение счетчика на единицу. Метод `headCount.getValue()` служит для чтения текущего значения счетчика:

```
Counter headCount, tailCount;
  tailCount = new Counter();
  headCount = new Counter();
   for (int flip = 0;  flip < 100;  flip++) {
     if (Math.random() < 0.5)
                 headCount.increment() ;
                 tailCount.increment() ;
          }
System.out.println(("Vypalo " + headCount.getValue() + " reshek.");
System.out.println(("Vyaplo " + tailCount.getValue() + " orlov.");
```

Ниже приводится код, содержащийся в файле TextIO.java. Класс `TextIO` используется в примерах настоящего приложения и в главах книги.

Файл TextIO.java

```
import java.io.*;
public class TextIO {

// *************************** Ввод/вывод ******************************

// Методы для работы с примитивными типами и типом String.
 // Вывод на консоль без лишних пробелов.
 // Вещественные значения выводятся округленными,
 // при этом максимальное количество знаков будет 10 или 11.
 // Чтобы вывести число с сохранением максимальной точности,
 // следует использовать метод TextIO.put(String.valueOf(x)).

public static void put(int x)     { put(x,0); }
                 // Прим.: поддерживаются типы byte и short!
public static void put(long x)    { put(x,0); }
public static void put(double x)  { put(x,0); }
                           // Поддерживается тип float.
public static void put(char x)    { put(x,0); }
public static void put(boolean x) { put(x,0); }
public static void put(String x)  { put(x,0); }
```

```
  // Методы для записи примитивных типов и типа String,
  // для вывода их на консоль с последующим символом перевода каретки
  // без лишних символов пробелов.

public static void putln(int x)      { put(x,0); newLine(); }
                       // Поддерживаются типы byte и short.
public static void putln(long x)     { put(x,0); newLine(); }
public static void putln(double x)   { put(x,0); newLine(); }
                        // Поддерживается тип float.
public static void putln(char x)     { put(x,0); newLine(); }
public static void putln(boolean x)  { put(x,0); newLine(); }
public static void putln(String x)   { put(x,0); newLine(); }

   // Методы для вывода примитивных типов и типа String на консоль
   // с минимальной шириной, заданной в качестве значения w,
   // символом перевода строки. Если ширина строки вывода
   // оказывается меньше w символов,
   // то она спереди заполняется пробелами.

public static void putln(int x, int w)     { put(x,w); newLine(); }
                       // Поддерживаются типы byte и short.
public static void putln(long x, int w)    { put(x,w); newLine(); }
public static void putln(double x, int w)  { put(x,w); newLine(); }
                        // Поддерживается тип float.
public static void putln(char x, int w)    { put(x,w); newLine(); }
public static void putln(boolean x, int w) { put(x,w); newLine(); }
public static void putln(String x, int w)  { put(x,w); newLine(); }

   // Методы для вывода символа возврата каретки.

   public static void putln() { newLine(); }

  // Методы вывода примитивных типов и типа String на консоль
  // с минимальной шириной, заданной в качестве значения переменной w.

public static void put(int x, int w)
{
  dumpString(String.valueOf(x), w);
}
```

```
                        // Поддерживаются типы byte и short.
public static void put(long x, int w)
  {
  dumpString(String.valueOf(x), w);
  }
public static void put(double x, int w)
  {
  dumpString(realToString(x), w);
  }
                        // Поддерживается тип float.
public static void put(char x, int w)
  {
  dumpString(String.valueOf(x), w);
  }
public static void put(boolean x, int w)
  {
  dumpString(String.valueOf(x), w);
  }
public static void put(String x, int w)
  {
  dumpString(x, w);
  }
// Методы чтения примитивных типов, а также типов "words" и "lines".
// Методы "getln..." игнорируют лишние введенные символы вплоть
// до символа возврата каретки (включая его).
// Слово "word" читается с помощью метода getlnWord()
// и представляет собой последовательность символов (не пробелов).
// Строка "line" читается с помощью методов getlnString() и getln()
// и состоит из всех символов до символа CR;
// Символ CR (возврат каретки) не возвращается этими методами,
// он читается и затем игнорируется.
// Все методы игнорируют символы пробелов в конце строки
// и символ перевода каретки и ищут следующий символ, отличный от
// пробела. Исключение составляют методы getAnyChar(), peek().
// Метод getln() может возвратить пустую строку,
// методы getChar() и getlnChar() могут возвратить
// пробел и перевод строки ('\n').
// Метод peek() позволяет найти следующий символ в потоке ввода,
// не удаляя его из потока.
```

```
// В качестве допустимых логических значений могут быть использованы
// следующие значения типа "words": true, false, t, f, yes,
// no, y, n, 0 или 1; допускаются прописные буквы.
// Методы не допускают появление ошибок.
// Если обнаруживается ошибка,
// то методы предлагают пользователю повторить ввод.
//=============  Список методов  ==========================
//            getByte()      getlnByte()    getShort()     getlnShort()
//            getInt()       getlnInt()     getLong()      getlnLong()
//            getFloat()     getlnFloat()   getDouble()    getlnDouble()
//            getChar()      getlnChar()    peek()         getAnyChar()
//            getWord()      getlnWord()    getln()        getString()
//            getlnString()
//
// (Метод getlnString эквивалентен методу getln consistency.)

public static byte getlnByte()
{ byte x=getByte();       emptyBuffer();  return x; }
public static short getlnShort()
{ short x=getShort();     emptyBuffer();  return x; }
public static int getlnInt()
{ int x=getInt();         emptyBuffer();  return x; }
public static long getlnLong()
{ long x=getLong();       emptyBuffer();  return x; }
public static float getlnFloat()
{ float x=getFloat();     emptyBuffer();  return x; }
public static double getlnDouble()
{ double x=getDouble();   emptyBuffer();  return x; }
public static char getlnChar()
{ char x=getChar();       emptyBuffer();  return x; }
public static boolean getlnBoolean()
{ boolean x=getBoolean(); emptyBuffer();  return x; }
public static String getlnWord()
{ String x=getWord();     emptyBuffer();  return x; }
public static String getlnString()
{ return getln(); }  // same as getln()
public static String getln() {
      StringBuffer s = new StringBuffer(100);
```

```
      char ch = readChar();
      while (ch != '\n') {
         s.append(ch);
         ch = readChar();
      }
      return s.toString();
   }



public static byte getByte()
{ return (byte)readInteger(-128L,127L); }
public static short getShort()
{ return (short)readInteger(-32768L,32767L); }
public static int getInt()
{ return (int)readInteger((long)Integer.MIN_VALUE,
(long)Integer.MAX_VALUE);
}
public static long getLong()
{ return readInteger(Long.MIN_VALUE, Long.MAX_VALUE); }
public static char getAnyChar()
{ return readChar(); }
public static char peek()
{ return lookChar(); }
public static char getChar()
  {
 // пропускаем пробелы и символы, затем возвращаем следующий символ
char ch = lookChar();
  while (ch == ' ' || ch == '\n')
  {
    readChar();
    if (ch == '\n')
      dumpString("? ",0);
      ch = lookChar();
    }
    return readChar();
   }

   public static float getFloat()
     {
```

```
      float x = 0.0F;
      while (true) {
        String str = readRealString();
        if (str.equals("")) {
            errorMessage("Illegal floating point input.",
                         "Real number in the range " + Float.MIN_VALUE +
                         " to " + Float.MAX_VALUE);
        }
        else {
           Float f = null;
           try { f = Float.valueOf(str); }
           catch (NumberFormatException e) {
              errorMessage("Illegal floating point input.",
                           "Real number in the range " +
                            Float.MIN_VALUE + " to " + Float.MAX_VALUE);
              continue;
           }
           if (f.isInfinite()) {
           errorMessage("Floating point input outside of legal range.",
                        "Real number in the range " + Float.MIN_VALUE +
                        " to " + Float.MAX_VALUE);
              continue;
           }
           x = f.floatValue();
           break;
     }
 }
      return x;
}

public static double getDouble() {
   double x = 0.0;
   while (true) {
      String str = readRealString();
      if (str.equals("")) {
          errorMessage("Illegal floating point input",
                       "Real number in the range " + Double.MIN_VALUE
                     + " to " + Double.MAX_VALUE);
      }
```

```
      else {
         Double f = null;
         try { f = Double.valueOf(str); }
         catch (NumberFormatException e) {
            errorMessage("Illegal floating point input",
                         "Real number in the range " +
                           Double.MIN_VALUE + " to " +
                           Double.MAX_VALUE);
            continue;
         }
         if (f.isInfinite()) {
         errorMessage("Floating point input outside of legal range.",
                      "Real number in the range " + Double.MIN_VALUE +
                      " to " + Double.MAX_VALUE);
            continue;
         }
         x = f.doubleValue();
         break;
      }
   }
   return x;
}

public static String getWord() {
   char ch = lookChar();
   while (ch == ' ' || ch == '\n') {
      readChar();
      if (ch == '\n')
         dumpString("? ",0);
      ch = lookChar();
   }
   StringBuffer str = new StringBuffer(50);
   while (ch != ' ' && ch != '\n') {
      str.append(readChar());
      ch = lookChar();
   }
   return str.toString();
}
```

```
public static boolean getBoolean() {
   boolean ans = false;
   while (true) {
      String s = getWord();
      if (s.equalsIgnoreCase("true") || s.equalsIgnoreCase("t") ||
              s.equalsIgnoreCase("yes")  || s.equalsIgnoreCase("y") ||
              s.equals("1")) {
          ans = true;
          break;
       }
       else if (s.equalsIgnoreCase("false") ||
                 s.equalsIgnoreCase("f") ||
                 s.equalsIgnoreCase("no")  ||
                 s.equalsIgnoreCase("n") ||
                 s.equals("0")) {
          ans = false;
          break;
       }
       else
          errorMessage("Illegal boolean input value.",
                       "one of:  true, false, t, f, yes, no, y, n, 0,
                                 or 1");
   }
   return ans;
}
// ***************** Далее  следуют значения private ***********

// ************* Вспомогательные процедуры для ввода и вывода ********

private static InputStream in = System.in;
                                // новое имя для потока ввода
private static PrintStream out = System.out;
                               // новое имя для потока вывода
private static String buffer = null;  // чтение строки из потока ввода
private static int pos = 0;
                                // положение следующего необработанного
                                // символа в строке ввода
private static String readRealString()
{   // чтение символа в соответствии с синтаксисом представления
```

```
// чисел с плавающей точкой
StringBuffer s=new StringBuffer(50);
char ch=lookChar();
while (ch == ' ' || ch == '\n') {
    readChar();
    if (ch == '\n')
       dumpString("? ",0);
    ch = lookChar();
}
if (ch == '-' || ch == '+') {
    s.append(readChar());
    ch = lookChar();
    while (ch == ' ') {
       readChar();
       ch = lookChar();
    }
}
while (ch >= '0' && ch <= '9') {
    s.append(readChar());
    ch = lookChar();
}
if (ch == '.') {
   s.append(readChar());
   ch = lookChar();
   while (ch >= '0' && ch <= '9') {
       s.append(readChar());
       ch = lookChar();
   }
}
if (ch == 'E' || ch == 'e') {
   s.append(readChar());
   ch = lookChar();
   if (ch == '-' || ch == '+') {
       s.append(readChar());
       ch = lookChar();
   }
   while (ch >= '0' && ch <= '9') {
       s.append(readChar());
       ch = lookChar();
```

```
        }
    }
    return s.toString();
}
private static long readInteger(long min, long max)
{  // чтение длинного целого в заданных пределах
  long x=0;
  while (true) {
     StringBuffer s=new StringBuffer(34);
     char ch=lookChar();
     while (ch == ' ' || ch == '\n') {
           readChar();
           if (ch == '\n');
              dumpString("? ",0);
           ch = lookChar();
       }
       if (ch == '-' || ch == '+') {
           s.append(readChar());
           ch = lookChar();
           while (ch == ' ') {
              readChar();
              ch = lookChar();
           }
       }
       while (ch >= '0' && ch <= '9') {
           s.append(readChar());
           ch = lookChar();
       }
       if (s.equals("")){
           errorMessage("Illegal integer input.",
                        "Integer in the range " + min + " to " + max);
       }
       else {
           String str = s.toString();
           try {
              x = Long.parseLong(str);
           }
           catch (NumberFormatException e) {
              errorMessage("Illegal integer input.",
```

```
                        "Integer in the range " + min + " to " +
                         max);
            continue;
         }
         if (x < min || x > max) {
            errorMessage("Integer input outside of legal range.",
                        "Integer in the range " + min + " to " + max);
            continue;
         }
         break;
      }
   }
   return x;
}

private static String realToString(double x)
{
// Получение представления числа с плавающей точкой длиной
// в 10 или 11 (для отрицательного значения) символов.
   if (Double.isNaN(x))
      return "undefined";
   if (Double.isInfinite(x))
      if (x < 0)
         return "-INF";
      else
         return "INF";
   if (Math.abs(x) <= 5000000000.0 && Math.rint(x) == x)
      return String.valueOf((long)x);
   String s = String.valueOf(x);
   if (s.length() <= 10)
      return s;
   boolean neg = false;
   if (x < 0) {
      neg = true;
      x = -x;
      s = String.valueOf(x);
   }
   if (x >= 0.00005 && x <= 50000000 && (s.indexOf('E') == -1 &&
    s.indexOf('e') == -1))
```

```
     {  // оставляем максимальное число символов, равное 10
       s = round(s,10);
       s = trimZeros(s);
     }
     else if (x > 1)
{ // строим экспоненциальное представление с положительным
  // показателем
     long power = (long)Math.floor(Math.log(x)/Math.log(10));
       String exp = "E" + power;
       int numlength = 10 — exp.length();
       x = x / Math.pow(10,power);
       s = String.valueOf(x);
       s = round(s,numlength);
       s = trimZeros(s);
       s += exp;
   }
   else {
       long power = (long)Math.ceil(-Math.log(x)/Math.log(10));
       String exp = "E-" + power;
       int numlength = 10 — exp.length();
       x = x * Math.pow(10,power);
       s = String.valueOf(x);
       s = round(s,numlength);
       s = trimZeros(s);
       s += exp;
   }
   if (neg)
      return "-" + s;
   else
      return s;
}

private static String trimZeros(String num)
{  // используется в методе realToString
  if (num.indexOf('.') >= 0 && num.charAt(num.length() — 1) == '0') {
     int i = num.length() — 1;
     while (num.charAt(i) == '0')
        i--;
     if (num.charAt(i) == '.')
```

```
        num = num.substring(0,i);
     else
        num = num.substring(0,i+1);
  }
  return num;
}

private static String round(String num, int length)
{  // используется в методе realToString
 if (num.indexOf('.') < 0)
    return num;
 if (num.length() <= length)
    return num;
 if (num.charAt(length) >= '5' && num.charAt(length) != '.') {
    char[] temp = new char[length+1];
    int ct = length;
    boolean rounding = true;
    for (int i = length-1; i >= 0; i--) {
       temp[ct] = num.charAt(i);
       if (rounding && temp[ct] != '.') {
          if (temp[ct] < '9') {
             temp[ct]++;
             rounding = false;
          }
          else
             temp[ct] = '0';
       }
       ct--;
    }
    if (rounding) {
       temp[ct] = '1';
       ct--;
    }
    // ct равно -1 или 0
    return new String(temp,ct+1,length-ct);
}
 else
    return num.substring(0,length);
```

```
}
private static void dumpString(String str, int w)
{   // вывод строки на консоль
    for (int i=str.length(); i<w; i++)
    out.print(' ');
    for (int i=0; i<str.length(); i++)
    if ((int)str.charAt(i) >= 0x20 && (int)str.charAt(i) != 0x7F)
                        // нет символов, подлежащих удалению
       out.print(str.charAt(i));
    else if (str.charAt(i) == '\n' || str.charAt(i) == '\r')
       newLine();
}

private static void errorMessage(String message, String expecting)
{   // Сообщение пользователю об ошибке и предложение повторить ввод.
    newLine();
    dumpString("  *** Error in input: " + message + "\n", 0);
    dumpString("  *** Expecting: " + expecting + "\n", 0);
    dumpString("  *** Discarding Input: ", 0);
    if (lookChar() == '\n')
       dumpString("(end-of-line)\n\n",0);
    else {
       while (lookChar() != '\n')
          out.print(readChar());
       dumpString("\n\n",0);
    }
    dumpString("Please re-enter: ", 0);
    readChar();  // игнорирует символы в конце
}

private static char lookChar()
{  // возвращает следующий введенный символ
return next character from input
  if (buffer == null || pos > buffer.length())
    fillBuffer();
  if (pos == buffer.length())
     return '\n';
  return buffer.charAt(pos);
}
```

```
private static char readChar()
  {  // возвращает следующий символ в потоке ввода и игнорирует его
   char ch = lookChar();
   pos++;
   return ch;
}
private static void newLine()
  {   // выводит на консоль символ CR
   out.println();
   out.flush();
}

private static boolean possibleLinefeedPending = false;
private static void fillBuffer()
  {    // Ожидает пользовательский ввод,
       // помещает ввод в буфер.
   StringBuffer b = new StringBuffer();
   out.flush();
   try {
      int ch = in.read();
      if (ch == '\n' && possibleLinefeedPending)
         ch = in.read();
      possibleLinefeedPending = false;
      while (ch != -1 && ch != '\n' && ch != '\r') {
          b.append((char)ch);
          ch = in.read();
      }
      possibleLinefeedPending = (ch == '\r');
      if (ch == -1) {
         System.out.println("\n*** Found an end-of-file while trying
                                      to read from standard input!");
         System.out.println("*** Maybe your Java system doesn't
                                         implement standard input?");
         System.out.println("*** Program will be terminated.\n");
         throw new RuntimeException("End-of-file on standard input.");
      }
   }
   catch (IOException e) {
     System.out.println("Unexpected system error on input.");
```

```
      System.out.println("Terminating program.");
       System.exit(1);
    }
    buffer = b.toString();
    pos = 0;
}

private static void emptyBuffer()
  {   // игнорирует оставшуюся часть строки ввода
   buffer = null;
  }
}
```

Приложение 5

Краткая справка по апплетам

В этом приложении описаны некоторые элементы API, часто используемые при работе с апплетами.

Класс *Component*

- `java.awt.Component`

Основан на классе `java.lang.Object`. Имплементирует интерфейс `java.awt.image.ImageObserver`, `java.awt.MenuContainer`, `java.io.Serializable`.

Класс *java.awt.Button*

Создает кнопку с текстом-меткой.

Методы класса.

- `viod addActionListener(ActionListener l)`

  Вставляет прослушиватель событий.

- `void addNotify ()`

  Делает данный компонент видимым, сопоставляет его ресурсу экрана.

- `string getActionCommand()`

  Возвращает имя команды, выполняемой при возникновении события.

- `String getLabel()`

  Возвращает метку кнопки.

- `protected String paramString()`

  Возвращает строку состояния.

- `protected void processActionEvent(ActionEvent e)`

  Обрабатывает событие кнопки, направляя его зарегистрированному объекту прослушивания событий.

- `protected void processEvent(AWTEvent e)`

  Обрабатывает событие.

- `Void removeActionListener(ActionListener l)`

  Удаляет прослушиватель событий действия из кнопки.

- `Void setActionCommand(String command)`

  Устанавливает имя команды обработки событий действия.

- `Void setLabel(String label)`

  Задает строку, отображаемую на кнопке.

Класс *java.awt.Canvas*

Представляет прямоугольную область (холст), в которой помещаются компоненты и элементы.

Методы класса.

- `void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.

- `void paint(Graphics g)`

  Используется для перерисовывания холста.

Класс *java.awt.Checkbox*

Имплементирует интерфейс `java.awt.ItemSelectable`.

Поле для отметки — графический компонент, который может иметь два состояния (`true` и `false`) (отмечено и не отмечено).

Методы класса.

- `Void addItemListener(ItemListener l)`

  Вставляет прослушиватель событий.

- `Void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.

- `CheckboxGroup getCheckboxGroup()`

  Определяет группу, к которой относится данное поле.

- `String getLabel()`

  Получает метку данного поля.

- `Object[] getSelectedObjects()`

  Возвращает массив (длина 1) меток поля или `null`, если поле не отмечено.

- `Boolean getState()`

  Определяет, отмечено ли данное поле.

- `protected String paramString()`
  Возвращает строку параметра.
- `protected void processEvent(AWTEvent e)`
  Обрабатывает событие.
- `protected void processItemEvent(ItemEvent e)`
  Обрабатывает событие, направляя его зарегистрированному обработчику `ItemListener`.
- `Void removeItemListener(ItemListener l)`
  Удаляет прослушиватель события.
- `void setCheckboxGroup(CheckboxGroup g)`
  Задает группу полей.
- `void setLabel(String label)`
  Задает метку поля.
- `void setState(boolean state)`
  Задает состояние поля.

Класс *java.awt.Choice*

Имплементирует интерфейс `java.awt.ItemSelectable`.

Меню списка выбора. Методы класса.

- `void add(String item)`
  Вставляет элемент в меню списка выбора.
- `void addItem(String item)`
  Вставляет элемент в список.
- `Void addItemListener(ItemListener l)`
  Вставляет прослушиватель событий.
- `Void addNotify()`
  Переопределяет метод `addNotify()` класса `java.awt.component`.
- `String getItem(int index)`
  Возвращает строку по номеру элемента.
- `Int getItemCount()`
  Возвращает число элементов в списке.
- `Int getSelectedIndex()`
  Возвращает индекс выбранного элемента.
- `String getSelectedItem()`
  Строка представления текущего выбора.

- `Object[] getSelectedObjects()`

  Возвращает массив (длина 1) выбранных элементов.
- `Void insert(String item, int index)`

  Вставляет элемент в указанное положение.
- `protected  String paramString()`

  Возвращает строку параметра состояния.
- `protected  void processEvent(AWTEvent e)`

  Обрабатывает событие.
- `protected  void processItemEvent(ItemEvent e)`

  Обрабатывает событие, перенаправляя его зарегистрированному обработчику.
- `Void remove(int position)`

  Удаляет элемент из списка по указанной позиции.
- `Void remove(String item)`

  Удаляет первый раз встречающийся элемент по строке.
- `Void removeAll()`

  Удаляет все элементы.
- `Void removeItemListener(ItemListener l)`

  Удаляет прослушиватель.
- `Void select(int pos)`

  Задает выбранный элемент по номеру элемента.
- `Void select(String str)`

  Задает выбранный элемент по имени.

Класс *java.awt.Container*

Контейнер, служит для вставки в него прочих графических компонентов.

Методы класса.

- `Component add(Component comp)`

  Вставить компонент в конец контейнера.
- `Component add(Component comp, int index)`

  Вставить компонент в указанную позицию.
- `Void add(Component comp, Object constraints)`

  Вставить компонент в конец контейнера с ограничениями.

- `Void add(Component comp, Object constraints, int index)`

  Вставить компонент с ограничениями в указанную позицию.
- `Component add(String name, Component comp)`

  Вставить компонент в контейнер.
- `Void addContainerListener(ContainerListener l)`

  Вставить прослушиватель событий.
- `protected void addImpl(Component comp, Object constraints, int index)`

  Вставить компонент в указанную позицию.
- `Void addNotify()`

  Позволить контейнеру быть видимым для ресурсов экрана.
- `Void doLayout()`

  Заставить контейнер разместить компоненты.
- `Component findComponentAt(int x, int y)`

  Найти видимый дочерний элемент, расположенный в заданной позиции.
- `Component findComponentAt(Point p)`

  Найти видимый дочерний элемент, содержащий заданную точку.
- `Float getAlignmentX()`

  Возвращает способ выравнивания по оси `x`.
- `Float getAlignmentY()`

  Возвращает способ выравнивания по оси `y`.
- `Component getComponent(int n)`

  Получить n-ный компонент контейнера.
- `Component getComponentAt(int x, int y)`

  Получить компонент, содержащий позицию по указанным координатам.
- `Component getComponentAt(Point p)`

  Получить компонент, содержащий заданную точку.
- `Int getComponentCount()`

  Получить количество компонентов в панели.
- `Component[] getComponents()`

  Получить число всех компонентов.
- `Insets getInsets()`

  Получить размер границы контейнера.
- `LayoutManager getLayout()`

  Получить управление конфигурацией контейнера.

- `Dimension getMaximumSize()`

  Возвращает максимальный размер контейнера.
- `Dimension getMinimumSize()`

  Возвращает минимальный размер контейнера.
- `Dimension getPreferredSize()`

  Возвращает предпочтительный размер контейнера.
- `Void invalidate()`

  Выключает контейнер.
- `Boolean isAncestorOf(Component c)`

  Проверяет, является ли компонент родительским в иерархии компонентов.
- `Void list(PrintStream out, int indent)`

  Печатает контейнер, отправляя его в поток вывода.
- `Void list(PrintWriter out, int indent)`

  Выводит контейнер на указанное устройство начиная с заданной позиции.
- `Void paint(Graphics g)`

  Рисует контейнер.
- `Void paintComponents(Graphics g)`

  Рисует все компоненты в этом контейнере.
- `protected  String paramString()`

  Возвращает строку состояния контейнера.
- `Void print(Graphics g)`

  Печатает контейнер.
- `Void printComponents(Graphics g)`

  Печатает все компоненты контейнера.
- `protected  void processContainerEvent(ContainerEvent e)`

  Обрабатывает события контейнера, направляя их зарегистрированному объекту `ContainerListener`.
- `protected  void processEvent(AWTEvent e)`

  Обрабатывает событие контейнера.
- `Void remove(Component comp)`

  Удаляет указанный компонент из контейнера.
- `Void remove(int index)`

  Удаляет компонент по индексу.
- `Void removeAll()`

  Удаляет все компоненты контейнера.

- `Void removeContainerListener(ContainerListener l)`

  Удаляет прослушиватель события из контейнера.

- `Void removeNotify()`

  Делает контейнер невидимым, удаляя связь контейнера с экранными ресурсами.

- `Void setCursor(Cursor cursor)`

  Задает вид курсора.

- `Void setFont(Font f)`

  Задает шрифт для контейнера.

- `Void setLayout(LayoutManager mgr)`

  Задает менеджер для контейнера.

- `Void update(Graphics g)`

  Обновляет контейнер.

- `Voidv validate()`

  Действует на контейнер и все его компоненты.

- `protected  void validateTree()`

  Последовательно действует на все компоненты в дереве вложений.

Класс *java.awt.Panel*

Метод класса.

- `Void addNotify()`

  Позволяет панели быть видимой для ресурсов экрана.

Класс *java.awt.ScrollPane*

Контейнер с автоматическим механизмом прокручивания (панель прокрутки) вертикально и горизонтально для одного дочернего компонента.

Режимы прокручивания:

- `static int SCROLLBARS_ALWAYS` — полоса прокрутки появляется только тогда, когда необходимо;
- `static int SCROLLBARS_AS_NEEDED` — полоса прокрутки присутствует всегда;
- `static int SCROLLBARS_NEVER` — полоса прокрутки не показывается никогда.

Методы класса.

- ```
  protected  void addImpl(Component comp, Object constraints,
                          int index)
  ```

  Вставляет компонент.
  ```

- `Void doLayout()`

  Задает внешний вид контейнера.
- `Adjustable getHAdjustable()`

  Возвращает объект `Adjustable` по отношению к горизонтальной прокрутке.
- `Int getHScrollbarHeight()`

  Высота прокрутки.
- `Int getScrollbarDisplayPolicy()`

  Политика отображения прокрутки.
- `Point getScrollPosition()`

  Положение прокрутки.
- `Adjustable getVAdjustable()`

  Возвращает объект `Adjustable` по отношению к вертикальной прокрутке.
- `Dimension getViewportSize()`

  Величина панели прокрутки.
- `Int getVScrollbarWidth()`

  Ширина вертикальной панели прокрутки.
- `String paramString()`

  Строковый параметр контейнера прокрутки.
- `Void printComponents(Graphics g)`

  Выводит компонент в панели контейнера прокрутки.
- `Void setLayout(LayoutManager mgr)`

  Задает менеджер внешнего вида.
- `Void setScrollPosition(int x, int y)`

  Прокручивает до указанной позиции.
- `Void setScrollPosition(Point p)`

  Порокручивает до указанной точки.

Класс *java.awt.Window*

Объект верхнего уровня, не имеет границ, панелей меню. По умолчанию задается тип `BorderLayout`. Объект `Window` должен содержать в себе либо фрейм, диалог или другое окно.

С окном связаны события:

- `WindowOpened`;
- `WindowClosed`.

Методы класса.

- `void addNotify()`

  Создает связь с ресурсами экрана.

- `Void addWindowListener(WindowListener l)`

  Вставляет прослушиватель.

- `Void applyResourceBundle(String rbName)`

  Загружает `ResourceBundle` по указанному имени.

- `Void dispose()`

  Высвобождает ресурсы экрана.

- `protected  void finalize()`

  Высвобождает методы ввода.

- `Component getFocusOwner()`

  Возвращает дочерний компонент, содержащий фокус, если только окно активно.

- `InputContext getInputContext()`

  Получает контекст ввода для окна.

- `Locale getLocale()`

  Получает объект `Locale` для окна.

- `Window[] getOwnedWindows()`

  Возвращает массив всех окон, вложенных в данное окно.

- `Window getOwner()`

  Возвращает владельца данного окна.

- `Toolkit getToolkit()`

  Возвращает `Toolkit`.

- `String getWarningString()`

  Возвращает строку предупреждения.

- `Void hide()`

  Скрывает окно и все его компоненты.

- `Boolean isShowing()`

  Проверяет, показано ли окно на экране.

- `Void pack()`

  Вынуждает окно учесть размер компонентов.

- `protected  void processEvent(AWTEvent e)`

  Обрабатывает событие.

- `protected void processWindowEvent(WindowEvent e)`

  Обрабатывает событие, перенаправляя его зарегистрированному обработчику `WindowListener`.

- `Void removeWindowListener(WindowListener l)`

  Удаляет указанный прослушиватель событий.

- `Void setCursor(Cursor cursor)`

  Задает рисунок курсора.

- `Void show()`

  Делает окно видимым.

- `Void toBack()`

  Отправляет окно на задний план.

- `Void toFront()`

  Выводит окно на передний план.

Класс *java.awt.Dialog*

Диалоговое окно — окно верхнего уровня с названием и границами. Оно, как правило, используется для осуществления пользовательского ввода. Окно может быть как модальным, так и немодальным. В окне могут создаваться следующие события:

- `WindowOpened`;
- `WindowClosing`;
- `WindowClosed`;
- `WindowActivated`;
- `WindowDeactivated`.

Конструкторы класса.

- `Dialog(Dialog owner)`

  Создает первоначально невидимое немодальное диалоговое окно с пустым заголовком.

- `Dialog(Dialog owner, String title)`

  Создает первоначально невидимое немодальное окно с указанным заголовком.

- `Dialog(Dialog owner, String title, boolean modal)`

  Создает первоначально невидимое окно с заголовком и заданной модальностью.

- `Dialog(Frame owner)`

  Создает первоначально невидимое немодальное окно без заголовка с указанным владельцем фрейма.

- `Dialog(Frame owner, boolean modal)`

  Создает первоначально невидимое окно с пустым заголовком и заданной модальностью.

- `Dialog(Frame owner, String title)`

  Создает первоначально невидимое немодальное окно с заголовком и владельцем фрейма.

- `Dialog(Frame owner, String title, boolean modal)`

  Создает первоначально невидимое окно с заданным владельцем фрейма, модальностью и заголовком.

Методы класса.

- `Void addNotify()`

  Предоставляет окну ресурс отображения на экране.

- `Void dispose()`

  Переопределяет метод `dispose()` класса `java.awt.Window`.

- `String getTitle()`

  Получает заголовок окна.

- `void hide()`

  Скрывает окно.

- `boolean isModal()`

  Модальное окно или нет?

- `boolean isResizable()`

  Изменяемы размеры окна?

- `protected  String paramString()`

  Строка параметра, содержащего состояние окна.

- `void setModal(boolean b)`

  Сделать окно модальным?

- `void setResizable(boolean resizable)`

  Позволить пользователю изменять размер окна?

- `void setTitle(String title)`

  Задать заголовок окна.

- `void show()`

  Сделать окно видимым.

Класс *java.awt.FileDialog*

Диалоговое окно, в котором пользователь может выбрать файл.

Конструкторы класса.

- `FileDialog(Frame parent)`

  Создает окно выбора файла.

- `FileDialog(Frame parent, String title)`

  Создает окно выбора файла с указанным заголовком.

- `FileDialog(Frame parent, String title, int mode)`

  Создать окно м указанным заголовком для загрузки или сохранения файла.

Методы класса.

- `void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.

- `String getDirectory()`

  Каталог файлового окна.

- `String getFile()`

  Получить файл.

- `FilenameFilter getFilenameFilter()`

  Задать фильтр имен файлов окна.

- `int getMode()`

  Определять тип окна — для загрузки файла или для сохранения файла.

- `protected  String paramString()`

  Возвращает строку параметра состояния окна.

- `void setDirectory(String dir)`

  Устанавливает каналы окна.

- `void setFile(String file)`

  Устанавливает конкретный файл для окна.

- `void setFilenameFilter(FilenameFilter filter)`

  Задает фильтр имен файлов.

- `void setMode(int mode)`

  Задает тип окна.

Класс *java.awt.Frame*

Имплементирует интерфейс `java.awt.MenuContainer`.

Фрейм — это окно верхнего уровня, с заголовком и границами. Может генерировать следующие события:

- `WindowOpened`;
- `WindowClosing`;
- `WindowClosed`;
- `WindowIconified`;
- `WindowDeiconified`;
- `WindowActivated`;
- `WindowDeactivated`.

Методы класса.

- `void addNotify()`

  Предоставляет ресурс отображения окна на экране.

- `protected  void finalize()`

  Переопределяет метод `finalize()` класса `java.awt.windows`.

- `static Frame[] getFrames()`

  Возвращает массив всех фреймов приложения.

- `Image getIconImage()`

  Получает рисунок для пиктограмм фрейма.

- `MenuBar getMenuBar()`

  Получает панель меню для фрейма.

- `int getState()`

  Получает состояние фрейма.

- `String getTitle()`

  Возвращает заголовок фрейма.

- `boolean isResizable()`

  Может ли пользователь изменять размер фрейма.

- `protected  String paramString()`

  Возвращает параметр `String` для фрейма.

- `void remove(MenuComponent m)`

  Удаляет указанную панель меню.

- `void removeNotify()`

  Удаляет ресурс экрана, делает фрейм недоступным для показа.

- `void setIconImage(Image image)`

  Использует рисунок в качестве пиктограммы для фрейма.
- `void setMenuBar(MenuBar mb)`

  Задает панель меню для фрейма.
- `void setResizable(boolean resizable)`

  Может ли пользователь изменять размер фрейма.
- `void setState(int state)`

  Задает состояние фрейма.
- `void setTitle(String title)`

  Задает заголовок фрейма.

Класс *java.awt.Label*

Метка используется для размещения текста в контейнере. Текст только для чтения.

Конструкторы класса.

- `Label()`

  Пустая метка.
- `Label(String text)`

  Метка с текстом.
- `Label(String text, int alignment)`

  Метка с текстом и параметром выравнивания.

Методы класса.

- `void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.
- `int getAlignment()`

  Получает способ выравнивания метки.
- `String getText()`

  Получает текст метки.
- `protected  String paramString()`

  Возвращает строку параметра состояния метки.
- `void setAlignment(int alignment)`

  Устанавливает способ выравнивания метки.
- `void setText(String text)`

  Задает текст метки.

Класс *java.awt.List*

Имплементирует интерфейс `java.awt.ItemSelectable`.

Список текстовых строк с прокруткой для множественного выбора.

Конструкторы класса.

- `List()`

  Новый список с прокруткой.

- `List(int rows)`

  Новый список с указанным количеством строк.

- `List(int rows, boolean multipleMode)`

  Новый список с указанным числом показываемых строк.

Методы класса.

- `void add(String item)`

  Вставляет элемент в конец списка.

- `void add(String item, int index)`

  Вставляет элемент в указанную позиция списка.

- `void addActionListener(ActionListener l)`

  Вставляет прослушиватель событий.

- `void addItemListener(ItemListener l)`

  Вставляет прослушиватель событий элементов списка.

- `void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.

- `void deselect(int index)`

  Снимает выбор с указанного элемента в списке.

- `String getItem(int index)`

  Получает элемент по номеру элемента в списке.

- `int getItemCount()`

  Получает число элементов в списке.

- `String[] getItems()`

  Получает массив элементов списка.

- `Dimension getMinimumSize()`

  Определяет минимальный размер списка с прокруткой.

- `Dimension getMinimumSize(int rows)`

  Получает минимальное количество строк.

- `Dimension getPreferredSize()`

  Получает предпочтительный размер списка.

- `Dimension getPreferredSize(int rows)`

  Получает предпочтительный размер списка в строках.

- `int getRows()`

  Получает количество видимых строк списка.

- `int getSelectedIndex()`

  Получает номер (индекс) выбранного элемента списка.

- `int[] getSelectedIndexes()`

  Получает индексы выбранных элементов списка.

- `String getSelectedItem()`

  Получает выбранный элемент списка.

- `String[] getSelectedItems()`

  Получает выбранные элементы списка.

- `Object[] getSelectedObjects()`

  Возвращает выбранные элементы в виде массива объектов.

- `int getVisibleIndex()`

  Получает индекс элемента, который был сделан видимым при последнем обращении к методу `makeVisible`.

- `boolean isIndexSelected(int index)`

  Проверяет, является ли указанный элемент списка выбранным элементом.

- `boolean isMultipleMode()`

  Проверяет, разрешен ли в списке множественный выбор.

- `void makeVisible(int index)`

  Делает элемент списка видимым.

- `protected String paramString()`

  Возвращает строку параметра состояния списка с прокруткой.

- `protected void processActionEvent(ActionEvent e)`

  Обрабатывает событие, направляя его зарегистрированному объекту `ActionListener`.

- `protected void processEvent(AWTEvent e)`

  Обрабатывает событие списка с прокруткой.

- `protected void processItemEvent(ItemEvent e)`

  Обрабатывает событие, направляя его зарегистрированному объекту `ItemListener`.

- `void remove(int position)`

  Удаляет элемент по указанной позиции.

- `void remove(String item)`

  Удаляет первое вхождение элемента в списке.

- `void removeActionListener(ActionListener l)`

  Удаляет указанный прослушиватель события из списка.

- `void removeAll()`

  Удаляет все элементы из списка.

- `void removeItemListener(ItemListener l)`

  Удаляет указанный прослушиватель.

- `void replaceItem(String newValue, int index)`

  Заменяет элемент списка на новый.

- `void select(int index)`

  Производит выбор указанного элемента списка.

- `void setMultipleMode(boolean b)`

  Задает флаг, определяющий то, разрешен ли в списке множественный выбор.

Класс *java.awt.Scrollbar*

Имплементирует интерфейс `java.awt.Adjustable`.

Панель прокрутки.

- `static int HORIZONTAL`

  Горизонтальная панель прокрутки.

- `static int VERTICAL`

  Вертикальная панель прокрутки.

Конструкторы класса.

- `Scrollbar()`

  Создает новую панель прокрутки.

- `Scrollbar(int orientation)`

  Создает новую панель прокрутки с указанием ориентации панели.

- `Scrollbar(int orientation, int value, int visible, int minimum, int maximum)`

  Создает панель прокрутки с указанием ориентации, начальным значением, видимостью, максимальным и минимальным значениями.

Методы класса.

- `void addAdjustmentListener(AdjustmentListener l)`

  Создает прослушиватель, получающий события `AdjustmentEvent`.
- `void addNotify()`
- `int getBlockIncrement()`

  Получает инкремент панели прокрутки.
- `int getMaximum()`

  Получает максимальный размер панели прокрутки.
- `int getMinimum()`

  Получает минимальный размер панели прокрутки.
- `int getOrientation()`

  Определяет ориентацию панели прокрутки.
- `int getUnitIncrement()`

  Получает единицу инкремента панели прокрутки.
- `int getValue()`

  Получает текущее значение.
- `int getVisibleAmount()`

  Получает значение величины видимой части панели прокрутки.
- `protected String paramString()`

  Возвращает строку параметра состояния.
- `protected void processAdjustmentEvent(AdjustmentEvent e)`

  Обрабатывает события, направляя зарегистрированному объекту `AdjustmentListener`.
- `protected void processEvent(AWTEvent e)`

  Обрабатывает событие.
- `void removeAdjustmentListener(AdjustmentListener l)`

  Удаляет указанный прослушиватель событий `AdjustmentEvent`.
- `void setBlockIncrement(int v)`

  Задает инкремент.
- `void setMaximum(int newMaximum)`

  Получает максимальное значение панели прокрутки.
- `void setMinimum(int newMinimum)`

  Задает минимальное значение панели прокрутки.
- `void setOrientation(int orientation)`

  Задает ориентацию панели прокрутки.

- `void setUnitIncrement(int v)`

  Задает единицу инкремента.
- `void setValue(int newValue)`

  Устанавливает значение панели прокрутки на укзанное.
- `void setValues(int value, int visible, int minimum, int maximum)`

  Устанавливает значения свойств панели прокрутки.
- `void setVisibleAmount(int newAmount)`

  Устанавливает видимую часть панели прокрутки.

Класс *java.awt.TextComponent*

Суперкласс для всех компонентов редактирования текста.

Методы класса.

- `void addTextListener(TextListener l)`

  Вставляет прослушиватель событий.
- `int getCaretPosition()`

  Получает положение курсора.
- `String getSelectedText()`

  Получает выбранный текст.
- `int getSelectionEnd()`

  Получает положение последнего символа выбранного текста.
- `int getSelectionStart()`
- `String getText()`
- `boolean isEditable()`
- `protected  String paramString()`
- `protected  void processEvent(AWTEvent e)`
- `protected  void processTextEvent(TextEvent e)`
- `void removeNotify()`
- `void removeTextListener(TextListener l)`
- `void select(int selectionStart, int selectionEnd)`
- `void selectAll()`
- `void setCaretPosition(int position)`
- `void setEditable(boolean b)`
- `void setSelectionEnd(int selectionEnd)`
- `void setSelectionStart(int selectionStart)`
- `void setText(String t)`

Класс *java.awt.TextArea*

Объект для редактирования (или только для чтения) многострочного текста.

- `static int SCROLLBARS_BOTH`

  Прокрутка по вертикали и горизонтали.

- `static int SCROLLBARS_HORIZONTAL_ONLY`

  Горизонтальная прокрутка.

- `static int SCROLLBARS_NONE`

  Нет прокрутки.

- `static int SCROLLBARS_VERTICAL_ONLY`

  Вертикальная прокрутка.

Конструкторы класса.

- `TextArea()`

  Создать новое текстовое поле.

- `TextArea(int rows, int columns)`

  Создать новое поле с указанным числом столбцов и рядов.

- `TextArea(String text)`

  Создать поле с заданным текстом.

- `TextArea(String text, int rows, int columns)`

  Создать поле с заданным текстом и по заданным размерам.

- `TextArea(String text, int rows, int columns, int scrollbars)`

  Создать поле с текстом по заданным размерам и с панелью прокрутки.

Методы класса.

- `void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.

- `void append(String str)`

  Вставляет текст.

- `int getColumns()`

  Получает количество столбцов текстовой области.

- `Dimension getMinimumSize()`

  Определяет минимальный размер текстовой области.

- `Dimension getMinimumSize(int rows, int columns)`

  Определяет минимальный размер текстовой области, выраженный в количестве строк и столбцов.

- `Dimension getPreferredSize()`

  Определяет предпочтительный размер текстовой области.

- `Dimension getPreferredSize(int rows, int columns)`

  Определяет предпочтительный размер текстовой области, выраженный в количестве строк и столбцов.
- `int getRows()`

  Определяет количество строк текстовой области.
- `int getScrollbarVisibility()`

  Определяет номер используемой прокрутки.
- `void insert(String str, int pos)`

  Вставляет текст в указанную позицию.
- `protected  String paramString()`

  Возвращает строку параметра состояния текстовой области.
- `void replaceRange(String str, int start, int end)`

  Удаляет текст, расположенный между заданными позициями.
- `void setColumns(int columns)`

  Задает количество столбцов текстовой области.
- `void setRows(int rows)`

  Задает количество строк текстовой области.

Класс *java.awt.TextField*

Текстовое поле для редактирования одной строки текста.

Конструкторы класса.

- `TextField()`

  Создать текстовое поле.
- `TextField(int columns)`

  Создать текстовое пол с указанным числом столбцов.
- `TextField(String text)`

  Создать текстовое поле с указанным текстом.
- `TextField(String text, int columns)`

  Создать текстовое поле с указанным текстом и заданным числом столбцов.

Методы класса.

- `void addActionListener(ActionListener l)`

  Вставить указанный прослушиватель событий для однострочного текстового поля.
- `void addNotify()`

  Переопределяет метод `addNotify()` класса `java.awt.component`.

- `boolean echoCharIsSet()`

  Задан ли режим эхо (отображения вводимых символов).
- `int getColumns()`

  Получает количество столбцов поля.
- `char getEchoChar()`

  Получает символ, установленный для эхо.
- `Dimension getMinimumSize()`

  Получает минимальный размер поля.
- `Dimension getMinimumSize(int columns)`

  Получает минимальный размер поля, выраженный в количестве столбцов.
- `Dimension getPreferredSize()`

  Получает предпочтительный размер поля.
- `Dimension getPreferredSize(int columns)`

  Получает предпочтительный размер поля, выраженный в количестве столбцов.
- `protected String paramString()`

  Возвращает строку параметра состояния поля.
- `protected void processEvent(AWTEvent e)`

  Обрабатывает событие.
- `void removeActionListener(ActionListener l)`

  Удаляет прослушиватель.
- `void setColumns(int columns)`

  Задает количество столбцов.
- `void setEchoChar(char c)`

  Задает символ для эхо.
- `void setText(String t)`

  Задает текст.

Класс *JComponent*

Класс `javax.swing.JComponent` имплементирует интерфейс `java.io.Serializable`.

Базовый класс для создания компонентов `Swing`.

Класс *javax.swing.AbstractButton*

Имплементирует интерфейсы `java.awt.ItemSelectable`, `javax.swing.SwingConstants`.

Значения.

- `protected ActionListener actionListener`
- `static String BORDER_PAINTED_CHANGED_PROPERTY`
- `protected ChangeEvent changeEvent`
- `protected ChangeListener changeListener`
- `static String CONTENT_AREA_FILLED_CHANGED_PROPERTY`
- `static String DISABLED_ICON_CHANGED_PROPERTY`
- `static String DISABLED_SELECTED_ICON_CHANGED_PROPERTY`
- `static String FOCUS_PAINTED_CHANGED_PROPERTY`
- `static String HORIZONTAL_ALIGNMENT_CHANGED_PROPERTY`
- `static String HORIZONTAL_TEXT_POSITION_CHANGED_PROPERTY`
- `static String ICON_CHANGED_PROPERTY`
- `protected ItemListener itemListener`
- `static String MARGIN_CHANGED_PROPERTY`
- `static String MNEMONIC_CHANGED_PROPERTY`
- `protected ButtonModel model`
- `static String MODEL_CHANGED_PROPERTY`
- `static String PRESSED_ICON_CHANGED_PROPERTY`
- `static String ROLLOVER_ENABLED_CHANGED_PROPERTY`
- `static String ROLLOVER_ICON_CHANGED_PROPERTY`
- `static String ROLLOVER_SELECTED_ICON_CHANGED_PROPERTY`
- `static String SELECTED_ICON_CHANGED_PROPERTY`
- `static String TEXT_CHANGED_PROPERTY`
- `static String VERTICAL_ALIGNMENT_CHANGED_PROPERTY`
- `static String VERTICAL_TEXT_POSITION_CHANGED_PROPERTY`

Конструкторы класса.

- `AbstractButton()`

Класс *javax.swing.JButton*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Кнопка для нажатия.

Конструкторы класса.

- `JButton()`

  Создает кнопку без метки (текстовой или иконки).

- `JButton(Icon icon)`

  Кнопка с пиктограммой.

- `JButton(String text)`

  Кнопка с текстом.

- `JButton(String text, Icon icon)`

  Кнопка с текстом и пиктограммой.

Класс *javax.swing.plaf.basic.BasicArrowButton*

Имплементирует интерфейс `javax.swing.SwingConstants`.

Кнопка — стрелка.

Конструкторы класса.

- `BasicArrowButton(int direction)`

Класс *javax.swing.plaf.metal.MetalScrollButton*

Металлическая кнопка.

Конструкторы класса.

- ```
  MetalScrollButton(int direction, int width,
                    boolean freeStanding)
  ```

Класс *javax.swing.plaf.metal.MetalComboBoxButton*

Конструкторы класса.

- ```
  MetalComboBoxButton(JComboBox cb, Icon i, boolean onlyIcon,
                      CellRendererPane  ane, JList list)
  ```

- ```
  MetalComboBoxButton(JComboBox cb, Icon i, CellRendererPane pane,
                      JList list)
  ```

Класс *javax.swing.JMenuItem*

Имплементирует интерфейс `javax.accessibility.Accessible`, `javax.swing.MenuElement`.

Элемент меню (кнопка в списке).

Конструкторы класса.

- `JMenuItem()`

  Элемент меню без текста и пиктограмм.
  ```

- `JMenuItem(Icon icon)`

  Элемент меню с пиктограммой.
- `JMenuItem(String text)`

  Элемент меню с текстом.
- `JMenuItem(String text, Icon icon)`

  Элемент меню с текстом и пиктограммой.
- `JMenuItem(String text, int mnemonic)`

  Элемент меню с текстом и клавишей быстрого доступа.

Класс *javax.swing.JCheckBoxMenuItem*

Имплементирует интерфейс `javax.accessibility.Accessible`, `javax.swing.SwingConstants`.

Элемент меню с выбором (может быть отмечено с помощью флажка).

Класс *javax.swing.JMenu*

Имплементирует интерфейс `javax.accessibility.Accessible`, `javax.swing.MenuElement`.

Элемент меню.

Конструкторы класса.

- `JMenu()`

  `JMenu` без текста.
- `JMenu(String)`

  `JMenu` с текстом.
- `JMenu(String s, boolean b)`

Класс *javax.swing.JRadioButtonMenuItem*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Элемент меню — радио-кнопка.

Конструкторы класса.

- `JRadioButtonMenuItem()`

  `JRadioButtonMenuItem` без текста и пиктограммы.
- `JRadioButtonMenuItem(Icon icon)`

  `JRadioButtonMenuItem` с пиктограммой.
- `JRadioButtonMenuItem(Icon icon, boolean selected)`

  С пиктограммой и указанным состоянием, но без текста.

- `JRadioButtonMenuItem(String text)`

  `JRadioButtonMenuItem` с текстом.
- `JRadioButtonMenuItem(String text, boolean b)`

  С текстом и заданным состоянием.
- `JRadioButtonMenuItem(String text, Icon icon)`

  `JRadioButtonMenuItem` с текстом и пиктограммой.
- `JRadioButtonMenuItem(String text, Icon icon, boolean selected)`

  С текстом, пиктограммой и заданным состоянием.

Класс *javax.swing.JToggleButton*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Кнопка с двумя состояниями.

Конструкторы класса.

- `JToggleButton()`

  Ненажатая кнопка переключения без текста и пиктограмм.
- `JToggleButton(Icon icon)`

  Ненажатая кнопка переключения без текста, но с пиктограммой.
- `JToggleButton(Icon icon, boolean selected)`

  Кнопка с заданным состоянием и пиктограммой, но без текста.
- `JToggleButton(String text)`

  Кнопка с текстом, без пиктограмм, не нажата.
- `JToggleButton(String text, boolean selected)`

  Кнопка с тестом и заданным состоянием.
- `JToggleButton(String text, Icon icon)`

  Кнопка с текстом и пиктограммой, не нажата.
- `JToggleButton(String text, Icon icon, boolean selected)`

  Кнопка с текстом, пиктограммой и указанным состоянием.

Класс *javax.swing.JCheckBox*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Поле для флажка.

Конструкторы класса.

- `JCheckBox()`

  Неотмеченное поле без текста и пиктограмм.

- `JCheckBox(Icon icon)`

  Неотмеченное поле без текста, но с пиктограммой.
- `JCheckBox(Icon icon, boolean selected)`

  Поле с пиктограммой и заданным состоянием.
- `JCheckBox(String text)`

  Поле без пиктограмм, но с текстом, неотмеченное.
- `JCheckBox(String text, boolean selected)`

  Поле с текстом, указанным состоянием, без пиктограмм.
- `JCheckBox(String text, Icon icon)`

  Поле без отметки, с пиктограммой и текстом.
- `JCheckBox(String text, Icon icon, boolean selected)`

  Поле с заданным состоянием, пиктограммой и текстом.

Класс *javax.swing.JRadioButton*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Радио-кнопка.

Конструкторы класса.

- `JRadioButton()`

  Неотмеченная кнопка без текста и без пиктограмм.
- `JRadioButton(Icon icon)`

  Неотмеченная кнопка без текста, но с пиктограммой.
- `JRadioButton(Icon icon, boolean selected)`

  Кнопка с заданным состоянием и с пиктограммой, но без текста.
- `JRadioButton(String text)`

  Кнопка с текстом, неотмеченная, без пиктограмм.
- `JRadioButton(String text, boolean selected)`

  Кнопка с текстом, заданным состоянием, без пиктограмм.
- `JRadioButton(String text, Icon icon)`

  Кнопка с текстом, пиктограммой, неотмеченная.
- `JRadioButton(String text, Icon icon, boolean selected)`

  Кнопка с заданным состоянием, текстом и пиктограммой.

Класс *javax.swing.plaf.basic.BasicInternalFrameTitlePane*

Класс для работы с панелью заголовка.
Конструктор класса.

- `BasicInternalFrameTitlePane(JInternalFrame f)`

Методы класса.

- `Void addNotify()`

  Сообщает компоненту о том, что в нем сейчас содержится дочерний компонент.

- `protected void addSubComponents()`

  Вставляет дочерний компонент.

- `protected void addSystemMenuItems(JMenu systemMenu)`

  Вставляет элемент меню.

- `protected void assembleSystemMenu()`

  Создает меню.

- `protected void createButtons()`

  Метод для создания кнопок.

- `protected LayoutManager createLayout()`

  Метод для создания конфигураций.

- `protected PropertyChangeListener createPropertyChangeListener()`

  Создает прослушиватель событий, связанных с изменением свойств.

- `protected JMenu createSystemMenu()`

  Создает меню.

- `protected JMenuBar createSystemMenuBar()`

  Создает панель меню.

- `protected void enableActions()`

  Включает обработчики событий.

- `protected void installDefaults()`

  Активизирует параметры по умолчанию.

- `protected void installListeners()`

  Использует заданные прослушиватели событий.

- `protected void installTitlePane()`

  Использует созданную панель заголовка.

- `void paintComponent(Graphics g)`

  Вызывает метод прорисовки.

- `protected void postClosingEvent(JInternalFrame frame)`

  Создает во фрейме событие, сходное с событием `WINDOW_CLOSING`.

- `Void removeNotify()`

  Сообщение компоненту о том, что он более не содержит дочерних компонентов.

- `protected void setButtonIcons()`

  Задает пиктограммы для кнопок.

- `protected void showSystemMenu()`

  Отображает меню.

- `protected void uninstallDefaults()`

  Отменяет параметры по умолчанию.

Класс *javax.swing.JColorChooser*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Панель для работы с цветом.

Конструкторы класса.

- `JColorChooser()`

  Панель выбора цвета, белый цвет.

- `JColorChooser(Color initialColor)`

  Панель выбора цвета с заданным цветом.

- `JColorChooser(ColorSelectionModel model)`

  Панель с заданной переменной `ColorSelectionModel`.

Класс *javax.swing.JComboBox*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `java.awt.event.ActionListener`, `java.awt.ItemSelectable`, `javax.swing.event.ListDataListener`.

Комбо-бокс.

Конструкторы класса.

- `JComboBox()`

  `JComboBox` с моделью по умолчанию.

- `JComboBox(ComboBoxModel aModel)`

  `JComboBox` на основе существующей модели `ComboBoxModel`.

- `JComboBox(Object[] items)`

  `JcomboBox` с указанными элементами массива.
- `JComboBox(Vector items)`

  `JComboBox` с элементами в соответствии с указанным вектором.

Класс *javax.swing.JFileChooser*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Механизм для выбора файла.

Конструкторы класса.

- `JFileChooser()`

  Выбор файла, текущий каталог пользователя.
- `JFileChooser(File currentDirectory)`

  Выбор файла с указанным каталогом.
- `JFileChooser(File currentDirectory, FileSystemView fsv)`

  Выбора файла с указанным каталогом и видом `FileSystemView`.
- `JFileChooser(FileSystemView fsv)`

  Выбор файла с указанным видом `FileSystemView`.
- `JFileChooser(String currentDirectoryPath)`

  Выбор файла по указанной папке.
- `JFileChooser(String currentDirectoryPath, FileSystemView fsv)`

  Выбора файла по указанной папке с указанным видом.

Методы класса.

- `boolean accept(File f)`

  `true` — файл следует отобразить.
- `void addActionListener(ActionListener l)`

  Вставка прослушивателя `ActionListener`.
- `void addChoosableFileFilter(FileFilter filter)`

  Вставка фильтра.
- `void approveSelection()`

  Нажата кнопка **OK**.
- `void cancelSelection()`

  Нажата кнопка **Cancel**.
- `void changeToParentDirectory()`

  Переход к родительскому каталогу.

- `void ensureFileIsVisible(File f)`

  Видимость файла.
- `protected  void fireActionPerformed(String command)`

  Генерация события.
- `FileFilter getAcceptAllFileFilter()`

  Фильтр `AcceptAll`.
- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext`, связанный с данным `JFileChooser`.
- `JComponent getAccessory()`

  Возвращает компонент.
- `int getApproveButtonMnemonic()`

  Возвращает значение кнопки.
- `String getApproveButtonText()`

  Возвращает текстовое значение кнопки.
- `String getApproveButtonToolTipText()`

  Возвращает подсказку для кнопки.
- `FileFilter[] getChoosableFileFilters()`

  Возвращает список доступных пользователю фильтров.
- `File getCurrentDirectory()`

  Возвращает текущий каталог.
- `String getDescription(File f)`

  Возвращает описание файла.
- `String getDialogTitle()`

  Возвращает заголовок диалогового окна.
- `int getDialogType()`

  Возвращает тип диалогового окна.
- `FileFilter getFileFilter()`

  Возвращает текущий тип фильтра.
- `int getFileSelectionMode()`

  Возвращает тип выбора файла.
- `FileSystemView getFileSystemView()`

  Возвращает тип представления файла.
- `FileView getFileView()`

  Возвращает представление файла.

- Icon getIcon(File f)
- String getName(File f)
- File getSelectedFile()
- File[] getSelectedFiles()
- String getTypeDescription(File f)
- FileChooserUI getUI()
- String getUIClassID()
- boolean isDirectorySelectionEnabled()
- boolean isFileHidingEnabled()
- boolean isFileSelectionEnabled()
- boolean isMultiSelectionEnabled()
- boolean isTraversable(File f)
- protected String paramString()
- void removeActionListener(ActionListener l)
- boolean removeChoosableFileFilter(FileFilter f)
- void rescanCurrentDirectory()
- void resetChoosableFileFilters()
- void setAccessory(JComponent newAccessory)
- void setApproveButtonMnemonic(char mnemonic)
- void setApproveButtonMnemonic(int mnemonic)
- void setApproveButtonText(String approveButtonText)
- void setApproveButtonToolTipText(String toolTipText)
- void setCurrentDirectory(File dir)
- void setDialogTitle(String dialogTitle)
- void setDialogType(int dialogType)
- void setFileFilter(FileFilter filter)
- void setFileHidingEnabled(boolean b)
- void setFileSelectionMode(int mode)
- void setFileSystemView(FileSystemView fsv)
- void setFileView(FileView fileView)
- void setMultiSelectionEnabled(boolean b)
- void setSelectedFile(File file)
- void setSelectedFiles(File[] selectedFiles)
- protected void setup(FileSystemView view)
- int showDialog(Component parent, String approveButtonText)

- `int showOpenDialog(Component parent)`
- `int showSaveDialog(Component parent)`
- `void updateUI()`

Класс *javax.swing.JInternalFrame*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.RootPaneContainer`, `javax.swing.WindowConstants`.

Легковесный фрейм.

Конструкторы класса.

- `JInternalFrame()`

  Создает фрейм без возможности изменения размера, незакрываемый, не разворачиваемый в полное окно, без пиктограммы, без заголовка.

- `JInternalFrame(String title)`

  `JinternalFrame` без изменения размера, незакрываемый, не разворачиваемый, без пиктограммы, но с заголовком.

- `JInternalFrame(String title, boolean resizable)`

  `JinternalFrame` с изменяемым размером, незакрываемый, не разворачиваемый, без пиктограммы, но с заголовком.

- ```
  JInternalFrame(String title, boolean resizable,
                 boolean closable)
  ```

  `JinternalFrame` с изменением размера, закрываемый, не разворачиваемый, без пиктограммы, с заголовком.

- ```
  JInternalFrame(String title, boolean resizable,
                 boolean closable, boolean maximizable)
  ```

  `JinternalFrame` с изменением размера, закрываемый, разворачиваемый, с заголовком.

- ```
  JInternalFrame(String title, boolean resizable,
                 boolean closable, boolean maximizable,
                 oolean iconifiable)
  ```

  `JInternalFrame` с заголовком, изменяемыми размерами, закрываемый, максимизируемый и с пиктограммой.

Класс *javax.swing.JInternalFrame.JDesktopIcon*

Имплементирует интерфейс `javax.accessibility.Accessible`

Класс *javax.swing.JLabel*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.SwingConstants`.
  ```

Метка — короткая строка или рисунок.

Конструкторы класса.

- `JLabel()`

  `Jlabel` без рисунка и текста.

- `JLabel(Icon image)`

  `JLabel` с пиктограммой.

- `Label(Icon image, int horizontalAlignment)`

  `JLabel` с пиктограммой и горизонтальным выравниванием.

- `JLabel(String text)`

  `JLabel` с указанным текстом.

- `JLabel(String text, Icon icon, int horizontalAlignment)`

  `JLabel` с текстом, пиктограммой, горизонтальным выравниванием.

- `JLabel(String text, int horizontalAlignment)`

  `JLabel` с текстом, горизонтальным выравниванием.

Класс *javax.swing.DefaultListCellRenderer*

Имплементирует интерфейсы `javax.swing.ListCellRenderer`, `java.io.Serializable`.

Отображает элемент списка.

Конструкторы класса.

`DefaultListCellRenderer()`
Создает объект, который используется для отображения элемента в списке.

Методы класса.

- `Component getListCellRendererComponent(JList list, Object value, int index, boolean isSelected, boolean cellHasFocus)`

  Возвращает компонент, используемый для отображения заданного значения.

Класс *javax.swing.table.DefaultTableCellRenderer*

Имплементирует интерфейсы `TableCellRenderer`, `Serializable`.

Стандартный класс для отображения отдельных ячеек в таблице `JTable`.

Конструкторы класса.

- `DefaultTableCellRenderer()`
  Создает объект для отображения ячеек таблицы.

Методы класса.

- `Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)`

  Метод используется для задания конфигурации объекта, отображающего ячейки таблицы.

- `Void setBackground(Color c)`

  Переопределяет метод `JComponent.setForeground`.

- `Void setForeground(Color c)`

  Переопределяет метод `JComponent.setForeground`.

- `protected void setValue(Object value)`

- `void updateUI()`

  Сообщает, что объект внешнего вида `Look&Feel` был изменен.

Класс *javax.swing.tree.DefaultTreeCellRenderer*

Имплементирует интерфейс `TreeCellRenderer`.

Отображает элемент дерева.

Значения.

- `protected Color backgroundNonSelectionColor`

  Цвет фона, когда узел не выбран.

- `protected Color backgroundSelectionColor`

  Цвет фона, когда узел выбран.

- `protected Color borderSelectionColor`

  Цвет границы, когда узел выбран.

- `protected Icon closedIcon`

  Пиктограмма для показа неразвернутых неконечных узлов дерева.

- `protected Icon leafIcon`

  Пиктограмма для показа конечных узлов дерева (листьев).

- `protected Icon openIcon`

  Пиктограмма для отображения развернутых неконечных узлов дерева.

- `protected Boolean selected`

  Выбран ли узел.

- `protected Color textNonSelectionColor`

  Цвет невыбранного узла.

- `protected Color textSelectionColor`

  Цвет выбранного узла.

Методы класса.

- `Color getBackgroundNonSelectionColor()`

  Возвращает цвет фона невыбранного узла.

- `Color getBackgroundSelectionColor()`

  Возвращает цвет фона выбранного узла.

- `Color getBorderSelectionColor()`

  Возвращает цвет границы.

- `Icon getClosedIcon()`

  Возвращает пиктограмму, используемую для отображения неконечного неразвернутого узла.

- `Icon getDefaultClosedIcon()`

  Пиктограмма для неразвернутого неконечного узла, используемая по умолчанию.

- `Icon getDefaultLeafIcon()`

  Пиктограмма для отображения конечного узла (листа), используемая по умолчанию.

- `Icon getDefaultOpenIcon()`

  Пиктограмма, используемая по умолчанию для отображения неразвернутого узла.

- `Icon getLeafIcon()`

  Пиктограмма, используемая для отображения конечного узла (листа).

- `Icon getOpenIcon()`

  Пиктограмма, используемая для отображения развернутого узла.

- `Dimension getPreferredSize()`

  Переопределяет метод `JComponent.getPreferredSize`.

- `Color getTextNonSelectionColor()`

  Цвет текста при невыбранном узле.

- `Color getTextSelectionColor()`

  Цвет текста при выбранном узле.

- `Component getTreeCellRendererComponent(JTree tree, Object value, boolean sel, boolean expanded, boolean leaf, int row, boolean hasFocus)`

  Задает конфигурацию компонента.

- `Void paint(Graphics g)`

  Отображает объект.

- `Void setBackground(Color color)`

  Задает цвет фона.
- `Void setBackgroundNonSelectionColor(Color newColor)`

  Цвет фона для невыбранных узлов.
- `Void setBackgroundSelectionColor(Color newColor)`

  Цвет фона для выбранных узлов.
- `Void setBorderSelectionColor(Color newColor)`

  Цвет границы.
- `Void setClosedIcon(Icon newIcon)`

  Пиктограмма для неразвернутых узлов.
- `Void setFont(Font font)`

  Устанавливает шрифт.
- `Void setLeafIcon(Icon newIcon)`

  Пиктограмма для конечного узла (листа).
- `void setOpenIcon(Icon newIcon)`

  Пиктограмма для развернутого узла.
- `Void setTextNonSelectionColor(Color newColor)`

  Цвет шрифта для выбранного узла.
- `Void setTextSelectionColor(Color newColor)`

  Цвет шрифта для выбранного узла.

Класс *javax.swing.JlayeredPane*

Имплементирует интерфейс `Accessible`.

Используется для создания многослойных панелей.

Методы класса.

- `protected void addImpl(Component comp, Object constraints, int index)`

  Вставляет компонент в контейнер по указанному индексу.
- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext` для компонента `Jcomponent`.
- `int getComponentCountInLayer(int layer)`

  Возвращает число дочерних объектов в слое.
- `Component[] getComponentsInLayer(int layer)`

  Возвращает массив компонентов слоя.

- `Int getIndexOf(Component c)`

  Возвращает индекс компонента.

- `Int getLayer(Component c)`

  Возвращает атрибут слоя для указанного компонента.

- `static int getLayer(JComponent c)`

  Получает свойство слоя для компонента `JComponent`.

- `static JLayeredPane getLayeredPaneAbove(Component c)`

  Возвращает первый компонент типа `JlayeredPane`, содержащий указанный компонент.

- `protected  Integer getObjectForLayer(int layer)`

  Возвращает объект типа `Integer`, связанный с указанным слоем.

- `Int getPosition(Component c)`

  Получает относительное положение компонента в слое.

- `Int highestLayer()`

  Возвращает максимальное значение слоя для всех текущих дочерних объектов.

- `protected  int insertIndexForLayer(int layer, int position)`

  Определяет положение для вставки нового дочернего объекта на основе информации о слое и о требуемом положении вставляемого объекта.

- `Boolean isOptimizedDrawingEnabled()`

  Возвращает `false`, если компоненты могут покрывать друг друга, т. е. когда оптимизации рисования невозможна.

- `Int lowestLayer()`

  Возвращает самый нижний слой из всех дочерних объектов.

- `Void moveToBack(Component c)`

  Перемещает компонент вниз текущего слоя.

- `Void moveToFront(Component c)`

  Перемещает компонент вверх на текущем слое.

- `Void paint(Graphics g)`

  Рисует панель `JLayeredPane` в заданном графическом контексте.

- `protected  String paramString()`

  Возвращает строковое представление панели `JLayeredPane`.

- `static void putLayer(JComponent c, int layer)`

  Задает свойство (номер) слоя для компонента `JComponent`.

- `Void remove(int index)`

  Удаляет из слоя компонент по индексу.

- `Void setLayer(Component c, int layer)`

  Задает атрибут слоя для указанного компонента, делая его самым верхним в слое.

- `Void setLayer(Component c, int layer, int position)`

  Задает атрибут слоя для указанного компонента, а также его положение в слое.

- `Void setPosition(Component c, int position)`

  Перемещает компонент в позицию `position`, причем 0 соответствует самая верхняя позиция, а `-1` — самая нижняя позиция.

Класс *javax.swing.JdesktopPane*

Имплементирует интерфейс `Accessible`.

Методы класса.

- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext`, связанный с компонентом `Jcomponent`.

- `JInternalFrame[] getAllFrames()`

  Возвращает все фреймы `JinternalFrames`, отображенные на экране.

- `JInternalFrame[] getAllFramesInLayer(int layer)`

  Возвращает все фреймы `JinternalFrames`, отображенные на экране в указанном слое.

- `DesktopManager getDesktopManager()`

  Возвращает объект `DesktopManger`.

- `DesktopPaneUI getUI()`

  Возвращает объект `Look&Feel`.

- `String getUIClassID()`

  Возвращает имя объекта `Look&Feel`.

- `Boolean isOpaque()`

  Возвращает `true`, если компонент прорисовывает все свои пикселы.

- `protected  String paramString()`

  Возвращает строковое представление объекта `JDesktopPane`.

- `Void setDesktopManager(DesktopManager d)`

  Задает объект `DesktopManger`.

- `Void setUI(DesktopPaneUI ui)`

  Задает объект `Look&Feel`.

- `Void updateUI()`

  Сообщение о том, что объект `Look&Feel` был изменен.

Класс *javax.swing.JList*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.Scrollable`.

Список для выбора.

Конструкторы класса.

- `JList()`

  `JList` с пустой моделью.

- `JList(ListModel dataModel)`

  `JList` с заданной моделью.

- `JList(Object[] listData)`

  `JList` с элементами, указанными в массиве.

- `JList(Vector listData)`

  `JList` с элементами, указанными в векторе.

Методы класса.

- `void addListSelectionListener(ListSelectionListener listener)`

  Вставка прослушивателя событий.

- `void addSelectionInterval(int anchor, int lead)`

  Выбор элементов.

- `void clearSelection()`

  Очистка выбора после `isSelectionEmpty()` (если вернулось `true`).

- `protected  ListSelectionModel createSelectionModel()`

  Возвращает `DefaultListSelectionModel`.

- `void ensureIndexIsVisible(int index)`

  Если `Jlist` отображается с использованием `Jviewport` и поле не полностью видимо, то производится прокрутка.

- `protected  void fireSelectionValueChanged(int firstIndex, int lastIndex, boolean isAdjusting)`

  Уведомляет `ListSelectionListeners` об изменении модели.

- `AccessibleContext getAccessibleContext()`

  Получает `AccessibleContext`, связанный с `Jcomponent`.

- `int getAnchorSelectionIndex()`

  Возвращает первый аргумент использованного метода `addSelectionInterval` или `setSelectionInterval`.

- `Rectangle getCellBounds(int index1, int index2)`

  Возвращает границы для `JList`.

- `ListCellRenderer getCellRenderer()`

  Объект, отображающий элементы списка.

- `int getFirstVisibleIndex()`

  Индекс ячейки верхнего левого угла или −1.

- `int getFixedCellHeight()`
- `int getFixedCellWidth()`
- `int getLastVisibleIndex()`
- `int getLeadSelectionIndex()`
- `int getMaxSelectionIndex()`
- `int getMinSelectionIndex()`
- `ListModel getModel()`
- `Dimension getPreferredScrollableViewportSize()`
- `Object getPrototypeCellValue()`
- `int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)`
- `boolean getScrollableTracksViewportHeight()`
- `boolean getScrollableTracksViewportWidth()`
- `int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)`
- `int getSelectedIndex()`
- `int[] getSelectedIndices()`
- `Object getSelectedValue()`
- `Object[] getSelectedValues()`
- `Color getSelectionBackground()`
- `Color getSelectionForeground()`
- `int getSelectionMode()`
- `ListSelectionModel getSelectionModel()`
- `ListUI getUI()`
- `String getUIClassID()`
- `boolean getValueIsAdjusting()`

- `int getVisibleRowCount()`
- `Point indexToLocation(int index)`
- `boolean isSelectedIndex(int index)`
- `boolean isSelectionEmpty()`
- `int locationToIndex(Point location)`
- `protected  String paramString()`
- `void removeListSelectionListener(ListSelectionListener listener)`
- `void removeSelectionInterval(int index0, int index1)`
- `void setCellRenderer(ListCellRenderer cellRenderer)`
- `void setFixedCellHeight(int height)`
- `void setFixedCellWidth(int width)`
- `void setListData(Object[] listData)`
- `void setListData(Vector listData)`
- `void setModel(ListModel model)`
- `void setPrototypeCellValue(Object prototypeCellValue)`
- `void setSelectedIndex(int index)`
- `void setSelectedIndices(int[] indices)`
- `void setSelectedValue(Object anObject, boolean shouldScroll)`
- `void setSelectionBackground(Color selectionBackground)`
- `void setSelectionForeground(Color selectionForeground)`
- `void setSelectionInterval(int anchor, int lead)`
- `void setSelectionMode(int selectionMode)`
- `void setSelectionModel(ListSelectionModel selectionModel)`
- `void setUI(ListUI ui)`
- `void setValueIsAdjusting(boolean b)`
- `void setVisibleRowCount(int visibleRowCount)`
- `void updateUI()`

Класс *javax.swing.JMenuBar*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.MenuElement`.

Панель меню.

Методы класса.

- `JMenu add(JMenu c)`

  Вставка меню в конец панели меню.

- `void addNotify()`

  Вместо `JComponent.addNotify` — регистрация панели с использованием текущего `KeyboardManager`.
- `AccessibleContext getAccessibleContext()`
- `Component getComponent()`
- `Component getComponentAtIndex(int i)`

  Компонент по его индексу.
- `int getComponentIndex(Component c)`

  Индекс по компоненту.
- `JMenu getHelpMenu()`

  Меню помощи.
- `Insets getMargin()`

  Отступ между краем и меню.
- `JMenu getMenu(int index)`

  Меню по указанной позиции.
- `int getMenuCount()`

  Количество элементов в панели меню.
- `SingleSelectionModel getSelectionModel()`
- `MenuElement[] getSubElements()`

  Элементы меню в панели меню.
- `MenuBarUI getUI()`

  Текущий интерфейс панели меню.
- `String getUIClassID()`

  Имя класса интерфейса.
- `boolean isBorderPainted()`

  Закрашены ли границы.
- `boolean isManagingFocus()`

  Внутреннее управление событиями фокуса.
- `boolean isSelected()`

  Возвращает `true`, если панель содержит выбранные элементы.
- `void menuSelectionChanged(boolean isIncluded)`

  Имплементация элемента `MenuElement`.
- `protected  void paintBorder(Graphics g)`

  Рисует границу панели меню, если `BorderPainted` равно `true`.

- `protected  String paramString()`

  Строковое представление панели `JMenuBar`.

- `void processKeyEvent(KeyEvent e, MenuElement[] path, MenuSelectionManager manager)`
- `void processMouseEvent(MouseEvent event, MenuElement[] path, MenuSelectionManager manager)`
- `void removeNotify()`
- `void setBorderPainted(boolean b)`

  Закрашивать ли границы.

- `void setHelpMenu(JMenu menu)`

  Задает меню помощи, появляющееся при нажатии кнопки **Help**.

- `void setMargin(Insets m)`

  Зазор между краем панели меню и элементами меню.

- `void setSelected(Component sel)`

  Задает выбранный компонент.

- `void setSelectionModel(SingleSelectionModel model)`
- `void setUI(MenuBarUI ui)`
- `void updateUI()`

Класс *javax.swing.plaf.basic.BasicInternalFrameTitlePane*

Методы класса.

- `Void addNotify()`

  Сообщение о том, что объект обладает родительским компонентом.

- `protected  void addSubComponents()`
- `protected  void addSystemMenuItems(JMenu systemMenu)`
- `protected  void assembleSystemMenu()`
- `protected  void createActions()`
- `protected  void createButtons()`
- `protected  LayoutManager createLayout()`
- `protected  PropertyChangeListener createPropertyChangeListener()`
- `protected  JMenu createSystemMenu()`
- `protected  JMenuBar createSystemMenuBar()`
- `protected  void enableActions()`
- `protected  void installDefaults()`
- `protected  void installListeners()`

- `protected void installTitlePane()`
- `void paintComponent(Graphics g)`

  Вызывает метод прорисовки пользовательского интерфейса.
- `protected void postClosingEvent(JInternalFrame frame)`

  Направляет событие типа `WINDOW_CLOSING` фрейму.
- `Void removeNotify()`

  Сообщение о том, что компонент больше не имеет родительского объекта.
- `protected void setButtonIcons()`
- `protected void showSystemMenu()`
- `protected void uninstallDefaults()`

Класс *javax.swing.JOptionPane*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Диалоговая панель.

Класс *javax.swing.JPanel*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Легковесный контейнер.

Конструкторы класса.

- `JPanel()`

  Создает `JPanel` с поддержкой двойной буферизации.
- `JPanel(boolean isDoubleBuffered)`

  Создает `Jpanel` с указанием стратегии двойной буферизации.
- `JPanel(LayoutManager layout)`

  Создает `JPanel` с заданным менеджером `LayoutManager`.
- `JPanel(LayoutManager layout, boolean isDoubleBuffered)`

  Указаны менеджер и стратегия буферизации.

Методы класса.

- `AccessibleContext getAccessibleContext()`
- `String getUIClassID()`
- `protected String paramString()`
- `void updateUI()`

Класс *javax.swing.colorchooser.AbstractColorChooserPanel*

Абстрактный класс для создания панелей выбора цветов.

Методы класса.

- `protected abstract void buildChooser()`
- `protected Color getColorFromModel()`
- `ColorSelectionModel getColorSelectionModel()`
- `abstract String getDisplayName()`
- `abstract Icon getLargeDisplayIcon()`
- `abstract Icon getSmallDisplayIcon()`
- `void installChooserPanel(JColorChooser enclosingChooser)`

  Вызывается вставкой панели в объект `JColorChooser`.
- `Void paint(Graphics g)`

  Рисует компонент.
- `Void uninstallChooserPanel(JColorChooser enclosingChooser)`

  Удаляет панель из объекта `JColorChooser`.
- `abstract void updateChooser()`

  Метод вызывается автоматически при изменении состояния модели.

Класс *javax.swing.JPopupMenu*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.MenuElement`.

Выпадающее меню.

Конструкторы класса.

- `JPopupMenu()`
- `JPopupMenu(String label)`

  Указан заголовок.

Методы класса.

- `JMenuItem add(Action a)`

  Вставка элемента меню в конец с указанием объекта `Action`.
- `JMenuItem add(JMenuItem menuItem)`

  Вставка элемента в конец меню.
- `JMenuItem add(String s)`

  Создание элемента меню с указанным текстом, вставка его в конец меню.

- `void addPopupMenuListener(PopupMenuListener l)`

  Вставка параметра прослушивателя.
- `void addSeparator()`
- `protected PropertyChangeListener createActionChangeListener(JMenuItem b)`
- `protected  void firePopupMenuCanceled()`
- `protected  void firePopupMenuWillBecomeInvisible()`
- `protected  void firePopupMenuWillBecomeVisible()`
- `AccessibleContext getAccessibleContext()`
- `Component getComponent()`
- `Component getComponentAtIndex(int i)`
- `int getComponentIndex(Component c)`
- `static boolean getDefaultLightWeightPopupEnabled()`
- `Component getInvoker()`
- `String getLabel()`
- `Insets getMargin()`
- `SingleSelectionModel getSelectionModel()`
- `MenuElement[] getSubElements()`
- `PopupMenuUI getUI()`
- `String getUIClassID()`
- `void insert(Action a, int index)`
- `void insert(Component component, int index)`
- `boolean isBorderPainted()`
- `boolean isLightWeightPopupEnabled()`
- `boolean isVisible()`
- `void menuSelectionChanged(boolean isIncluded)`
- `void pack()`
- `protected  void paintBorder(Graphics g)`
- `protected  String paramString()`
- `void processKeyEvent(KeyEvent e, MenuElement[] path,`
  `MenuSelectionManager manager)`
- `void processMouseEvent(MouseEvent event, MenuElement[] path,`
  `MenuSelectionManager manager)`
- `void remove(Component comp)`
- `void remove(int pos)`
- `void removePopupMenuListener(PopupMenuListener l)`

- `void setBorderPainted(boolean b)`
- `static void setDefaultLightWeightPopupEnabled(boolean aFlag)`
- `void setInvoker(Component invoker)`
- `void setLabel(String label)`
- `void setLightWeightPopupEnabled(boolean aFlag)`
- `void setLocation(int x, int y)`
- `void setPopupSize(Dimension d)`
- `void setPopupSize(int width, int height)`
- `void setSelected(Component sel)`
- `void setSelectionModel(SingleSelectionModel model)`
- `void setUI(PopupMenuUI ui)`
- `void setVisible(boolean b)`
- `void show(Component invoker, int x, int y)`
- `void updateUI()`

Класс *javax.swing.plaf.basic.BasicComboPopup*

Имплементирует интерфейс `javax.swing.plaf.basic.ComboPopup`.

Методы класса.

- `protected void autoScrollDown()`
- `protected void autoScrollUp()`
- `protected Rectangle computePopupBounds(int px, int py, int pw, int ph)`
- `protected void configureList()`

  Используется для конфигурирования списка, созданного при помощи метода `createList()`.
- `protected void configurePopup()`

  Используется для конфигурирования объекта `JPopupMenu` (т. е. объекта `BasicComboPopup`).
- `protected void configureScroller()`

  Используется для конфигурирования объекта `JscrollPane`, созданного при помощи метода `createScroller()`.
- `protected MouseEvent convertMouseEvent(MouseEvent e)`
- `protected ItemListener createItemListener()`

  Создает прослушиватель событий, следящих за изменениями в выбранном объекте `JComboBox`.

- `protected KeyListener createKeyListener()`

  Создает прослушиватель клавиш, который возвращается методом `ComboPopup.getKeyListener()`.

- `protected JList createList()`

  Создает объект `Jlist`, который используется для отображения элементов модели.

- `protected ListDataListener createListDataListener()`

  Создает прослушиватель для определения вставки и удалений элементов.

- `protected MouseListener createListMouseListener()`

  Создает прослушиватель событий мыши.

- `protected MouseMotionListener createListMouseMotionListener()`

  Создает прослушиватель событий движения мыши.

- `protected ListSelectionListener createListSelectionListener()`

  Создает прослушиватель событий выбора списка.

- `protected MouseListener createMouseListener()`

  Создает прослушиватель событий мыши, который возвращается методом `ComboPopup.getMouseListener()`.

- `protected MouseMotionListener createMouseMotionListener()`

  Создает прослушиватель движений мыши, который возвращается методом `ComboPopup.getMouseMotionListener()`.

- `protected PropertyChangeListener createPropertyChangeListener()`

  Создает прослушиватель изменений свойств компонента `JComboBox`.

- `protected JScrollPane createScroller()`

  Создает объект `JscrollPane`, содержащий список.

- `protected void delegateFocus(MouseEvent e)`

  Метод для определения того, куда устанавливать фокус при возникновении всплывающего окна.

- `KeyListener getKeyListener()`

  Имплементация метода `ComboPopup.getKeyListener()`.

- `JList getList()`

  Имплементация метода `ComboPopup.getList()`.

- `MouseListener getMouseListener()`

  Имплементация метода `ComboPopup.getMouseListener()`.

- `MouseMotionListener getMouseMotionListener()`

  Имплементация метода `ComboPopup.getMouseMotionListener()`.

- `protected int getPopupHeightForRowCount(int maxRowCount)`
- `void hide()`

  Имплементация метода `ComboPopup.hide()`.
- `protected void installComboBoxListeners()`

  Вставка прослушивателя в объект `JComboBox`.
- `protected void installComboBoxModelListeners(ComboBoxModel model)`
- `protected void installKeyboardActions()`
- `protected void installListListeners()`

  Вставка прослушивателя, вызывается методом `configureList()`.
- `Boolean isFocusTraversable()`
- `Void show()`

  Имплементация метода `ComboPopup.show()`.
- `protected void startAutoScrolling(int direction)`

  Метод для обработки автоматического скроллирования.
- `protected void stopAutoScrolling()`
- `protected void togglePopup()`

  Делает всплывающее меню видимым (если было невидимо) и невидимым (если было видимо).
- `protected void uninstallComboBoxModelListeners(ComboBoxModel model)`
- `void uninstallingUI()`

  Вызывается при удалении пользовательского интерфейса.
- `protected void uninstallKeyboardActions()`
- `protected void updateListBoxSelectionForEvent(MouseEvent anEvent, boolean shouldScroll)`

  Метод используется прослушивателями событий.

Класс *javax.swing.plaf.metal.MetalComboBoxUI.MetalComboPopup*

Всплывающее меню для `MetalComboBox`.

Методы класса.

- `Void delegateFocus(MouseEvent e)`

  Метод для определения того, куда направить фокус при появлении всплывающего меню (используется в обработчиках событий).

Класс *javax.swing.JProgressBar*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.SwingConstants`.

Полоса прогресса процесса.

Конструкторы класса.

- `JProgressBar()`

  Горизонтальная полоса прогресса.

- `JProgressBar(BoundedRangeModel newModel)`

  Создает горизонтальную полосу прогресса.

- `JProgressBar(int orient)`

  Полоса прогресса с указанной ориентацией, либо `JProgressBar.VERTICAL`, либо `JProgressBar.HORIZONTAL`.

- `JProgressBar(int min, int max)`

  Горизонтальная полоса.

- `JProgressBar(int orient, int min, int max)`

  Задана ориентация и минимум с максимумом.

Методы класса.

- `void addChangeListener(ChangeListener l)`

  Вставка прослушивателя событий `ChangeListener`.

- `protected  ChangeListener createChangeListener()`

- `protected  void fireStateChanged()`

  Вызывается событие.

- `AccessibleContext getAccessibleContext()`

  Контекст `AccessibleContext`.

- `int getMaximum()`

  Максимальное значение модели.

- `int getMinimum()`

  Минимальное значение модели.

- `BoundedRangeModel getModel()`

- `int getOrientation()`

  Ориентация полосы.

- `double getPercentComplete()`

  Процент завершенности.

- `String getString()`

  Текущее значение строки.

- `ProgressBarUI getUI()`
- `String getUIClassID()`
- `int getValue()`

  Текущее значение модели.
- `boolean isBorderPainted()`

  Имеет ли полоса границу.
- `boolean isStringPainted()`

  Выводит ли полоса строку.
- `protected  void paintBorder(Graphics g)`

  Рисует границу полосы, если `BorderPainted` — `true`.
- `protected  String paramString()`
- `void removeChangeListener(ChangeListener l)`

  Удаляет прослушиватель событий.
- `void setBorderPainted(boolean b)`

  Прорисовывает ли полоса свои границы.
- `void setMaximum(int n)`

  Максимум.
- `void setMinimum(int n)`

  Минимум.
- `void setModel(BoundedRangeModel newModel)`

  Задает модель.
- `void setOrientation(int newOrientation)`

  Ориентация полосы: `JProgressBar.VERTICAL` или `JProgressBar.HORIZONTAL`.
- `void setString(String s)`

  Значение строки.
- `void setStringPainted(boolean b)`

  Выводится ли строка.
- `void setUI(ProgressBarUI ui)`
- `void setValue(int n)`

  Текущее значение.
- `void updateUI()`

Класс *javax.swing.JRootPane*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Контейнер окон, фреймов, панелей.

Класс *javax.swing.JScrollBar*

Имплементирует интерфейс `javax.accessibility.Accessible`, `java.awt.Adjustable`.

Полоса прокрутки.

Конструкторы класса.

- `JScrollBar()`

  Создает вертикальную полосу прокрутки.

- `JScrollBar(int orientation)`

  Создает вертикальную полосу прокрутки с заданной ориентацией.

- `JScrollBar(int orientation, int value, int extent, int min, int max)`

  Создает вертикальную полосу прокрутки с заданной ориентацией, максимумом и минимумом.

Методы класса.

- `void addAdjustmentListener(AdjustmentListener l)`

  Вставляет прослушиватель `AdjustmentListener`.

- `protected void fireAdjustmentValueChanged(int id, int type, int value)`

- `AccessibleContext getAccessibleContext()`

  Получает `AccessibleContext`.

- `int getBlockIncrement()`

- `int getBlockIncrement(int direction)`

  Возвращает величину изменения для полосы прокрутки.

- `int getMaximum()`

  Максимальная единица для полосы прокрутки — `максимум` для `extent`.

- `Dimension getMaximumSize()`

- `int getMinimum()`

  Минимальное значение (как правило 0).

- `Dimension getMinimumSize()`

- `BoundedRangeModel getModel()`

  Возвращает модель (четыре свойства `minimum`, `maximum`, `value`, `extent`).

- `int getOrientation()`

  Ориентация класса.
- `ScrollBarUI getUI()`
- `String getUIClassID()`

  Имя класса `LookAndFeel` для данного компонента.
- `int getUnitIncrement()`
- `int getUnitIncrement(int direction)`

  Величина изменения при прокрутке.
- `int getValue()`

  Значение полосы прокрутки.
- `boolean getValueIsAdjusting()`

  Перетащен ли ползунок.
- `int getVisibleAmount()`

  Видимая часть полосы прокрутки.
- `protected  String paramString()`

  Строка представления полосы прокрутки.
- `void removeAdjustmentListener(AdjustmentListener l)`

  Удаляет прослушиватель событий `AdjustmentEventlistener`.
- `void setBlockIncrement(int blockIncrement)`

  Задает свойство — величина изменения при прокрутке.
- `void setEnabled(boolean x)`

  Компонент может менять положение ползунка прокрутки.
- `void setMaximum(int maximum)`

  Максимальное значение модели.
- `void setMinimum(int minimum)`

  Минимальное значение модели.
- `void setModel(BoundedRangeModel newModel)`

  Задает модель (четыре свойства: `minimum`, `maximum`, `value`, `extent`).
- `void setOrientation(int orientation)`

  Задает ориентацию панели прокрутки `VERTICAL` или `HORIZONTAL`.
- `void setUnitIncrement(int unitIncrement)`

  Устанавливает свойство `unitIncrement`.
- `void setValue(int value)`

  Значение панели прокрутки.

- `void setValueIsAdjusting(boolean b)`

  Устанавливает значение свойства `valueIsAdjusting`.

- `void setValues(int newValue, int newExtent, int newMin, int newMax)`

  Задает четыре свойства `BoundedRangeModel`.

- `void setVisibleAmount(int extent)`

  Задает свойства видимости полосы.

- `void updateUI()`

Класс *javax.swing.JScrollPane.ScrollBar*

Имплементирует интерфейсы `javax.swing.plaf.UIResource`.

Класс *javax.swing.JScrollPane*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.ScrollPaneConstants`.

Панель с прокруткой.

Конструкторы класса.

- `JScrollPane()`

  Создает пустую панель, горизонтальная и вертикальная полосы прокрутки появятся при необходимости.

- `JScrollPane(Component view)`

  Создает панель, которая отображает содержимое заданного компонента. Полосы прокрутки появляются при необходимости, если размеры панели оказываются малы для отображения того, что на ней расположено.

- `JScrollPane(Component view, int vsbPolicy, int hsbPolicy)`

  Создает панель с загруженным видом по указанным политикам прокрутки по вертикали и горизонтали.

- `JScrollPane(int vsbPolicy, int hsbPolicy)`

  Создает пустую панель с заданными политиками прокрутки по вертикали и по горизонтали.

Методы класса.

- `JScrollBar createHorizontalScrollBar()`

  Используется в `ScrollPaneUI` для создания горизонтальной прокрутки.

- `JScrollBar createVerticalScrollBar()`

  Используется в `ScrollPaneUI` для создания вертикальной прокрутки.

- `protected  JViewport createViewport()`

  Возвращает `JViewport`.
- `AccessibleContext getAccessibleContext()`

  Получает `AccessibleContext`.
- `JViewport getColumnHeader()`

  Возвращает заголовок колонки.
- `Component getCorner(String key)`

  Возвращает компонент.
- `JScrollBar getHorizontalScrollBar()`

  Возвращает горизонтальную полосу прокрутки.
- `int getHorizontalScrollBarPolicy()`

  Возвращает значение политики горизонтальной полосы прокрутки.
- `JViewport getRowHeader()`

  Возвращает заголовок строки.
- `ScrollPaneUI getUI()`

  Возвращает объект `Look and Feel` для компонента.
- `String getUIClassID()`
- `JScrollBar getVerticalScrollBar()`

  Возвращает вертикальную полосу прокрутки.
- `int getVerticalScrollBarPolicy()`

  Возвращает политику вертикальной полосы прокрутки.
- `JViewport getViewport()`

  Возвращает `JViewport`.
- `Border getViewportBorder()`

  Возвращает значение свойства `viewportBorder`.
- `Rectangle getViewportBorderBounds()`

  Границы `viewportBorder`.
- `boolean isOpaque()`

  `True`, если компонент прорисовывает все пикселы.
- `boolean isValidateRoot()`

  Вызов `revalidate()` всех дочерних компонентов для `JScrollPane`.
- `protected  String paramString()`

  Строка, представляющая `JScrollPane`.
- `void setColumnHeader(JViewport columnHeader)`

  Удаление старого `columnHeader`.

- `void setColumnHeaderView(Component view)`

  Создает вид заголовка столбца, вставляет его в `viewport`.
- `void setCorner(String key, Component corner)`

  Создает дочерний элемент и, если есть место, показывает его.
- `void setHorizontalScrollBar(JScrollBar horizontalScrollBar)`

  Вставляет полосу прокрутки по горизонтали.
- `void setHorizontalScrollBarPolicy(int policy)`

  Задает появление горизонтальной полосы прокрутки на панели.
- `void setLayout(LayoutManager layout)`

  Задает менеджер вида для панели.
- `void setRowHeader(JViewport rowHeader)`

  Удаляет старый заголовок строки.
- `void setRowHeaderView(Component view)`

  Создает вид с заголовком строк, вставляет в панель.
- `void setUI(ScrollPaneUI ui)`

  Создает объект `ScrollPaneUI`.
- `void setVerticalScrollBar(JScrollBar verticalScrollBar)`

  Вставляет горизонтальную полосу прокрутки.
- `void setVerticalScrollBarPolicy(int policy)`

  Когда появляется вертикальная полоса прокрутки? Определяет политику.
- `void setViewport(JViewport viewport)`

  Удаляет старый, задает новый вид.
- `void setViewportBorder(Border viewportBorder)`

  Устанавливает границу вокруг вида.
- `void setViewportView(Component view)`

  Создает вид.
- `void updateUI()`

Класс *javax.swing.JSeparator*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.SwingConstants`.

Разделитель групп в меню.

Класс *javax.swing.JPopupMenu.Separator*

Разделитель для всплывающего меню.

Методы класса.

- `String getUIClassID()`

  Возвращает имя класса `Look&Feel`.

Класс *javax.swing.JToolBar.Separator*

Разделитель для панели меню.

Конструкторы класса.

- `JToolBar.Separator()`

  Создает разделитель размером, заданным по умолчанию для данного класса `Look&Feel`.

- `JToolBar.Separator(Dimension size)`

  Создает новый разделитель с указанным размером.

Методы класса.

- `Dimension getMaximumSize()`

  Возвращает максимальный размер разделителя.

- `Dimension getMinimumSize()`

  Возвращает минимальный размер разделителя.

- `Dimension getPreferredSize()`

  Возвращает предпочтительный размер разделителя.

- `Dimension getSeparatorSize()`

  Возвращает размер разделителя.

- `String getUIClassID()`

  Возвращает имя класса `Look&Feel`.

- `Void setSeparatorSize(Dimension size)`

  Задает размер разделителя.

Класс *javax.swing.JSlider*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.SwingConstants`.

Графический ползунок выбора элемента.

Класс *javax.swing.JSplitPane*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Панель для разделения двух компонентов.

Класс *javax.swing.JTabbedPane*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `java.io.Serializable`, `javax.swing.SwingConstants`.

Компонент перехода от группы к группе с использованием клавиши табуляции.

Класс *javax.swing.JTable*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.event.CellEditorListener`, `javax.swing.event.ListSelectionListener`, `javax.swing.Scrollable`, `javax.swing.event.TableColumnModelListener`, `javax.swing.event.TableModelListener`.

Двумерная таблица.

Константы класса.

- `static int AUTO_RESIZE_ALL_COLUMNS`

  При изменении размеров изменяет размер столбцов. Пропорционально для всех столбцов.

- `static int AUTO_RESIZE_LAST_COLUMN`

  Изменение размеров столбцов индивидуально для каждого столбца.

- `static int AUTO_RESIZE_NEXT_COLUMN`

  Последующий столбец изменяется противоположно предыдущему.

- `static int AUTO_RESIZE_OFF`

  Столбцы не меняют размер автоматически.

- `static int AUTO_RESIZE_SUBSEQUENT_COLUMNS`

  Столбцы форматируются с сохранением общей ширины.

- `protected boolean autoCreateColumnsFromModel`

  Для задания параметров столбцов таблица запрашивает `TableModel`.

- `protected int autoResizeMode`

  Автоматическое изменение размеров.

- `protected TableCellEditor cellEditor`

- `protected boolean cellSelectionEnabled`

- `protected TableColumnModel columnModel`

  Модель столбцов таблицы `TableColumnModel`.

- `protected TableModel dataModel`

  Модель таблицы `TableModel`.

- `protected Hashtable defaultEditorsByColumnClass`
- `protected Hashtable defaultRenderersByColumnClass`
- `protected int editingColumn`
- `protected int editingRow`
- `protected Component editorComp`
- `protected Color gridColor`
- `protected Dimension preferredViewportSize`
- `protected int rowHeight`
- `protected int rowMargin`
- `protected boolean rowSelectionAllowed`
- `protected Color selectionBackground`
- `protected Color selectionForeground`
- `protected ListSelectionModel selectionModel`
- `protected boolean showHorizontalLines`
- `protected boolean showVerticalLines`
- `protected JTableHeader tableHeader`

Конструкторы класса.

- `JTable()`

  Создает таблицу `JTable`, инициализирует ее моделями по умолчанию.

- `JTable(int numRows, int numColumns)`

  Создает таблицу с указанным числом рядов и столбцов.

- `JTable(Object[][] rowData, Object[] columnNames)`

  Создает таблицу по двум массивам, содержащим данные и названия столбцов.

- `JTable(TableModel dm)`

  Создает таблицу по указанной модели данных.

- `JTable(TableModel dm, TableColumnModel cm)`

  Создает таблицу по указанным моделям данных и моделям столбцов.

- ```
  JTable(TableModel dm, TableColumnModel cm,
         ListSelectionModel sm)
  ```

  Создает таблицу `JTable` по указанным моделям данных, столбцов и модели выбора.
  ```

- `JTable(Vector rowData, Vector columnNames)`

  Создает таблицу, отображающую значения `rowData` со столбцами, имена которых хранятся в `columnNames`.

Методы класса.

- `void addColumn(TableColumn aColumn)`

  Вставляет столбец `aColumn` в конец массива столбцов в модели столбцов.

- `void addColumnSelectionInterval(int index0, int index1)`

  Вставляет столбцы с положения `index1` до `index1`.

- `void addNotify()`

  Вызывает `configureEnclosingScrollPane`.

- `void addRowSelectionInterval(int index0, int index1)`

  Вставляет ряды.

- `void clearSelection()`

  Убирает выбранные столбцы и ряды.

- `void columnAdded(TableColumnModelEvent e)`

  Сообщает прослушивателю о создании столбца, т. е. о вставке столбца в модель.

- `int columnAtPoint(Point point)`

  Возвращает индекс столбца.

- `void columnMarginChanged(ChangeEvent e)`

  Сообщает о событии удаления столбца в связи с изменением границ.

- `void columnMoved(TableColumnModelEvent e)`

  Сообщает об изменении положения столбца.

- `void columnRemoved(TableColumnModelEvent e)`

  Сообщает о событии удаления столбца из модели.

- `void columnSelectionChanged(ListSelectionEvent e)`

  Сообщает об изменении модели `TableColumnModel`.

- `protected  void configureEnclosingScrollPane()`

- `int convertColumnIndexToModel(int viewColumnIndex)`

  Возвращает индекс столбца в модели.

- `int convertColumnIndexToView(int modelColumnIndex)`

  Возвращает индекс столбца в виде.

- `protected  TableColumnModel createDefaultColumnModel()`

  Возвращает модель столбцов по умолчанию `DefaultTableColumnModel`.

- `void createDefaultColumnsFromModel()`

  Создает столбцы по умолчанию из модели данных с использованием методов `getColumnCount()` и `getColumnClass()` (интерфейс `TableModel`).

- `protected TableModel createDefaultDataModel()`

  Возвращает объект модели таблицы `DefaultTableModel`.

- `protected void createDefaultEditors()`

  Создает редактор ячейки для работы со значениями типов `Objects`, числовых и логических типов.

- `protected void createDefaultRenderers()`

- `protected ListSelectionModel createDefaultSelectionModel()`

  Возвращает модель выбора по умолчанию `DefaultListSelectionModel`.

- `protected JTableHeader createDefaultTableHeader()`

  Возвращает заголовок таблицы по умолчанию `JTableHeader`.

- `static JScrollPane createScrollPaneForTable(JTable aTable)`

- `boolean editCellAt(int row, int column)`

  Начинает редактирование ячейки в указанной строке и заданном столбце.

- `boolean editCellAt(int row, int column, EventObject e)`

  Редактирование ячейки в указанном столбце и строке, если ячейка может быть редактируема.

- `void editingCanceled(ChangeEvent e)`

  Вызывается при отмене редактирования (программными средствами).

- `void editingStopped(ChangeEvent e)`

  Вызывается при завершении редактирования (программными средствами).

- `AccessibleContext getAccessibleContext()`

  Получает контекст `AccessibleContext`.

- `boolean getAutoCreateColumnsFromModel()`

  Создаются столбцы по умолчанию в соответствии с моделью.

- `int getAutoResizeMode()`

- `TableCellEditor getCellEditor()`

  Редактор ячейки.

- `TableCellEditor getCellEditor(int row, int column)`

- `Rectangle getCellRect(int row, int column,`
  `boolean includeSpacing)`

- `TableCellRenderer getCellRenderer(int row, int column)`
- `boolean getCellSelectionEnabled()`

  Истина, если допускается одновременный выбор ряда и столбца.
- `TableColumn getColumn(Object identifier)`

  Возвращает объект столбца `TableColumn` по идентификатору.
- `Class getColumnClass(int column)`

  Возвращает тип столбца.
- `int getColumnCount()`

  Возвращает число столбцов в модели столбцов, это число может отличаться от числа столбцов в модели таблицы.
- `TableColumnModel getColumnModel()`

  Возвращает модель `TableColumnModel`, содержащую всю информацию о столбцах таблицы.
- `String getColumnName(int column)`

  Возвращает имя столбца.
- `boolean getColumnSelectionAllowed()`

  Возвращает истину, если столбец может быть выбран.
- `TableCellEditor getDefaultEditor(Class columnClass)`

  Возвращает редактор, используемый в `TableColumn`.
- `TableCellRenderer getDefaultRenderer(Class columnClass)`
- `int getEditingColumn()`

  Индекс редактируемого столбца.
- `int getEditingRow()`

  Индекс редактируемого ряда.
- `Component getEditorComponent()`
- `Color getGridColor()`

  Цвет для рисования сетки.
- `Dimension getIntercellSpacing()`

  Промежуток между ячейками.
- `TableModel getModel()`
- `Dimension getPreferredScrollableViewportSize()`

  Предпочтительный размер вида.
- `int getRowCount()`

  Возвращает число рядов в таблице.

- `int getRowHeight()`

  Высота таблицы.

- `int getRowMargin()`

  Величина зазора между рядами.

- `boolean getRowSelectionAllowed()`

  Истина, если ряд можно выбрать.

- `int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)`

  Возвращает (в зависимости от ориентации таблицы) `visibleRect.height` или `visibleRect.width`.

- `boolean getScrollableTracksViewportHeight()`

  Возвращает ложь, если высота вида не задает высоту таблицы.

- `boolean getScrollableTracksViewportWidth()`

  Возвращает ложь, если ширина вида не задает высоту таблицы.

- `int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)`

  Интервал прокрутки, требуемый для отображения целого нового ряда или столбца (в зависимости от ориентации).

- `int getSelectedColumn()`

  Возвращает индекс первого выбранного столбца (−1 в случае отсутствия выбора).

- `int getSelectedColumnCount()`

  Количество выбранных столбцов.

- `int[] getSelectedColumns()`

  Возвращает массив индексов всех выбранных столбцов.

- `int getSelectedRow()`

  Возвращает индекс первого выбранного ряда (−1 в случае отсутствия выбора).

- `int getSelectedRowCount()`

  Количество выбранных рядов.

- `int[] getSelectedRows()`

  Индексы всех выбранных рядов.

- `Color getSelectionBackground()`

  Возвращает цвет фона выбранных ячеек.

- `Color getSelectionForeground()`

  Возвращает основной цвет выбранных ячеек.

- `ListSelectionModel getSelectionModel()`

  Возвращает `ListSelectionModel`.

- `boolean getShowHorizontalLines()`

  Истина в случае, если между ячейками рисуется горизонтальная линия.

- `boolean getShowVerticalLines()`

  Истина в случае, если между ячейками рисуется вертикальная линия.

- `JTableHeader getTableHeader()`

  Возвращает `tableHeader`.

- `String getToolTipText(MouseEvent event)`
- `TableUI getUI()`
- `String getUIClassID()`

  Имя класса `Look and Feel` для компонента.

- `Object getValueAt(int row, int column)`

  Возвращает значение ячейки по указанному ряду и столбцу.

- `protected  void initializeLocalVars()`

  Инициализирует свойства таблицы значениями по умолчанию.

- `boolean isCellEditable(int row, int column)`

  Возвращает истину, если ячейка в указанном ряду и столбце может быть редактируемой.

- `boolean isCellSelected(int row, int column)`

  Возвращает истину, если указанная ячейка выбрана.

- `boolean isColumnSelected(int column)`

  Возвращает истину, если указанный столбец (по индексу) выбран.

- `boolean isEditing()`

  Возвращает истину, если происходит редактирование ячейки.

- `boolean isManagingFocus()`
- `boolean isRowSelected(int row)`

  Возвращает истину, если выбран указанный ряд.

- `void moveColumn(int column, int targetColumn)`

  Перемещает столбец в новое положение.

- `protected  String paramString()`

  Возвращает строковое представление данной таблицы `JTable`.

- `Component prepareEditor(TableCellEditor editor,`
  `int row, int column)`

- `Component prepareRenderer(TableCellRenderer renderer, int row, int column)`
- `void removeColumn(TableColumn aColumn)`

  Удаляет столбец `aColumn` из массива столбцов таблицы `JTable`.
- `void removeColumnSelectionInterval(int index0, int index1)`

  Удаляет столбцы с `index0` по `index1` включительно.
- `void removeEditor()`

  Удаляет объект редактора.
- `void removeRowSelectionInterval(int index0, int index1)`

  Снимает выбор с рядов в интервале от `index0` до `index0`.
- `void reshape(int x, int y, int width, int height)`

  Вызывает `super.reshape()`.
- `protected  void resizeAndRepaint()`

  Равносильно `revalidate()` после `repaint()`.
- `int rowAtPoint(Point point)`

  Возвращает индекс ряда.
- `void selectAll()`

  Выделяет все ряды, столбцы и ячейки в таблице.
- `void setAutoCreateColumnsFromModel(boolean createColumns)`
- `void setAutoResizeMode(int mode)`
- `void setCellEditor(TableCellEditor anEditor)`
- `void setCellSelectionEnabled(boolean flag)`

  Разрешен ли одновременный выбор строк и столбцов в таблице.
- `void setColumnModel(TableColumnModel newModel)`
- `void setColumnSelectionAllowed(boolean flag)`

  Разрешен ли выбор столбцов в модели.
- `void setColumnSelectionInterval(int index0, int index1)`

  Выбирает столбцы с `index0` по `index1`.
- `void setDefaultEditor(Class columnClass, TableCellEditor editor)`

  Задает редактор по умолчанию для `TableColumn`.
- `void setDefaultRenderer(Class columnClass, TableCellRenderer renderer)`
- `void setEditingColumn(int aColumn)`

  Задает переменную для редактирования столбца.

- `void setEditingRow(int aRow)`

  Задает переменную для редактирования строки.

- `void setGridColor(Color newColor)`

  Задает цвет для рисования сетки.

- `void setIntercellSpacing(Dimension newSpacing)`

  Задает ширину и высоту отступов между ячейками.

- `void setModel(TableModel newModel)`

  Устанавливает модель таблицы, задавая новое значение `newModel` и регистрирует для прослушивания событий.

- `void setPreferredScrollableViewportSize(Dimension size)`

  Задает предпочтительный размер вида таблицы.

- `void setRowHeight(int newHeight)`

  Задает высоту рядов.

- `void setRowMargin(int rowMargin)`

  Задает пробел между рядами.

- `void setRowSelectionAllowed(boolean flag)`

  Устанавливает, могут ли ряды в модели быть удалены.

- `void setRowSelectionInterval(int index0, int index1)`

  Выбирает ряды с `index0` по `index1`.

- `void setSelectionBackground(Color selectionBackground)`

  Задает цвет фона для выбранных ячеек.

- `void setSelectionForeground(Color selectionForeground)`

  Задает основной цвет для выбранных ячеек.

- `void setSelectionMode(int selectionMode)`

- `void setSelectionModel(ListSelectionModel newModel)`

  Задает модель выбора ячеек.

- `void setShowGrid(boolean b)`

  Рисуется ли сетка вокруг ячейки.

- `void setShowHorizontalLines(boolean b)`

  Устанавливается, будет ли рисоваться горизонтальная линия между ячейками.

- `void setShowVerticalLines(boolean b)`

  Устанавливается, будет ли рисоваться вертикальная линия между ячейками.

- `void setTableHeader(JTableHeader newHeader)`

  Задает новый `tableHeader`.

- `void setUI(TableUI ui)`

  Задает объект `Look and Feel`.

- `void setValueAt(Object aValue, int row, int column)`

  Задает значение для указанной ячейки.

- `void sizeColumnsToFit(boolean lastColumnOnly)`
- `void sizeColumnsToFit(int resizingColumn)`

  Изменяет размер одного или нескольких столбцов таким образом, что общая ширина столбцов таблицы `JTable` будет равна ширине таблицы.

- `void tableChanged(TableModelEvent e)`
- `void updateUI()`

  Сообщение об изменении объекта `Look and Feel`.

- `void valueChanged(ListSelectionEvent e)`

  Вызывается при изменении объекта выбора.

Класс *javax.swing.table.JTableHeader*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.event.TableColumnModelListener`.

Класс *javax.swing.text.JTextComponent*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.Scrollable`.

Текстовый компонент.

Класс *javax.swing.JeditorPane*

Конструкторы класса.

- `JEditorPane()`
  Создает новую панель `JEditorPane`.
- `JditorPane(String url)`

  Создает новую панель `JEditorPane` на основе указанного URL.

- `JEditorPane(String type, String text)`
- Создает новую панель `JEditorPane` с заданным текстом.
- `JEditorPane(URL initialPage)`
  Создает панель `JEditorPane` на основе указанного URL источника.

Методы класса.

- `Void addHyperlinkListener(HyperlinkListener listener)`

  Создает гиперссылку, изменения которой будут отслеживаться (например, изменения, связанные с выбором ссылки или переходом по ней).

- `protected  EditorKit createDefaultEditorKit()`

  Создает компонент `PlainEditorKit`, вызываемый по умолчанию во время первого обращения к панели.

- `static EditorKit reateEditorKitForContentType(String type)`

  Создает метку для заданного типа.

- `void fireHyperlinkUpdate(HyperlinkEvent e)`

  Информирует прослушивателей событий о событиях гиперссылки.

- `AccessibleContext getAccessibleContext()`

  Получает `AccessibleContext`, связанный с панелью `JEditorPane`.

- `String getContentType()`

  Получает тип содержания панели.

- `EditorKit getEditorKit()`

  Получает объект `EditorKit`, используемый для обработки содержания панели.

- `EditorKit getEditorKitForContentType(String type)`

  Получает объект `EditorKit`, используемый для заданного типа содержания панели.

- `URL getPage()`

  Получает текущий URL, отображаемый в панели.

- `Dimension getPreferredSize()`

  Предпочтительный размер панели `JEditorPane` изменен в сравнении с тем, что задан в суперклассе.

- `boolean getScrollableTracksViewportHeight()`

  Возвращает `true`, если объект `Viewport` заставляет объект `Scrollable` изменять высоту в соответствии с высотой объекта `Viewport`.

- `boolean getScrollableTracksViewportWidth()`

  Возвращает `true`, если объект `Viewport` заставляет установить ширину в объекте `Scrollable` в соответствии со своей шириной.

- `protected  InputStream getStream(URL page)`

  Получает поток от заданного URL, который будет загружен с помощью метода `setPage`.

- `String getText()`

  Возвращает текст из текстового компонента `TextComponent`.
- `String getUIClassID()`

  Получает идентификатор класса пользовательского интерфейса.
- `boolean isManagingFocus()`

  Выключает возможность работы с табуляцией после получения фокуса.
- `protected String paramString()`

  Возвращает строку, представляющую панель `JEditorPane`.
- `protected void processComponentKeyEvent(KeyEvent e)`

  Обрабатывает нажатие клавиш табуляции.
- `Void read(InputStream in, Object desc)`

  Метод инициализируется в потоках.
- `static void registerEditorKitForContentType(String type, classname)`

  Задает соответствия имен и типов, используемые по умолчанию.
- `static void registerEditorKitForContentType(String type, classname, ClassLoader loader)`

  Задает соответствия имен и типов, используемые по умолчанию.
- `Void removeHyperlinkListener(HyperlinkListener listener)`

  Удаляет прослушиватель событий гиперссылки.
- `Void replaceSelection(String content)`

  Удаляет выбранное содержание и заменяет новым, указанным в строке.
- `protected void scrollToReference(String reference)`

  Прокручивает вид к заданному положению.
- `void setContentType(String type)`

  Задает тип содержания, с которым работает редактор.
- `Void setEditorKit(EditorKit kit)`

  Задает объект `EditorKit` для работы с содержанием.
- `void setEditorKitForContentType(String type, EditorKit k)`

  Связывает объект `EditorKit` с заданным типом.
- `Void setPage(String url)`

  Задает текущий URL.
- `Void setPage(URL page)`

  Задает текущий URL.

- `Void setText(String t) omponent c)`

  Задает текст компонента `TextComponent`.

Класс *javax.swing.JtextPane*

Конструкторы класса.

- `JTextPane()`

  Новая панель `JTextPane`.

- `JTextPane(StyledDocument doc)`

  Новая панель `JtextPane` с указанной моделью документа.

Методы класса.

- `Style addStyle(String nm, Style parent)`

  Вставляет стиль в иерархию стилей.

- `protected  EditorKit createDefaultEditorKit()`

  Создает объект `EditorKit`, используемый по умолчанию.

- `AttributeSet getCharacterAttributes()`

  Получает атрибуты символа в данной позиции.

- `MutableAttributeSet getInputAttributes()`

  Получает атрибуты ввода для панели.

- `Style getLogicalStyle()`

  Возвращает стиль параграфа для данной позиции.

- `AttributeSet getParagraphAttributes()`

  Возвращает атрибуты параграфа для данной позиции.

- `Boolean getScrollableTracksViewportWidth()`

  Возвращает `true`, если объект `Viewport` заставляет объект `Scrollable` устанавливать ширину в соответствии с шириной объекта `Viewport`.

- `Style getStyle(String nm)`

  Возвращает ранее заданный стиль.

- `StyledDocument getStyledDocument()`

  Возвращает модель редактора.

- `protected  StyledEditorKit getStyledEditorKit()`

  Получает объект `StyledEditorKit`.

- `String getUIClassID()`

  Возвращает идентификатор класса пользовательского интерфейса.

- `Void insertComponent(C`

  Вставляет компонент в документ вместо того, который выбран в текущий момент.

- `Void insertIcon(Icon g)`

  Вставляет пиктограмму в документ взамен той, что выбрана на текущий момент.

- `protected  String paramString()`

  Возвращает строковое представление панели `JTextPane`.

- `Void removeStyle(String nm)`

  Удаляет ранее вставленный стиль по имени.

- `Void replaceSelection(String content)`

  Удаляет выбранное содержимое, заменяя его новым.

- `Void setCharacterAttributes(AttributeSet attr, boolean replace)`

  Задает указанные атрибуты для символьного содержания.

- `Void setDocument(Document doc)`

  Связывает редактор с текстовым документом.

- `Void setEditorKit(EditorKit kit)`

  Связывает установленный объект редактора с содержанием, которое будет обрабатываться.

- `Void setLogicalStyle(Style s)`

  Устанавливает стиль параграфа для выбранного положения.

- `Void setParagraphAttributes(AttributeSet attr, boolean replace)`

  Вставляет указанные атрибуты в применении к параграфу.

- `Void setStyledDocument(StyledDocument doc)`

  Связывает редактор с текстовым документом.

Класс *javax.swing.JtextArea*

Конструкторы класса.

- `JTextArea()`

  Создает новую текстовую панель `TextArea`.

- `JTextArea(Document doc)`

  Создает новый объект `JTextArea` с указанной моделью документа, прочие аргументы такие, как те, что используются по умолчанию (`null`, `0`, `0`).

- `JTextArea(Document doc, String text, int rows, int columns)`

  Создает новый объект `JTextArea` с указанным количеством строк и столбцов по указанной модели.

- `JTextArea(int rows, int columns)`

  Создает новый пустой объект `TextArea` с указанным количеством строк.

- `JTextArea(String text)`

  Создает новый объект `TextArea` с заданным текстом.

- `JTextArea(String text, int rows, int columns)`

  Создает новый объект `TextArea` с заданным текстом и количеством строк и столбцов.

Методы класса.

- `Void append(String str)`

  Вставляет указанный текст в конец документа.

- `protected Document createDefaultModel()`

  Создает модель, которая будет использоваться по умолчанию, если во время создания документа не будет указана другая модель.

- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext`, связанный с панелью `JtextArea`.

- `Int getColumns()`

  Возвращает количество столбцов в объекте `TextArea`.

- `protected int getColumnWidth()`

  Получает ширину столбцов.

- `Int getLineCount()`

  Определяет количество строк в текстовой области.

- `Int getLineEndOffset(int line)`

  Определяет отступ для заданной строки.

- `Int getLineOfOffset(int offset)`

  Определяет отступ.

- `Int getLineStartOffset(int line)`

  Определяет отступ в указанной строке.

- `Boolean getLineWrap()`

  Политика переноса текста.

- `Dimension getPreferredScrollableViewportSize()`

  Предпочтительный размер объекта `Viewport` в панели с прокруткой `JScrollPane`.

- `Dimension getPreferredSize()`

  Возвращает предпочтительную ширину текстовой области `TextArea`.

- `protected  int getRowHeight()`

  Определяет высоту строки.

- `Int getRows()`

  Возвращает количество строк в объекте `TextArea`.

- `Boolean getScrollableTracksViewportWidth()`

  Возвращает `true`, если объект `Viewport` заставляет устанавливать ширину в объекте `Scrollable` в соответствии со своей шириной.

- `Int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)`

  Получает величину, которая используется для того, чтобы компонент смог отобразить целиком целую строку или столбец (в зависимости от ориентации).

- `Int getTabSize()`

  Количество символов, соответствующих табуляции.

- `String getUIClassID()`

  Возвращает идентификатор класса пользовательского интерфейса.

- `Boolean getWrapStyleWord()`

  Получает стиль переноса строк в текстовой области.

- `Void insert(String str, int pos)`

  Вставляет заданный текст в указанную позицию.

- `Boolean isManagingFocus()`

  Выключает клавишу табуляции при получении объектом фокуса.

- `protected  String paramString()`

  Возвращает строковое представление объекта `JTextArea`.

- `protected  void processComponentKeyEvent(KeyEvent e)`

  Обработка событий клавиши табуляции.

- `Void replaceRange(String str, int start, int end)`

  Замена текста с указанным начальной и конечной позициями новым текстом.

- `Void setColumns(int columns)`

  Устанавливает количество столбцов в текстовой области `TextArea`.

- `Void setFont(Font f)`

  Задает текущий шрифт.

- `Void setLineWrap(boolean wrap)`

  Задает политику переноса строки в текстовой области.

- `Void setRows(int rows)`

  Устанавливает количество строк в текстовой области.

- `Void setTabSize(int size)`

  Устанавливает количество символов, соответствующее одной табуляции.

- `Void setWrapStyleWord(boolean word)`

  Задает стиль переноса.

Класс *javax.swing.JtextField*

Имплементирует интерфейс `javax.swing.SwingConstants`.

Конструкторы класса.

- `JTextField()`

  Создает новое текстовое поле `TextField`.

- `JTextField(Document doc, String text, int columns)`

  Создает новое текстовое поле `JTextField` с указанным документом, текстом и количеством столбцов.

- `JTextField(int columns)`

  Создает новое текстовое поле (пустое) `TextField` с указанным числом столбцов.

- `JTextField(String text)`

  Создает новое текстовое поле `TextField` с заданным текстом.

- `JTextField(String text, int columns)`

  Создает новое текстовое поле `TextField` с заданным текстом и количеством столбцов.

- Методы класса.

- `Void addActionListener(ActionListener l)`

  Вставляет прослушиватель событий для событий текстового поля.

- `protected  Document createDefaultModel()`

  Создает модель документа, которая будет использоваться по умолчанию.

- `protected  void fireActionPerformed()`

  Сообщает прослушивателям событий о наступлении события.

- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext`, связанный с полем `JTextField`.

- `Action[] getActions()`

  Получает список команд редактора.

- `Int getColumns()`

  Возвращает количество столбцов в объекте текстового поля `TextField`.

- `protected  int getColumnWidth()`

  Получает ширину столбца.

- `Int getHorizontalAlignment()`

  Возвращает горизонтальное выравнивание текста.

- `BoundedRangeModel getHorizontalVisibility()`

  Получает модель видимости текстового поля.

- `Dimension getPreferredSize()`

  Возвращает предпочтительные размеры объекта текстовой области `TextField`.

- `Int getScrollOffset()`

  Получает отступ при прокрутке.

- `String getUIClassID()`

  Получает идентификатор класса пользовательского интерфейса.

- `Boolean isValidateRoot()`

  Вызывается для задания параметров обновления текстового поля.

- `protected  String paramString()`

  Возвращает строковое представление текстового поля `JTextField`.

- `void postActionEvent()`

  Обрабатывает событие текстового поля, направляя его зарегистрированным объектам прослушивателей `ActionListener`.

- `Void removeActionListener(ActionListener l)`

  Удаляет указанный прослушиватель событий.

- `Void scrollRectToVisible(Rectangle r)`

  Прокручивает поле вправо или влево.

- `Void setActionCommand(String command)`

  Задает строку команды для события.

- `Void setColumns(int columns)`

  Задает количество столбцов в текстовом поле `TextField`.

- `Void setFont(Font f)`

  Задает текущий шрифт.

- `Void setHorizontalAlignment(int alignment)`

  Задает параметр горизонтального выравнивания текста.

- `Void setScrollOffset(int scrollOffset)`

  Задает отступ для прокрутки.

Класс *javax.swing.tree.DefaultTreeCellEditor.DefaultTextField*

Конструктор класса.

- `DefaultTreeCellEditor.DefaultTextField(Border border)`

  Создает объект `DefaultTreeCellEditor.DefaultTextField`.

Методы класса.

- `Border getBorder()`

  Переопределяет метод `JComponent.getBorder`, возвращает текущие границы.

- `Font getFont()`

  Получает шрифт, используемый в компоненте.

- `Dimension getPreferredSize()`

  Переопределяет метод `JTextField.getPreferredSize`. Возвращает предпочтительный размер с учетом типа шрифта.

Класс *javax.swing.JpasswordField*

Конструкторы класса.

- `JPasswordField()`

  Создает новое поля для ввода пароля `JPasswordField` в документе по умолчанию без начального текста и шириной 0 столбцов.

- `JPasswordField(Document doc, String txt, int columns)`

  Создает новое поля для ввода пароля `JPasswordField` с использованием заданного документа с начальным текстом и указанной шириной.

- `JPasswordField(int columns)`

  Создает новое пустое поле для ввода пароля `JPasswordField` с заданным количеством столбцов.

- `JPasswordField(String text)`

  Создает новое поле для ввода пароля `JPasswordField` с указанным текстом.

- `JPasswordField(String text, int columns)`

  Создает новое поле для ввода пароля `JPasswordField` с заданным текстом и числом столбцов.

Методы класса.

- `Void copy()`

  Копирует выбранный фрагмент в системный буфер.

- `Void cut()`

  Копирует выбранный фрагмент в системный буфер и удаляет из документа.

- `Boolean echoCharIsSet()`

  Возвращает `true`, если объект `JPasswordField` имеет заданный набор символов для отображения текста.

- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext` для пароля `JPasswordField`.

- `Char getEchoChar()`

  Возвращает символ.

- `char[] getPassword()`

  Возвращает текст, содержащийся в этом текстовом компоненте.

- `String getText()`

  Устарел. Заменен методом `getPassword()`.

- `String getText(int offs, int len)`

  Устарел. Заменен методом `getPassword()`.

- `String getUIClassID()`

  Возвращает имя класса `Look&Feel`, соответствующее компоненту.

- `protected String paramString()`

  Возвращает строку, представляющую панель для ввода текста `JPasswordField`.

- `Void setEchoChar(char c)`

  Задает символ для панели для ввода текста.

Класс *javax.swing.JtoolBar*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.SwingConstants`

Конструкторы класса.

- `JToolBar()`

  Создает новую панель инструментов.

- `JToolBar(int orientation)`

  Создает панель инструментов с указанной ориентацией.

Методы класса.

- `JButton add(Action a)`

  Вставляет кнопку `JButton` для заданного действия.

- `protected void addImpl(Component comp, constraints, int index)`

  Вставляет компонент в контейнер с заданным индексом.

- `Void addSeparator()`

  Вставляет разделитель в конец панели инструментов.

- `Void addSeparator(Dimension size)`

  Вставляет разделитель в конец панели инструментов с указанным размером.

- `protected PropertyChangeListener(JButton b)`
- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext`.

- `Component getComponentAtIndex(int i)`

  Возвращает компонент по указанному индексу.

- `Int getComponentIndex(Component c)`

  Возвращает индекс по компоненту.

- `Insets getMargin()`

  Возвращает отступ между границей панели инструментов и кнопками, расположенными на панели инструментов.

- `Int getOrientation()`

  Возвращает текущую ориентацию панели инструментов.

- `ToolBarUI getUI()`

  Возвращает интерфейс UI.

- `String getUIClassID()`

  Возвращает имя объекта `Look&Feel`, с помощью которого отображается данный компонент.

- `Boolean isBorderPainted()`

  Проверяет, следует ли отображать границы.

- `Boolean isFloatable()`

  Возвращает `true`, если объект `Toolbar` разрешено перетаскивать.

- `protected void paintBorder(Graphics g)`

  Рисует границы панели инструментов, если свойство `BorderPainted` равно `true`.

- `protected String paramString()`

  Возвращает строковое представление панели `JToolBar`.

- `Void remove(Component comp)`

  Удаляет компонент из панели инструментов.

- `Void setBorderPainted(boolean b)`

  Задает, следует ли рисовать границы панели.

- `Void setFloatable(boolean b)`

  Может ли панель инструментов быть плавающей.
- `Void setMargin(Insets m)`

  Устанавливает отступ кнопок на панели инструментов от границ панели.
- `Void setOrientation(int o)`

  Задает ориентацию панели инструментов.
- `Void setUI(ToolBarUI ui)`

  Задает объект `Look&Feel`.
- `Void updateUI()`

  Сообщение от объекта `UIFactory` об изменении объекта `Look&Feel`.

Класс *javax.swing.JtoolTip*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Методы класса.

- `AccessibleContext getAccessibleContext()`

  Задает объект `AccessibleContext`.
- `JComponent getComponent()`

  Возвращает компонент, к которому относится объект `JToolTip`.
- `String getTipText()`

  Возвращает текст, который отображается, когда виден объект `JToolTip`.
- `ToolTipUI getUI()`

  Возвращает объект `Look&Feel`.
- `String getUIClassID()`

  Возвращает имя класса для объекта `Look&Feel`.
- `protected  String paramString()`

  Возвращает строковое представление объекта `JToolTip`.
- `Void setComponent(JComponent c)`

  Задает компонент, для которого создано описание `JToolTip`.
- `Void setTipText(String tipText)`

  Задает текст для показа, когда видим объект `JToolTip`.
- `Void updateUI()`

  Сообщение об объекте `UIFactory` о смене объекта `Look&Feel`.

Класс *javax.swing.Jtree*

Имплементирует интерфейсы `javax.accessibility.Accessible`, `javax.swing.Scrollable`.

Конструкторы класса.

- `JTree()`

  Создает объект `JTree`.

- `JTree(Hashtable value)`

  Создает объект `Jtree` по заданному объекту `Hashtable`.

- `JTree(Object[] value)`

  Создает объект `Jtree`, каждый элемент является дочерним для нового корневого элемента, который не отображается.

- `JTree(TreeModel newModel)`

  Создает объект `JTree` по заданной модели.

- `JTree(TreeNode root)`

  Создает объект.

- `JTree(TreeNode root, boolean asksAllowsChildren)`

  Создает объект `Jtree` с заданным корневым объектом `TreeNode`, корневой узел отображается. Дочерние узлы отображаются в зависимости от значения второго параметра.

- `JTree(Vector value)`

  Создает объект `JTree` с дочерними элементами, расположенными в векторе `Vector`, новый корневой узел не отображается.

Методы класса.

- `void addSelectionInterval(int index0, int index1)`

  Задает интервал выбора в промежутке от `index0` до `index1` включительно.

- `void addSelectionPath(TreePath path)`

  Вставляет узел.

- `void addSelectionPaths(TreePath[] paths)`

  Вставляет выделения.

- `void addSelectionRow(int row)`

  Вставляет выделения ряда.

- `void addSelectionRows(int[] rows)`

  Вставляет выделения рядов.

- `void addTreeExpansionListener(TreeExpansionListener tel)`

  Вставляет прослушиватель событий `TreeExpansion`.

- `void addTreeSelectionListener(TreeSelectionListener tsl)`

  Вставляет прослушиватель событий `TreeSelection`.
- `void addTreeWillExpandListener(TreeWillExpandListener tel)`

  Вставляет прослушиватель событий `TreeWillExpand`.
- `void cancelEditing()`

  Удаляет текущую сессию редактирования.
- `void clearSelection()`

  Снимает выделение.
- `rotected  void clearToggledPaths()`

  Очищает выделения.
- `void collapsePath(TreePath path)`

  Сворачивает выделенный узел, делает его видимым.
- `void collapseRow(int row)`

  Сворачивает узел строки.
- `String convertValueToText(Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`

  Вызывается для преобразования заданного значения в текст.
- `protected static TreeModel createTreeModel(Object value)`

  Возвращает объект `TreeModel` для указанного объекта.
- `protected  TreeModelListener createTreeModelListener()`

  Создает и возвращает объект `TreeModelHandler`.
- `void expandPath(TreePath path)`

  Раскрывает и делает выдимым указанный узел.
- `void expandRow(int row)`

  Раскрывает и делает видимым узел строки.
- `void fireTreeCollapsed(TreePath path)`

  Сообщает прослушивателям событий о наступлении события данного типа.
- `void fireTreeExpanded(TreePath path)`

  Сообщает прослушивателям событий о наступлении события данного типа.
- `void fireTreeWillCollapse(TreePath path)`

  Сообщает прослушивателям событий о наступлении события данного типа.
- `void fireTreeWillExpand(TreePath path)`

  Сообщает прослушивателям событий о наступлении события данного типа.
- `protected  void fireValueChanged(TreeSelectionEvent e)`

  Сообщает прослушивателям событий о наступлении события данного типа.

- `AccessibleContext getAccessibleContext()`

  Получает все объекты `AccessibleContext`, связанные с данным компонентом `Jcomponent`.

- `TreeCellEditor getCellEditor()`

  Возвращает редактор, используемый для редактирования элементов дерева.

- `TreeCellRenderer getCellRenderer()`

  Возвращает объект `TreeCellRenderer`, используемый для отображения ячеек.

- `TreePath getClosestPathForLocation(int x, int y)`

  Возвращает путь к узлу, наиболее близкому к точке `x`, `y`.

- `int getClosestRowForLocation(int x, int y)`

  Возвращает ряд, наиболее близкий к точке `x`, `y`.

- `protected static TreeModel getDefaultTreeModel()`

  Возвращает созданный объект `TreeModel`.

- `protected  Enumeration (TreePath parent)`

  Возвращает объект `Enumeration`.

- `TreePath getEditingPath()`

  Возвращает путь к редактируемому элементу.

- `Enumeration getExpandedDescendants(TreePath parent)`

  Возвращает объект `Enumeration` для путей, расположенных ниже текущего в объекте `TreePath`.

- `boolean getInvokesStopCellEditing()`

  Возвращает индикатор прекращения редактирования.

- `Object getLastSelectedPathComponent()`

  Возвращает последний компонент в выбранном текущем пути.

- `TreePath getLeadSelectionPath()`

  Возвращает путь для последнего добавленного узла.

- `int getLeadSelectionRow()`

  Возвращает строку последнего вставленного узла строки.

- `int getMaxSelectionRow()`

  Получает последний выделенный ряд.

- `int getMinSelectionRow()`

  Возвращает первый выделенный ряд.

- `TreeModel getModel()`

  Возвращает объект `TreeModel`.

- `protected TreePath[] getPathBetweenRows(int index0, int index1)`

  Возвращает объект `JTreePath` для путей в интервале от `index0` до `index1` включительно.

- `Rectangle getPathBounds(TreePath path)`

  Возвращает объект `Rectangle` в котором рисуется заданный узел.

- `TreePath getPathForLocation(int x, int y)`

  Возвращает путь к узлу по заданному положению.

- `TreePath getPathForRow(int row)`

  Возвращает путь к указанному ряду.

- `Dimension getPreferredScrollableViewportSize()`

  Возвращает предпочтительный размер отображения объекта `JTree`.

- `Rectangle getRowBounds(int row)`

  Возвращает объект `Rectangle`, в котором отображен узел, расположенный в указанном ряду.

- `int getRowCount()`

  Возвращает число рядов, отображенных в данный момент.

- `int getRowForLocation(int x, int y)`

  Возвращает ряд, расположенный по указанным координатам.

- `int getRowForPath(TreePath path)`

  Возвращает ряд, отображающий узел с заданным путем.

- `int getRowHeight()`

  Возвращает высоту рядов.

- `int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)`

  Возвращает инкремент (высоту или ширину).

- `boolean getScrollableTracksViewportHeight()`

  Возвращает `false`, если высота объекта `JViewport` не задает высоту таблицы.

- `boolean getScrollableTracksViewportWidth()`

  То же, что и в предыдущем пункте, но для ширины.

- `int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)`

  Возвращает инкремент при прокрутке.

- `boolean getScrollsOnExpand()`

  Возвращает `true`, если скроллирование привело к первоначально скрытому элементу.

- `int getSelectionCount()`

  Возвращает число выделенных элементов.

- `TreeSelectionModel getSelectionModel()`

  Возвращает модель выделения, объект `TreeSelectionModel`.

- `TreePath getSelectionPath()`

  Возвращает путь к первому выделенному узлу.

- `TreePath[] getSelectionPaths()`

  Возвращает пути для всех выделений.

- `int[] getSelectionRows()`

  Возвращает все выделенные ряды.

- `boolean getShowsRootHandles()`

  Возвращает `true`, если отображены метки корневых узлов.

- `String getToolTipText(MouseEvent event)`

  Переопределяет метод `getToolTipText` в компоненте `JComponent`, чтобы получить возможность отображать пояснения.

- `TreeUI getUI()`

  Возвращает объект `Look&Feel` для данного компонента.

- `String getUIClassID()`

  Возвращает имя класса `Look&Feel` для отображения компонента.

- `int getVisibleRowCount()`

  Возвращает число строк в отображаемой области.

- `boolean hasBeenExpanded(TreePath path)`

  Возвращает `true`, если узел был когда-либо развернут.

- `boolean isCollapsed(int row)`

  Возвращает `true`, если узел в указанном ряду свернут.

- `boolean isCollapsed(TreePath path)`

  Возвращает `true`, если узел по указанному пути свернут.

- `boolean isEditable()`

  Возвращает `true`, если дерево можно редактировать.

- `boolean isEditing()`

  Возвращает `true`, если дерево редактируется в данный момент.

- `boolean isExpanded(int row)`

  Возвращает `true`, если узел в указанном ряду в данное время развернут.

- `boolean isExpanded(TreePath path)`

  Возвращает `true`, если узел по указанному пути в данный момент развернут.

- `boolean isFixedRowHeight()`

  Возвращает `true`, если высота ряда фиксирована.

- `boolean isLargeModel()`

  Возвращает `true`, если дерево сконфигурировано для большой модели.

- `boolean isPathEditable(TreePath path)`

  Возвращает значение `isEditable`.

- `boolean isPathSelected(TreePath path)`

  Возвращает `true`, если элемент по заданному пути в данное время выбран.

- `boolean isRootVisible()`

  Возвращает `true`, если корневой узел дерева отображен.

- `boolean isRowSelected(int row)`

  Возвращает `true`, если узел, расположенный в указанном ряду, выбран.

- `boolean isSelectionEmpty()`

  Возвращает `true`, если выбор пуст.

- `boolean isVisible(TreePath path)`

  Возвращает `true`, если элемент, расположенный по указанному пути, видим.

- `void makeVisible(TreePath path)`

  Делает видимым узел по заданному пути.

- `protected  String paramString()`

  Возвращает строковое представление объекта `JTree`.

- `protected  void (Enumeration toRemove)`

  Удаляет младшие узлы.

- `void removeSelectionInterval(int index0, int index1)`

  Удаляет узлы в диапазоне от `index0` до `index1` включительно.

- `void removeSelectionPath(TreePath path)`

  Удаляет узел по указанному пути из выделения.

- `void removeSelectionPaths(TreePath[] paths)`

  Удаляет узлы из выделения.

- `void removeSelectionRow(int row)`

  Удаляет путь по указанному ряду из выделения.

- `void removeSelectionRows(int[] rows)`

  Удаляет пути по рядам.

- `void (TreeExpansionListener tel)`

  Удаляет прослушиватель событий `TreeExpansion`.

- `Void removeTreeSelectionListener(TreeSelectionListener tsl)`

  Удаляет прослушиватель событий `TreeSelection`.

- `void removeTreeWillExpandListener(TreeWillExpandListener tel)`

  Удаляет прослушиватель событий `TreeWillExpand`.

- `void scrollPathToVisible(TreePath path)`

  Делает видимыми все компоненты по указанному пути, раскрывая их.

- `void scrollRowToVisible(int row)`

  Прокручивает к указанному ряду.

- `void setCellEditor(TreeCellEditor cellEditor)`

  Задает редактор ячейки.

- `void setCellRenderer(TreeCellRenderer x)`

  Задает объект `TreeCellRenderer`, который будет использоваться для отображения ячеек.

- `void setEditable(boolean flag)`

  Определяет, является ли дерево редактируемым.

- `void setInvokesStopCellEditing(boolean newValue)`

  Определяет, что привело к прекращению редактирования.

- `void setLargeModel(boolean newValue)`

  Должен ли интерфейс пользователя работать с большой моделью.

- `void setModel(TreeModel newModel)`

  Задает модель `TreeModel`.

- `void setRootVisible(boolean rootVisible)`

  Видим ли корневой узел модели.

- `void setRowHeight(int rowHeight)`

  Устанавливает высоту ячейки.

- `void setScrollsOnExpand(boolean newValue)`

  Нужно ли отображать максимально возможное число дочерних элементов при развертывании узла.

- `void setSelectionInterval(int index0, int index1)`

  Выделяет узлы в интервале от `index0` до `index1` включительно.

- `void setSelectionModel(TreeSelectionModel selectionModel)`

  Задает модель выбора для дерева.
- `void setSelectionPath(TreePath path)`

  Выделяет узел по заданному пути.
- `void setSelectionPaths(TreePath[] paths)`

  Выделяет узлы по массиву путей.
- `void setSelectionRow(int row)`

  Выделяет узел по заданному ряду.
- `void setSelectionRows(int[] rows)`

  Выделяет узлы по массиву номеров рядов.
- `void setShowsRootHandles(boolean newValue)`

  Отображены ли метки узлов.
- `void setUI(TreeUI ui)`

  Задает объект `Look&Feel`.
- `void setVisibleRowCount(int newCount)`

  Задает число рядов для отображения.
- `void startEditingAtPath(TreePath path)`

  Выбирает узел по указанному пути и начинает редактирование.
- `boolean stopEditing()`

  Завершает текущую сессию редактирования.
- `void treeDidChange()`

  Вызывается, если дерево изменилось настолько, что требуется изменить границы, но не требуется изменения структуры дерева (разворачивания узлов).
- `void updateUI()`

  Сообщение от объекта `UIManager` об изменении объекта `Look&Feel`.

Класс *javax.swing.JViewport*

Имплементирует интерфейс `javax.accessibility.Accessible`.

Методы класса.

- `void addChangeListener(ChangeListener l)`

  Вставляет прослушиватель `ChangeListener` для определения событий изменения размеров, положения объекта `Viewport`.

- `protected void addImpl(Component child, constraints, int index)`

  Задает дочерний компонент в объекте `Viewport`.

- `protected LayoutManager createLayoutManager()`

  Создает объект `LayoutManager`.

- `protected JViewport.ViewListener createViewListener()`

  Создает прослушиватель.

- `protected void firePropertyChange(String propertyName, oldValue, Object newValue)`

  Сообщает прослушивателю об изменении свойств.

- `protected void fireStateChanged()`

  Сообщает прослушивателю об изменении состояния.

- `AccessibleContext getAccessibleContext()`

  Получает объект `AccessibleContext`.

- `Dimension getExtentSize()`

  Возвращает размер видимой части.

- `Insets getInsets()`

  Возвращает размеры в виде набора (0, 0, 0, 0), т. к. границы для объекта `Jviewport` не определены.

- `Insets getInsets(Insets insets)`

  Возвращает объект `Insets` с параметрами объекта `JViewport`.

- `Component getView()`

  Возвращает дочерний элемент для объекта `JViewport` или `null`.

- `Point getViewPosition()`

  Возвращает координаты верхнего левого угла вида `Viewport`, если вида нет, то возвращает (0, 0).

- `Rectangle getViewRect()`

  Возвращает объект `Rectangle` с положением, возвращаемым `getViewPosition`, и размером `getExtentSize()`.

- `Dimension getViewSize()`

  Размер объекта вида `JViewport`.

- `boolean isBackingStoreEnabled()`

  Возвращает `true`, если вид поддерживает внеэкранное хранение изображений.

- `boolean isOptimizedDrawingEnabled()`

  Объект `JViewport` использует переопределенный метод, возвращающий `false`.

- `void paint(Graphics g)`

  Создает рисунок с использованием заэкранного холста.

- `protected  String paramString()`

  Возвращает строковое представление `JViewport`.

- `void remove(Component child)`

  Удаляет дочерний элемент объекта `JViewport`.

- `void removeChangeListener(ChangeListener l)`

  Удаляет прослушиватель событий.

- `void repaint(long tm, int x, int y, int w, int h)`

  Производит перерисовку.

- `void reshape(int x, int y, int w, int h)`

  Задает границы для вида.

- `void scrollRectToVisible(Rectangle contentRect)`

  Объект `Rectangle` становится видимым после прокрутки.

- `void setBackingStoreEnabled(boolean x)`

  Возвращает `true`, если вид поддерживает скрытое хранение изображений.

- `void setBorder(Border border)`

  Установка границ.

- `void setExtentSize(Dimension newExtent)`

  Задание размеров.

- `void setView(Component view)`

  Установка дочернего элемента.

- `void setViewPosition(Point p)`

  Задает координаты, которые будут отображены в верхнем левом углу вида.

- `void setViewSize(Dimension newSize)`

  Задает координаты в верхнем левом углу и размер вида.

- `Dimension toViewCoordinates(Dimension size)`

  Преобразует координаты в пикселах в координаты вида.

- `Point toViewCoordinates(Point p)`

  Преобразует координаты точки из координат в пикселах в координаты вида.