С изменениями версии Java 5.0

И.Н. Блинов В.С. Романчик

Java 2

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Игорь Блинов – доцент Белорусского государственного университета, кандидат физико-математических наук, Sun Certified Programmer. Более двенадцати лет преподает программирование на C/C++, Java и более пяти лет – проектирование корпоративных систем с использованием RUP, UML и распределенное Web-программирование. Руководит лабораторией IBA-БГУ и группой по разработке коммерческих и учебных проектов.

Валерий Романчик – доцент, кандидат физико-математических наук, заведующий кафедрой численных методов и программирования Белгосуниверситета. Около тридцати лет преподает на механико-математическом факультете БГУ курс «Методы программирования», а в последние годы – и спецкурс «Web-программирование». Автор более 80 статей и книг. В сфере его интересов – вопросы, связанные с использованием численных методов и компьютерных технологий для моделирования физических процессов в задачах механики, а также вопросы преподавания.

Коды примеров размещены на Web-ресурсе http://bsu.iba.by в разделе «Учебники».

С изменениями версии Java 5.0

И.Н. Блинов В.С. Романчик ПРАКТИЧЕСКОЕ РУКОВОДСТВО Java 2

С изменениями версии Java 5.0

И.Н. Блинов В.С. Романчик

Java 2

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Семантика и синтаксис языка
Объектно-ориентированный подход
Графический интерфейс на базе Swing
Многопоточность
Внутренние и анонимные классы
Технологии создания корпоративных систем
Сервлеты, JSP, JDBC
Основы технологии Struts
Основы HTML и XML

ТЕСТОВЫЕ ЗАДАНИЯ
ЗАДАЧИ ДЛЯ ВЫПОЛНЕНИЯ

УНИВЕРСИТЕТСКОЕ ИЗДАТЕЛЬСТВО

И.Н. Блинов, В.С. Романчик

Java 2

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Минск
УП «УниверсалПресс»
2005

УДК 004.432.45
ББК 32.973.26-018.1
Б69

Рекомендовано к изданию
Ученым советом факультета прикладной математики
и информатики БГУ

Авторы:
доцент кафедры ИПМОАП БГУ,
кандидат физико-математических наук **И.Н. Блинов**
доцент, зав. кафедрой численных методов и программирования БГУ,
кандидат физико-математических наук **В.С. Романчик**

Рецензенты:
профессор, докт. физ.-матем. наук В.Б. Таранчук;
доц. канд. физ.-матем. наук И.М. Галкин;
доц. канд. техн. наук А.Е. Люлькин;
доц. канд. физ.-матем. наук С.В. Суздаль

Блинов, И.Н.
Б69 Java 2: практ. рук. / И.Н. Блинов, В.С. Романчик. – Мн.: УниверсалПресс, 2005. – 400 с.
ISBN 985-6699-37-1

Книга предназначена для начинающих и продолжающих изучение Java-технологий. В первой части даны основы языка Java и концепции объектно-ориентированного программирования. Во второй части изложены наиболее важные аспекты применения языка. В третьей части приведены основы программировании распределенных информационных систем с применением сервлетов, JSP, баз данных и собственных тегов разработчика.

В конце каждой главы даны тестовые вопросы по материалу главы и задания для выполнения по рассмотренной теме. В приложениях приведены дополнительные материалы, относящиеся к использованию HTML, XML в информационных системах, основанных на применении Java-технологий, а также краткое описание популярной технологии Struts для разработки распределенных систем, объединяющей возможности J2EE и XML.

Дополнительно представлены расширения версии платформы JSDK 1.5.0 (Java 5.0.)

УДК
ББК

ISBN 985-6699-37-1

ПРЕДИСЛОВИЕ

Эта книга является обобщением материала, который авторы предлагали студентам механико-математического факультета и факультета прикладной математики и информатики БГУ, а также слушателям курсов повышения квалификации учебного центра IBA при изучении компьютерных технологий. Изучение Java-технологий по учебному плану следует за изучением технологий C++, поэтому авторы при изложении материала довольно часто ссылаются на соответствующие структуры языка C++. При этом знания языка C++ не требуется и никакой зависимости от языка C++ не используется, кроме ссылок на похожий синтаксис языка и на общие концепции объектно-ориентированного программирования. Книгу можно использовать для обучения программированию на языке Java с "нуля".

Интересы авторов, направленные на обучение, определили структуру данной книги. Книга предназначена для начинающих изучение Java-технологий и для продолжающих обучение на среднем уровне. Авторы считают, что "профессионалов" обучить нельзя, ими становятся только после участия в разработке нескольких серьезных Java-проектов. В то же время данный курс может служить ступенькой к мастерству. Прошедшие обучение по данному курсу успешно сдают различные экзамены, получают международные сертификаты и в состоянии участвовать в командной разработке достаточно больших проектов.

Книга разбита на три логических части и состоит из глав. В первой части даны основы языка Java и концепции объектно-ориентированного программирования. Во второй части изложены наиболее важные аспекты применения языка, в частности коллекции и многопоточность. В третьей части приведены основы программировании распределенных информационных систем с применением сервлетов, JSP и баз данных, также даны основные принципы создания собственных библиотек тегов.

В конце каждой главы находятся тестовые вопросы по материалу данной главы и задания для выполнения по рассмотренной теме. Ответы и пояснения к тестовым вопросам сгруппированы в отдельный блок.

В приложениях приведены дополнительные материалы, относящиеся к использованию HTML, XML в информационных системах, основанных на применении Java-технологий, а также краткое описание популярной технологии Struts для разработки распределенных систем, в полной мере объединяющей возможности J2EE и XML.

Рассмотренный материал относится к программированию на языке Java 2, однако дополнительно представлены нововведения версии платформы JSDK 1.5.0, которую создатели языка назвали Java 5.0.

АВТОРЫ БЛАГОДАРЯТ

В создании этой книги нам помогли многие коллеги. Без их участия этот проект, возможно, был бы до сих пор незавершенным.

Выражаем благодарность директору отделения интернет-технологий компании IBA В.Ю. Никуленко – за техническую и технологическую поддержку проекта, директору учебного центра IBA В.К. Дюбкову – за финансирование проекта, а также директору отделения управления персоналом И.И. Врублевскому – за поддержку и расширение проекта.

Благодарим наших рецензентов: проректора Белорусского государственного педагогического университета, профессора, докт. физ.-матем. наук В.Б. Таранчука, доцентов механико-математического факультета БГУ И.М. Галкина, А.Е. Люлькина, С.В. Суздаля, сотрудника IBA В.В. Леоновича за ценные замечания и предложения.

Выражаем свою признательность студентам факультетов прикладной математики и информатики и механико-математического Белгосуниверситета А. Домино, А. Орлову, А. Шелуховскому, А. Шуманскому, Д. Зюкову, Д. Глиндзичу, А. Казакевичу, А. Полюху, П. Краснопевцеву, П. Шавелю, К. Чернышу, С. Доманову, В. Козловскому, Л. Паримской за подготовку и тестирование целого ряда примеров, иллюстраций и диаграмм.

Часть 1.
Основы языка Java

В первой части книги излагаются вопросы, относящиеся к основам языка Java и концепции объектно-ориентированного программирования: типы данных, операторы, классы и интерфейсы, наследование классов и полиморфизм.

Глава 1
ПРИЛОЖЕНИЯ И АППЛЕТЫ. КЛАССЫ И ОБЪЕКТЫ

Обзор языка Java 2

Язык Java – это объектно-ориентированный, платформенно-независимый язык программирования, используемый для разработки распределенных приложений, работающих в сети Internet. Проект Java был представлен корпорацией Sun Microsystems в 1995 году. Система программирования Java позволяет использовать World Wide Web (WWW) для распространения небольших интерактивных прикладных программ – апплетов. Они размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте как часть документа WWW. Апплет имеет весьма ограниченный доступ к ресурсам компьютера клиента, поэтому он может предоставлять произвольный мультимедийный интерфейс и выполнять сложные вычисления без риска повреждения данных на диске. Другим видом программ являются приложения Java, представляющие собой переносимые коды, которые могут выполняться на любом компьютере, независимо от его архитектуры. Генерируемый при этом виртуальный код представляет набор инструкций для выполнения на интерпретаторе виртуального кода – виртуальной Java-машине (JVM – Java Virtual Machine). Широкое распространение получили сервлеты и JSP (Java Server Pages), предоставляющие клиентам возможность доступа к серверным приложениям и базам данных.

Язык Java использует синтаксис языка C++. Основные отличия от C++ связаны с необходимостью уменьшения размеров программ и увеличения требований к безопасности переносимых приложений, работающих в сети. Java не поддерживает указателей (наиболее опасного средства языка C++), так как возможность работы с произвольными адресами памяти через безтиповые указатели позволяет игнорировать защиту памяти.

Системная библиотека классов языка содержит классы и пакеты, реализующие различные базовые возможности языка. Методы классов, включенных в эти библиотеки, вызываются из JVM во время интерпретации Java-программы. В Java все объекты программы расположены в динамической памяти (heap) и доступны по объектным ссылкам, которые, в свою очередь, хранятся в стеке (stack). Это решение исключило непосредственный доступ к памяти, но усложнило работу с элементами массивов и сделало ее менее эффективной по сравнению с программами на C++. Необходимо отметить, что объектные ссылки языка Java содержат информацию о классе объектов, на которые они ссылаются, так что объектные ссылки – это не указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки. В Java пересмотрена и концепция динамического распределения памяти: отсутствуют способы освобождения динамически выделенной памяти. Вместо этого реализована система автоматического освобождения памяти (сборщик мусора), выделенной с помощью оператора **`new`**.

Стремление разработчиков упростить Java-программы и сделать их более понятными привело к необходимости удаления из языка файлов-заголовков (h-файлов) и препроцессорной обработки. Файлы-заголовки C++, содержащие прототипы классов и распространяемые отдельно от двоичного кода этих классов, усложняют управление версиями, что дает возможность несанкционированного доступа к частным данным. В Java-программах спецификация класса и его реализация всегда содержатся в одном и том же файле.

Java не поддерживает структуры и объединения, являющиеся частными случаями классов в C++. Язык Java не поддерживает перегрузку операторов и typedef, беззнаковые целые (если не считать таковым **`char`**), а также использование методами аргументов по умолчанию. В Java отсутствует множественное наследование, существуют конструкторы, но отсутствуют деструкторы (применяется автоматическая сборка мусора), не используется оператор **`goto`** и слово **`const`**, хотя они являются зарезервированными словами языка.

Наиболее существенные новые возможности, появившиеся в Java, – это интерфейсы (аналог абстрактных классов C++) и многопоточность (возможность одновременного выполнения частей кода).

Изменения в версии J2SE 5.0

В версии языка J2SE 5.0 внесены некоторые изменения и усовершенствования:

- введено понятие шаблона класса;
- поддерживаются перечисления типов;
- упрощен обмен информацией между примитивными типами данных и их классами-оболочками;
- разрешено определение метода с переменным количеством параметров;
- возможен статический импорт констант и методов;
- улучшен механизм формирования коллекций;
- добавлен форматированный консольный ввод/вывод;
- увеличено число математических методов;
- введены новые способы управления потоками;
- добавлены новые возможности в ядре и др.

Простое приложение

Изучение любого языка программирования удобно начинать с программы вывода обычного сообщения:

```
// пример # 1 : простое приложение: First.java
public class First {
     public static void main (String[] args) {
System.out.println("Первая программа на Java!");
      }
}
```

Здесь класс **First** используется для того, чтобы определить метод **main()** который запускается автоматически интерпретатором Java. Метод **main()** содержит аргументы-параметры командной строки **String[] args** в виде массива строк и является открытым (**public**) членом класса. Это означает, что метод **main()** виден и доступен любому классу. Ключевое слово **static** объявляет методы и переменные класса, используемые для работы с классом в целом, а не только с объектом класса. Символы верхнего и нижнего регистров здесь различаются, как и в C++. Вывод строки "Первая программа на Java!" в примере осуществляет метод **println()** (**ln** – переход к новой строке после вывода) свойства **out** класса **System**, который включается в пакет автоматически

вместе с пакетом **lang**. Приведенную программу необходимо поместить в файл с расширением **.java**, имя которого совпадает с именем класса.

Простейший способ компиляции – вызов строчного компилятора:

```
javac First.java
```

При успешной компиляции создается файл **First.class**. Выполнить этот виртуальный код можно с помощью интерпретатора Java:

```
java First
```

Чтобы выполнить приложение, необходимо загрузить и установить последнюю версию пакета, например с сайта **java.sun.com**. При инсталляции рекомендуется указывать для размещения корневой каталог. Если JSDK установлена в директории (для Windows) **c:\jdk1.5.0**, то каталог, который компилятор Java будет рассматривать как корневой для иерархии пакетов, можно задавать с помощью переменной среды окружения **CLASSPATH** в виде:

```
CLASSPATH=.;c:\jdk1.5.0\src.zip
```

Переменной задано еще одно значение '**.**' для использования текущей директории, например **c:\temp**, в качестве рабочей для хранения своих собственных приложений.

Чтобы можно было вызывать сам компилятор и другие исполняемые программы, переменную **PATH** нужно проинициализировать в виде

```
PATH=c:\jdk.5.0\bin
```

Этот путь указывает на месторасположение файлов **javac.exe** и **java.exe**. В различных версиях путь к JSDK может указываться различными способами.

Следующая программа отображает в окне консоли аргументы командной строки метода **main()**. Аргументы представляют массив строк, разделенных пробелами, значения которых присваиваются объектам массива **String[] args**. Объекту **args[0]** присваивается значение первой строки и т.д. Количество аргументов определяется значением **args.length**.

```
/* пример # 2 : вывод аргументов командной строки :
OutArgs.java */
public class OutArgs {
     public static void main(String[] args) {
          for (int j = 0; j < args.length; j++)
System.out.println("Apr #" + j + "-> " + args[j]);
     }
}
```

Запуск этого приложения с помощью следующей командной строки:

```
java OutArgs 2005 argument-2 "Java string"
```

приведет к выводу на консоль следующей информации:

```
Арг #0-> 2005
Арг #1-> argument-2
Арг #2-> Java string
```

Аргументы командной строки могут быть использованы как один из способов ввода строковых данных.

В следующем примере рассматривается ввод строки из потока ввода, связанного с консолью.

```
// пример # 3 : ввод строки с консоли : InputStr.java
import java.io.*;//подключение пакета классов
public class InputStr {
public static void main(String[] args) {
/* байтовый поток ввода System.in передается конст-
руктору при создании объекта класса InputStreamReader
*/
      InputStreamReader is =
            new InputStreamReader (System.in);
/* производится буферизация данных, исключающая необ-
ходимость обращения к источнику данных при выполнении
операции чтения */
      BufferedReader bis = new BufferedReader(is);
            try {
      System.out.println(
      "Введите Ваше имя и нажмите <Enter>:");
/*чтение строки из буфера; метод readLine() требует
обработки возможной ошибки при вводе с консоли в бло-
ке try */
            String name = bis.readLine();
            System.out.println("Привет, " + name);
            } catch (IOException e) {
            System.out.print("ошибка ввода " + e);
            }
      }
}
```

Обработка исключительной ситуации **IOException**, которая возникает в операциях ввода/вывода, осуществляется в методе **main()** с помощью реализации блока **try-catch**.

В результате запуска приложение предложит пользователю ввести имя и нажать ввод, после этого в консоль будет выведен текст приветствия.

В этом примере использованы конструкторы и методы классов **InputStreamReader** и **BufferedReader** из библиотеки **java.io,** в

частности метод **readLine()** для чтения строки из буфера, связанного с потоком ввода **System.in**. Подробности действий указанных классов будут рассмотрены в главе, посвященной потокам ввода/вывода.

Простой апплет

Одной из целей создания языка Java было создание апплетов – небольших программ, запускаемых Web-браузером. Поскольку апплеты должны быть безопасными, они ограничены в своих возможностях, хотя остаются мощным инструментом поддержки Web-программирования на стороне клиента.

```
// пример # 4 : простой апплет : FirstApplet.java
import java.awt.*;
public class FirstApplet extends java.applet.Applet {
      private String date;
      public void init() {
            date = new java.util.Date().toString();
      }
      public void paint(Graphics g) {
            g.drawString("Апплет запущен:", 50, 15);
            g.drawString(date, 50, 35);
      }
}
```

Для вывода текущего времени и даты в этом примере был использован объект **Date** из пакета **java.util**. Метод **toString()** используется для преобразования информации, содержащейся в объекте, в строку для последующего вывода в апплет с помощью метода **drawString()**. Цифровые параметры этого метода обозначают горизонтальную и вертикальную координаты начала рисования строки, считая от левого верхнего угла апплета.

Апплету не нужен метод **main()** – код его запуска помещается в метод **init()** или **paint()**. Для запуска апплета нужно поместить ссылку на его класс в HTML-документ и просмотреть этот документ Web-браузером, поддерживающим Java. При этом можно обойтись очень простым фрагментом (тегом) **<applet>** в HTML-документе **view.html**:

```
<html><body>
<applet code= FirstApplet.class width=300
height=300>
</applet></body></html>
```

Сам файл **FirstApplet.class** при таком к нему обращении должен находиться в той же директории, что и HTML-документ. Исполнителем HTML-документа является браузер Microsoft Internet Explorer или какой-либо другой, поддерживающий Java.

Результат выполнения документа **view.html** изображен на рис.1.1.

Рис. 1.1. Запуск и выполнение апплета

Классы и объекты

Классы в Java содержат переменные-члены класса, а также методы и конструкторы. Основные отличия от классов C++: все функции определяются внутри классов и называются методами; невозможно создать метод, не являющийся методом класса, или объявить метод вне класса; ключевое слово inline как в C++ не поддерживается; спецификаторы доступа **public**, **private**, **protected** воздействуют только на то, перед чем они стоят, а не на участок от одного до другого спецификатора, как в C++; элементы по умолчанию не устанавливаются в **private**, а доступны для классов из данного пакета. Объявление класса имеет вид:

```
[спецификаторы] class имя_класса [extends суперкласс]
[implements список_интерфейсов]{/*определение класса*/}
```

Спецификатор доступа к обычному внешнему классу может быть **public** (класс доступен объектам данного пакета и вне пакета), **final** (класс не может иметь подклассов), **abstract** (класс может содержать абстрактные методы, объект такого класса создать нельзя). По умолчанию спецификатор устанавливается в friendly (класс доступен в данном пакете). Данное слово при объявлении вообще не используется и не является ключевым словом языка.

```
// пример # 5 : простой пример класса : Subject.java
class Subject {
      public String name;
      private int age;
      public Subject() { //конструктор
            name = "NoName";
```

```
            age = 0;
      }
      public Subject(String n) {//конструктор
            name = n;
      }
      public void setAge(int a) {//метод
            age = a;
      }
      public int getAge() {//метод
            return age;
      }

      void show() { //метод
            System.out.println("Имя: " + name
            + ", Возраст: " + age);
      }
}
```

Класс **Subject** содержит два поля **name** и **age**, помеченные как **public** и **private**. Значение поля **age** можно изменять только при помощи методов, например **setAge()**. Поле **name** доступно и напрямую через объект класса **Subject**. Доступ к методам и **public**-полям данного класса осуществляется только после создания объекта данного класса.

```
/* пример # 6 : создание объекта, доступ к полям и
методам объекта : SubjectDemo.java */
public class SubjectDemo {
      public static void main(String[] args) {
            Subject ob = new Subject("Балаганов");
            ob.name = "Шура Балаганов";
            //ob.age = 19;// поле недоступно
            ob.setAge(19);
            ob.show();
      }
}
```

Компиляция и выполнение данного кода приведут к выводу на консоль следующей информации:

Имя: Шура Балаганов, Возраст: 19

Классы из примеров 5 и 6 можно сохранять перед компиляцией в одном файле **SubjectDemo.java**, причем имя этому файлу дается по имени **public** класса, то есть **SubjectDemo**.

Объект класса создается за два шага. Сначала объявляется ссылка на объект класса. Затем с помощью оператора **new** создается экземпляр объекта, например:

```
String str; //объявление ссылки
str = new String(); //создание объекта
```

Однако эти два действия обычно объединяют в одно:

```
String str = new String();/*объявление ссылки и создание объекта*/
```

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору. Операция присваивания для объектов означает, что две ссылки будут указывать на один и тот же участок памяти.

Операции сравнения ссылок не имеют особого смысла. Для сравнения значений объектов необходимо использовать соответствующие методы, например **equals()**. Этот метод наследуется в каждый класс из суперкласса **Object**, который лежит в корне дерева иерархии классов. **x.equals(y)** возвращает **true**, если содержимое объектов **x** и **y** эквивалентно, как, например:

```
/* пример # 7 : сравнение строк и объектов :
ComparingStrings.java */
public class ComparingStrings {
  public static void main(String[] args) {
      String s1, s2;
      s1 = "abc";
      s2 = s1; // переменная ссылается на ту же строку
System.out.println("сравнение ссылок " + (s1 == s2));
// результат true
//создание нового объекта добавлением символа
      s1 += 'd';
//s1-="a";//ошибка, вычитать строки нельзя
System.out.println("сравнение ссылок " + (s1 == s2));
// результат false
//создание нового объекта копированием
      s2 = new String(s1);
System.out.println("сравнение ссылок " + (s1 == s2));
// результат false
System.out.println("сравнение значений "
      + s1.equals(s2)); // результат true
      }
}
```

Объекты можно преобразовывать один к другому с помощью соответствующих методов и конструкторов. В следующем примере с консоли (из потока **System.in)** считываются три цифровые строки, преобразуются в объекты класса **Integer** с помощью метода **Integer.valueOf()**, а затем объекты класса **Integer** преобразуются в целые числа типа **int** с помощью метода **intValue()** этого же класса.

```
/* пример # 8 : преобразование массива строк в массив
целых чисел : ConsoleStringToInt.java */
import java.io.*;
public class ConsoleStringToInt {
     public static void main(String[] args) {
//буферизация входного потока
          BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
//объявление массива строк
          String masStr[] = new String[3];
//объявление массива целых чисел
          int masInt[] = new int[3];
          try {
System.out.println("введите три целых числа,");
System.out.println("после каждого нажмая Enter ->");
          for (int i = 0; i < 3; i++) {
//чтение строки, содержащей целое число
               masStr[i] = br.readLine();
//преобразование строки в целое число
masInt[i] = Integer.valueOf(masStr[i]).intValue();
               }
     System.out.print("массив: ");
               for (int i = 0; i < 3; i++) {
               System.out.print(masInt[i] + "; ");
               }
// обработка ошибки чтения с консоли
          } catch (IOException e) {
          System.out.print("ошибка ввода " + e);
/* обработка ошибки при неправильном формате целого
числа */
          } catch (NumberFormatException e) {
          System.out.print(
          "неправильный формат числа " + e);
          }
     }
}
```

В качестве примера в консоль в ответ на запрос следует ввести некоторые целые числа и получить, например, следующий результат:

```
Введите три целых числа,
после каждого нажмая Enter ->
35
7
58
массив: 35; 7; 58;
```

При вводе символа, не являющегося цифрой, на экран будет выдано сообщение об ошибке при попытке его преобразования в целое число.

В языке Java используются однострочные и блочные комментарии **//** и **/* */**, аналогичные комментариям, применяемым в C++. Введен также новый вид комментария **/** ** */**, который может содержать дескрипторы вида:

@author – задает сведения об авторе;

@exception – задает имя класса исключения;

@param – описывает параметры, передаваемые методу;

@return – описывает тип, возвращаемый методом;

@throws – описывает исключение, генерируемое методом.

Из java-файла, содержащего такие комментарии, соответствующая утилита **javadoc.exe** может извлекать информацию для документирования классов и сохранения ее в виде HTML-документа.

Задания к главе 1

Вариант А

1. Написать приложение, выводящее n строк с переходом и без перехода на новую строку.
2. Написать приложение для ввода пароля из командной строки и сравнения его со строкой-образцом.
3. Написать программу ввода целых чисел как аргументов командной строки, подсчета их суммы (произведения) и вывода результата на консоль.
4. Написать приложение, выводящее фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания. Для получения последней даты и времени использовать класс **Date** из пакета **java.util** (Объявление объекта **Date d = new Date();**) или статический метод класса **System.currentTimeMillis()**.
5. Написать апплет с выводом в окно фамилии разработчика и даты.
6. Добавить комментарий в программы 1-7 в виде **/**** коммен-тарий ***/**, с помощью программы **javadoc.exe** извлечь эту

документацию в HTML-документ и просмотреть полученную страницу Web-браузером.

Вариант В

Ввести с консоли n целых чисел и поместить их в массив. На консоль вывести:

1. Четные и нечетные числа.
2. Числа, которые делятся на 3 или на 9.
3. Числа, которые делятся на 5 или на 10.
4. Наибольший общий делитель и наименьшее общее кратное этих чисел.
5. Простые числа.
6. “Счастливые” числа.
7. Числа Фибоначчи: f0 = f1 = 1, f (n) = f (n–1) + f (n–2).
8. Числа-палиндромы, значения которых в прямом и обратном порядке совпадают.
9. Период десятичной дроби p = m/n для первых двух целых положительных чисел n и m, расположенных подряд.
10. Построить треугольник Паскаля для первого положительного числа.

Тестовые задания к главе 1

Вопрос 1.1.

Дан код:

```
class Quest1 {
     private static void main (String a) {
             System.out.println("Java 2");
     }  }
```

Какие исправления необходимо сделать, чтобы класс **Quest1** стал запускаемым приложением? (выберите 2 правильных варианта)

1) объявить класс **Quest1** как **public**;
2) заменить параметр метода **main()** на **String[] a**;
3) заменить параметр доступа к методу **main()** на **public**;
4) убрать параметр из объявления метода **main()**.

Вопрос 1.2.

Выберите истинные утверждения:

1) в Java можно использовать оператор **goto**;
2) в Java можно создавать методы, не принадлежащие ни одному классу;
3) Java поддерживает множественное наследование классов;
4) в Java нельзя перегрузить оператор;

5) Java поддерживает многопоточность.

Вопрос 1.3.

Дан код:

```
class Quest3 {
      public static void main(String s[ ]) {
      String args;
      System.out.print(args + s);
     } }
```

Результатом компиляции кода будет:

1) ошибка компиляции: метод **main()** содержит неправильное имя параметра;
2) ошибка компиляции: переменная **args** используется до инициализации;
3) ошибка компиляции: несовпадение типов параметров при вызове метода **print()**;
4) компиляция без ошибок.

Вопрос 1.4.

Дан код:

```
public class Quest4 {
      public static void main(String[] args) {
      byte b[]=new byte[80];
      for (int i=0;i<b.length;i++)
            b[i]=(byte)System.in.read();
  System.out.print("Ok");  } }
```

Результатом компиляции и запуска будет:

1) вывод: Ok;
2) ошибка компиляции: так как метод **read()** может порождать исключительную ситуацию типа **IOException**;
3) ошибка компиляции: так как длина массива **b** может не совпадать с длиной считываемых данных;
4) ошибка времени выполнения: так как массив уже проинициализирован.

Вопрос 1.5.

Дан код:

```
public class Quest5{
      public static void main(){
            System.out.print("A");     }
      public static void main(String args){
            System.out.print("Б");     }
```

```
public static void main(String[] args){
      System.out.print("B");     }  }
```

Что будет выведено в результате компиляции и запуска:

1) ошибка компиляции;
2) Б;
3) ВБА;
4) В;
5) АБВ.

Глава 2
ТИПЫ ДАННЫХ. ОПЕРАТОРЫ. МАССИВЫ

Базовые типы данных и литералы

В языке Java определено восемь базовых типов данных, размер каждого из которых остается неизменным независимо от платформы. Беззнаковых типов в Java не существует.

Тип	Размер (бит)	По умолчанию	Значения (диапазон или максимум)
boolean	8	false	true, false
byte	8	0	–128..127
char	16	'\u0000'	0..65535
short	16	0	–32768..32767
int	32	0	-2147483648..2147483647
long	64	0	922372036854775807L
float	32	0.0f	3.40282347E+38
double	64	0.0	1.797693134486231570E+308

В отличие от C++ тип **char** использует формат UNICODE длиной два байта, что позволяет использовать множество наборов символов, включая иероглифы.

В Java используются целочисленные литералы: **1024**, **015** – восьмеричное значение, **0x51** – шестнадцатеричное значение. Целочисленные литералы создают значение типа **int**. если необходимо определить длинный литерал типа **long**, в конце указывается символ **L** (например: **0xffffL**). Литералы с плавающей точкой записываются в виде **1.918** или в экспоненциальной форме **0.112E-05** и относятся к типу **double**. Если необходимо определить литерал типа **float**, то в конце следует добавить символ **F**. Символьные литералы определяются в апострофах (**'a'**, **'\n'**, **'\141'**, **'\u005a'**). Строки заключаются в двойные апострофы и представляют собой объекты. Литералами считаются булевские значения **true** и **false**, а также **null** – значение по умолчанию для объектов.

В арифметических выражениях автоматически выпоняются расширяющие преобразования типа **byte → short → int → long → float → double**. Java автоматически расширяет тип каждого **byte**

или **short** операнда до **int**. Для сужающих преобразований необходимо производить явное преобразование вида `(тип)значение`. Например:

```
byte b=(byte)128;
```

Указанное в примере преобразование в большинстве случаев необязательно, так как, например, при операциях присваивания литералов преобразования выполняются автоматически. Java не позволяет присваивать переменной значение более длинного типа, если только это не константы. Исключение составляют операторы инкремента (++), декремента (--) и сокращенные операторы (+=, /= и т.д.). При инициализации полей класса и локальных переменных с использованием арифметических операторов автоматически выполняется приведение литералов к объявленному типу без его явного указания, если только их значения находятся в допустимых пределах. При явном преобразовании возможно усечение значения.

Имена переменных не могут начинаться с цифры, в именах не могут использоваться символы арифметических и логических операторов, а также символ '**#**'. Применение символов '**$**' и '**_**' допустимо, в том числе и в первой позиции имени.

```
/* пример # 1 : типы данных и операции над ними :
TypeByte.java */
class TypeByte {
      public static void main(String[] args) {
            int i = 3;
            byte b = 1, b1 = 1 + 2;
            //b = b1 + 1; //ошибка приведения типов
                     b = (byte)(b1 + 1);//0
            //b = -b; //ошибка приведения типов
                     b = (byte)-b;//1
            //b = +b1; //ошибка приведения типов
                     b = (byte) + b1; //2
                     b1 *= 2; //3
                     b1++; //4
            //b = i; //ошибка приведения типов
                     b = (byte)i; //5
                     b += i++; //работает!!!  //6
                  float f = 1.1f;
                     b /= f; //работает!!!  //7
      }
}
```

Переменные базовых типов, объявленные как члены класса, хранят нулевые значения, соответствующие своему типу. Если переменные объявлены как локальные переменные в методе, то перед использованием они обязательно должны быть проинициализированы.

Классы-оболочки

Кроме базовых типов данных широко используются соответствующие классы-оболочки (wrapper-классы): **Boolean**, **Character**, **Integer**, **Byte**, **Short**, **Long**, **Float**, **Double**. Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.

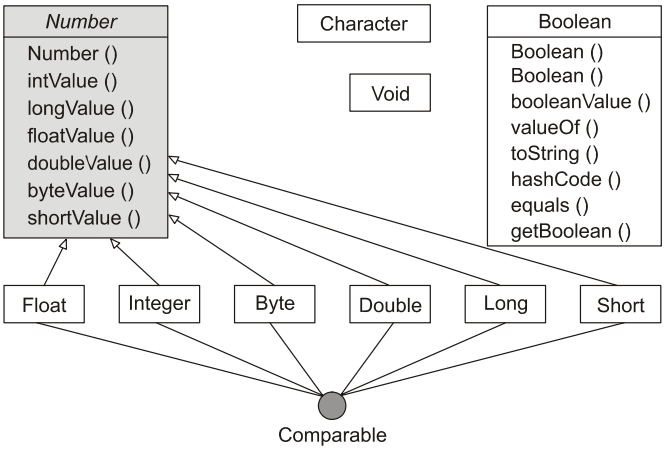

Рис. 2.1. Классы-оболочки

Объекты этих классов представляют ссылки на участки динамической памяти, в которой хранятся их значения, и являются классами-оболочками для значений базовых типов. Указанные классы находятся в библиотеке **java.lang,** являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable**, представляющий собой интерфейс для работы со всеми скалярными типами.

Объекты классов-оболочек не могут принимать участия в арифметических операциях и по умолчанию получают значение **null**.

Переменную базового типа можно преобразовать к соответствующему объекту, передав ее значение конструктору при объявлении объекта. Для приведения объектов к другому типу используются также методы **valueOf(), toString()** и др. Объекты класса могут быть преобразованы к любому базовому типу методами **intValue()**, **digit()**, **longValue()** и др.

Однако класс **Character** не наследуется от **Number**, так как этому классу нет необходимости поддерживать интерфейс классов, предназначенных для хранения результатов арифметических операций. Класс **Character** имеет целый ряд специфических методов для обработки символьной информации. У класса **Character**, в отличие от других классов оболочек, не существует конструктора с параметром типа **String**.

```
/* пример # 2 : преобразование типов данных :
Types.java */
public class Types {
      public static void main(String[] args) {
      Float f1 = new Float(10.01); //double в Float
      String s1 = Float.toString(0f); //float в String
      String s2 = String.valueOf(f1); //Float в String
      Byte b = Byte.valueOf("120"); //String в Byte
      double d = b.doubleValue(); //Byte в double
      short s = (short) d; //double в short
            /* Character в int */
      Character ch = new Character('3');
      int i = Character.digit(ch.charValue(), 10);
      System.out.println("s1=" + s1 + ", s2=" + s2);
      System.out.print("b=" + b + ", s=" + s
            + ", d=" + d + ", i=" + i);
      }
}
```

Результатом выполнения данного кода будет:

```
s1=0.0, s2=10.01
b=120, s=120, d=120.0, i=3
```

Статический метод **valueOf(String str)** класса **Number** переопределен для всех подклассов, соответствующих примитивным типам, и выполняет действия по преобразованию значения, заданного в виде строки, к значению соответствующего объектного типа данных.

Методы вида **byteValue()**, **intValue()** и др. преобразуют значение объектного типа данных к базовому типу. Определены для всех объектных типов.

Статические методы **getNumericValue(char ch)** или **digit (char ch, int radix)** преобразуют символ к соответствующему целочисленному значению типа **int** в заданной системе счисления.

Java включает два класса для работы с высокоточной арифметикой – **BigInteger** и **BigDecimal**, которые поддерживают целые числа и числа с фиксированной точкой произвольной точности.

Строка в Java представляет объект класса **String**. При работе со строками можно использовать перегруженную операцию "**+**" объединения строк, а также методы класса **String**. Строковые константы заключаются в двойные кавычки и не заканчиваются символом **'\0'**, это не ASCII-строки, а массивы символов.

Классы-оболочки в J2SE 5.0

Серьезных различий между базовыми типами и классами-оболочками стало меньше. Теперь разрешено участие объектов в арифметических операциях, в том числе с использованием базовых типов:

```
Integer j = new Integer(1);
Integer k = ++j;
int i = 2;
k = i + j + k;
```

Однако следующий код генерирует исключительную ситуацию **NullPointerException** при попытке присвоить базовому типу значение **null** объекта класса **Integer**.

```
class NewProperties {
      static Integer i = null;
      public static void main(String[] args) {
            int j = i; //генерация исключения
      }
}
```

Несмотря на то, что значения базовых типов могут быть присвоены объектам классов-оболочек, сравнение объектов между собой происходит по ссылкам.

```
int i = 7;
Integer oa = i;
Integer ob = i;
System.out.print("oa == i" + (oa == i));//true
System.out.print("ob == i" + (ob == i));//true
System.out.print("oa == ob" + (oa == ob));//false
System.out.print("oa.equals(i)"
                + oa.equals(i)
                + oa.equals(ob));//true
```

Значение базового типа может быть передано в метод **equals()**. Однако ссылка на базовый тип не может вызывать методы:

```
boolean b = i.equals(oa));//ошибка компиляции
```

При инициализации объекта класса-оболочки значением базового типа преобразование типов необходимо указывать явно, то есть код

```
Float f = 7;//правильно будет (float)7 вместо 7
```

вызывает ошибку компиляции.

Операторы

Операторы Java практически совпадают с операторами C++ и имеют такой же приоритет. Поскольку указатели отсутствуют, то отсутствуют операторы `*`, `&`, `->`, delete для работы с ними.

Операции над целыми числами: **`+, –, *, %, /, ++, ––`** и битовые операции **`&`**, **`|`**, **`^`**, **`~`** аналогичны операциям C++. Деление на ноль целочисленного типа вызывает исключительную ситуацию, переполнение не контролируется.

Операции над числами с плавающей запятой практически те же, что и в C++, но по стандарту IEEE 754 введены понятие бесконечности **`+Infinity`** и **`–Infinity`** и значение **`NaN`** (Not a Number), которое может быть получено, например, при извлечении квадратного корня из отрицательного числа.

Арифметические операторы

+	Сложение	/	Деление
+ =	Сложение (с присваиванием)	/=	Деление (с присваиванием)
–	Бинарное вычитание и унарное изменение знака	**%**	Остаток от деления
– =	Вычитание (с присваиванием)	**%=**	Остаток от деления (с присваиванием)
*	Умножение	++	Инкремент
*=	Умножение (с присваиванием)	– –	Декремент

Битовые операторы

|	*Или*	>>	Сдвиг вправо
|=	*Или* (с присваиванием)	>>=	Сдвиг вправо (с присваиванием)
&	*И*	>>>	Сдвиг вправо с появлением нулей
&=	*И* (с присваиванием)	>>>=	Сдвиг вправо с появлением нулей и присваиванием
^	Исключающее *или*	<<	Сдвиг влево
^=	Исключающее *или* (с присваиванием)	<<=	Сдвиг влево с присваиванием
~	Унарное отрицание		

Операторы отношения

<	Меньше	>	Больше
<=	Меньше либо равно	>=	Больше либо равно
==	Равно	!=	Не равно

Применяются для сравнения символов, целых и вещественных чисел, а также для сравнения ссылок при работе с объектами.

Логические операторы

||	*Или*	**&&**	*И*
!	Унарное отрицание		

К операторам относится также оператор определения принадлежности типу **instanceof**, оператор **[ ]** и тернарный оператор **?:** (if-then-else).

Логические операции выполняются над значениями типа **boolean** (**true** или **false**).

```
// пример # 3 : битовые операторы : Operators.java
public class Operators {
  public static void main(String[] args) {
System.out.println("5%1=" + 5%1 + "  5%2=" + 5%2);
           int b1 = 0xe;//14 или 1110
           int b2 = 0x9;//9  или 1001
           int i = 0;
System.out.println(b1 + "|" + b2 + " = " + (b1|b2));
System.out.println(b1 + "&" + b2 + " = " + (b1&b2));
System.out.println(b1 + "^" + b2 + " = " + (b1^b2));
System.out.println(     "~" + b2 + " = " + ~b2);
System.out.println(b1 + ">>" + ++i
                + " = " + (b1>>i));
System.out.println(b1 + "<<" + i
                + " = " + (b1<<i++));
System.out.println(b1 + ">>>" + i +" = " + (b1>>>i));
     }
}
```

Результатом выполнения данного кода будет:

```
5%1=0  5%2=1
14|9 = 15
14&9 = 8
14^9 = 7
~9 = -10
```

```
14>>1 = 7
14<<1 = 28
14>>>2 = 3
```

Тернарный оператор "**?**" используется в выражениях:

```
booleanexp ? value0 : value1
```

Если `booleanexp` равно **true**, вычисляется значение `value0` и оно становится результатом выражения, иначе результатом является значение `value1`.

Оператор **instanceof** возвращает значение **true**, если объект является экземпляром данного класса**,** например:

```
Font obj = new Font("Courier", 1, 18);
if (obj instanceof java.awt.Font) {/*операторы*/}
```

Числовые параметры при объявлении объекта класса **Font** указывают на стиль и размер шрифта.

Результатом действия оператора **instanceof** будет истина, если объект является объектом одного из подклассов класса, на принадлежность к которому проверяется данный объект, но не наоборот. Проверка на принадлежность объекта к классу **Object** всегда даст истину как результат. Результат применения этого оператора по отношению к **null** всегда ложь, потому что **null** нельзя причислить к какому-либо типу. В тоже время литерал **null** можно передавать в методы по ссылке на любой объектный тип и использовать в качестве возвращаемого значения.

Операторы управления

Оператор **if** и три вида операторов цикла аналогичны операторам C++:

```
if (boolexp) { /*операторы*/}
else { /*операторы*/ }//может отсутствовать
while (boolexp) { /*операторы*/ }

do { /*операторы*/ }
while (boolexp);

for(exp1; boolexp; exp3){ /*операторы*/ }
```

Циклы выполняются, пока булевское выражение `boolexp` равно **true**. Аналогично C++ используется и оператор **switch:**

```
switch(exp) {
     case exp1:/*операторы, если exp==exp1*/
          break;
```

```
      case exp2:/*операторы, если exp==exp2*/
            break;
      default: /* операторы Java */
}
```

При совпадении условий вида exp**==**exp1 выполняются подряд все блоки операторов до тех пор, пока не встретится оператор **break**.

Расширение возможностей получили оператор прерывания цикла **break** и оператор прерывания итерации цикла **continue**, которые можно использовать с меткой, например:

```
/* пример # 4 : выход за цикл, помеченный меткой :
DemoLabel.java */
public class DemoLabel {
      public static void main(String[] a) {
            int j = -3;
            OUT: while(true) {
                  for(;;)
                  while (j < 10) {
                  if (j == 0)
                        break OUT;
                  else {
                        j++;
                        System.out.println(j);
                        }
                  }
            }
            System.out.println("end");
      }
}
```

Здесь оператор **break** разрывает цикл, помеченный меткой **OUT**.

Цикл for в J2SE 5.0

Появилась возможность при работе с массивами и коллекциями получать доступ к их элементам без использования индексов или итераторов.

```
 int[] array = {1, 3, 5, 11};
  for(int i : array)
  System.out.printf("%d ", i);//вывод всех элементов
```

В то же время изменить значения элементов массива с помощью такого цикла нельзя.

Массивы

Массивы элементов базовых типов состоят из значений, проиндексированных начиная с нуля. Все массивы в языке Java являются динамическими, поэтому для создания массива требуется выделение памяти с помощью оператора **new** или инициализации. Значения элементов неинициализированных массивов, для которых выделена память, устанавливается в нуль. Имена массивов являются ссылками. Для объявления ссылки на массив можно записать пустые квадратные скобки после имени типа, например: **int a[]**. Аналогичный результат получится при записи **int[] a**.

```
/* пример # 5 : замена отрицательных элементов масси-
ва на максимальный : FindReplace.java */
public class FindReplace {
 public static void main(String[] args) {
      int myArray[];//объявление ссылки
//объявление с инициализацией значениями по умолчанию
      int mySecond[] = new int[100];
      /*объявление с инициализацией */
      int a[] = {5, 10, 0, -5, 16, -2};
      int max = a[0];
 //поиск max-элемента
  for(int i = 0; i < a.length; i++)
        if(max < a[i])
                 max = a[i];
   for(int i = 1; i < a.length; i++)   { //замена
        if( a[i] < 0 ) a[i] = max;
           mySecond[i] = a[i];
   System.out.println("a[" + i + "]= " + a[i]);
         }
      myArray = a; //установка ссылки на массив a
      }
}
```

В результате выполнения программы будет выведено:

```
a[0]= 5
a[1]= 10
a[2]= 0
a[3]= 16
a[4]= 16
a[5]= 16
```

Присваивание **mySecond[i]=a[i]** приведет к тому, что части элементов массива **mySecond**, а именно шести, будут присвоены значения элементов массива **a**. Остальные элементы **mySecond** сохранят значения,

полученные при инициализации, то есть нули. Если же присваивание организовать в виде **mySecond=a** или **myArray=a**, то оба массива, участвующие в присваивании, получат ссылку на массив **a**, то есть оба будут содержать по шесть элементов и ссылаться на один и тот же участок памяти.

Многомерных массивов в Java не существует, но можно объявлять массивы массивов. Для задания начальных значений массивов существует специальная форма инициализатора, например:

```
int arr[][] = {       { 1 },
                      { 2, 3 },
                      { 4, 5, 6 },
                      { 7, 8, 9, 0 }
                  };
```

Первый индекс указывает на порядковый номер массива, например **arr[2][0]** указывает на первый элемент третьего массива, а именно на значение **4**.

В следующей программе создаются и инициализируются массивы массивов равной длины (матрицы) и выполняется произведение одной матрицы на другую.

```
/* пример # 6  : произведение двух матриц :
Matrix.java */
public class Matrix {
     private int[][] a;
     Matrix(int n, int m) {
// создание и заполнение случайными значениями
     a = new int[n][m];
     for (int i = 0; i < n; i++)
          for (int j = 0; j < m; j++)
               a[i][j] = (int) (Math.random() * 5);
          show();
     }
public Matrix(int n, int m, int k) {
     a = new int[n][m];
     for (int i = 0; i < n; i++)
          for (int j = 0; j < m; j++) {
               a[i][j] = k;
          }
          if(k!=0) show();
}
public void show() {
System.out.println("Матрица : " + a.length
               + " на " + a[0].length);
     for (int i = 0; i < a.length; i++) {
```

```
            for (int j = 0; j < a[0].length; j++)
                  System.out.print(a[i][j] + " ");
            System.out.println();
      }
}
public static void main(String[] args) {
      int n = 2, m = 3, l = 4;
      Matrix p = new Matrix(n, m);
      Matrix q = new Matrix(m, l);
      Matrix r = new Matrix(n, l, 0);
      for (int i = 0; i < p.a.length; i++)
            for (int j = 0; j < q.a[0].length; j++)
            for (int k = 0; k < p.a[0].length; k++)
                  r.a[i][j] += p.a[i][k] * q.a[k][j];
      System.out.println("Произведение матриц: ");
                  r.show();
    }
}
```

Так как значения элементам массивов присваиваются при помощи метода **random()**, то одним из вариантов выполнения кода может быть следующий:

```
Матрица : 2 на 3
1 1 1
2 4 1
Матрица : 3 на 4
0 1 2 3
3 1 0 4
3 4 0 4
Произведение матриц:
Матрица : 2 на 4
6 6 2 11
15 10 4 26
```

Если объект создан внутри класса, то он имеет прямой доступ к полям, объявленным как **private**.

Массивы объектов при объявлении в действительности представляют собой массивы ссылок, проинициализированных по умолчанию значением **null**.

```
/* пример # 7 : копирование массива :
ArrayCopyDemo.java */
public class ArrayCopyDemo {
```

```
     public static void main(String[] args) {
int  mas1[] = { 1, 2, 3 },
     mas2[] = { 4, 5, 6, 7, 8, 9 };
          System.out.print("mas1[]: ");
          show(mas1);
          System.out.print("\nmas2[]: ");
          show(mas2);
//копирование массива mas1[] в mas2[]
          System.arraycopy(mas1, 0, mas2, 2, 3);
/*   0 – mas1[] копируется начиная с первого элемента,
     2 – элемент, с которого начинается замена,
     3 – количество копируемых элементов */
     System.out.println("\n после arraycopy(): ");
     System.out.print("mas1[]: ");
          show(mas1);
     System.out.print("\n mas2[]: ");
          show(mas2);
     }
     private static void show(int[] mas) {
          int i;
          for (i = 0; i < mas.length; i++)
               System.out.print(" " + mas[i]);
     }
}
mas1[]:  1 2 3
mas2[]:  4 5 6 7 8 9
после arraycopy():
mas1[]:  1 2 3
mas2[]:  4 5 1 2 3 9
```

Все массивы хранятся в куче (heap), одной из подобластей памяти, выделенной системой для работы виртуальной машины. Определить общий объем памяти и объем свободной памяти можно с помощью методов **totalMemory()** и **freeMemory()** класса **Runtime**.

```
/* пример # 8  : информация о состоянии оперативной
памяти : RuntimeDemo.java */
public class RuntimeDemo {
     public static void main(String[] args) {
          Runtime rt = Runtime.getRuntime();
          System.out.println("Полный объем памяти: "
                    + rt.totalMemory());
          System.out.println("Свободная память: "
                    + rt.freeMemory());
```

```
            double d[] = new double[10000];
      System.out.println("Свободная память после"
      + " объявления массива: " + rt.freeMemory());
            try {
            rt.exec("mspaint"); //запуск mspaint.exe
            } catch (java.io.IOException e) {
            System.out.println(e.getMessage());
            }
      System.out.println("Свободная память после "
      + "запуска mspaint.exe: " + rt.freeMemory());
      }
}
```

В результате выполнения этой программы может быть выведена, например, следующая информация:

```
Полный объем памяти: 2031616
Свободная память: 1903632
Свободная память после объявления массива: 1823336
Свободная память после запуска mspaint.exe: 1819680
```

Объект класса **Runtime** создается при помощи вызова статического метода **getRuntime()**, возвращающего объект **Runtime**, который ассоциирован с данным приложением. Запуск внешних приложений осуществляется с помощью метода **exec()**, в качестве одного из параметров которого может применяться строка с именем запускаемого приложения. Внешнее приложение использует для своей работы память операционной системы.

Класс Math

Рассмотрим пример обработки значения случайного числа, полученного с помощью метода **random()** класса **Math**. В классе **Math** существует ряд других полезных методов, таких как **floor()**, **ceil()**, **rint()**, **round()**, **max(**параметр, параметр**)**, **min(**параметр, параметр**)**, которые выполняют задачи по округлению, поиску экстремальных значений, нахождению ближайшего целого и т.д.

```
/* пример # 9 : использование методов класса Math :
MathMethods.java */
public class MathMethods {
      public static void main(String[] args) {
            final int MAX_VAL = 10;
            double d, max = 0, min = MAX_VAL;
      d = Math.random() * MAX_VAL;
      System.out.println("d = " + d);
      System.out.println("Округленное до ближайшего"
```

```
+ " целого =" + Math.round(d));
      System.out.println("Наибольшее целое, "
+ " <= исходного числа =" + Math.floor(d));
      System.out.println("Наименьшее целое, "
+ " >= исходного числа =" + Math.ceil(d));
      System.out.println("Ближайшее целое значение"
+ " исходного числа =" + Math.rint(d));
      }
}
```

Один из вариантов выполнения кода представлен ниже:

```
d = 0.08439575016076173
Округленное до ближайшего целого =0
Наибольшее целое, <= исходного числа =0.0
Наименьшее целое, >= исходного числа =1.0
Ближайшее целое значение исходного числа =0.0
```

Перечисления в J2SE 5.0

Типобезопасные перечисления (typesafe enums) в Java представляют собой значительно более серьезный механизм, чем в C/C++, хотя внешне очень похожи:

```
enum MyDay { MORNING, NIGHT , DAY, EVENING }
```

В качестве простейшего примера можно рассмотреть следующий код.

```
/* пример # 10  : применение перечисления :
DemoEnum.java */
public class DemoEnum {
      enum MyDay { MORNING, NIGHT, DAY, EVENING }
      public static void main(String args[]) {
            MyDay day = MyDay.NIGHT;
            System.out.print("Мое время: ");
            switch (day) {
                  case NIGHT : System.out.print(day);
            }
      }
   }
```

Результатом выполнения будет:

```
Мое время: NIGHT
```

Перечисления – это к тому же классы, которые могут содержать поля и методы, реализовывать интерфейсы. Каждый тип **enum** содержит статический метод **values()**, который возвращает массив, содержащий все элементы перечисления в порядке их объявления.

```
/* пример # 11  : применение перечисления :
MethodDemo.java */
import static java.lang.Math.*;
public class MethodDemo {
      public enum Shape {
            RECTANGLE, TRIANGLE, CIRCLE;
            double square(double x, double y) {
                  switch (this) {
                        case RECTANGLE :
                              return x * y;
                        case TRIANGLE :
                              return x * y / 2;
                        case CIRCLE :
                              return pow(x, 2) * PI;
                  }
      throw new AssertionError("неизвестная фигура: "
                  + this);
            }
      }
      public static void main(String args[]) {
            double x = 2, y = 3;
            for (Shape sh : Shape.values())
            System.out.printf("%s = %f%n", sh,
                  sh.square(x, y));
      }
}
```

Каждый из элементов перечисления в данном случае представляет собой арифметическую операцию, ассоциированную с методом **square()**. Без **throw** код не будет компилироваться, что позволяет указать на возможную ошибку при появлении необъявленной фигуры. Поэтому и при добавлении нового элемента необходимо добавлять соответствующий ему **case**.

Однако на перечисления накладывается целый ряд ограничений.

Запрещено:

- их наследовать;
- делать их абстрактными;
- создавать экземпляры, используя ключевое слово **new**.

Задания к главе 2

Вариант А

В приведенных ниже заданиях необходимо вывести внизу фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания. Для получения последней даты и времени следует использовать класс **`Date`**. Добавить комментарии в программы в виде **`/**`** `комментарий` **`*/`**, извлечь эту документацию в HTML-файл и просмотреть полученную страницу Web-браузером.

1. Ввести n строк с консоли, найти самую короткую строку. Вывести эту строку и ее длину.
2. Ввести n строк с консоли. Найти самую длинную строку. Вывести найденную строку и ее длину.
3. Ввести n строк с консоли. Упорядочить и вывести строки в порядке возрастания их длин, а также значения их длин.
4. Ввести n строк с консоли. Упорядочить строки и вывести эти строки в порядке убывания длины, а также значения их длин.
5. Ввести n строк с консоли. Вывести на консоль те строки, длина которых меньше средней, а также их длины.
6. Ввести n строк с консоли. Вывести на консоль те строки, длина которых больше средней, а также их длины.
7. Написать программы решения задач 1–6, осуществляя ввод строк как аргументов командной строки.

Вариант В

Ввести с консоли n – размерность матрицы a[n][n]. Задать значения элементов матрицы в интервале значений от n1 до n2 с помощью датчика случайных чисел.

1. Упорядочить строки (столбцы) матрицы в порядке возрастания значений элементов k-го столбца (строки).
2. Выполнить циклический сдвиг заданной матрицы на k позиций вправо (влево, вверх, вниз).
3. Найти и вывести наибольшее число возрастающих (убывающих) элементов матрицы, идущих подряд.
4. Найти сумму элементов матрицы, расположенных между первым и вторым положительными элементами каждой строки.
5. Транспонировать квадратную матрицу.
6. Вычислить норму матрицы.
7. Повернуть матрицу на 90^{o} против часовой стрелки.
8. Вычислить определитель матрицы.

9. Построить матрицу, вычитая из элементов каждой строки матрицы ее среднее арифметическое.
10. Найти максимальный элемент(ы) в матрице и удалить из матрицы строку(и) и столбец(цы), в которой он (они) находится.
11. Уплотнить матрицу, удаляя из нее строки и столбцы, заполненные нулями.
12. В матрице найти минимальный элемент и переместить его на место заданного элемента путем перестановки строк и столбцов.
13. Преобразовать строки матрицы таким образом, чтобы элементы, равные нулю, располагались после всех остальных.

Тестовые задания к главе 2

Вопрос 2.1.

Какие из следующих строк скомпилируются без ошибки?

1) `float f = 7.0;`
2) `char c = "z";`
3) `byte b = 255;`
4) `boolean n = null;`
5) `int i = 32565;`
6) `int j = 'ъ'.`

Вопрос 2.2.

Какие варианты записи оператора условного перехода корректны?

1) `if (i<j) { System.out.print("-1-"); }`
2) `if (i<j) then  System.out.print("-2-");`
3) `if i<j { System.out.print("-3-"); }`
4) `if [i<j] System.out.print("-4-");`
5) `if (i<j) System.out.print("-5-");`
6) `if {i<j} then System.out.print("-6-");.`

Вопрос 2.3.

Какие из следующих слов являются ключевыми или резервированными словами в Java?

1) `if;`
2) `then;`
3) `goto;`
4) `extend;`
5) `case.`

Вопрос 2.4.

Какие из следующих идентификаторов являются корректными?

1) `2int;`
2) `int_#;`
3) `_int;`
4) `_2_;`
5) `$int;`
6) `#int.`

Вопрос 2.5.

Какое из выражений выведет **`-1.0`**?

1) `System.out.print(Math.floor(-1.51));`
2) `System.out.print(Math.round(-1.51));`
3) `System.out.print(Math.ceil(-1.51));`
4) `System.out.print(Math.min(-1.51));`
5) `System.out.print(Math.max(-1.51)).`

Вопрос 2.6.

Дан код:

```
public class Quest6 {
public static void main(String[] args){
      int a[] = new int[]{1,2,3};
      System.out.print(a[1]); } }
```

В результате компиляции и запуска будет выведено:

1) 1;
2) 2;
3) ошибка компиляции: неправильная инициализация;
4) ошибка компиляции: не определен размер массива;
5) ошибка времени выполнения.

Вопрос 2.7.

Какие из приведенных объявлений массивов корректны?

```
int a1[] = {};
int a2[] = new int[]{1,2,3};
int a3[] = new int[](1,2,3);
int a4[] = new int[3];
int a5[] = new int[3]{1,2,3};
```

1) a1;
2) a2;
3) a3;
4) a4;
5) a5.

Вопрос 2.8.

Дан код:

```
public class Quest8{
    static int j = 2;
 public static void result(int i){
     i *= 10;
     j += 2;    }
    public static void main(String[] args){
     char i = '1';
     result(i);
     System.out.println(i + " " + j);    } }
```

В результате компиляции и запуска будет выведено:

1) ошибка компиляции: параметр метода **`result()`** не сочетается с передаваемой переменной ;
2) 10 4;
3) 1 4;
4) 1 2;
5) 10 2.

Глава 3
КЛАССЫ

Классы и отношения

Классы определяют структуру и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.

Классом называется описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Каждый класс имеет свое имя, отличающее его от других классов, и относится к определенному пакету. Имя класса в пакете должно быть уникальным. Физически пакет представляет собой каталог, в который помещаются программные файлы, содержащие реализацию классов. Классы позволяют разбить поведение сложных систем на простое взаимодействие взаимосвязанных объектов.

При проектировании системы необходимо не только идентифицировать сущности, но и указать, как они соотносятся друг с другом.

Отношением называется связь между классами. В объектно-ориентированном проектировании особое значение имеют четыре типа отношений: зависимости, обобщения, ассоциации и реализации.

Зависимостью (Dependency) называется отношение использования, определяющее, что изменение состояния объекта одного класса может повлиять на объект другого класса, который его использует, причем обратное в общем случае неверно. Зависимости применяются тогда, когда экземпляр одного класса использует экземпляр другого, например в качестве параметра метода.

Обобщение (Generalization) означает, что объекты подкласса могут использоваться всюду, где встречаются объекты суперкласса, но ни в коем случае не наоборот. Определение суперкласса является более общим, чем определение подкласса. Подкласс наследует свойства родителя (атрибуты и методы). Идентификация суперклассов и подклассов осуществляется с использованием модели предметной области, так как с ее помощью возможен анализ всех понятий в более общих и абстрактных терминах. В итоге улучшается понимание кода (особенно для систем с сотнями классов), уменьшается объем повторяемой информации.

Например, понятия **CashPayment**, **CreditPayment**, **CheckPayment** очень похожи одно на другое, и в этом случае разумно организовать их в иерархию обобщения – специализации классов. Класс **Payment** представляет более общее понятие, а его подклассы – специализированные свойства.

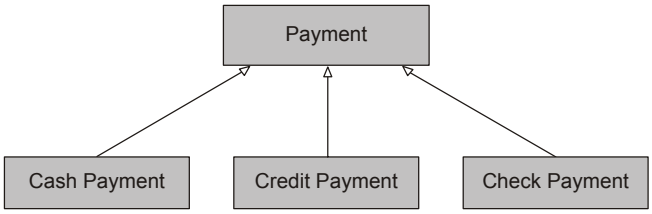

Рис. 3.1. Отношение обобщения

Подкласс создается в случаях, если:

- он имеет дополнительные атрибуты, интересующие разработчика;
- он имеет дополнительные ассоциации, интересующие разработчика;
- ему соответствует понятие, управляемое, обрабатываемое, реагирующее или используемое способом, отличным от способа, определенного суперклассом или другими подклассами;
- он представляет объекту поведение, которое отлично от поведения, определяемого суперклассом или другими подклассами.

Реализацией (Realization) называется отношение между классификаторами (классами, интерфейсами), при котором один из них описывает контракт (интерфейс сущности), а другой гарантирует его выполнение.

Ассоциации (Association) показывают, что объекты одного класса связаны с объектами другого класса и отражают некоторое отношение между ними. В этом случае можно перемещаться (с помощью вызова методов) от объектов одного класса к объектам другого. Агрегация – ассоциация, моделирующая взаимосвязь “часть/целое” между классами, которые в тоже время могут быть равноправными. Оба класса при этом находятся на одном концептуальном уровне, и ни один не является более важным, чем другой.

Тело класса в системе Java может содержать объявление полей данных, конструкторов, методов, внутренних классов и интерфейсов, а также логические блоки, используемые обычно для инициализации полей.

Переменные класса и константы

Данные – члены класса, которые называются полями или переменными класса, объявляются в классе следующим образом:

```
спецификатор тип имя;
```

В языке Java могут использоваться переменные класса, объявленные один раз для всего класса со спецификатором **static** и одинаковые для всех экземпляров класса, или переменные экземпляра, создаваемые для каждого экземпляра класса. Переменные объявляются со спецификаторами доступа **public**, **private**, **protected** или по умолчанию без спецификатора. Кроме данных – членов класса в классе используются локальные переменные и параметры методов. Переменные со спецификатором **final** являются константами. Спецификатор **final** можно использовать для переменной, объявленной в методе, а также для параметра метода.

Объявить и проинициализировать значения переменных класса и локальных переменных метода, а также параметры метода можно следующим образом:

```
/* пример # 1 : типы атрибутов и переменных :
MyClass.java */
class MyClass {
  int x; // переменная экземпляра класса
  int y = 2; // переменная экземпляра класса
  final int YEAR = 2003; // константа
  static int bonus; // переменная класса
  static int b = 1; // переменная класса
  void init(int z){// параметр метода
      z = 3;// переинициализация
      int a;// локальная переменная метода
      a = 4;// инициализация
  }
}
```

В приведенном примере использованы данные базовых типов, не являющиеся ссылками на объекты. Данные могут быть ссылками, назначить которым реальные объекты можно с помощью оператора **new**.

Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать с четырьмя категориями видимости между элементами классов:

- классы и подклассы в том же пакете (по умолчанию);
- независимые классы (**private**);
- подклассы в текущем и других пакетах (**protected**);
- классы, которые не являются подклассами и не входят в тот же пакет (**public**).

Элемент (атрибут или метод), объявленный **public**, доступен из любого места вне класса. Все, что объявлено **private**, доступно только внутри класса и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется по умолчанию. Если же необходимо, чтобы элемент был доступен из другого пакета, но только подклассам того класса, которому он принадлежит, нужно объявить такой элемент со спецификатором **protected**.

Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия по инициализации объекта. Конструктор имеет то же имя, что и класс; вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса. Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

Деструкторы в языке Java не используются, объекты уничтожаются сборщиком мусора после прекращения их использования. Аналогом деструкторов являются методы **finalize()**. Исполняющая среда языка Java будет вызывать их каждый раз, когда сборщик мусора будет уничтожать объекты этого класса, которым не соответствует ни одна ссылка.

```
/* пример # 2 : перегрузка конструктора :
NewBook.java */
class NewBook {
      private String title, publisher;
      private float price;
  public NewBook() {
      title = "NoTitle";
      }
  public NewBook(String t, String pub, float p) {
      title = new String(t);
      publisher = pub;
      price = p;
      }
}
```

Объект класса **NewBook** может быть создан двумя способами, вызывающими один из конструкторов:

```
NewBook tips1; // объявление
tips1 = new NewBook();// инициализация
NewBook tips2 = new NewBook("Java2", "Ноутон", 9.f);
```

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору.

Если конструктор в классе не определен, Java предоставляет конструктор по умолчанию, который инициализирует объект значениями по умолчанию. Если же конструктор с параметрами определен, то конструктор по умолчанию становится недоступным и для его вызова необходимо явное объявление такого конструктора.

В следующем примере объявлен класс **Locate** с двумя полями (атрибутами), конструктором и методами для инициализации и извлечения значений атрибутов.

```
/* пример # 3 : вычисление расстояния между точками :
Distance.java */
class Locate {
      private double x, y; /*по умолчанию x=0 и y=0 */
  public Locate(){
      x = 1;
      y = 1;
      }
  public void setX(double a){
      x = a;
      }
  void setY(double b){ /*видимость по умолчанию*/
      y = b;
      }
  public double getX(){
      return x;
      }
  public double getY(){
      return y;
      }
}
public class Distance {
      public static void main(String[] args) {
//локальные переменные не являются членами класса
     Locate t1 = new Locate();
     Locate t2 = new Locate();
```

```
      double dx, dy, distance;
       t1.setX(5);
       t1.setY(10);
       t2.setX(2);
       t2.setY(6);
       dx = t1.getX() - t2.getX();
       dy = t1.getY() - t2.getY();
       /* вычисление расстояния */
       distance = Math.sqrt(dx*dx + dy*dy);
       //distance = Math.hypot(dx, dy);//java 5.0
System.out.print("расстояние равно: " + distance);
      }
}
```

В результате будет выведено:

```
расстояние равно: 5.0
```

Здесь используется статические методы **sqrt()** (или **hypot()**) из класса **Math**, которые вызываются без объявления объекта указанного класса. Класс **Math** содержит только статические методы для физических и технических расчетов, а также константы **E** и **PI**.

Методы

Все функции Java объявляются только внутри классов и называются методами. Определение метода имеет вид:

```
возвращаемый_тип methodName(список_аргументов){
      //код
      return value;//если нужен
}
```

Если метод не возвращает значение, ключевое слово **return** может отсутствовать, тип возвращаемого значения в этом случае будет **void**. Вместо пустого списка аргументов тип **void** не указывается. Вызов методов осуществляется из объекта или класса (для статических методов):

```
object_name.methodName(список_аргументов);
```

Отметим, что методы-конструкторы вызываются автоматически при создании объекта класса с помощью оператора **new**. Автоматически вызывается метод **main()** при загрузке приложения, содержащего класс с методом **main()**. На протяжении жизненного цикла апплетов автоматиче-

ски запускаются методы **init()**, **start()**, **stop()**, **paint()**, **destroy()**.

Для того чтобы создать метод, нужно внутри объявления класса написать объявление метода и затем реализовать его тело. Объявление метода, как минимум, должно содержать тип возвращаемого значения (возможен **void**) и имя метода. В приведенном ниже объявлении метода элементы, заключенные в квадратные скобки, являются необязательными.

```
[доступ][static][abstract][final][native][synchronized]
возвращаемый_тип methodName(список _аргументов) [throws
список_исключений]
```

Как и для атрибутов, спецификатор доступа к методам может быть **public, private, protected** и friendly (по умолчанию). При этом методы суперкласса можно перегружать или переопределять в подклассе.

Статические методы и атрибуты

Атрибуты, объявленные как **static**, являются общими для всех объектов класса и называются переменными класса. Если один объект изменит значение такого атрибута, то это изменение увидят все объекты. Для работы со статическими атрибутами используются статические методы, объявленные со спецификатором **static**. Такие методы являются методами класса и не содержат указателя **this** на конкретный объект.

```
// пример # 4 : статический метод : MyStatic.java
class MyStatic {
   private static int x = 1;
   public static int getX(){
          return x;
     }
}
```

Вызов статического метода возможен с помощью указания: имя_класса.имя_метода, например:

```
int y = MyStatic.getX();
```

Можно вызывать такие методы, например из класса **Math,** без объявления объекта:

```
float z = Math.max(x, y);
double rd = Math.random(); // случайное значение
```

Статический метод можно вызывать также с использованием имени объекта, но такой вызов не будет логически корректным, хотя и не приведет к ошибке компиляции.

Переопределение статических методов класса не имеет практического смысла, так как обращение к статическому атрибуту или методу осущест-

вляется по большей части посредством задания имени класса, которому они принадлежат.

Модификатор final

Модификатор **final** используется для определения констант. Методы, объявленные как **final**, нельзя замещать в подклассах. Например:

```
/* пример # 5 : final-поля и методы : B.java */
class A {
      final int T = -273;
      public final void method() {
            System.out.println("final метод");
            T = 0;//ошибка!
      }
}
class B extends A {
      public void method(){ // ошибка!
       }
}
```

Константа может быть объявлена как поле класса, но не проинициализирована. В этом случае она должна быть проинициализирована в логическом блоке или конструкторе. Значение по умолчанию константа получить не может в отличие от атрибутов с прочими модификаторами. Константы могут объявляться в методах как локальные или как параметры метода. В обоих случаях значения таких констант изменять нельзя.

```
/* пример # 6 : инициализация final-полей и их ис-
пользование : DemoFinalFields.java */
public class DemoFinalFields {
      final int NUM; //неинициализированная константа
      public DemoFinalFields() {
      //инициализация в конструкторе
            NUM = 10; //только один раз!!!
      }
      //или инициализация в логическом блоке
            //{ NUM = 10; }
      static int initFinal() {
            //объявление и использование
            final int N = 1;
            return N + 1;
      }
      static char finalParam(final char c) {
//c = '\u0256'; нельзя изменять значение константы
```

```
            return (char) (c + initFinal());
      }
      public static void main(String[] args) {
            char ch = '\u0041'; // ch = 'A';
System.out.println("-> " + finalParam(ch)); // -> C
      DemoFinalFields df = new DemoFinalFields();
            System.out.println(df.NUM + 5); // 15
      }
}
```

Абстрактные методы

Абстрактные методы размещаются в абстрактных классах или интерфейсах, тела у таких методов отсутствуют и должны быть реализованы в подклассах.

```
/* пример # 7 : абстрактный класс и метод :
AbstractClass.java */
public abstract class AbstractClass{
      public abstract void abstractMethod();
}
```

Абстрактный класс может содержать и абстрактные, и неабстрактные методы, может и не содержать ни одного абстрактного метода.

Модификатор native

Приложение на языке Java может вызывать методы, написанные на языке C++. Такие методы объявляются с ключевым словом **native**, которое сообщает компилятору, что метод реализован в другом месте. Например:

```
public native int outFunction(int num);
```

Методы, помеченные **native**, можно переопределять обычными методами в подклассах.

Модификатор synchronized

При использовании нескольких потоков необходимо синхронизировать методы, обращающиеся к общим данным. Когда интерпретатор обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении. Вызов методов **notify()**, **notifyAll()**, **wait()** класса **Object** (суперкласса для всех классов языка Java) предполагает использование модификатора **synchronized**, так как эти методы предназначены для работы с потоками.

Передача объектов в методы

Объекты в методы передаются по ссылке, поэтому если в методе изменить значение поля объекта, то это изменение коснется исходного объекта.

```
/* пример # 8 : передача по ссылке : DemoRef.java */
class Point implements Cloneable{
      private int x = 1;
      public int getX() {
            return x;
      }
      public void setX(int value) {
            x = value;
      }
}
public class DemoRef {
      private static void meth(Point p) {
            p.setX(10);
      }
      public static void main(String[] args) {
            Point point = new Point();
            System.out.println("x = " + point.getX());
            meth(point);
            System.out.println("x = " + point.getX());
      }
}
```

В результате будет выведено:

```
x = 1
x = 10
```

Вызов одних методов класса из других методов этого же класса возможен без имени объекта или класса. Кроме того, в методе **meth()** ссылка **p** указывает на тот же самый объект, что и ссылка **point** в методе **main()**. Чтобы избежать подобных ситуаций, следует создавать в методе локальную копию объекта в виде:

```
Point pCopy = (Point)p.clone();
```

Для этого необходимо реализовать пустой интерфейс **Cloneable**, а также переопределить в классе **Point** метод **clone()**:

```
public Object clone()
            throws CloneNotSupportedException {
            return super.clone();
      }
```

Логические блоки

При описании класса могут быть использованы логические блоки. Логическим блоком называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса.

```
{ /* код */ }
```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов как текущего класса, так и не принадлежащих ему. При создании объекта класса они вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса. Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки **this**, которая, как и в C++, представляет собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static**. В этом случае он вызывается всегда перед вызовом статического метода данного класса.

```
/* пример # 9 : использование логических блоков при
объявлении класса : DemoLogic.java */
public class DemoLogic {
      {
      System.out.println("логический блок(1) x="
            + this.x);
      }
private int x = 1;
    public DemoLogic() {
        System.out.println("конструктор");
           }
    int getX(){
           return x;
           }
      {
      x = 2;
      System.out.println("логический блок(2) x=" + x);
      }
      public static void main(String[] args) {
            DemoLogic obj = new DemoLogic();
            System.out.println("значение x="
```

```
                + obj.getX());
        }
}
```

В результате выполнения этой программы будет выведено:

```
логический блок(1) x=0
логический блок(2) x=2
конструктор
значение x=2
```

В первой строке вывода поле **x** получит значение по умолчанию, так как память для него выделена при создании объекта, а значение еще не проинициализировано. Во второй строке выводится значение **x** равное **2**, так как после инициализации атрибута класса был вызван логический блок, изменивший его значение.

Классы-шаблоны в J2SE 5.0

С помощью шаблонов можно создавать родовые (generic) классы и методы, что позволяет использовать более строгую типизацию, например при работе с коллекциями. Применение классов-шаблонов для создания типизированных коллекций будет рассмотрено в главе 10.

Пример класса-шаблона с двумя параметрами:

```
class MyTempl < T1, T2 > {
      T1 value1;
      T2 value2;
      }
```

Здесь **T1**, **T2** – фиктивные типы, которые используются при объявлении атрибутов класса. Компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект. Объект класса **MyTempl** можно создать, например, следующим образом:

```
MyTempl < Integer, Byte > ob =
            new MyTempl < Integer, Byte > ();
```

В предложенном примере полностью приведено объявление класса **CurrentType** с конструкторами и методами.

```
/* пример # 10 : создание и использование объектов
параметризованного класса : DemoTemplate.java */
class CurrentType < T > {
      T value;
      CurrentType(){}
      CurrentType(T value){
            this.value = value;
      }
```

```
      T getValue() {
            return this.value;
      }
      void setValue (T value) {
            this.value = value;
      }
}
public class DemoTemplate {
      public static void main(String [] args){
       CurrentType < Integer > ob1;
            ob1 = new CurrentType < Integer > ();
            ob1.setValue(new Integer(7));
       System.out.println(ob1.getValue());
       CurrentType < String > ob2 =
            new CurrentType <String> ("Java");
       System.out.println(ob2.getValue());
      }
}
```

В классе **DemoTemplate** созданы два объекта: **ob1** на основе типа **Integer** и **ob2** на основе типа **String** при помощи различных конструкторов.

Можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
class MyTemplExt < T extends Number > {
      T value;
      }
```

Такая запись говорит о том, что в качестве типа **T** разрешено применять только классы, являющиеся наследниками класса **Number**.

В качестве параметров классов запрещено применять базовые типы.

Методы-шаблоны в J2SE 5.0

Параметризованный метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра, и может быть записан, например, в виде:

```
< T extends Type > returnType methodName(T arg) {}
< T > returnType methodName(T arg) {}
```

Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после **extends**. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

```
/* пример # 11 : параметризованный метод :
GenericMethod.java */
class GenericMethod {
public static <T extends Number> byte asByte(T num) {
                return num.byteValue();
          }
      public static void main(String [] args) {
      System.out.println(asByte(new Integer(7)));
      System.out.println(asByte(new Float(7.f)));
      System.out.println(
asByte(new Character('7')));/* ошибка компиляции */
      }
}
```

Объекты типа **Integer** и **Float** являются подклассами абстрактного класса **Number**, поэтому компиляция проходит без затруднений. Класс **Character** не обладает вышеуказанным свойством, и его объект не может передаваться в метод **asByte(T num)**.

Методы с переменным числом параметров в J2SE 5.0

Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод.

```
/* пример # 12 : определение количества аргументов
метода : DemoVarargs.java */
class DemoVarargs {
 public static int getArgCount(Object...args) {
  for(int i = 0; i < args.length; i++)
   System.out.println("Arg #" + i
           + ": " + args[i].toString());
  return args.length;
 }
 public static void main(String args[]) {
  System.out.println(
     getArgCount(7, "No", new Boolean("TruE")));
   Integer [] i = {1, 2, 3, 4, 5};
  System.out.println(getArgCount(i));
 }
}
```

В примере приведен простейший метод с переменным числом параметров. Метод **getArgCount()** выводит все переданные ей аргументы

и возвращает их количество. При передаче параметров в метод из них автоматически создается массив. Второй вызов метода в примере позволяет передать в метод массив.

Чтобы передать массив в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип2[]... args){}
```

Можно попытаться передать в метод два массива или массив с объектом:

```
/* пример # 13 : передача двух массивов :
DemoVarArgs2.java */
class DemoVarArgs2 {
 //   static int showArg(Integer...args) {
      static int showArg(Object...args) {
  for(int i = 0; i < args.length; i++)
   System.out.println("Arg #" + i
           + ": " + args[i].toString());
  return args.length;
 }
 public static void main(String args[]) {
  Integer i[] = {0, 3, 5, 7};
  System.out.println(showArg(i, i));
  System.out.println(showArg(i, 5));
     }
}
```

Ошибка возникает при попытке передачи двух параметров разных типов **Integer** и **Integer[]**. Если же в объявлении метода параметр **Integer** изменить на **Object**, ссылка на который может принять и объект, и массив, то компиляция и выполнение кода пройдут без ошибок.

Методы с переменным числом аргументов могут быть перегружены:

```
void show(Integer...args) {}
void show(String...args) {}
```

Не существует также ограничений и на переопределение подобных методов.

Единственным ограничением является то, что параметр вида **Object...args** должен быть последним в объявлении метода, например:

```
void methodName(Тип1 obj, Тип2... args){}
```

Задания к главе 3

Вариант А

1. Определить класс **Vector** размерности n. Реализовать методы сложения, вычитания, умножения, инкремента, декремента, индек-

сирования. Определить массив из m объектов. Каждую из пар векторов передать в методы, возвращающие их скалярное произведение и длины. Вычислить и вывести углы между векторами.

2. Определить класс **Vector** размерности n. Определить несколько конструкторов. Реализовать методы для вычисления модуля вектора, скалярного произведения, сложения, вычитания, умножения на константу. Объявить массив объектов. Написать метод, который для заданной пары векторов будет определять, являются ли они коллинеарными или ортогональными.
3. Определить класс **Vector** в R^3. Реализовать методы для проверки векторов на ортогональность, проверки пересечения неортогональных векторов, сравнения векторов. Создать массив из m объектов. Определить, какие из векторов компланарны.
4. Определить класс **Matrix** размерности (n x n). Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения матриц. Объявить массив объектов. Создать методы, вычисляющие первую и вторую нормы матрицы $\| a \|_1 = \max\limits_{1 \le i \le n} \sum\limits_{j=1}^{n} (a_{ij}), \|a\|_2 = \max\limits_{1 \le j \le n} \sum\limits_{i=1}^{n} (a_{ij})$. Определить, какая из матриц имеет наименьшую первую и вторую нормы.
5. Определить класс **Matrix** размерности (m x n). Класс должен содержать несколько конструкторов. Объявить массив объектов. Передать объекты в метод, меняющий местами строки с максимальным и минимальным элементами k-го столбца. Создать метод, который изменяет i-ю матрицу путем возведения ее в квадрат.
6. Определить класс **Цепная дробь** $A = a_0 + \cfrac{x}{a_1 + \cfrac{x}{a_2 + \cfrac{x}{a_3 + \ldots}}}$

   Определить методы сложения, вычитания, умножения, деления. Вычислить значение для заданного n, x, a[n].
7. Определить класс **Дробь** в виде пары (m,n). Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения и деления дробей. Объявить массив из k дробей, ввести/вывести значения для массива дробей. Создать массив объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента массива.
8. Определить класс **Complex**. Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания,

умножения, деления, присваивания комплексных чисел. Создать два вектора размерности n из комплексных координат. Передать их в метод, который выполнит их сложение.

9. Определить класс **Квадратное уравнение**. Класс должен содержать несколько конструкторов. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать массив объектов и определить наибольшие и наименьшие по значению корни.
10. Определить класс **Булева матрица** (**BoolMatrix**) размерности (n x m). Класс должен содержать несколько конструкторов. Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.
11. Построить класс **Булев вектор** (**BoolVector**) размерности n. Определить несколько конструкторов. Реализовать методы для выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.
12. Определить класс **Множество символов** мощности n. Написать несколько конструкторов. Реализовать методы для определения принадлежности заданного элемента множеству; пересечения, объединения, разности двух множеств. Создать методы сложения, вычитания, умножения (пересечения), индексирования, присваивания. Создать массив объектов и передавать пары объектов в метод другого класса, который строит множество, состоящее из элементов, входящих только в одно из заданных множеств.
13. Определить класс **Polynom** степени n. Создать методы для сложения и умножения объектов. Объявить массив из m полиномов и передать его в метод, вычисляющий сумму полиномов массива. Определить класс **Rational Polynom** с полем типа **Polynom**. Определить метод для сложения: $R = \frac{p_1(x)}{Q_1(x)} + \frac{p_2(x)}{Q_2(x)}$ и методы для ввода/вывода.
14. Определить класс **Нелинейное уравнение** для двух переменных. Написать несколько конструкторов. Создать методы для сложения и умножения объектов. Реализовать метод определения корней методом бисекции.

Вариант В

Создать классы, спецификации которых приведены ниже. Определить конструктор и методы **`set`**`Тип()`, **`get`**`Тип()`, **`show`**`Информ()`. Реали-

зовать класс в консольном приложении. Определить дополнительно методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль.

1. **Student**: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа.
   Создать массив объектов. Вывести:
   a) список студентов заданного факультета;
   b) списки студентов для каждого факультета и курса;
   c) список студентов, родившихся после заданного года;
   d) список учебной группы.
2. **Customer**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета.
   Создать массив объектов. Вывести:
   a) список покупателей в алфавитном порядке;
   b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.
3. **Patient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, диагноз.
   Создать массив объектов. Вывести:
   a) список пациентов, имеющих данный диагноз;
   b) список пациентов, номер медицинской карты у которых находится в заданном интервале.
4. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.
   Создать массив объектов. Вывести:
   a) список абитуриентов, имеющих неудовлетворительные оценки;
   b) список абитуриентов, средний балл у которых выше заданного;
   c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).
5. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.
   Создать массив объектов. Вывести:
   a) список книг заданного автора;
   b) список книг, выпущенных заданным издательством;
   c) список книг, выпущенных после заданного года.
6. **House**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.
   Создать массив объектов. Вывести:
   a) список квартир, имеющих заданное число комнат;
   b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
   c) список квартир, имеющих площадь, превосходящую заданную.

7. **Phone**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров.
   Создать массив объектов. Вывести:
   a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;
   b) сведения об абонентах, которые пользовались междугородной связью;
   c) сведения об абонентах в алфавитном порядке.
8. **Car**: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.
   Создать массив объектов. Вывести:
   a) список автомобилей заданной марки;
   b) список автомобилей заданной модели, которые эксплуатируются больше n лет;
   c) список автомобилей заданного года выпуска, цена которых больше указанной.
9. **Product**: id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество.
   Создать массив объектов. Вывести:
   a) список товаров для заданного наименования;
   b) список товаров для заданного наименования, цена которых не превосходит заданную;
   c) список товаров, срок хранения которых больше заданного.
10. **Train**: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс).
    Создать массив объектов. Вывести:
    a) список поездов, следующих до заданного пункта назначения;
    b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;
    c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.
11. **Bus**: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег.
    Создать массив объектов. Вывести:
    a) список автобусов для заданного номера маршрута;
    b) список автобусов, которые эксплуатируются больше 10 лет;
    c) список автобусов, пробег у которых больше 100000 км.
12. **Aeroflot**: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.
    Создать массив объектов. Вывести:
    a) список рейсов для заданного пункта назначения;

b) список рейсов для заданного дня недели;
c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

Тестовые задания к главе 3

Вопрос 3.1.

Дан код:

```
public class Quest1{
 static int i;
 public static void main(String[] args){
 System.out.print(i);
 }}
```

В результате при компиляции и запуске будет выведено:

1) ошибка компиляции: переменная `i` использована до инициализации;
2) `null`;
3) 1;
4) 0.

Вопрос 3.2.

Какие из ключевых слов могут быть использованы при объявлении конструктора?

1) `private;`
2) `final;`
3) `native;`
4) `abstract;`
5) `protected.`

Вопрос 3.3.

Как следует вызвать конструктор класса **`Quest3`**, чтобы в результате выполнения кода была выведена на консоль строка "Конструктор".

```
public class Quest3 {
  Quest3 (int i){ System.out.print("Конструктор");
}
    public static void main(String[] args){
      Quest3 s= new Quest3();
      //1
    }
 public Quest3()     {
      //2
    }
```

```
    {
    //3
  } }
```

1) вместо //1 написать `Quest3(1);`
2) вместо //2 написать `Quest3(1);`
3) вместо //3 написать **`new`** `Quest3(1);`
4) вместо //3 написать `Quest3(1).`

Вопрос 3.4.

Какие из следующих утверждений истинные?

1) nonstatic-метод не может быть вызван из статического метода;
2) static-метод не может быть вызван из нестатического метода;
3) private-метод не может быть вызван из другого метода этого класса;
4) final-метод не может быть статическим.

Вопрос 3.5.

Дан код:

```
public class Quest5 {
          {System.out.print("1");}
   static{System.out.print("2");}
     Quest5(){System.out.print("3");}
     public static void main(String[] args) {
           System.out.print("4");
     } }
```

В результате при компиляции и запуске будет выведено:

1) 1234;
2) 4;
3) 34;
4) 24;
5) 14.

Вопросы:

1. Какие свойства имеет метод класса, если он объявлен как **`public final`**.
2. Объяснить отличия базовых типов Java от C++. Чем отличаются типы **`boolean`** и **`Boolean`**, **`int`** и **`Integer`**, **`String`** и **`char[]`**?
3. Изменение каких данных из одного экземпляра класса влечет их изменение для всех экземпляров класса?
4. Какие свойства имеет метод, если он объявлен как **`public static`**?

5. Как вызвать метод класса, объявленный со спецификатором **`private`**?
6. Как создать объект класса, если он имеет единственный конструктор, объявленный со спецификатором **`private`**?

Глава 4

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ. КЛАСС Object

Наследование

Класс (подкласс) может наследовать переменные и методы другого класса (суперкласса), используя ключевое слово **extends**. Подкласс имеет доступ ко всем открытым переменным и методам (кроме **private**) родительского класса, как будто они находятся в подклассе. В то же время подкласс может иметь методы с тем же именем, параметрами и возвращаемым значением, что и методы суперкласса. В этом случае подкласс переопределяет методы родительского класса. Это часть механизма ООП, который называется полиморфизмом. В следующем примере переопределяемый метод **show()** находится в двух классах **Bird** и **Eagle**. По принципу полиморфизма вызывается метод, наиболее близкий к текущему объекту.

```
/* пример # 1 : наследование класса и переопределение
метода : BirdSample.java */
class Bird {
      private String name;
      private float price;
public Bird(float p, String str) { //конструктор
      name = str;
      price = p;
      }
public float getPrice(){
      return price;
      }
public String getName(){
      return name;
      }
void show(){
    System.out.println("название: " + name
          + ", цена: " + price);
      }
}
class Eagle extends Bird {
    private boolean fly;
```

```
public Eagle(float p, String str, boolean f) {
 super(p, str); //вызов конструктора суперкласса
 fly = f;
}
void show(){
System.out.println("название:" + getName()
      + ", цена: " + getPrice() + ", полет:" + fly);
   }
}
public class BirdSample {
        public static void main(String[] args) {
      Bird b1 = new Bird(0.85F, "Гусь");
      Bird b2 = new Eagle(10.55F, "Белый Орел", true);
      b1.show(); // вызов show() класса Bird
      b2.show(); // вызов show() класса Eagle
  }
}
```

Объект **b1** создается при помощи вызова конструктора класса **Bird**, и, соответственно, при вызове метода **show()** вызывается версия метода из класса **Bird**. При создании объекта **b2** ссылка типа **Bird** инициализируется объектом типа **Eagle**. При таком способе инициализации ссылка на суперкласс получает доступ к методам, переопределенным в подклассе.

При объявлении совпадающих по сигнатуре полей в суперклассе и подклассах их значения не переопределяются и никак не пересекаются, то есть существуют в одном объекте независимо друг от друга. В этом случае задача извлечения требуемого значения определенного поля, принадлежащего классу в цепочке наследования, ложится на программиста.

```
/* пример # 2 : доступ к полям с одинаковыми именами
при наследовании : DemoAB.java */
class A {
 int x = 1, y = 2;
    public A() {
        y = getX();
    System.out.println("в классе A после вызова"
+   " getX()  x=" + x + "  y=" + y);
    }
public int getX(){
     System.out.println("в классе A");
    return x;
     }
}
```

```
class B extends A {
 int x = 3, y = 4;
    public B() {
     System.out.println("в классе B  x=" + x
                + "  y=" + y);
    }
public int getX(){
     System.out.println("в классе B");
    return x;
     }
}
public class DemoAB {
    public static void main (String[] args) {
        A objA = new B();
        B objB = new B();
System.out.println(objA.x);
System.out.println(objB.x);
     }
}
```

В результате выполнения данного кода последовательно будет выведено:

```
в классе B
в классе A после вызова getX()  x=1  y=0
в классе B  x=3  y=4
в классе B
в классе A после вызова getX()  x=1  y=0
в классе B  x=3  y=4
x=1
x=3
```

В случае создания объекта **objA** инициализацией ссылки на класс **A** объектом класса **B** был получен доступ к полю **x** класса **A**. Во втором случае при создании объекта **objB** класса **B** был получен доступ к полю **x** класса **B**. Однако, воспользовавшись преобразованием типов вида: **((B)objA).x** или **((A)objB).x**, легко можно получить доступ к полю **x** из соответствующего класса.

Одну из сторон полиморфизма методов иллюстрирует конструктор класса **A** в виде:

```
public A() {
   y = getX();
        }
```

Метод **getX()** содержится как в классе **A**, так и в классе **B**. При создании объекта класса **B** одним из способов:

```
A objA = new B();
B objB = new B();
```

в любом случае сначала вызывается конструктор класса **A**. Но так как создается объект класса **B**, то вызывается метод **getX()**, соответственно принадлежащий классу **B**, который в свою очередь оперирует полем **x**, еще не проинициализированным для класса **B**. В результате **y** получит значение **x** по умолчанию, т.е. нуль.

Нельзя создать подкласс для класса, объявленного со спецификатором **final**:

```
// класс First не может быть суперклассом
final class First {/*код*/}
// следующий класс невозможен
class Second extends First{/*код*/}
```

Использование super и this

Ключевое слово **super** используется для вызова конструктора суперкласса и для доступа к члену суперкласса. Например:

```
super(список_параметров); /* вызов конструктора суперкласса с передачей параметров или без нее*/
super.i = n; /* обращение к атрибуту суперкласса */
super.methodName(); // вызов метода суперкласса
```

Вторая форма **super** подобна ссылке **this** на экземпляр класса. Третья форма специфична для Java и обеспечивает вызов переопределенного метода, причем если в суперклассе этот метод не определен, то будет осуществляться поиск по цепочке наследования до тех пор, пока метод не будет найден. Каждый экземпляр класса имеет неявную ссылку **this** на себя, которая передается также и методам. После этого можно, например, вместо артибута **price** писать **this.price,** хотя и не обязательно.

Следующий код показывает, как, используя **this**, можно строить одни конструкторы на основе других.

```
// пример # 3 : this в конструкторе : Locate3D.java
class Locate3D {
  private int x, y, z;
 public Locate3D(int x, int y, int z) {
   this.x = x;
   this.y = y;
   this.z = z;
          }
 public Locate3D() {
     this(-1, -1, -1);
          }
}
```

В этом классе второй конструктор для завершения инициализации объекта обращается к первому конструктору. Такая конструкция применяется в случае, когда в классе имеется несколько конструкторов и требуется добавить конструктор по умолчанию.

Ссылка **this** используется в методе для уточнения того, о каких именно переменных **x** и **y** идет речь в каждом отдельном случае, а конкретно для доступа к переменной класса из метода, если в методе есть локальная переменная с тем же именем. Инструкция **this()** должна быть единственной в вызывающем конструкторе и быть первой по счету выполняемой операцией.

Переопределение методов и полиморфизм

Способность Java делать выбор метода, исходя из типа времени выполнения, называется динамическим полиморфизмом. Поиск метода происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не достигнут **Object** – суперкласс для всех классов.

Отметим, что статические методы могут быть переопределены в подклассе, но не могут быть полиморфными, так как их вызов не затрагивает объекты.

Если два метода с одинаковыми именами и возвращаемыми значениями находятся в одном классе, то списки их параметров должны отличаться. Такие методы являются перегружаемыми (overloading). Если метод подкласса совпадает с методом суперкласса (порождающего класса), то метод подкласса переопределяет (overriding) метод суперкласса. Все методы Java являются виртуальными (ключевое слово virtual, как в C++, не используется). Переопределение методов является основой концепции динамического связывания, реализующей полиморфизм. Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию метода вызвать, основываясь на типе объекта, на который имеется ссылка. Таким образом, тип объекта определяет версию метода на этапе выполнения. В следующем примере рассматривается реализация полиморфизма на основе динамического связывания. Так как суперкласс содержит методы, переопределенные подклассами, то объект суперкласса будет вызывать методы различных подклассов, в зависимости от того, на объект какого подкласса у него имеется ссылка.

```
/* пример # 4 : динамическое связывание методов :
DynDispatch.java */
class A {
    int i, j;
public A(int a, int b) {
 i = a;
```

```
 j = b;
 }
void show() { // вывод i и j
     System.out.println("i и j: " + i + " " + j);
      }
}
class B extends A {
    int k;
public B(int a, int b, int c) {
      super(a, b);
      k = c;
             }
   void show() {
 /* вывод k: переопределенный метод show() из A */
     super.show(); // вывод значений из A
     System.out.println("k: " + k);
   }
}
class C extends B {
   int m;
public C(int a, int b, int c, int d) {
      super(a, b, c);
      m = d;
        }
void show() {
/* вывод m: переопределенный метод show() из B */
    super.show(); //вывод значений из B
      // show();
/*нельзя!!! метод будет вызывать сам себя, что приве-
дет к ошибке во время выполнения */
    System.out.println("m: " + m);
       }
  }
public class DynDispatch {
    public static void main(String[] args) {
      A Aob;
      B Bob = new B(1, 2, 3);
      C Cob = new C(5, 6, 7, 8);
      Aob = Bob; // установка ссылки на Bob
  Aob.show(); // вызов show() из B
  System.out.println();
     Aob = Cob; // установка ссылки на Cob
  Aob.show(); // вызов show() из C
       }
 }
```

Результат:

```
i и j: 1 2
k: 3
i и j : 5 6
k:7
m: 8
```

Следует помнить, что при вызове **show()** обращение **super** всегда происходит к ближайшему суперклассу.

Перегрузка методов

Метод может быть перегруженным, т.е. существует несколько его версий с одним и тем же именем, но с различным списком параметров. Перегрузка может ограничиваться одним классом или несколькими классами, но обязательно находящимися в одной цепочке наследования. Следует отметить, что статические методы могут перегружаться нестатическими и наоборот.

При вызове перегруженных методов следует избегать ситуаций, когда компилятор будет не в состоянии выбрать тот или иной метод, как, например, в случае:

```
/* пример # 5 : вызов перегруженного метода :
DemoCD.java */
   class ClassC {}
   class ClassD extends ClassC{}
   public class DemoCD {
     static void show(ClassC obj1, ClassD obj2){
     System.out.println(
          "первый метод show(ClassC, ClassD)");
     }
     static void show(ClassD obj1, ClassC obj2){
     System.out.println(
          "второй метод show(ClassD, ClassC)");
     }
     static void show(Object obj1, Object obj2){
     System.out.println(
          "третий метод show(Object, Object)");
     }
     public static void main(String[] args) {
          ClassC c = new ClassC();
          ClassD d = new ClassD();
          Object ob= new Object();
          show(c,d);//1_первый метод
```

```
            show(d,c);//2_второй метод
            show(c,c);//3_третий метод
      //show(d,d);// 4_ошибка компиляции
            show(ob, ob);//5_третий метод
            show(c,ob);//6_третий метод
            show(ob,d);//7_третий метод
      }
}
```

В первом, втором и пятом случаях передаваемые параметры в метод **show()** полностью совпадают с параметрами при объявлении метода. В третьем случае первый и второй методы не годятся для использования, так как одним из параметров этих методов является объект класса **ClassD**, а определение вызываемого метода идет вверх по цепочке наследования для параметров, поэтому в данном случае будет вызван метод с параметрами типа **Object**. Аналогичная ситуация возникает в шестом и седьмом случаях. В четвертом случае все три метода **show()** одинаково подходят для вызова, поэтому возникнет ошибка компиляции. Чтобы избежать неопределенности, следует использовать явное преобразование типов, например:

```
show(d,(ClassC)d);
show(d,(Object)d);
```

Каждый из вариантов вызовет в итоге соответствующий ему метод **show()**.

В следующем примере экземпляр подкласса создается с помощью **new,** ссылка на него передается объекту суперкласса. При вызове из суперкласса соответственно вызывается метод подкласса.

```
/* пример # 6 : динамический вызов метода :
Dispatch.java */
class A {
     void myMethod() {/* private и protected исполь-
зовать нельзя, так как метод при наследовании стано-
вится недоступным*/
       System.out.println("метод класса A");
      }
      void myMethod(int i) {
System.out.println("метод класса A с аргументом");
      }
}
class B extends A {
      void myMethod(int i) {
System.out.println("метод класса B с аргументом");
            }
    }
public class C extends B {
      {
```

```
      System.out.println("класс C");
      }
      void myMethod() {
            System.out.println("метод класса C");
            }
}
public class Dispatch {
      public static void main(String[] args) {
            A obj1 = new B();
            obj1.myMethod();
            A obj2 = new C();
            obj2.myMethod();
            obj2.myMethod(10);
      }
}
```

В результате будет выведено:

```
метод класса A
класс C
метод класса C
метод класса B с аргументом
```

При первом обращении вызывается метод **myMethod()** из класса **A** как унаследованный. При втором обращении вызывается метод **myMethod()** из класса **C** как переопределенный. В последнем случае вызывается метод **myMethod(int i)** из класса **B** как унаследованный.

Полиморфизм и расширяемость

В объектно-ориентированном программировании применение наследования предоставляет возможность расширения и дополнения программного обеспечения, имеющего сложную структуру с большим количеством классов и методов. В задачи базового класса в этом случае входит определение интерфейса (как способа взаимодействия) для всех наследников.

В следующем примере приведение к базовому типу происходит в выражении:

```
Stone s1 = new White();
Stone s2 = new Black();
```

Базовый класс **Stone** предоставляет общий интерфейс для своих классов-наследников. Порожденные классы перекрывают эти определения для обеспечения уникального поведения.

```
// пример # 7 : полиморфизм : StoneRandom.java
class Stone {
   public void info() {/*пустая реализация*/}
}
```

```
class White extends Stone {
  public void info() {
System.out.println("добавлен белый каменный шар");
      }
}
class Black extends Stone {
 public void info() {
System.out.println("добавлен черный каменный шар");
      }
}
public class StoneRandom {
   public static Stone randStone() {
   switch((int)(Math.random() * 2)){
     case 0:  return new Black();
     case 1:  return new White();
      default: return null;
/*if((int)(Math.random() * 2)==0) return new Black();
else return new White();как альтернативный и не очень
удачный вариант. Почему? */
    }
}
public static void main(String[] args) {
   Stone[] s = new Stone[15];
  for(int i = 0; i < s.length; i++)
/* заполнение массива камнями */
     s[i] = randStone();
  for(int i = 0; i < s.length; i++)
     s[i].info();// вызов полиморфного метода
     }
}
```

Главный класс **StoneRandom** содержит **static** метод **randStone()**, который возвращает ссылку на случайно выбранный объект подкласса класса **Stone** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **Black** или **White**. Метод **main()** содержит массив из ссылок **Stone,** заполненный вызовами **randStone()**. На этом этапе известно, что имеется некоторое множество ссылок на объекты базового типа и ничего больше (не больше, чем знает компилятор). Когда происходит перемещение по этому массиву, метод **info()** вызывается для каждого случайным образом выбранного объекта.

Если понадобится в дальнейшем добавить систему, например класс **Green**, то это потребует только переопределения метода **info()** и добавления одной строки в код метода **randStone()**, что делает систему легко расширяемой.

Статические методы и полиморфизм

К статическим методам принципы полиморфизма неприменимы. При использовании ссылки для доступа к статическому члену компилятор при выборе метода или поля учитывает тип ссылки, а не тип объекта, ей присвоенного.

```
/* пример # 8 : поведение статического метода :
StaticDemo.java */
class StaticA {
      public static void show(){
            System.out.println(
            "метод show() из StaticA");
      }
}
class StaticB extends StaticA {}
class StaticC extends StaticB {
      public static void show(){
            System.out.println(
            "метод show() из StaticC");
      }
}
public class StaticDemo {
      public static void main(String[] args) {
            StaticA s1 = new StaticC();
            StaticB s2 = new StaticC();
            StaticC s3 = new StaticC();
            s1.show();
            s2.show();
            s3.show();
      }
}
```

В результате выполнения данного кода будет выведено:

```
метод show() из StaticA
метод show() из StaticA
метод show() из StaticC
```

При таком способе инициализации объектов **s1** и **s2**, метод **show()** будет вызван из суперклассов **StaticA** и **StaticB** соответственно. Для объекта **s3** будет вызван собственный метод **show()**, что следует из способа объявления объекта. Если же спецификатор **static** убрать из объявления методов, то вызовы методов будут осуществляться в соответствии с принципами полиморфизма, т.е. будет вызван метод из класса **StaticC**.

Класс Object

На вершине иерархии классов находится класс **Object**. Ссылочная переменная типа **Object** может обращаться к объекту любого другого класса, кроме того, переменная типа **Object** может указывать на любой массив, так как массивы реализуются как классы. В классе **Object** определен набор методов, который наследуется всеми классами. Следует отметить два метода: **equals()** и **toString()**. Метод **equals()** при сравнении двух объектов возвращает истину, если объекты эквивалентны, и ложь – в противном случае. Если требуется сравнивать объекты класса, созданного программистом, то этот метод необходимо переопределять в этом классе. Метод **toString()** возвращает строку с описанием объекта в виде:

```
getClass().getName() + '@' +
      Integer.toHexString(hashCode())
```

Метод вызывается автоматически, когда объект выводится методами **println()**, **print()** и некоторыми другими.

Метод **hashCode()** переопределен, как правило, в каждом классе Java и возвращает число, являющееся уникальным идентификатором объекта, зависящем в большинстве случаев только от значения объекта. Метод **hashCode()** возвращает хэш-код объекта, который вычисляется по принципу – различные по содержанию объекты одного и того же типа имеют различные хэш-коды, с другой стороны, принято соглашение “все равные по значению объекты одного типа имеют одинаковые хэш-коды”.

При создании классов также рекомендуется переопределять методы **hashCode()** и **toString()**, чтобы адаптировать их действия для создаваемого типа.

```
/* пример # 9 : переопределение методов equals() и
toString() : Point.java */
class Point {
      protected byte b;
      protected String str;
      public Point(byte n, String s) {
            b = n;
            str = s;
      }
      public Point() {
            this((byte)0, "NoName");
      }
      public boolean equals(Object obj) {
            if (obj instanceof Point)
```

```
      return (this.b == ((Point) obj).b) &&
            (str.equals(((Point) obj).str));
            return false;
      }
      public String toString() {
            return getClass().getName() + "@"
                  + " name=" + str + "  b=" + b;
      }
}
class PointZ extends Point{
      short s = 100;
}
```

Метод **equals()** переопределяется для класса **Point** таким образом, чтобы убедиться в том, что полученный объект является объектом типа **Point** или одним из его наследников, а также сравнить содержимое полей **b** и **str,** соответственно у вызывающего и передаваемого объектов. Метод **toString()** переопределен таким образом, что кроме стандартной информации о пакете, в котором находится класс **Point,** и самого имени класса, выводит значения полей объекта, вызвавшего этот метод, вместо хэш-кода, как это делается в классе **Object**.

Следует обратить внимание на вызов одного конструктора из другого с передачей ему параметров, а также на преобразование значения типа **int** к типу **byte**, так как данное преобразование не выполняется по умолчанию из-за возможной потери информации.

```
/* пример # 10 : иллюстрация работы методов equals()
и toString() : PointDemo.java */
public class PointDemo {
      public static void main(String[] args) {
            Point p1 = new Point((byte) 1, "Петров");
            Point p2 = new Point((byte) 1, "Петров");
            PointZ p3 = new PointZ();
            Point p4 = new Point();
            System.out.println(p1.equals(p2));
            System.out.println(p1.equals(p3));
            System.out.println(p4.equals(p3));
            System.out.println(p3.equals(p4));
            System.out.println(p1.toString());
      }
}
```

В результате выполнения данного кода будет выведено следующее:

```
true
false
```

```
true
true
com.mypack.Point@ name=Петров  b=1
```

Переопределенный таким образом метод **equals()** позволяет сравнивать объекты суперкласса с объектами подклассов, но только по тем полям, которые являются общими.

"Сборка мусора"

Так как объекты создаются динамически с помощью операции **new**, то желательно знать механизм ликвидации объектов и способ освобождения памяти для более позднего перераспределения. Java автоматически выполняет освобождение памяти, занимаемой объектом, с помощью механизма "сборки мусора". Когда никаких ссылок на объект не существует, то есть все ссылки на него вышли из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена. "Сборка мусора" происходит нерегулярно во время выполнения программы. Форсировать "сборку мусора" невозможно, можно лишь "рекомендовать" ее выполнить вызовом метода **gc()**, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным.

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций используется механизм finalization. Чтобы использовать finalization, необходимо определить метод **finalize()**. Виртуальная машина вызывает этот метод всегда, когда она собирается уничтожить объект данного класса. Внутри метода **finalize()** нужно определить действия, которые должны быть выполнены до уничтожения объекта. Непосредственно перед освобождением памяти для объекта вызывается метод **finalize()**.

Метод **finalize()** имеет следующую сигнатуру:

```
protected void finalize(){
    // код завершения
}
```

Ключевое слово **protected** запрещает доступ к **finalize()** кодам, определенным вне этого класса. Метод **finalize()** вызывается только перед самой "сборкой мусора", а не тогда, когда объект выходит из области действия идентификаторов, то есть невозможно определить, когда **finalize()** будет выполнен. В принципе, этот метод может быть вообще не выполнен.

```
/* пример # 11 : демонстрация сборки мусора :
FinalizeDemo.java */
```

```
class Demo {
      private int a;
      public Demo(int a) {
            this.a = a;
      }
      protected void finalize() {
      System.out.println("объект удален, a=" + a);
      }
}
public class FinalizeDemo {
      public static void main(String[] args) {
            Demo d1 = new Demo(1);
            d1 = null;
            Demo d2 = new Demo(2);
            Object d3 = d2;       //1
            //Object d3 = new Demo(3); //2
            d2 = d1;
      System.gc();//просьба выполнить "сборку мусора"
      }
}
```

В результате выполнения этого кода перед вызовом метода **gc()** без ссылки останется только один объект.

```
объект удален, a=1
```

Если закомментировать строку 1 и снять комментарий со строки 2, то перед выполнением **gc()** ссылку потеряют уже два объекта.

```
объект удален, a=1
объект удален, a=2
```

Задания к главе 4

Вариант А

Реализовать агрегирование. При создании класса агрегируемый класс объявляется как атрибут (локальная переменная, параметр метода).

1. Создать объект класса **Строка**, используя классы **Слово**, **Символ**.
2. Создать объект класса **Абзац**, используя класс **Строка**.
3. Создать объект класса **Страница**, используя класс **Абзац**.
4. Создать объект класса **Текст**, используя классы **Страница**, **Слово**.
5. Создать объект класса **Абзац**, используя класс **Слово**.
6. Создать объект класса **Страница**, используя класс **Слово**.
7. Создать объект класса **Страница**, используя классы **Строка**, **Слово**.

8. Создать объект класса **Текст**, используя класс **Абзац**.
9. Создать объект класса **Автомобиль**, используя класс **Колесо**.
10. Создать объект класса **Самолет**, используя класс **Крыло**.
11. Создать объект класса **Беларусь**, используя класс **Область**.
12. Создать объект класса **Планета**, используя класс **Материк**.
13. Создать объект класса **Звездная система**, используя классы **Планета**, **Звезда**, **Луна**.
14. Создать объект класса **Компьютер**, используя классы **Винчестер**, **Дисковод**, **ОЗУ**.

Вариант В

При выполнении данного задания использовать соответствующие условия из варианта В предыдущей главы. Построить модель программной системы с применением отношений (обобщения, ассоциации, использования, реализации) между классами. Задать атрибуты и методы классов. Ввести (если необходимо) дополнительные классы.

1. Система **Факультатив**. **Преподаватель** объявляет запись на **Курс**. **Студент** записывается на **Курс**, обучается и по окончании **Преподаватель** выставляет **Оценку**, которая сохраняется в **Архиве**. **Студентов**, **Преподавателей** и **Курсов** при обучении может быть несколько.
2. Система **Платежи**. **Клиент** имеет **Счет** в банке и **Кредитную Карту (КК)**. **Клиент** может оплатить **Заказ**, сделать платеж на другой **Счет**, заблокировать **КК** и аннулировать **Счет**. **Администратор** может заблокировать **КК** за превышение кредита.
3. Система **Больница**. **Пациенту** назначается лечащий **Врач**. **Врач** может сделать назначение **Пациенту** (процедуры, лекарства, операции). **Медсестра** или другой **Врач** выполняют назначение. **Пациент** может быть выписан из **Больницы** по окончании лечения, при нарушении режима или иных обстоятельствах.
4. Система **Вступительные экзамены**. **Абитуриент** регистрируется на **Факультет**, сдает **Экзамены**. **Преподаватель** выставляет **Оценку**. Система подсчитывает средний балл и определяет **Абитуриентов**, зачисленных в учебное заведение.
5. Система **Библиотека**. **Читатель** оформляет **Заказ** на **Книгу**. Система осуществляет поиск в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. При невозвращении **Книги Читателем** он может быть занесен **Администратором** в “черный список”.
6. Система **Конструкторское бюро**. **Заказчик** представляет **Техническое Задание (ТЗ)** на проектирование многоэтажного **До-**

ма. **Конструктор** регистрирует **ТЗ**, определяет стоимость проектирования и строительства, выставляет **Заказчику Счет** за проектирование и создает **Бригаду Конструкторов** для выполнения Проекта.

7. Система **Телефонная станция**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**, может попросить **Администратора** сменить номер и отказаться от услуг. **Администратор** изменяет номер, **Услуги** и временно отключает **Абонента** за неуплату.
8. Система **Автобаза**. **Диспетчер** распределяет заявки на **Рейсы** между **Водителями** и назначает для этого **Автомобиль**. **Водитель** может сделать заявку на ремонт. **Диспетчер** может отстранить **Водителя** от работы. **Водитель** делает отметку о выполнении **Рейса** и состоянии **Автомобиля**.
9. Система **Интернет-магазин**. **Администратор** добавляет информацию о **Товаре**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** регистрирует **Продажу** и может занести неплательщиков в “черный список”.
10. Система **Железнодорожная касса**. **Пассажир** делает **Заявку** на станцию назначения, время и дату поездки. Система регистрирует **Заявку** и осуществляет поиск подходящего **Поезда**. **Пассажир** делает выбор **Поезда** и получает **Счет** на оплату. **Администратор** вводит номера **Поездов**, промежуточные и конечные станции, цены.
11. Система **Городской транспорт**. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом. При поломке на **Маршрут** должен выходить резервный транспорт или увеличиваться интервал движения.
12. Система **Аэрофлот**. **Администратор** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. Каждый **Рейс** выполняется **Самолетом** с определенной вместимостью и дальностью полета. **Рейс** может быть отменен из-за погодных условий в **Аэропорту** отлета или назначения. **Аэропорт** назначения может быть изменен в полете из-за технических неисправностей, о которых сообщил командир.

Вариант С

Реализовать агрегирование для приведенных ниже классов с созданием методов соответствующих специфике классов.

1. **Fraction ← ComplexFraction.**
2. **Complex ← ComplexPolynom.**
3. **Fraction ← FractionArray.**
4. **Fraction ← FractionPolynom.**

5. **Complex ← ComplexArray.**
6. **Complex ← ComplexFraction.**
7. **Polynom ← ComplexPolynom.**
8. **Polynom ← FractionPolynom.**

Тестовые задания к главе 4

Вопрос 4.1.

Дан код:

```
class Base {}
class A extends Base {}
 public class Quest{
    public static void main(String[] args){
      Base b = new Base();
      A ob = (A) b;
    } }
```

Результатом компиляции и запуска будет:

1) компиляция и выполнение без ошибок;
2) ошибка во время компиляции;
3) ошибка во время выполнения.

Вопрос 4.2.

Классы **A** и **Quest2** находятся в одном файле. Что необходимо изменить в объявлении класса **Quest2**, чтобы оно было корректным?

```
public class A{}
class Quest2 extends A, Object {}
```

1) необходимо убрать спецификатор **public** перед **A**;
2) необходимо добавить спецификатор **public** к **Quest2**;
3) убрать после **extends** один из классов;
4) класс **Object** нельзя указывать явно.

Вопрос 4.3.

Дан код:

```
class A {A(int i) {}}  // 1
class B extends A {}   // 2
```

Какие из следующих утверждений верны? (выберите 2)

1) компилятор пытается создать по умолчанию конструктор для класса **A**;
2) компилятор пытается создать по умолчанию конструктор для класса **B**;
3) ошибка во время компиляции в строке 1;
4) ошибка во время компиляции в строке 2.

Вопрос 4.4.

Дан код, находящийся в файле **Quest.java**:

```
public class Base{
    Base(){
      int i = 1;
      System.out.print(i);
    } }
public class Quest4 extends Base{
    static int i;
    public static void main(String [] args){
      Quest4 ob = new Quest4();
      System.out.print(i);
    } }
```

В результате компиляции и запуска будет выведено:

1) ошибка компиляции;
2) 0;
3) 10;
4) 1;
5) ошибка выполнения.

Вопрос 4.5.

Что будет результатом компиляции и выполнения следующего кода?

```
class Q {
    private void show(int i){
      System.out.println("1");
    } }
class  Quest5 extends Q{
    public void show(int i){
      System.out.println("2");
    }
    public static void main(String[] args){
      Q ob = new  Quest5();
      int i = '1'; //1
      ob.show(i);
    } }
```

1) ошибка компиляции: метод **show()** недоступен;
2) ошибка времени выполнения: метод **show()** недоступен;
3) ошибка компиляции: несовпадение типов в строке 1;
4) 2;
5) 1.

Вопрос 4.6.

Что будет результатом компиляции и выполнения следующего кода?

```
class Q {
```

```
    void mQ(int i) {
        System.out.print("mQ" + i);
    } }
class Quest6 extends Q {
    public void mQ(int i) {
        System.out.print("mQuest" + i);
    }
          public void mP(int i) {
        System.out.println("mP" + i);
    }
    public static void main(String args[]) {
       Q ob = new Quest6(); //1
        ob.mQ(1); //2
        ob.mP(1); //3
    } }
```

1) mQ1 mP1;
2) mQuest1 mP1;
3) ошибка компиляции в строке //1;
4) ошибка компиляции в строке //2;
5) ошибка компиляции в строке //3.

Вопрос 4.7.

Как следует вызвать конструктор класса **A**, чтобы в результате выполнения кода была выведена на консоль строка в " Конструктор A ".

```
class A{
  A(int i){ System.out.print("Конструктор A");     }
}
public class Quest extends A{
    public static void main(String[] args){
      Quest s= new Quest();
      //1
    }
      public Quest(){
      //2
    }
   public void show()     {
      //3
    } }
```

1) вместо //1 написать **A(1);**
2) вместо //1 написать **super(1);**
3) вместо //2 написать **super(1);**
4) вместо //2 написать **A(1);**
5) вместо //3 написать **super(1).**

Глава 5

АБСТРАКТНЫЕ КЛАССЫ И МЕТОДЫ. ИНТЕРФЕЙСЫ. ПАКЕТЫ

Абстрактные классы

Абстрактные классы объявляются с ключевым словом **abstract** и содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Объекты таких классов создать нельзя, но можно создать объекты подклассов, которые реализуют эти методы. Абстрактные классы могут содержать и полностью реализованные методы.

```
/* пример # 1 : абстрактные методы и классы:
AbstractDemo.java */
abstract class Square {
   abstract int squareIt(int i); //абстрактный метод
   public void show() {
      System.out.println("обычный метод");
      }
}
//squareIt() должен быть реализован подклассом Square
class SquareReal extends Square {
   public int squareIt(int i) {
            return i*i;
                 }
}
public class AbstractDemo {
   public static void main(String[] args) {
// Square ob1 = new Square(); нельзя создать объект!
       Square ob2 = new SquareReal();
     System.out.println("10 в квадрате равно "
                + ob2.squareIt(10));
       ob2.show();
      }
}
```

В результате будет получено:

```
10 в квадрате равно 100
```

обычный метод

Ссылка на абстрактный суперкласс **ob2** инициализируется объектом подкласса, в котором реализованы все абстрактные методы суперкласса. Реализованные методы суперкласса вызываются с помощью этой ссылки. С помощью этой ссылки вызываются обычные реализованные методы абстрактного класса, если они не переопределены в подклассе.

Интерфейсы

Интерфейсы представляют полностью абстрактные классы: ни один из объявленных методов не может быть реализован внутри интерфейса. Все объявленные методы автоматически трактуются как **public** и **abstract**, а все атрибуты – как **public**, **static** и **final**. Класс может реализовывать любое число интерфейсов, указываемых после ключевого слова **implements**, дополняющего определение класса. На множестве интерфейсов также определена иерархия по наследованию, но она не имеет отношения к иерархии классов. В языке Java интерфейсы обеспечивают большую часть той функциональности, которая в C++ представляется с помощью механизма множественного наследования. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов. Определение интерфейса имеет вид:

```
[public] interface имя [extends I1,I2,…,IN]
{/*реализация интерфейса*/}
```

Реализация интерфейсов классом может иметь вид:

```
[доступ] class имя_класса implements I1,I2,…,IN
{/*код класса*/}
```

Здесь **I1,I2, …,IN** перечень используемых интерфейсов. Класс, который реализует интерфейс, должен предоставить полную реализацию всех методов, объявленных в интерфейсе. Кроме этого, данный класс может объявлять свои собственные методы. Если класс расширяет интерфейс, но полностью не реализует его методы, то этот класс должен быть объявлен как **abstract.**

```
/* пример # 2 : интерфейс и его реализации :
InterfacesDemo.java */
interface Shape {
      double PI = Math.PI;
      double getSquare(); //объявление методов
      void showParameters();
}
class Rectangle implements Shape { /* реализация ин-
терфейса */
```

```
      double a, b;
      Rectangle(double a, double b) {
            this.a = a;
            this.b = b;
      }
      public double getSquare() {
            return a * b;
      }
      public void showParameters() {
            System.out.println("стороны: a=" + a
                        + " b=" + b);
      }
}
class Circle implements Shape {
      double r;
      Circle(double r) {
            this.r = r;
      }
      public double getSquare() {
            return 2 * PI * r * r;
      }
      public void showParameters() {
            System.out.println("радиус: r=" + r);
      }
}
/* метод getSquare() в следующем абстрактном классе
не реализован */
abstract class Triangle implements Shape {
      double a, b, angle;
      Triangle(double a, double b, double angle) {
            this.a = a;
            this.b = b;
            this.angle = angle;
      }
      public void showParameters() {
            System.out.println("стороны: a=" + a
                  + " b=" + b);
            System.out.println(
            "угол между ними angle=" + angle);
      }
}
public class InterfacesDemo {
```

```
      public static void main(String[] args) {
            Rectangle r = new Rectangle(5, 9.95);
            Circle c = new Circle(7.01);
            printFeatures(r);
            printFeatures(c);
      }
      public static void printFeatures(Shape f) {
            System.out.println("площадь:"
      + f.getSquare() + " \n параметры фигуры - >");
            f.showParameters();
      }
}
```

Класс **InterfacesDemo** содержит метод **printFeatures()**, который вызывает методы объекта, передаваемого ему в качестве параметра. Вначале ему передается объект, соответствующий прямоугольнику, затем кругу (объекты **c** и **r**). Каким образом метод **printFeatures()** может обрабатывать объекты двух различных классов? Все дело в типе передаваемого этому методу аргумента – класса, реализующего интерфейс **Shape**. Вызывать, однако, можно только те методы, которые были объявлены в интерфейсе.

В следующем примере объявляется объектная ссылка, которая использует интерфейсный тип. Такая ссылка может указывать на экземпляр любого класса, который реализует объявленный интерфейс. Когда вызывается метод через такую ссылку, то будет вызываться его правильная версия, основанная на текущем экземпляре класса. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

```
/* пример # 3 : динамический вызов методов :
TestCall.java */
interface Call{
      int NUM = 10; // по умолчанию final
}
interface Callback extends Call{
void callB(); /* abstract по умолчанию */
      //int i; // ошибка, если нет инициализации
      //void method(){} /* ошибка, так как абстрактный
      метод не может иметь тела! */
      }
class Client implements Callback {
   public void callB() {
        System.out.println("callB() из класса Client:"
                  + " NUM = "+ NUM);
      }
}
```

```
class XClient implements Callback {
  public void callB() {
     System.out.print("callB() из класса XClient: ");
     System.out.println("NUM в квадрате = "
                 + (NUM*NUM));
   }
}
public class TestCall{
     public static void main(String[] args) {
       Callback c = new Client();/*ссылка на интер-
фейсный тип */
        Client cl = new Client();
       XClient ob = new XClient();
   c.callB();
   c = ob; // присваивается ссылка на другой объект
   c.callB();
   //cl = ob;//ошибка! разные ветви наследования
   }
}
```

Результат:

```
callB() из класса Client: NUM = 10
callB() из класса XClient: NUM в квадрате = 100
```

Невозможно приравнивать ссылки на классы, находящиеся в разных ветвях наследования. По этой же причине ошибку вызовет попытка объявления объекта в виде:

```
Client cl = new XClient();
```

Пакеты

Любой класс Java относится к определенному пакету, который может быть неименованным. Оператор **package**, помещаемый в начале исходного программного файла, определяет именованный пакет, т.е. область в пространстве имен классов, где определяются имена классов, содержащихся в этом файле. Например:

```
package com;
```

При этом программный файл будет помещен в подкаталог с названием **com**. Внутри указанной области можно выделить подобласти:

```
package com.mypack;
```

Действие оператора **package** аналогично действию объявления директории на имена файлов. При использовании классов перед именем класса через точку надо добавлять имя пакета, к которому относится дан-

ный класс. Общую форма исходного файла, содержащего код Java, может быть следующая:

```
одиночный оператор package (необязателен);
любое количество операторов import (необязательны);
одиночный открытый (public) класс (необязателен)
любое количество классов пакета (необязательны)
```

Каждый класс добавляется в пакет при компиляции. Для добавления класса в какой-то пакет этот пакет указывается после слова **package**. Например:

```
// пример # 4 : простейший пакет : Items.java
package com.mypack;
public class Items {
      private String name;
      public double price;
      int count;
public Items(String n, double p, int c) {
 name = n;
 price = p;
 count = c;
}
public void show() {
   if(count == 0)
      System.out.print(name
            + "-->> в продаже отсутствует ");
   else
      System.out.println(name + ": $"
            + price + " количество: " + count);
      }
}
```

Файл начинается с объявления того, что данный класс принадлежит пакету **com.mypack.** Другими словами, это означает, что файл **Items.java** находится в каталоге **mypack**, который, в свою очередь, находится в каталоге **com.** Нельзя переименовывать пакет, не переименовав каталог, в котором хранятся его классы. Чтобы получить доступ к классу из другого пакета, перед именем такого класса указывается имя пакета: **com.mypack.Items**. Чтобы избежать таких длинных имен, используется ключевое слово **import.** Например:

```
import com.mypack.Items;
```

или

```
import com.mypack.*;
```

Во втором варианте импортируется весь пакет. Доступ к классу из другого пакета можно осуществить следующим образом:

```
// пример # 5 : доступ к пакету : ViewItems.java
package first;
import com.mypack.Items;
public class ViewItems {
 public static void main(String[] args) {
   Items current[] = new Items[3];
   current[0] = new Items ("Хлеб", 500.07, 100);
   current[1] = new Items ("Молоко", 650.1, 0);
   current[2] = new Items ("Кефир", 670.0, 25);
   for(int i = 0; i < 3; i++) current[i].show();
     }
}
```

Если пакет не существует, то его необходимо создать до первой компиляции, если пакет не указан, класс добавляется в пакет без имени (unnamed). При этом каталог unnamed не создается.

Статический импорт в J2SE 5.0

Константы и статические методы класса можно использовать без указания принадлежности к классу, если применить статический импорт, как это показано в следующем примере.

```
// пример # 6 : статический импорт : ImportDemo.java
import static java.lang.Math.*;
public class ImportDemo {
     public static void main(String[] args) {
           double radius = 3;
           System.out.println(2 * PI * radius);
           System.out.println(floor(cos(PI)));
     }
}
```

Если необходимо получить доступ только к одной константе класса, например **Math.E**, или интерфейса, то статический импорт производится в следующем виде:

```
import static java.lang.Math.E;
```

Задания к главе 5

Вариант А

Реализовать абстрактные классы или интерфейсы, а также наследование и полиморфизм для следующих классов:

1. Абстрактный класс **Книга** (Шифр, Автор, Название, Год, Издательство). Подклассы **Справочник** и **Энциклопедия**.

2. **interface Abiturient ← abstract class Student ← class Student Of Faculty**.
3. **interface Сотрудник ← class Инженер ← class Руководитель**.
4. **interface Учебное Заведение ← class Колледж ← class Университет**.
5. **interface Здание ← abstract class Общественное Здание ← class Театр**.
6. **interface Mobile ← abstract class Siemens Mobile ← class Model**.
7. **interface Корабль ← abstract class Военный Корабль ← class Авианосец**.
8. **interface Врач ← class Хирург ← class Нейрохирург**.
9. **interface Корабль ← class Грузовой Корабль ← class Танкер**.
10. **interface Диск ← abstract class Директория ← class Файл**.

Вариант В

В заданиях варианта В главы 4, где возможно, заменить объявления суперклассов объявлениями абстрактных классов или интерфейсов.

Выполнить аналогичное описание логики системы для заданий, приведенных ниже и использовать их при включении новых возможностей в разрабатываемую систему как это рекомендовано в варианте В последующих глав.

В следующих заданиях требуется создать суперкласс (абстрактный класс, интерфейс) и определить общие методы для данного класса. Создать подклассы, в которых добавить специфические свойства и методы. Часть методов переопределить. Создать массив объектов суперкласса и заполнить объектами подклассов. Объекты подклассов идентифицировать конструктором по имени или идентификационному номеру. Использовать объекты подклассов для моделирования реальных ситуаций и объектов.

1. Создать суперкласс **Транспортное средство** и подклассы **Автомобиль**, **Велосипед**, **Повозка**. Подсчитать время и стоимость перевозки пассажиров и грузов каждым транспортным средством.
2. Создать суперкласс **Грузоперевозчик** и подклассы **Самолет**, **Поезд**, **Автомобиль**. Определить время и стоимость перевозки для указанных городов и расстояний.

3. Создать суперкласс **Пассажироперевозчик** и подклассы **Самолет**, **Поезд**, **Автомобиль**. Определить время и стоимость передвижения.
4. Создать суперкласс **Учащийся** и подклассы **Школьник** и **Студент**. Создать массив объектов суперкласса и заполнить этот массив объектами. Показать отдельно студентов и школьников.
5. Создать суперкласс **Музыкальный инструмент** и классы **Ударный, Струнный**, **Духовой**. Создать массив объектов **Оркестр**. Выдать состав оркестра.
6. Определить суперкласс **Множество** и подкласс **Кольцо** (операции сложения и умножения, обе коммутативные и ассоциативные, связанные законом дистрибутивности). Ввести кольца целых чисел многочленов, систему классов целых чисел, сравнимых по модулю. Кольцо является полем, если в нем определена операция деления, кроме деления на ноль.
7. Создать абстрактный класс **Работник фирмы** и подклассы **Менеджер**, **Аналитик**, **Программист**, **Тестировщик**, **Дизайнер**.
8. Создать суперкласс **Домашнее животное** и подклассы **Собака**, **Кошка**, **Попугай**. С помощью конструктора установить имя каждого животного и его характеристики.
9. Создать базовый класс **Садовое дерево** и производные классы **Яблоня**, **Вишня**, **Груша** и другие. С помощью конструктора автоматически установить номер каждого дерева. Принять решение о пересадке каждого дерева в зависимости от возраста и плодоношения.

Тестовые задания к главе 5

Вопрос 5.1.

Какие из фрагментов кода скомпилируются без ошибки?

1)

```
import java.util.*;
package First;
class My{/* тело класса*/}
```

2)

```
package mypack;
import java.util.*;
public class First{/* тело класса*/}
```

3)

```
/*комментарий */
package first;
import java.util.*;
class First{/* тело класса*/}
```

Вопрос 5.2.

Дан код:

```
abstract class QuestBase {
 static int i;
abstract void show();
 }
public class Quest2 extends QuestBase {
 public static void main(String[] args){
 boolean[] a = new boolean[3];
 for(i = 0; i < a.length; i++)
 System.out.print(" " + a[i]);
 } }
```

В результате при компиляции и запуске будет выведено:

1) false false false;
2) ошибка компиляции: массив **a** использован прежде, чем проинициализирован;
3) ошибка компиляции: **Quest2** должен быть объявлен как **abstract;**
4) ошибка времени выполнения: будет сгенерировано исключение **IndexOutOfBounds;**
5) true true true.

Вопрос 5.3.

Какие определения интерфейса **MyInterface** являются корректными?

1)
```
interface MyInterface{
   public int result(int i){return(i++);}}
```
2)
```
interface MyInterface{
   int result(int i);}
```
3)
```
public interface MyInterface{
   public static int result(int i);}
```
4)
```
public interface MyInterface{
   public final static int i;
   {i=0;}
   public abstract int result(int i);}
```
5)
```
public interface MyInterface{
   public final static int i;
   public abstract int result(int i);}
```

Вопрос 5.4.

Дан код (классы находятся в разных пакетах):

```
package my;
```

```
class Q {
   protected void method1() {
System.out.print("1, ");}
   public void method2() {System.out.print("2, ");}
   private void method3() {System.out.print("3, ");}
   void method4() {System.out.print("4, ");}
}//следующий класс в другом пакете
package my.other;
import my.*;
class Quest4 extends Q {
  public static void main(String[] a) {
    Q ob = new Q();
    ob.method1();  // 1
    ob.method2();  // 2
    ob.method3();  // 3
    ob.method4();  // 4
}}
```

В результате при компиляции и запуске **`java Quest4`** будет выведено:

1) 1, 2, 3, 4;
2) код вызовет ошибку компиляции в строке 1;
3) код вызовет ошибку компиляции в строке 2;
4) код вызовет ошибку компиляции в строке 3;
5) код вызовет ошибку компиляции в строке 4.

Вопрос 5.5.

Какие определения методов в данном интерфейсе не создадут ошибки при компиляции?

```
public interface MyInterface {
  Error runtime();              //   1
  public Class show();          //   2
  protected int method();       //   3
  private String get();         //   4
  abstract void put(Byte Byte); //   5
   }
```

1) 1;
2) 2;
3) 3;
4) 4;
5) 5.

Глава 6
ВНУТРЕННИЕ И ВЛОЖЕННЫЕ КЛАССЫ

В Java можно определить (вложить) один класс внутри определения другого класса, что позволяет группировать классы, логически связанные друг с другом, и динамично управлять доступом к ним. С одной стороны, обоснованное использование в коде внутренних классов делает его более эффектным и понятным. С другой стороны, применение внутренних классов есть один из способов сокрытия кода, так как внутренний класс может быть абсолютно недоступен и не виден вне класса-владельца. Внутренние классы также используются в качестве блоков прослушивания событий (глава 12).

Вложенные классы могут быть статическими, объявляемыми с модификатором **static**, и нестатическими. Статические классы могут обращаться к членам включающего класса не напрямую, а только через его объект. Нестатические внутренние классы имеют доступ ко всем переменным и методам своего класса-владельца.

Внутренние (inner) классы

Нестатические вложенные классы принято называть внутренними (inner) классами. Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде внешнего класса. Объект внутреннего класса всегда ассоциируется (скрыто хранит ссылку) с создавшим его объектом внешнего класса – так называемым внешним (enclosing) объектом. Внешний и внутренний классы могут выглядеть, например, так:

```
class Owner{
   //поля и методы
[доступ] class Inner [extends Cl][implements Interf]{
   // поля и методы
                        }
}
```

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к допустимому содержимому внутреннего класса только после создания объекта внутреннего класса. Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (**final static**). Внутренние классы имеют право наследовать другие классы, реализовывать

интерфейсы и выступать в роли объектов наследования. Допустимо наследование следующего вида:

```
class ExtendOwner extends Owner {
      class ExtendInner extends Inner {}
         Inner ref = new ExtendInner();
}
```

Если внутренний класс наследуется обычным образом, то он теряет доступ к полям своего внешнего класса, в котором он был объявлен.

```
class New extends Owner.Inner {
      New(Owner ob){
            ob.super();
    }
}
```

В данном случае конструктор класса **New** должен быть объявлен с параметром типа **Owner**, что позволит получить доступ к ссылке на внутренний класс, наследуемый классом **New**.

При объявлении внутреннего класса могут использоваться модификаторы **final**, **abstract**, **private**, **protected**, **public**.

```
/* пример # 1 : простой внутренний класс :
SimpleDemo.java */
class Owner {
      class Content {
            private int count = 220;
            public int getCount() {
                  return count;
            }
      }
      void show() {
            Content c = new Content();
            System.out.println(c.getCount());
      }
}
public class SimpleDemo {
      public static void main(String[] args) {
            Owner ob = new Owner();
            ob.show();
      //Content content = new Content();//недоступен
      }
}
```

Внутренний класс, объект которого объявлен внутри **show()**, выглядит так же, как и любой другой класс.

```
/* пример # 2 : создание ссылки на внутренний класс :
SimpleRef.java */
class Base {
      protected class Content {
            private int count = 71;
            public void showContent() {
            System.out.println("count = " + count);
            }
      }
      Content get() {
            return new Content();
      }
}
public class SimpleRef {
      public static void main(String[] args) {
            Base b = new Base();
            Base.Content ob = b.get();
            ob.showContent();
      }
}
```

Если необходимо создать объект внутреннего класса где-нибудь, кроме нестатического метода внешнего класса, то нужно определить тип этого объекта как **OwnerName.InnerName**. Это можно сделать только в том случае, если внутренний класс не объявлен как **private**.

Простой пример практического применения взаимодействия класса-владельца и внутреннего нестатического класса проиллюстрирован на следующем примере.

```
/* пример # 3 : взаимодействие внешнего и внутреннего
классов : AnySession.java */
class Student {
      private int id;
      private Exam[] exams;
      Student(int id) {
            this.id = id;
      }
      class Exam {
            private String name;
            private int mark;
            private boolean passed;
            Exam(String name) {
                  this.name = name;
                  passed = false;
            }
```

```
            void passExam() {
                  passed = true;
            }
            void setMark(int mark) {
                  this.mark = mark;
            }
            int getMark() {
                  return mark;
            }
            int getPassedMark() {
                  final int PASSED_MARK=4;
                  return PASSED_MARK;
            }
            public String getName() {
                  return name;
            }
            public boolean isPassed() {
                  return passed;
            }
      }//окончание внутреннего класса
      public void setExams(String[] nameExams) {
            exams = new Exam[nameExams.length];
            for (int i = 0; i < nameExams.length; i++)
                  exams[i] = new Exam(nameExams[i]);
      }

      public void passExams(int[] marks) {
            for (int i = 0; i < exams.length; i++) {
                  exams[i].setMark(marks[i]);
if (exams[i].getMark() > exams[i].getPassedMark())
                        exams[i].passExam();
            }
      }
      public void checkExams() {
            for (int i = 0; i < exams.length; i++)
                  if (exams[i].isPassed())
      System.out.println(
            exams[i].getName() + " сдан");
                  else
      System.out.println(
            exams[i].getName() + " не сдан");
            }
}
```

```
class AnySession {
      public static void main(String[] args) {
      Student stud = new Student(822201);
      String exams[] = { "Mechanics", "Programming" };
            stud.setExams(exams);
            int marks[] = { 2, 9 };
            stud.passExams(marks);
            stud.checkExams();
                  }
}
```

В результате будет выведено:

```
Mechanics не сдан
Programming сдан
```

Внутренний класс определяет сущность предметной области "экзамен" (класс **Exam**), которая обычно непосредственно связана в информационной системе с объектом класса **Student**. Класс **Exam** в данном случае определяет только методы доступа к своим атрибутам. Класс **Student** включает методы по созданию и инициализации массива объектов внутреннего класса с любым количеством экзаменов, который однозначно идентифицирует текущую успеваемость студента.

Внутренний класс может быть объявлен также внутри метода или логического блока внешнего класса. В этом случае видимость класса регулируется областью видимости того блока, в котором он объявлен. Но внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода.

```
/* пример # 4 : внутренний класс, объявленный внутри
метода : InnerInBlockDemo.java */
interface I {
      void showI();
}
class Owner {
      private int x = 1;
      static int y = 2;
      public void show(int a1, final int z) {
            int b1 = a1 + z;
            System.out.println("в классе Owner");
//abstract или final - допустимые спецификаторы
      class Inner implements I {
                  public void showI() {
      System.out.println("в Inner =" + (x + y + z));
                  //к переменным a1 и b1 нет доступа
                  }
```

```
            }
            Inner in = new Inner();
            in.showI();
      }
}
public class InnerInBlockDemo {
      public static void main(String[] args) {
            new Owner().show(3, 4);
      }
}
```

Выведено будет:

```
в классе Owner
в Inner =7
```

Класс **Inner** объявлен в методе **show()**, и соответственно объекты этого класса можно создавать только внутри этого метода, из любого другого места внешнего класса внутренний класс недоступен.

Вложенные (nested) классы

Если не существует необходимости в связи объекта внутреннего класса с объектом внешнего класса, то есть смысл сделать такой класс статическим.

При объявлении такого внутреннего класса присутствует служебное слово **static**, и такой класс называется вложенным (nested). Если класс вложен в интерфейс, то он становится статическим по умолчанию. Такой класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать его объект, а напрямую доступны только статические поля и методы внешнего класса. Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости.

```
/* пример # 5 : вложенный класс :
NestedClassDemo.java */
class Owner3 {
      static int y = 1, z = 2;
      public void method1() {
//вызов нестатического метода статического класса
            new Nested().method2();
      }
```

```
      //abstract, final, private, public - допустимы
      protected static class Nested {
            public static void showN() {
                  System.out.println("z=" + z
                        + " y=" + Owner3.y);
            }
            public void method2() {
                  System.out.println("non-static");
                  }
      }
}
public class NestedClassDemo {
      public static void main(String[] args) {
            Owner3.Nested.showN(); //статический
            Owner3 ob = new Owner3();
            ob.method1();
      }
}
```

В результате будет выведено:

```
z=2 y=1
non-static
```

В результате создан объект внешнего класса, а метод внешнего класса вызвал нестатический метод вложенного класса. Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Анонимные (anonymous) классы

Можно объявить анонимный (безымянный) класс, который может расширить другой класс или реализовать интерфейс. Объявление такого класса выполняется одновременно с созданием его объекта посредством оператора **new**.

```
class Base {
  void show(){}
}
//объявление класса и его тела
Base ob = new Base(){
  void show(){
//новая реализация
  }
};
```

Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Конструкторы ни определять, ни переопределять нельзя. Анонимные классы допускают вложенность друг в друга, что очень сильно запутывает код и делает эти конструкции непонятными, поэтому эти возможности обычно не используются.

```
/* пример # 6 : анонимные классы и логические блоки :
AnonymousDemo.java */
abstract class A {
      private char c = 'A';
      A() {
      }
      A(char c) {
            this.c = c;
      }
      public char getC() {
            return c;
      }
      public abstract int getNum();
}
class AnonymousDemo {
      static int j = 2;
      static A ob1 = new A((char) 57) {
      //A(char c) {ch = c + 1;}
// ошибка! Конструктор переопределять нельзя
            {
      System.out.println("первый анонимный класс");
            }
            public int getNum() {
                  return Character.digit(getC(), 10);
            }
      };
      public static void main(String[] args) {
            System.out.println(ob1.getNum());
            A ob2 = new A() {
                  int i = 1;
                  {
      System.out.println("второй анонимный класс");
                  }
                  public int getNum() {
            i = show(); //вызов собственного метода
      return i + Character.getNumericValue(getC());
                  }
```

```
                int show() {
                      return i + j;
                }
          };
          System.out.println(ob2.getNum());
     }
}
```

В результате будет выведено:

```
первый анонимный класс
9
второй анонимный класс
13
```

При запуске приложения происходит инициализация полей и выполнение логических блоков класса, содержащего метод **main()**, поэтому сначала выполняются логические блоки при объявлении поля **ob1**. Затем в **main()** поле **ob1** вызывает метод **getNum()**, реализованный анонимным (безымянным) классом. Процесс создания объекта **ob2** и последующего вызова метода **getNum()** отличаются только тем, что объект **ob2** создается как локальный объект метода **main()**. В этом случае нет необходимости строить различные реализации класса **A**, для того чтобы получить немного отличающиеся версии метода.

Задания к главе 6

Вариант А

1. Создать класс **Notepad** (записная книжка) с внутренним классом или классами, с помощью объектов которого могут храниться несколько записей на одну дату.
2. Создать класс **Payment** (покупка) с внутренним классом, с помощью объектов которого можно сформировать покупку из нескольких товаров.
3. Создать класс **Account** (счет) с внутренним классом, с помощью объектов которого можно хранить информацию обо всех операциях со счетом (снятие, платежи, поступления).
4. Создать класс **Зачетная Книжка** с внутренним классом, с помощью объектов которого можно хранить информацию о сессиях, зачетах, экзаменах.
5. Создать класс **Department** (отдел фирмы) с внутренним классом, с помощью объектов которого можно хранить информацию обо всех должностях отдела и обо всех сотрудниках, когда-либо занимавших конкретную должность.
6. Создать класс **Catalog** (каталог) с внутренним классом, с помощью объектов которого можно хранить информацию об истории выдач книги читателям.

7. Создать класс **CCCP** с внутренним классом, с помощью объектов которого можно хранить информацию об истории изменения территориального деления на области и республики.
8. Создать класс **City** (город) с внутренним классом, с помощью объектов которого можно хранить информацию о проспектах, улицах, площадях.
9. Создать класс **CD** (mp3-диск) с внутренним классом, с помощью объектов которого можно хранить информацию о каталогах, подкаталогах и записях.
10. Создать класс **Mobile** с внутренним классом, с помощью объектов которого можно хранить информацию о моделях телефонов и их свойствах.

Вариант В

В заданиях варианта В главы 4 в одном из классов для сокрытия реализации использовать внутренний или вложенный класс. Для определения уникального поведения объекта одного из классов использовать анонимные классы.

Тестовые задания к главе 6

Вопрос 6.1.

Дан код:

```
abstract class A {
  private int x = 2;
  public int show() {return x;}
}
interface B {int show();}
class C {
      int x = 1;
  static class Nested extends A implements B {
    public int show() {return x++;}
}}
```

Какое значение будет возвращено при вызове метода **show()** из класса **Nested**?

1) 1;
2) 2;
3) 3;
4) ошибка времени выполнения.
5) ошибка компиляции.

Вопрос 6.2.

Что будет выведено в результате компиляции и выполнения следующего кода?

```
abstract class A {
      public int show() {return 1;}
}
class Quest2 {
      static int i = 2;
      static A ob1 = new A() {
      public int show() {return i++;} };
      static A ob2 = new A() {
      public int show() {return ++i;} };
public static void main(String[] args) {
      System.out.print(ob1.show());
      System.out.print(ob2.show());
}}
```

1) 34;
2) 24;
3) 33;
4) 23;
5) ошибка компиляции.

Вопрос 6.3.

Какие из объявлений корректны, если

```
class Owner{
      class Inner{
      } }
```

1) `new Owner.Inner();`
2) `Owner.new Inner();`
3) `new Owner.new Inner();`
4) `new Owner().new Inner();`
5) `Owner.Inner();`
6) `Owner().Inner().`

Вопрос 6.4.

Что будет выведено в результате компиляции и выполнения следующего кода?

```
abstract class Abstract {
      abstract Abstract meth();
}
class Owner {
      Abstract meth() {
            class Inner extends Abstract {
                  Abstract meth() {
                        System.out.print("inner ");
```

```
                        return new Inner();
                  }     }
            return new Inner();
      }
}
public abstract class Quest4 extends Abstract{
      public static void main(String a[]) {
            Owner ob = new Owner();
            Abstract abs = ob.meth();
            abs.meth();
      }
}
```

1) inner;
2) inner inner;
3) inner inner inner;
4) ошибка компиляции;
5) ошибка времени выполнения.

Вопрос 6.5.

```
class Owner {
  char A;          // 1
  void A() {}      // 2
  class A {}       // 3
}
```

В какой строке может возникнуть ошибка при компиляции данного кода?

1) 1;
2) 2;
3) 3;
4) компиляция без ошибок.

Глава 7
СТРОКИ

Системная библиотека Java содержит классы **String** и **StringBuffer**, поддерживающие работу со строками и определенные в пакете **java.lang**. Эти классы объявлены как **final**, что означает невозможность создания собственных порожденных классов со свойствами строки.

Класс String

Особенностью объекта класса **String** является то, что его значение не может быть изменено после создания объекта при помощи какого-либо метода, так как любое изменение приводит к созданию нового объекта. При этом ссылку на объект класса **String** можно изменить так, чтобы она указывала на другой объект.

Класс **String** поддерживает несколько конструкторов, например: **String()**, **String(String str)**, **String(byte asciichar[])**, **String(char[] c)**, **String(StringBuffer sbuf)** и др. Эти конструкторы используются для инициализации объектов класса. Например, при вызове конструктора **String(str.getBytes(),"Cp1251")**, где **str** – строка в формате Unicode, можно установить необходимый алфавит, в данном случае кириллицу. Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект типа **String**, на который можно установить ссылку. Таким образом, объект класса **String** можно создать, присвоив ссылке на класс значение существующего литерала, или с помощью оператора **new** и конструктора, например:

```
String s1 = "sun.com";
String s2 = new String("sun.com");
```

Класс **String** содержит следующие методы для работы со строками:

concat(String s) или "+" – слияние строк;

equals(Object ob) и **equalsIgnoreCase(String s**) – сравнение строк с учетом и без учета регистра соответственно;

compareTo(String s) и **compareToIgnoreCase (String s)** – лексикографическое сравнение строк с учетом и без учета регистра;

contentEquals(StringBuffer ob) – сравнение строки и содержимого объекта типа **StringBuffer**;

substring(int n, int m) – извлечение из строки подстроки длины **m-n**, начиная с позиции **n**;

substring(int n) – извлечение из строки подстроки, начиная с позиции **n**;

length() – определение длины строки;

valueOf(значение**)** – преобразование переменной базового типа к строке;

toUpperCase()/toLowerCase() – преобразование всех символов вызывающей строки в верхний/нижний регистр;

replace(char c1, char c2) – замена в строке всех вхождений первого символа вторым символом;

intern(String str) – занесение строки в "пул" литералов;

trim() – удаление всех пробелов в начале и конце строки;

charAt(int position) – возвращение символа из указанной позиции (нумерация с нуля);

getBytes(параметры**)**, **getChars(**параметры**)** – извлечение символов строки в виде массива байт или символов.

Во всех случаях вызова методов изменяющих строку создается новый объект типа **String**.

В следующем примере массив символов и целое число преобразуются в объекты типа **String** с использованием методов этого класса.

```
/* пример # 1 : использование методов :
DemoString.java */
public class DemoString {
          static int i;
     public static void main(String[] args) {
          char s[] = { 'J', 'a', 'v', 'a' };
//комментарий содержит результат выполнения кода
          String str = new String(s); //str="Java"
          i = str.length(); //i=4
          String num = String.valueOf(2); //num="2"
          str = str.toUpperCase(); //str="JAVA"
          num = str.concat(num); //num="JAVA2"
          str = str + "C";//str="JAVAC";
          char ch = str.charAt(2); //ch='V'
     i = str.lastIndexOf('A'); //i=3 (-1 если нет)
          num = num.replace('2', 'H'); //num="JAVAH"
i = num.compareTo(str); //i=5 между символами H и C
          str.substring(0, 4).toLowerCase(); //java
     }
}
```

Сохранить изменения в объекте класса **String** можно только с применением оператора присваивания, т.е. установкой ссылки на новый объект.

```
/* пример # 2 : передача строки по ссылке :
RefString.java */
public class RefString {
      static String changeStr(String st) {
            st = st.concat(" Microsystems");
            return st;
      }
      public static void main(String[] args) {
            String str = new String("Sun");
            changeStr(str);
      // str = changeStr(str);//сравнить результат!
            System.out.println(str);
      }
}
```

В результате будет выведена строка:

Sun

Так как объект был передан по ссылке, то любое изменение объекта в методе должно сохраняться и для исходного объекта, так как обе ссылки равноправны. Этого не происходит по той причине, что вызов метода **concat()** приводит к созданию нового объекта, на который ссылается локальная ссылка. Этот же объект возвращается оператором **return**, но возвращаемое значение ничему не присваивается, поэтому все изменения теряются. Если изменить код, как показано в комментарии, то все изменения объекта, произведенные в методе **changeStr()**, будут сохранены в объекте, объявленном в **main()**.

Далее рассмотрены особенности способов хранения и идентификации объектов на примере вызова метода **equals()**, сравнивающего строку **String** с указанным объектом и метода **hashCode()**, который вычисляет хэш-код объекта.

```
/* пример # 3 : сравнение ссылок и объектов :
EqualStrings.java */
public class EqualStrings {
   public static void main(String[] args) {
     String s1 = "Java";
     String s2 = "Java";
     String s3 = new String(s1);
System.out.println(s1 + "==" + s2
           + " : " + (s1==s2));//true
```

```
System.out.println(s1 + "==" + s3
          + " : " + (s1==s3));//false
System.out.println(s1 + " equals " + s2
          + " : " + s1.equals(s2));//true
System.out.println(s1 + " equals " + s3
          + " : " + s1.equals(s3));//true
System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
System.out.println(s3.hashCode());
   }
}
```

В результате, например, будет выведено:

```
Java==Java : true
Java==Java : false
Java equals Java : true
Java equals Java : true
2301506
2301506
2301506
```

Несмотря на то, что одинаковые по значению объекты расположены в различных участках памяти, значения их хэш-кодов совпадают.

В Java все ссылки хранятся в стеке, а объекты – в куче. При создании **s2** сначала создается ссылка, а затем этой ссылке устанавливается в соответствие объект. В данной ситуации **s2** ассоциируется с уже существующим литералом, так как объект **s1** уже сделал ссылку на этот литерал. При создании **s3** происходит вызов конструктора, то есть выделение памяти происходит раньше инициализации, и в этом случае в куче создается новый объект.

Существует возможность сэкономить память и переопределить ссылку с объекта на литерал при помощи вызова метода **intern()**.

```
// пример # 4 : применение intern() : DemoIntern.java
public class DemoIntern {
     public static void main(String[] args) {
        String s1 = "Java";//литерал и ссылка на него
        String s2 = new String("Java");
        System.out.println(s1 == s2); //false
        s2 = s2.intern();
        System.out.println(s1 == s2); //true
     }
}
```

В данной ситуации ссылка **s1** инициализируется литералом, обладающим всеми свойствами объекта вплоть до вызова методов. Вызов ме-

тода **intern()** организует поиск соответствующего значению объекта **s2** литерала (канонического представления строки) и при положительном результате возвращает ссылку на найденный литерал, а при отрицательном – заносит значение в пул и возвращает ссылку на него.

В следующем примере рассмотрена сортировка массива строк методом выбора.

```
// пример # 5 : сортировка : SortArray.java
public class SortArray  {
  public static void main(String[] args)  {
String a[] = {"  Vika", "Natasha  ", " Alina",
"   Dima ", "Denis  "};
for(int j = 0; j < a.length; j++) a[j] = a[j].trim();
for(int j = 0; j < a.length; j++)
  for(int i = j + 1; i < a.length; i++)
     if(a[i].compareTo(a[j]) < 0)   {
      String t = a[j];
       a[j] = a[i];
       a[i] = t;
            }
      int i = -1;
while(++i < a.length) System.out.print(a[i] + " ");
      }
}
```

Вызов метода **trim()** обеспечивает удаление всех начальных и конечных символов пробелов. Метод **compareTo()** выполняет лексикографическое сравнение строк между собой по правилам Unicode.

Класс StringBuffer

Класс **StringBuffer** является “близнецом” класса **String**, но, в отличие от последнего, содержимое и размеры объектов класса **StringBuffer** можно изменять. Объекты классов **StringBuffer** и **String** можно преобразовывать друг в друга. Конструктор класса **StringBuffer** может принимать в качестве параметра объект **String** и целочисленный (неотрицательный) размер буфера. Объекты этого класса можно преобразовать в объект класса **String** методом **toString()** или с помощью конструктора класса **String**.

Следует обратить внимание на следующие методы класса:

setLength(int n) – установка размера содержимого буфера;

ensureCapacity(int minimum) – установка гарантированного минимального размера буфера объекта;

capacity() – определение размера буфера объекта;

append(параметры**)** – добавление символов, значений базовых типов, массивов и строк;

insert(параметры**)** – вставка символа, объекта или строки в указанную позицию;

deleteCharAt(int index) – удаление символа;

delete(int start, int end) – удаление подстроки;

reverse() – обращение содержимого объекта.

В классе присутствуют также методы, аналогичные методам класса **String**, такие как **replace()**, **substring()**, **charAt()**, **length()**, **getChars()**, **indexOf()** и др.

```
/* пример # 6 : свойства объекта StringBuffer :
DemoStringBuffer.java */
public class DemoStringBuffer {
     public static void main(String[] args) {
          StringBuffer sb = new StringBuffer();
     System.out.println("длина ->" + sb.length());
     System.out.println("размер ->" + sb.capacity());
     //sb = "Java";//ошибка, только для класса String
          sb.append("Java");
          System.out.println("строка ->" + sb);
     System.out.println("длина ->" + sb.length());
     System.out.println("размер ->" + sb.capacity());
     System.out.println("реверс ->" + sb.reverse());
          }
}
```

Результатом выполнения данного кода будет:

```
длина ->0
размер ->16
строка ->Java
длина ->4
размер ->16
реверс ->avaJ
```

При создании объекта **StringBuffer** автоматически резервируется некоторый объем памяти, что в дальнейшем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе. Если длина строки **StringBuffer** после изменения превышает его размер, то емкость объекта автоматически увеличивается, оставляя при этом резерв для дальнейших изменений. С помощью метода **reverse()** можно быстро изменить порядок символов в объекте.

Если метод, вызываемый объектом **StringBuffer**, производит изменения в его содержимом, то это не приводит к созданию нового объекта, как в случае объекта **String**, а изменяет текущий объект **StringBuffer**.

```
/* пример # 7 : изменение объекта StringBuffer :
RefStringBuffer.java */
public class RefStringBuffer {
      static void changeStr(StringBuffer s) {
            s.append(" Microsystems");
      }
      public static void main(String[] args) {
            StringBuffer str =
                  new StringBuffer("Sun");
            changeStr(str);
            System.out.println(str);
      }
   }
```

В результате выполнения этого кода будет выведена строка:

Sun Microsystems

Объект **StringBuffer** передан в метод **changeStr()** по ссылке, поэтому все изменения объекта сохраняются и для вызывающего метода.

Для класса **StringBuffer** не переопределены методы **equals()** и **hashCode()**, т.е. сравнить содержимое двух объектов невозможно, к тому же хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**.

```
/* пример # 8 : сравнение объектов StringBuffer и их
хэш-кодов : EqualsStringBuffer.java */
public class EqualsStringBuffer {
      public static void main(String[] args) {
            StringBuffer sb1 =
                  new StringBuffer("Sun");
            StringBuffer sb2 =
                  new StringBuffer("Sun");
            System.out.print(sb1.equals(sb2));
            System.out.print(
            sb1.hashCode() == sb2.hashCode());
      }
}
```

Результатом выполнения данной программы будет дважды выведенное значение **false**.

Задания к главе 7

Вариант А

1. В каждом слове текста k-ю букву заменить заданным символом. Если k больше длины слова, корректировку не выполнять.
2. В русском тексте каждую букву заменить ее порядковым номером в алфавите. При выводе в одной строке печатать текст с двумя пробелами между буквами, в следующей строке внизу под каждой буквой печатать ее номер.
3. В тексте после буквы Р, если она не последняя в слове, ошибочно напечатана буква А вместо О. Внести исправления в текст.
4. В тексте слова заданной длины заменить указанной подстрокой, длина которой может не совпадать с длиной слова.
5. В тексте после k-го символа вставить заданную подстроку.
6. После каждого слова текста, заканчивающегося заданной подстрокой, вставить указанное слово.
7. В зависимости от признака (0 или 1) в каждой строке текста удалить указанный символ везде, где он встречается, или вставить его после k-го символа.
8. Из небольшого текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.
9. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
10. Удалить из текста его часть, заключенную между двумя символами, которые вводятся (например между скобками ‘(’ и ‘)’ или между звездочками ‘*’ и т.п.).
11. В тексте найти все пары слов, из которых одно является обращением другого.
12. Найти и напечатать, сколько раз повторяется в тексте каждое слово, которое встречается в нем.
13. В тексте найти и напечатать n символов (и их количество), встречающихся наиболее часто.
14. Найти, каких букв, гласных или согласных, больше в каждом предложении текста.
15. В стихотворении найти количество слов, начинающихся и заканчивающихся гласной буквой.
16. Напечатать без повторения слова текста, у которых первая и последняя буквы совпадают.
17. В тексте найти и напечатать все слова максимальной и все слова минимальной длины.
18. Напечатать квитанцию об оплате за телеграмму, если стоимость одного слова задана.

19. В стихотворении найти одинаковые буквы, которые встречаются во всех словах.
20. В тексте найти первую подстроку максимальной длины, не содержащую букв.
21. В тексте определить все согласные буквы, встречающиеся не более чем в двух словах.

Вариант В

1. В тексте нет слов, начинающихся одинаковыми буквами. Напечатать слова текста в таком порядке, чтобы последняя буква каждого слова совпадала с первой буквой последующего слова. Если все слова нельзя напечатать в таком порядке, найти такую цепочку, состоящую из наибольшего количества слов.
2. Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
3. Найти такое слово в первом предложении, которого нет ни в одном из остальных предложений.
4. Во всех вопросительных предложениях текста найти и напечатать без повторений слова заданной длины.
5. В каждом предложении текста поменять местами первое слово с последним, не изменяя длины предложения.
6. В предложении из n слов первое слово поставить на место второго, второе – на место третьего, и т.д., (n-1)-е слово – на место n-го, n-е слово поставить на место первого. В исходном и преобразованном предложениях между словами должны быть или один пробел, или знак препинания и один пробел.
7. Текст шифруется по следующему правилу: из исходного текста выбирается 1, 4, 7, 10-й и т.д. (до конца текста) символы, затем 2, 5, 8, 11-й и т.д. (до конца текста) символы, затем 3, 6, 9, 12-й и т.д. Зашифровать заданный текст.
8. На основании правила кодирования, описанного в предыдущем примере, расшифровать заданный набор символов.
9. Напечатать слова русского текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.
10. Рассортировать слова русского текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).
11. Слова английского текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по 1-й согласной букве слова.
12. Все слова английского текста рассортировать по возрастанию количества заданной буквы в слове. Слова с одинаковым количеством расположить в алфавитном порядке.

13. Ввести текст и список слов. Для каждого слова из заданного списка найти, сколько раз оно встречается в тексте, и рассортировать слова по убыванию количества вхождений.
14. Все слова текста рассортировать в порядке убывания их длин, При этом все слова одинаковой длины рассортировать в порядке возрастания в них количества гласных букв.

Тестовые задания к главе 7

Вопрос 7.1.

Дан код:

```
public class Quest1 {
   public static void main(String[] args) {
      String str = new String("java");
      int i=1;
      char j=3;
      System.out.println(str.substring(i,j));
}}
```

В результате при компиляции и запуске будет выведено:

1) ja;
2) av;
3) ava;
4) jav;
5) ошибка компиляции: заданы некорректные параметры для метода `substring()`.

Вопрос 7.2.

Какую инструкцию следует использовать, чтобы обнаружить позицию буквы **v** в строке **`str = "Java"?`**

1) `charAt(2,str);`
2) `str.charAt(2);`
3) `str.indexOf('v');`
4) `indexOf(str,'v');`

Вопрос 7.3.

Какие из следующих операций корректны при объявлении?

```
String s = new String("Java");
String t = new String();
String r = null;
```

1) `r = s + t + r;`
2) `r = s + t + 'r';`
3) `r = s & t & r;`
4) `r = s && t && r;`

Вопрос 7.4.

Дан код:

```
public class Quest4 {
   public static void main(String[] args) {
      String str="ava";
      char ch=0x74; // 74 - это код символа 'J'
      str=ch+str;
      System.out.print(str);
}}
```

В результате при компиляции и запуске будет выведено:

1) 74ava;
2) Java;
3) 0H74ava;
4) ошибка компиляции: недопустимая операция `ch+str`;
5) ошибка компиляции: недопустимое объявление `char ch=0x74`;
6) нет правильного ответа.

Вопрос 7.5.

Что будет результатом компиляции и выполнения следующего кода?

```
public class Quest5 {
   public static void main(String[] args) {
     StringBuffer s = new StringBuffer("you java");
      s.insert(2, "like ");
      System.out.print(s);
}}
```

1) `yolike u java;`
2) `you like java;`
3) `ylike ou java;`
4) `you java like;`
5) ошибка компиляции;
6) нет правильного ответа.

Часть 2.
Использование классов и библиотек

Во второй части книги рассмотрены вопросы использования классов из пакетов классов Java при работе с файлами, для хранения объектов, при создании пользовательских интерфейсов и др.

Из-за ограниченности объема книги подробное рассмотрение классов невозможно. Подробно классы и методы описаны в документации по языку Java, которую необходимо иметь каждому Java-программисту.

Глава 8
ФАЙЛЫ. ПОТОКИ ВВОДА/ВЫВОДА

Потоки ввода/вывода используются для передачи данных в файловые потоки или сетевые соединения.

Класс File

Для работы с файлами в приложениях Java используются классы из пакета **java.io**.

Класс **File** служит для хранения и обработки в качестве объектов каталогов и имен файлов. Этот класс не описывает способы работы с содержимым файла, но позволяет манипулировать такими свойствами файла, как права доступа, дата и время создания, путь в иерархии каталогов, создание, удаление, изменение имени файла и каталога и т.д.

Основные методы класса **File** и способы их применения рассмотрены в следующем примере.

```
/* пример # 1 : работа с файловой системой :
FileTest.java */
package com.learn;
import java.io.*;
import java.util.*;
public class FileTest {
public static void main(String[] args)
          throws IOException { /*отказ от обработки
     исключения */
```

```
//с объектом типа File ассоциируется файл на диске
      File fp = new File("com\\learn\\FileTest.java");
//другие способы создания объекта
      /*File fp = new File(
            "\\com\\learn", "FileTest.java");*/
      //File fp = new File("d:\\temp\\demo.txt");
      //File fp = new File("demo.txt");
      if(fp.isFile()){//если объект – дисковый файл
      System.out.println("Имя файла:\t"
                  + fp.getName());
      System.out.println("Путь к файлу:\t"
                  + fp.getPath());
      System.out.println("Абсолютный путь:\t"
                  + fp.getAbsolutePath());
      System.out.println("Размер файла:\t"
                  + fp.length());
      System.out.println(
            "Последняя модификация файла:\t"
            + fp.lastModified());
      System.out.println("Файл доступен для чтения:\t"
                  + fp.canRead());
      System.out.println("Файл доступен для записи:\t"
                  + fp.canWrite());
      System.out.println("Файл удален:\t"
                  + fp.delete());            }
            if(fp.createNewFile()){
      System.out.println("Файл "
                  + fp.getName() + " создан");
            }
            if(fp.exists()){
      System.out.println("temp файл "
                  + fp.getName() + " существует");
            }
            else
      System.out.println("temp файл "
                  + fp.getName() + " не существует");
//в объект типа File помещается каталог\директория
            File dir = new File("com\\learn");
if (dir.isDirectory())/*если объект является каталогом*/
      System.out.println("Каталог!");
            if(dir.exists()){//если каталог существует
      System.out.println("Dir "
```

```
                + dir.getName() + " существует");
                File [] files = dir.listFiles();
                System.out.println("");
          for(int i=0; i < files.length; i++){
          Date date =
                new Date(files[i].lastModified());
     System.out.println(files[i].getPath()
          + " \t| " + files[i].length() + "\t| "
          + date.toString());
//использовать toLocaleString() или toGMTString()
                }
          }
     }
}
```

Необходимая при работе с файлами обработка исключительной ситуации **IOException** в методе **main()** не осуществляется в соответствии с инструкцией **throws**.

У каталога (директории) как объекта класса **File** есть дополнительное свойство – просмотр списка имен файлов с помощью методов **list()**, **listFiles()**, **listRoots()**.

Потоки ввода/вывода

Потоки ввода последовательности байт являются подклассами абстрактного класса **InputStream**, потоки вывода последовательности байт – подклассами абстрактного класса **OutputStream**. При работе с файлами используются подклассы этих классов соответственно **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим файлом.

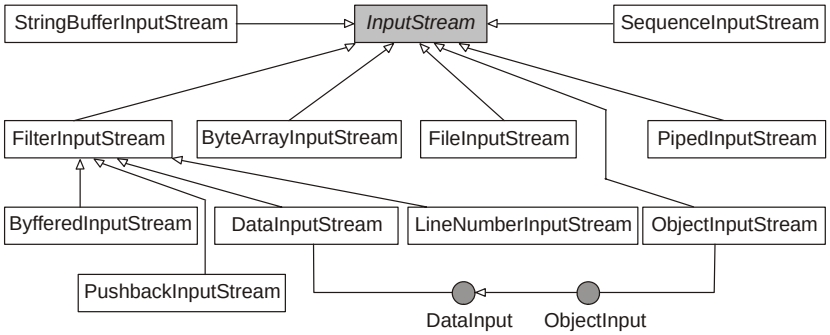

Рис. 8.1. Иерархия классов потоков ввода

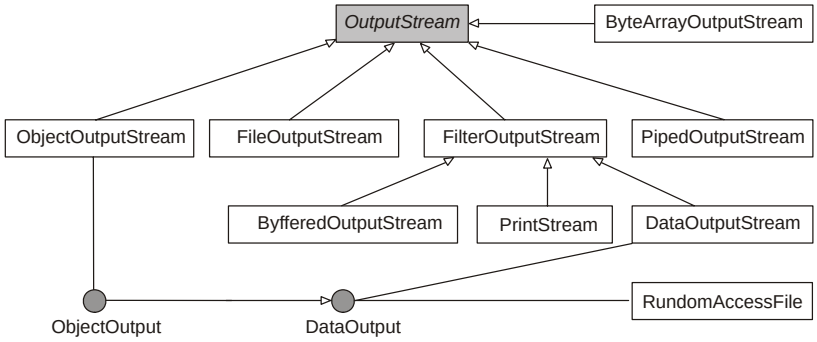

Рис. 8.2. Иерархия классов потоков вывода

Для чтения байта или массива байт используются перегружаемый метод **read()**/**read(byte[] b)** класса **FileInputStream**. Метод возвращает **–1**, если достигнут конец потока данных, поэтому возвращаемое значение имеет тип **int**. Начиная с версии 1.2 пакет **java.io** подвергся значительным изменениям. Появились новые классы, которые хотя и производят более скоростную обработку потоков, но, однако, полностью не перекрывают возможности классов предыдущей версии. Например, аналогом класса **FileInputStream** является класс **FileReader**.

```
/* пример # 2 : чтение по одному байту из потока ввода :
ReadBytes.java */
import java.io.*;
public class ReadBytes {
  public static void main(String[] args) {
    int b;
 try {
File f = new File("ReadBytes.java");
FileReader is = new FileReader(f);
//FileInputStream is = new FileInputStream(f);
     while ((b = is.read()) != -1) {
//прочитанные данные выводятся на консоль
    System.out.println("прочитан байт = " + b);
    }
   is.close(); //закрытие потока ввода
}
 catch (IOException e) {
System.out.println("ошибка файла: " + e);
           }
      }
}
```

Один из конструкторов **FileReader("ReadBytes.java")** или **FileInputStream("ReadBytes.java")** открывает поток **is** и связывает его с файлом **ReadBytes.java.** Для закрытия потока используется метод **close()**. При чтении из потока можно пропустить **n** байт с помощью метода **long skip(long n)**.

Для вывода байта или массива байт используются потоки вывода – объекты подкласса **FileWriter** суперкласса **Writer** или подкласса **FileOutputStream** суперкласса **OutputStream**. В следующем примере для вывода байта в поток используется метод **write()**.

```
// пример # 3 : вывод байта в поток : WriteBytes.java
import java.io.*;
public class WriteBytes {
   public static void main(String[] args){
     int pArray[] = {1, 2, 3, 5, 7, 11, 13, 17};
  try {
FileWriter os = new FileWriter("bytewrite.dat");
/*    FileOutputStream os =
      new FileOutputStream("bytewrite.dat" ); */
    for (int i = 0; i < pArray.length; i++)
         os.write(pArray[i]);
         os.close();
  } catch (IOException e) {
      System.out.println("ошибка файла: " + e);
           }
      }
}
```

Библиотека классов ввода/вывода содержит класс **DataInputStream** для фильтрации ввода, методы которого преобразуют введенные данные к базовым типам. Например, методы **readBoolean()**, **readByte()**, **readChar()**, **readInt()**, **readLong()**, **readFloat()**, **readDouble()** используются для ввода данных и преобразования к соответствующему типу.

Класс **DataOutputStream** позволяет записывать данные базового типа в поток вывода аналогично классу **DataInputStream** для ввода. При этом используются методы **writeBoolean()**, **writeChar()**, **writeInt()**, **writeLong()**, **writeFloat()**, **writeChars(String str)**. В следующем примере рассматривается запись данных в поток и чтение из потока.

```
/* пример # 4 : запись в поток и чтение :
DataStreams.java */
```

```
import java.io.*;
public class DataStreams {
   public static void main(String[] args){
/* запись данных в файл */
     try {
      FileOutputStream os = new FileOutputStream(
            new File("d:\\temp\\data.txt"));
      DataOutputStream ods = new DataOutputStream(os);
  // запись целого числа
      ods.writeInt(48);
  // запись вещественного числа
      ods.writeFloat(3.14159f);
  // запись логического значения
      ods.writeBoolean(true);
  // запись значения типа long
      ods.writeLong(725624);
      ods.close();
    }
      catch(IOException e){
      System.out.print("ошибка записи в файл: " + e);
            }
       try { // чтение данных из файла
      FileInputStream is = new FileInputStream(
            new File("d:\\temp\\data.txt"));
      DataInputStream ids = new DataInputStream(is);
   // чтение целого числа
      int tempi = ids.readInt();
      System.out.println(tempi);
   /* чтение вещественного числа */
      float tempf = ids.readFloat();
      System.out.println(tempf);
   /* чтение логического значения */
      boolean tempb = ids.readBoolean();
      System.out.println(tempb);
   // чтение значения типа long
      long templ = ids.readLong();
      System.out.println(templ);
      ids.close();
      } catch (IOException e) {
      System.out.print("ошибка " + e);
      }
      }
```

```
}
```

В отличие от классов **FileInputStream** и **FileOutputStream** класс **RandomAccessFile** позволяет осуществлять произвольный доступ к потокам как ввода, так и вывода. Поток рассматривается при этом как массив байт, доступ к элементам осуществляется с помощью метода **seek(long poz)**. Для создания потока можно использовать один из конструкторов:

```
RandomAccessFile(String name, String mode);
RandomAccessFile(File file, String mode);
```

Параметр **mode** равен **"r"** для чтения или **"rw"** для чтения и записи.

```
/* пример # 5 : запись и чтение из потока :
RandomFiles.java */
import java.io.*;
public class RandomFiles {
 public static void main(String[] args) {
   int dataArr[] = {12, 31, 56, 23, 27, 1, 43, 65};
   try {
     RandomAccessFile rf = new RandomAccessFile(
          "c:\\temp\\temp.txt", "rw");
    for (int i = 0; i < dataArr.length; i++)
        rf.writeInt(dataArr[i]); // запись в файл
/* чтение в обратном порядке */
    for (int i = dataArr.length - 1; i >= 0; i--) {
    rf.seek(i*4); /* длина каждой переменной типа int
равна 4 байта */
     System.out.println(rf.readInt());
                    }
          rf.close();
     } catch (IOException e) {
  System.out.println("ошибка доступа к файлу: " + e);
    }
  }
}
```

Предопределенные потоки

Класс **System** пакета **java.lang** содержит поле **in**, которое является ссылкой на объект класса **InputStream,** и поля **out**, **err** – ссылки на объекты класса **PrintStream**, объявленные со спецификаторами **public static** и являющиеся стандартными потоками ввода, вывода и вывода ошибок соответственно. Эти потоки связаны с консо-

лью, но могут быть переназначены на другое устройство. В отличие от Java 1.1 в языке Java 1.2 для консольного ввода используется не байтовый, а символьный поток. В этой ситуации для ввода используется подкласс **BufferedReader** абстрактного класса **Reader** и методы **read()** и **readLine()** для чтения символа и строки. Для назначения вывода текстовой информации в произвольный поток следует использовать класс **PrintWriter**, являющийся подклассом абстрактного класса **Writer**. После соединения с необходимым устройством вывода запись в поток производится с помощью методов **print()** и **println()**. Подклассы классов **Reader** и **Writer** дают возможность производить ввод/вывод информации с использованием формата Unicode.

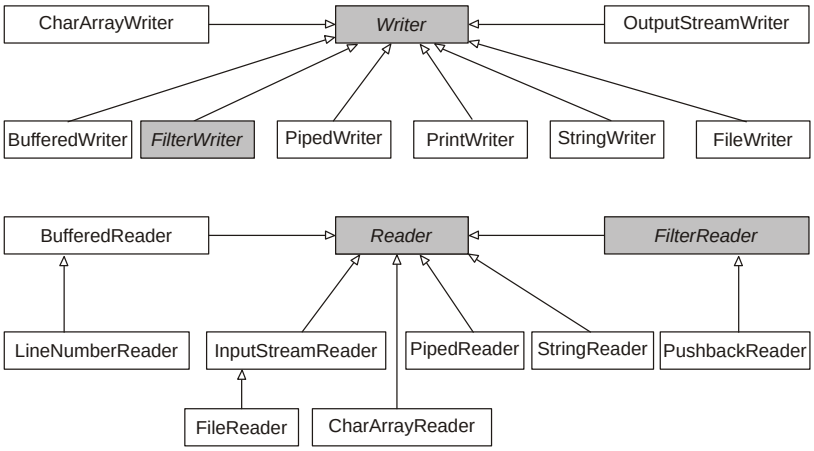

Рис. 8.3. Иерархия символьных потоков ввода/вывода

Следующая программа демонстрирует ввод строки, числа и символа с консоли и вывод на консоль и в файл.

```
// пример # 6 : ввод с консоли : ConsoleInput.java
import java.io.*;
public class ConsoleInput {
    public static void main(String[] args){
          InputStreamReader is =
               new InputStreamReader(System.in);
          BufferedReader bistream =
               new BufferedReader(is);
     try {
          char c;
          int number;
System.out.println("введите имя и нажмите <ввод>:");
   String nameStr = bistream.readLine();
```

```
   System.out.println(nameStr + " введите число:");
   String numberStr = bistream.readLine();
   number = Integer.valueOf(numberStr).intValue();
           System.out.println(nameStr
           + "  вы ввели число " + number);
           System.out.println(nameStr
           + "  введите символ:");
   c = (char)bistream.read();
           System.out.println(nameStr
           + "  вы ввели символ  " + c);
//вывод в файл
      PrintWriter ps = new PrintWriter(
           new FileWriter("res.txt"));
      ps.println("привет " + nameStr + c + number);
      ps.close();
      } catch (IOException e) {
      System.out.println("ошибка ввода " + e);
           }
      }
}
```

Для вывода данных в файл в текстовом формате использовался фильтрованный поток вывода **PrintWriter** и метод **println()**. После соединения этого потока с дисковым файлом посредством потока **FileWriter** становится возможным запись текстовой информации с помощью обычного метода **println()**.

Сериализация

Процесс преобразования объектов в потоки байт для хранения называется сериализацией. Процесс извлечения объекта из потока байт называется десериализацией. Для того чтобы объекты класса могли быть подвергнуты процессу сериализации, этот класс должен расширять интерфейс **Serializable**. Все подклассы такого класса также будут сериализованы. Многие стандартные классы реализуют этот интерфейс. Этот процесс заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора **static** или **transient**. Спецификатор **transient** означает, что поле, помеченное им, не может быть предметом сериализации.

Интерфейс **Serializable** не имеет методов, которые необходимо реализовать, поэтому его использование ограничивается упоминанием при объявлении класса. Все действия в дальнейшем производятся по умолчанию. Для записи объектов в поток необходимо использовать класс **ObjectOutputStream**. После этого достаточно вызвать метод

writeObject(Object ob) этого класса для сериализации объекта **ob** и пересылки его в выходной поток данных. Для чтения используется соответственно класс **ObjectInputStream** и метод **readObject()**, возвращающий ссылку на класс **Object**. После этого следует преобразовать полученный объект к нужному типу.

```
/* пример # 7 : класс для сериализации :
DemoSerial.java */
import java.io.*;
class DemoSerial implements Serializable {
      private long num;//сериализуется
      static double x; //сериализуется с ограничением
      private transient Short s; //не сериализуется
      transient int id; //не сериализуется
public DemoSerial(long n, double y, Short c, int i) {
            num = n;
            x = y;
            s = c;
            id = i;
      }
      public void show() {
            System.out.println("num = " + num
                  + "; x= " + x + ";  Short= " + s
                  + "; id= " + id);
      }
}
/* пример # 8 : запись сериализованного объекта в
файл и его десериализация : DemoSerialToFile.java */
import java.io.*;
public class DemoSerialToFile {
      public static void main(String[] args) {
            //создание и запись объекта
      DemoSerial ds = new DemoSerial(
            1, 10, new Short((short)20), 30);
      File fp = new File("d:\\temp", "demo.txt");
            try {
      ObjectOutputStream ostream =
            new ObjectOutputStream(
                  new FileOutputStream(fp));
                  ostream.writeObject(ds);
            } catch (IOException e) {
                  e.printStackTrace();
            }
            //чтение и вывод объекта
```

```
            File fp2 = new File("d:\\temp\\demo.txt");
            try {
            ObjectInputStream istream =
                  new ObjectInputStream(
                        new FileInputStream(fp2));
            //ds = null;//закомментировать следующую!
            DemoSerial d =
      new DemoSerial(2, 221, Short.valueOf("31"), 41);
   DemoSerial obj = (DemoSerial)istream.readObject();
            obj.show();
            } catch (ClassNotFoundException ce) {
                  ce.printStackTrace();
            } catch (FileNotFoundException fe) {
                  fe.printStackTrace();
            } catch (IOException ioe) {
                  ioe.printStackTrace();
            }
      }
}
```

В результате выполнения данного кода в консоль будет выведено:

```
num = 1; x= 221.0; Short= null; id= 0
```

В итоге поле **num** нового объекта **obj** сохранило значение, которое ему было присвоено до записи в файл. Поля **s** и **id** со спецификатором **transient** получили значения по умолчанию, соответствующие их типу (объектный тип всегда инициализируется по умолчанию значением **null**). Поле **x**, помеченное как статическое, получает то значение, которое имеет это поле для текущего объекта. Если же объекта данного типа нет в области видимости, то статическое поле получает значение, извлеченное из файла.

Для чтения и записи в поток значений отдельных полей объекта можно использовать соответственно методы внутренних классов:

ObjectInputStream.GetField
ObjectOutputStream.PutField.

Задания к главе 8

Вариант А

В следующих заданиях требуется ввести последовательность строк из текстового потока и выполнить указанные действия. При этом могут рассматриваться два варианта:

- каждая строка состоит из одного слова;
- каждая строка состоит из нескольких слов.

Имена входного и выходного файлов, а также абсолютный путь к ним могут быть введены как параметры командной строки или храниться в файле.

1. В каждой строке найти и удалить заданную подстроку.
2. В каждой строке стихотворения А. Блока найти и заменить заданную подстроку на подстроку иной длины.
3. В каждой строке найти слова, начинающиеся с гласной буквы.
4. Найти и вывести слова текста, для которых последняя буква одного слова совпадает с первой буквой следующего слова.
5. Найти в строке наибольшее число цифр, идущих подряд.
6. В каждой строке стихотворения С. Есенина подсчитать частоту повторяемости каждого слова из заданного списка и вывести эти слова в порядке возрастания частоты повторяемости.
7. В каждом слове сонета В. Шекспира заменить первую букву слова на прописную.
8. Определить частоту повторяемости букв и слов в стихотворении А. Пушкина.

Вариант B

Выполнить задания из варианта B глав 4 и 5, сохраняя объекты приложения в одном или нескольких файлах с применением механизма сериализации. Объекты могут содержать поля, помеченные как **`static`**, а также **`transient`**. Для изменения информации и извлечения информации в файле создать специальный класс-коннектор с необходимыми для выполнения этих задач методами.

Вариант C

При выполнении следующих заданий для вывода результатов создавать новую директорию и файл средствами класса **`File`**.

1. Создать и заполнить файл случайными целыми числами. Отсортировать содержимое файла по возрастанию.
2. Прочитать текст Java-программы и все слова **`public`** в объявлении атрибутов и методов класса заменить на слово **`private`**.
3. Прочитать текст Java-программы и записать в другой файл в обратном порядке символы каждой строки.
4. Прочитать текст Java-программы и в каждом слове длиннее двух символов все строчные символы заменить прописными.
5. В файле, содержащем фамилии студентов и их оценки, записать прописными буквами фамилии тех студентов, которые имеют средний балл более “7”.
6. Файл содержит символы, слова, целые числа и числа с плавающей запятой. Определить все данные, тип которых вводится из командной строки.

7. Из файла удалить все слова, содержащие от трех до пяти символов, но при этом из каждой строки должно быть удалено только максимальное четное количество таких слов.
8. Прочитать текст Java-программы и удалить из него все “лишние” пробелы и табуляции, оставив только необходимые для разделения операторов.
9. Из текста Java-программы удалить все виды комментариев.
10. Прочитать строки из файла и поменять местами первое и последнее слова в каждой строке.
11. Ввести из текстового файла, связанного с входным потоком, последовательность строк. Выбрать и сохранить m последних слов в каждой из последних n строк.
12. Из текстового файла ввести последовательность строк. Выделить отдельные слова, разделяемые пробелами. Написать метод поиска слова по образцу-шаблону. Вывести найденное слово в другой файл.
13. Сохранить в файл, связанный с выходным потоком, записи о телефонах и их владельцах. Вывести в файл записи, телефоны которых начинаются на k и на j.
14. Входной файл содержит совокупность строк. Строка файла содержит строку квадратной матрицы. Ввести матрицу в двумерный массив (размер матрицы найти). Вывести исходную матрицу и результат ее транспонирования.
15. Входной файл хранит квадратную матрицу по принципу: строка представляет собой число. Определить размерность. Построить 2-мерный массив, содержащий матрицу. Вывести исходную матрицу и результат ее поворота на 90 градусов по часовой стрелке.
16. В файле содержится совокупность строк. Найти номера строк, совпадающих с заданной строкой. Имя файла и строка для поиска – аргументы командной строки. Вывести строки файла и номера строк, совпадающих с заданной.

Тестовые задания к главе 8

Вопрос 8.1.

Можно ли изменить корневой каталог, в который вкладываются все пользовательские каталоги, используя объект **`myfile`** класса **`File`**? Если это возможно, то с помощью какой инструкции?

1) `myfile.chdir("NAME");`
2) `myfile.cd("NAME");`

3) `myfile.changeDir("NAME");`
4) методы класса **`File`** не могут изменять корневой каталог.

Вопрос 8.2.

Экземпляром какого класса является поле **`System.in`**?

1) `java.lang.System;`
2) `java.io.InputStream;`
3) `java.io.BufferedInputStream;`
4) `java.io.PrintStream;`
5) `java.io.Reader.`

Вопрос 8.3.

Какие из следующих операций можно выполнить применительно к файлу на диске с помощью методов объекта класса **`File`**?

1) добавить запись в файл;
2) вернуть имя родительской директории;
3) удалить файл;
4) определить, текстовую или двоичную информацию содержит файл.

Вопрос 8.4.

Какой абстрактный класс является суперклассом для всех классов, используемых для чтения байт?

1) `Reader;`
2) `FileReader;`
3) `ByteReader;`
4) `InputStream;`
5) `FileInputStream.`

Вопрос 8.5.

При объявлении какого из приведенных понятий может быть использован модификатор **`transient`**?

1) класса;
2) метода;
3) поля класса;
4) локальной переменной;
5) интерфейса.

Глава 9
ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Иерархия и способы обработки

Исключительные ситуации возникают во время выполнения программы, когда возникшая проблема не может быть решена в текущем контексте. Например: попытка индексации вне границ массива или деление на нуль. При возникновении исключения создается объект, описывающий это исключение. Ссылка на этот объект передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы. Если в коде используется метод, в котором может возникнуть исключительная ситуация, но не предусмотрена ее обработка, то ошибка возникает на этапе компиляции. При создании такого метода программист должен включить в код обработку исключений, которые может генерировать этот метод.

Каждой исключительной ситуации поставлен в соответствие некоторый класс. Если подходящего класса не существует, то он может быть создан разработчиком. Исключения являются наследниками суперкласса **`Throwable`** и его подклассов **`Error`** и **`Exception`** из пакета **`java.lang`**.

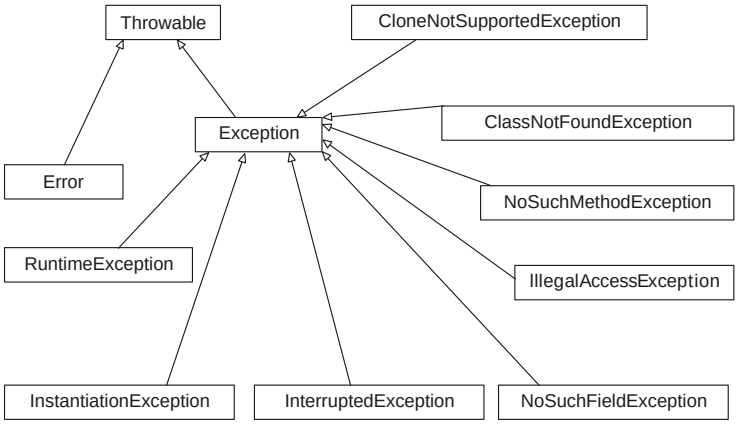

Рис. 9.1. Иерархия классов исключительных ситуаций

В качестве примера рассмотрим подробнее часто используемый подкласс **RuntimeException** класса **Exception** и порожденные от него классы, так называемые unchecked exception. Исключения этого типа, а также исключения типа **Error** возникают только во время выполнения программы. Это означает, что обработка непроверяемого исключения возлагается на программиста. Все остальные возможности возникновения исключительных ситуаций (checked exception) могут быть отслежены на этапе компиляции кода.

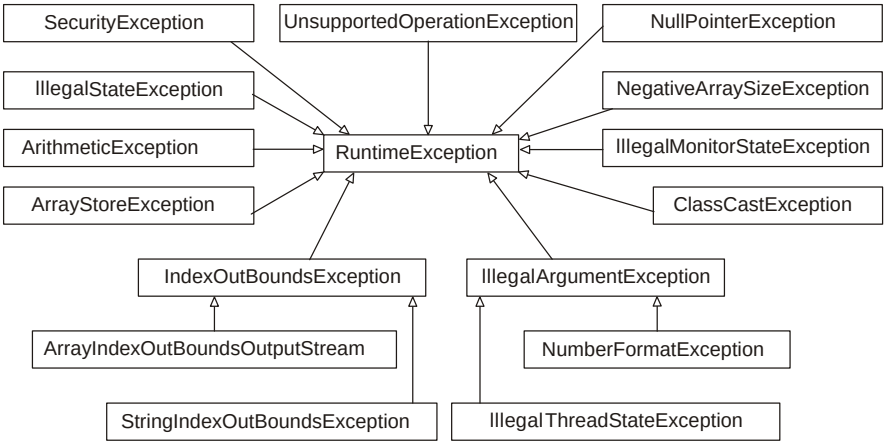

Рис. 9.2. Иерархия непроверяемых (unchecked) исключений

Обычно используется один из трех способов обработки исключений:

- перехват и обработка исключения в блоках **try-catch**;
- объявление исключений в секции **throws** метода и передача вызывающему методу;
- перехват исключения, преобразование его к другому классу и повторный вызов.

Первый подход можно рассмотреть на следующем примере. При арифметических операциях могут возникать исключительные ситуации типа **ArithmeticException**. Например:

```
public void doCalc() {
  try {
      int x = 7/0;
 } catch (ArithmeticException e) {
     String err = e.toString();
      System.out.println(err);
      }
}
```

Если при вычислениях возникает ошибка при делении на ноль, т.е. исключительная ситуация, то генерируется соответствующий объект и управление передается блоку **catch**, в котором обрабатывается данный тип исключения, иначе блок **catch** пропускается.

Второй подход демонстрируется на этом же примере. Если метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, он должен объявить о таком поведении с помощью ключевого слова **throws**, чтобы вызывающие методы могли защитить себя от этих исключений. В вызывающих методах должна быть предусмотрена обработка этих исключений. Форма объявления такого метода:

```
тип имя_метода(список аргументов)
        throws список_исключений { }
```

При этом сам объявляемый метод может содержать блоки **try-catch**, а может и не содержать их. Например, метод **doCalc()** можно объявить:

```
public void doCalc() throws ArithmeticException {
      int x = 7/0;
      }
```

Обрабатывать исключение при этом будет метод, вызывающий **doCalc()**:

```
public void myCalc(){
     try {
       doCalc();
      } catch (ArithmeticException e) {
  String err = e.toString();
  System.out.println(err);
     }
}
```

Третий подход будет рассмотрен ниже на примере создания пользовательских исключений.

Метод может обрабатывать несколько исключений.

```
/* пример # 1 : обработка двух типов исключений :
TwoException.java */
class TwoException {
    public static void main(String[] args) {
 try {
    int a = (int)(Math.random() * 2);
    System.out.println("a = " + a);
        int c[] = { 1/a };
        c[a] = 555;
      } catch(ArithmeticException e) {
```

```
      System.out.println("деление на 0" + e);
      } catch(ArrayIndexOutOfBoundsException e) {
System.out.print("превышение границ массива: " + e);
      }
       System.out.println("после блока try-catch");
      }
}
```

Исключение "деление на 0" возникнет при инициализации элемента массива **a=0**. В противном случае (при **a=1**) генерируется исключение "превышение границ массива" при попытке присвоить значение второму элементу массива **c[]**, который содержит только один элемент.

Подклассы исключений должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения. Например:

```
catch(Exception e){}/* суперкласс Exception перехва-
тит объекты всех своих подклассов */
catch(ArithmeticException e) {}/* не может быть вы-
зван, поэтому возникает ошибка компиляции */
```

Операторы **try** можно вкладывать друг в друга. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возникшему исключению, поиск будет развернут на одну ступень выше, и будут проверены разделы **catch** внешнего оператора **try**.

```
/* пример # 2 : вложенные блоки try-catch :
MultiTryCatch.java */
class MultiTryCatch {
      public static void main(String[] args) {
            try { //внешний блок
            int a = (int) (Math.random() * 2) - 1;
                  System.out.println("a = " + a);
              try { //внутренний блок
                        int b = 1/a;
            StringBuffer sb = new StringBuffer(a);
              } catch (NegativeArraySizeException e) {
            System.out.println(
            "недопустимый размер буфера: " + e);
                  }
            } catch (ArithmeticException e) {
            System.out.println("деление на 0" + e);
            }
      }
}
```

В результате запуска при **a=0** будет сгенерировано исключение **ArithmeticException**, а подходящий для его обработки блок **try–catch** является внешним по отношению к месту генерации исключения. Этот блок и будет задействован для обработки возникшей исключительной ситуации.

Оператор throw

Оператор **throw** используется для генерации исключения. Для этого может быть использован объект класса **Throwable** или объект его подкласса, а также ссылки на них. Общая форма записи инструкции **throw**:

```
throw объектThrowable;
```

При достижении этого оператора выполнение кода прекращается. Ближайший блок **try** проверяется на наличие соответствующего обработчика **catch**. Если он существует, управление передается ему, иначе проверяется следующий из вложенных операторов **try**.

Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор **throw** генерирует исключение, обрабатываемое в разделе **catch**, в котором генерируется другое исключение.

```
/* пример # 3 : генерация исключений :
ThrowGeneration.java */
public class ThrowGeneration {
      static void throwGen() {
      try {
         throw new ClassCastException("демонстрация");
      } catch (ClassCastException e) {
            System.out.println(
            "исключение внутри метода throwGen()");
     throw e;//генерация еще одного исключения
      }
}
public static void main(String[] args) {
  try {
      throwGen();
      } catch(ClassCastException e) {
            System.out.print(
      "обработка исключения вне метода: " + e);
            }
      }
}
```

Вызываемый метод **throwGen()** создает объект класса **ClassCastException** и генерирует исключение, перехватываемое обработчиком. Код обработчика сообщает о том, что сгенерировано исключение, а затем снова генерирует его, в результате чего непроверяемое исключение **ClassCastException**, как подкласс **RuntimeException**, передается обработчику исключений в методе **main()**, иначе компилятор потребовал бы обработки исключения в методе или отказа от нее с помощью инструкции **throws**.

Если метод генерирует исключение с помощью оператора **throw** и при этом блок **catch** в методе отсутствует, то для передачи обработки исключения вызывающему методу тип проверяемого (checked) класса исключений должен быть указан в операторе **throws** при объявлении метода для всех типов исключений, кроме исключений, являющихся подклассами класса **RuntimeException**.

```
/* пример # 4 : использование throws :
ThrowsSample.java */
public class ThrowsSample {
    static void method()
           throws IllegalAccessException {
      System.out.println("внутри метода");
     throw new IllegalAccessException(
                "демонстрация исключения");
    }
public static void main(String[] args) {
   try {
      method();
     } catch (IllegalAccessException e) {
     System.out.println("перехвачено: " + e);
          }
     }
}
```

Ключевое слово finally

Иногда нужно выполнить некоторые действия вне зависимости от того, произошло исключение или нет. В этом случае используется блок **finally**, который выполняется после инструкций **try** или **catch**. Например:

```
try {/*код, который может вызвать исключение*/}
catch(Exception1 e1) {/*обработка исключения e1*/}
catch(Exception2 e2) {/*обработка исключения e2*/}
finally {/*выполняется или после try, или после catch */}
```

Каждому разделу **try** должен соответствовать по крайней мере один раздел **catch** или блок **finally**. Блок **finally** часто используется для закрытия файлов и освобождения других ресурсов, захваченных для временного использования в начале выполнения метода. Код блока выполняется даже в том случае, если перед ним были выполнены инструкции вида **return**, **break**, **continue**. Приведем пример:

```
/* пример # 5 : выполнение блоков finally :
SampleFinally.java */
class SampleFinally {
      static void procA() {
      try {
        System.out.println("внутри метода procA()");
       throw
      new RuntimeException("демонстрация исключения");
      } finally {
System.out.println("блок finally метода procA()");
           }
      }
      static int procB() {
        try {
         System.out.println("внутри метода procB()");
           return 1;
       } finally {
      System.out.print("блок finally метода procB()");
           return 0;
           }
      }
public static void main(String[] args) {
    try {
       procA();
      } catch (Exception e) {
           e.printStackTrace();
           }
      System.out.println(procB());
      }
}
```

В методе **procA()** из-за генерации исключения происходит преждевременный выход из блока **try**, но до выхода из функции выполняется раздел **finally**. Метод **procB()** завершает работу выполнением стоящего в блоке **try** оператора **return,** но и при этом перед выходом из метода выполняется код блока **finally**.

Собственные исключения

Можно создать собственное исключение как подкласс класса **Exception** и затем использовать при обработке ситуаций, возникающих после вызова методов.

```
/* пример # 6 : метод, вызывающий исключение, создан-
ное программистом : MyBasic.java */
public class MyBasic {
      static double result(int i)
                  throws MyNotResultException {
            double d;
            try {
                  if((d = 100/i) < 0)
throw new MyNotResultException("мое исключение");
            else return d;
            } catch (ArithmeticException e) {
      throw new MyNotResultException("мое исключение"
      + "на основе другого исключения:", e);
            }
      }
      public static void main(String[] args) {
            try {
            double res = result(-11);//или 0, или 7;
            } catch (MyNotResultException e) {
      System.out.println(e.toString());
      System.out.println(e.getHiddenException());
            }
      }
}
```

При невозможности вычислить значение генерируется объект **ArithmeticException**, обработчик которого в свою очередь генерирует исключение **MyNotResultException**, используемое в качестве собственного исключения. Он принимает два аргумента. Один из них – сообщение, которое может быть выведено в поток ошибок; другой – реальное исключение, которое привело к вызову нашего исключения. Этот код показывает, как можно сохранить другую информацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину вызова **MyNotResultException**, он всего лишь должен вызвать метод **getHiddenException()**. Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки **MyNotResultException**.

```
/* пример # 7 : собственное исключение :
MyNotResultException.java */
public class MyNotResultException extends Exception {
           private Exception _myException;
 public MyNotResultException(String er, Exception e){
           super(er);
           _myException = e;
     }
 public MyNotResultException(String er){
                 super(er);
           }
     public Exception getHiddenException(){
           return (_myException);
     }
}
```

Задания к главе 9

Вариант А

Выполнить задания на основе варианта А главы 3, контролируя состояние потоков ввода/вывода. При возникновении ошибок, связанных с корректностью выполнения математических операций, генерировать и обрабатывать исключительные ситуации. Предусмотреть обработку исключений, возникающих при: нехватке памяти, отсутствии требуемой записи (объекта) в файле, недопустимом значении поля и т.д.

Вариант В

Выполнить задания из варианта В главы 4, реализуя собственные обработчики исключений и исключения ввода/вывода.

Тестовые задания к главе 9

Вопрос 9.1.

Дан код:

```
class Quest1{
     int counter;
     java.io.OutputStream out;
   Quest1(){/* инициализация out */}
   float inc() {
     try { counter++;
     out.write(counter); }
//комментарий
}}
```

Какой код достаточно добавить в метод **inc()** вместо комментария, чтобы компиляция прошла без ошибок? (выберите 2).

1) `catch (java.io.OutputStreamException e){};`
2) `catch (java.io.IOException e){};`
3) `catch (java.io.OutputException e){};`
4) `finally{};`
5) `return counter;`
6) `return;.`

Вопрос 9.2.

Какое значение будет возвращено при вызове **meth(5)**?

```
public int meth(int x) {
      int y = 010;                  //1
      try { y += x;                 //2
if(x<=5) throw new Exception(); //3
      y++; }                        //4
      catch(Exception e) { y--; }//5
return y; }                         //6
```

1) 12;
2) 13;
3) 14;
4) 15;
5) ошибка компиляции: невыполнимый код в строке 4.

Вопрос 9.3.

Какое значение будет возвращено при вызове **meth(12)**, если при вызове **mexcept(int x)** возникает исключительная ситуация **ArithmeticException**?

```
int meth(int x) {
      int count=0;
      try { count += x;
            count += mexcept(count);
            count++;
      }catch(Exception e) {
            --count;
      return count;
      }
      finally {
            count += 3;
      return count;
      }
}
```

1) 11;
2) 12;
3) 13;
4) 14;
5) ошибка компиляции из-за отсутствия **return** после блока **finally.**

Вопрос 9.4.

Какое из следующих определений метода **show()** может законно использоваться вместо комментария //КОД в классе **Quest4**?

```
class Base{
     public void show(int i) {/*реализация*/}
}
public class Quest4 extends Base{
     //КОД
}
```

1) `void show (int i) throws Exception {/*реализация*/}`
2) `void show (long i) throws IOException {/*реализация*/}`
3) `void show (short i){ /*реализация*/}`
4) `public void show (int i) throws IOException {/*реализация*/}`

Вопрос 9.5.

Дан код:

```
import java.io.*;
public class Quest5 {
    //ОБЪЯВЛЕНИЕ ioRead()
public static void main(String[] args) {
     try {
          ioRead();
     }catch(IOException e){}
     }}
```

Какое объявление метода **ioRead()** должно быть использовано вместо комментария, чтобы компиляция и выполнение кода прошли успешно?

1) `private static void ioRead() throws IOException{};`
2) `public static void ioRead() throw IOException{};`
3) `public static void ioRead(){};`
4) `public static void ioRead() throws Exception{}.`

Глава 10
ХРАНЕНИЕ И ОБРАБОТКА ОБЪЕКТОВ

Коллекции

Коллекции представляют собой реализацию абстрактных типов (структур) данных, поддерживающих две основные операции:

- вставка нового элемента в коллекцию;
- удаление элемента из коллекции.

В качестве дополнительных операций могут быть реализованы следующие: создать структуру данных, просмотреть элементы, подсчитать их количество и др.

Примером коллекции является стек (структура LIFO – Last In First Out), в котором всегда удаляется объект, вставленный последним. Для очереди (структура FIFO – First In First Out) используется другое правило удаления: всегда удаляется элемент, вставляемый первым. В абстрактных типах данных существует несколько видов очередей: двусторонние очереди, кольцевые очереди, обобщенные очереди, в которых запрещены повторяющиеся элементы. Стеки и очереди могут быть реализованы как на базе массива, так и на базе связного списка.

Коллекции объединены в библиотеку классов **`java.util`** и представляют собой контейнеры для хранения и манипулирования объектами. До появления Java 2 эта библиотека содержала классы только для работы с наиболее необходимыми структурами данных: **`Vector`**, **`Stack`**, **`Hashtable`**, **`BitSet`**, а также интерфейс **`Enumeration`** для работы с элементами этих классов. Коллекции, появившиеся в Java 2, представляют общую технологию хранения и доступа к объектам. Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Коллекции – это динамические массивы, связные списки, деревья, множества, хэш-таблицы, стеки, очереди. В интерфейсе **`Collection`** определены статические методы, которые работают на всех коллекциях. Другой способ работы с элементами коллекций – использование методов интерфейса **`Iterator`**, который обеспечивает средства перечисления содержимого коллекции.

Интерфейсы коллекции:

`Collection` – вершина иерархии коллекций;

`List` – расширяет коллекции для обработки списков;

`Set` – расширяет коллекции для обработки наборов (множеств), содержащих уникальные элементы;

`Map` – карта отображения вида "ключ-значение". Интерфейс **`Map`** будет рассмотрен ниже.

Все классы коллекций реализуют также интерфейс **`Serializable`**.

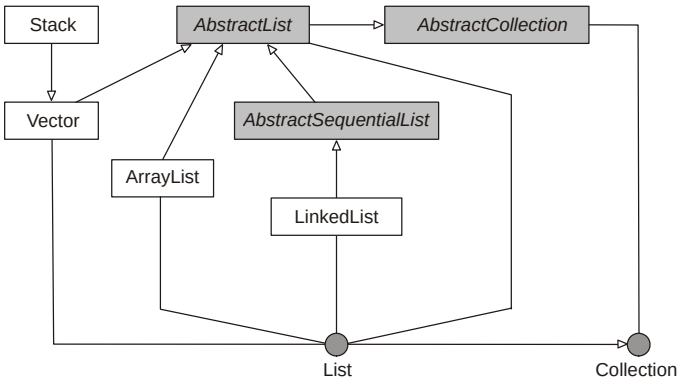

Рис. 10.1. Иерархия наследования списков

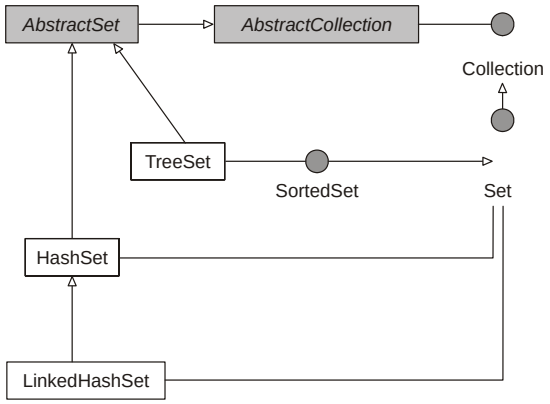

Рис. 10.2. Иерархия наследования множеств

Методы интерфейса **`Collection`**:

`boolean add(Object obj)` – добавляет **`obj`** к вызывающей коллекции и возвращает **`true`**, если объект добавлен, и **`false`**, если **`obj`** уже элемент коллекции. Так как **`Object`** – суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых;

`boolean addAll(Collection c)` – добавляет все элементы коллекции к вызывающей коллекции;

`void clear()` – удаляет все элементы из коллекции;

boolean contains(Object obj) – возвращает **true**, если вызывающая коллекция содержит элемент **obj**;

boolean equals(Object obj) – возвращает **true**, если коллекции эквивалентны;

boolean isEmpty() – возвращает **true**, если коллекция пуста;

Iterator iterator() – извлекает итератор;

boolean remove(Object obj) – удаляет **obj** из коллекции;

int size() – возвращает количество элементов в коллекции;

Object[] toArray() – копирует элементы коллекции в массив объектов.

Для работы с элементами коллекции применяются следующие интерфейсы:

Comparator – для сравнения объектов;

Iterator, **ListIterator**, **Map.Entry** – для перечисления и доступа к объектам коллекции.

Интерфейс **Iterator** используется для доступа к элементам коллекции. Итератор располагается в коллекции между элементами.

Методы интерфейса **Iterator:**

Object next() – возвращает объект, на который указывает итератор, и передвигает текущий указатель на следующий итератор, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод **next()** генерирует исключение **NoSuchElementException**;

boolean hasNext() – проверяет наличие следующего элемента, а в случае его отсутствия возвращает **false**. Итератор при этом остается неизменным;

void remove() – удаляет объект, возвращенный последним вызовом метода **next()**.

Интерфейс **ListIterator** расширяет интерфейс **Iterator** и предназначен в основном для работы со списками. Наличие методов **Object previous()**, **int previousIndex()** и **boolean hasPrevious()** обеспечивает обратную навигацию по списку. Метод **int nextIndex()** возвращает номер следующего итератора. Метод **void add(Object ob)** позволяет вставлять элемент в список в текущую позицию. Вызов метода **void set(Object ob)** производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

Интерфейс **Map.Entry** предназначен для извлечения ключей и значений карты с помощью методов **getKey()** и **getValue()** соответственно. Вызов метода **setValue(Object value)** заменяет значение, ассоциированное с текущим ключом.

Списки

Класс **ArrayList** – динамический массив объектных ссылок. Расширяет класс **AbstractList** и реализует интерфейс **List**. Класс имеет конструкторы:

```
ArrayList()
ArrayList(Collection c)
ArrayList(int capacity)
```

Практически все методы класса являются реализацией абстрактных методов из суперклассов и интерфейсов. Методы интерфейса **List** позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

void add(int index, Object obj) – вставляет **obj** в позицию, указанную в **index**;

void addAll(int index, Collection c) – вставляет в вызывающий список все элементы коллекции **c**, начиная с позиции **index**;

Object get(int index) – возвращает элемент в виде объекта из позиции **index**;

int indexOf(Object ob) – возвращает индекс указанного объекта;

Object remove(int index) – удаляет объект из позиции **index**.

Удаление элементов такой коллекции представляет собой ресурсоемкую задачу, поэтому объект **ArrayList** лучше всего подходит для хранения неизменяемых списков.

```
/* пример # 1 : работа со списком : DemoList1.java */
import java.util.*;
public class DemoList1 {
     public static void main(String[] args) {
          List c = new ArrayList();
//Collection c = new ArrayList();//попробуйте так!
          int i = 2, j = 5;
          c.add(new Integer(i));
          c.add(new Boolean("True"));
          c.add("<STRING>");
          c.add(2, Integer.toString(j) + "X");
                //заменить 2 на 5 !
          System.out.println(c);
          if (c.contains("5X"))
               c.remove(c.indexOf("5X"));
          System.out.println(c);
     }
}
```

В результате на консоль будет выведено:

```
[2, true, 5X, <STRING>]
[2, true, <STRING>]
```

Для доступа к элементам списка может использоваться интерфейс **ListIterator**, в то же время класс **ArrayList** обладает аналогичными методами, в частности **Object set(int index, Object ob)**, который позволяет заменить элемент списка без итератора, возвращая при этом удаляемый элемент.

```
/* пример # 2 : замена и удаление элементов :
DemoList2.java */
import java.util.*;
public class DemoList2 {
      static ListIterator it;
      public static void main(String[] args) {
            ArrayList a = new ArrayList();
            int index;
      System.out.println("коллекция пуста: "
                  + a.isEmpty());
            Character ch = new Character('b');
            a.add(ch);
            for (char c = 'a'; c < 'h'; ++c)
                  a.add(new Character(c));
      System.out.println(a + "число элементов:"
                        + a.size());
it = a.listIterator(2); //извлечение итератора списка
      it.add("new"); //добавление элемента без замены
      System.out.println(
            a + "добавление элемента в позицию");
System.out.println("число элементов стало:"
            + a.size());
                  //сравнить методы
index = a.lastIndexOf(ch); //index = a.indexOf(ch);
            a.set(index, "rep"); //замена элемента
            System.out.println(a + "замена элемента");
            a.remove(6); //удаление элемента
      System.out.println(a + "удален 6-й элемент");
      System.out.println("коллекция пуста: "
                        + a.isEmpty());
      }
}
```

Коллекция **LinkedList** реализует связанный список. В отличие от массива, который хранит объекты в последовательных ячейках памяти, связанный список хранит объекты отдельно, но вместе со ссылками на следующее и предыдущее звенья последовательности. В списке, состоящем из N элементов, существует N+1 позиций итератора.

В добавление ко всем имеющимся методам в **LinkedList** реализованы методы **void addFirst(Object ob)**, **void addLast(Object ob)**, **Object getFirst()**, **Object getLast()**, **Object removeFirst()**, **Object removeLast()** добавляющие, извлекающие, удаляющие и извлекающие первый и последний элементы списка соответственно.

```
/* пример # 3 : добавление и удаление элементов :
DemoLinkedList.java */
import java.util.*;
public class DemoLinkedList {
      public static void main(String[] args){
      LinkedList aL = new LinkedList();
      for(int i = 10; i <= 20; i++)
            aL.add("" + i);
      Iterator it = aL.iterator();
      while(it.hasNext())
            System.out.print(it.next() + " -> ");
            ListIterator list = aL.listIterator();
            list.next();
      System.out.println("\n" + list.nextIndex()
                  + "-й индекс");
      //удаление элемента с текущим индексом
            list.remove();
      while(list.hasNext())
            list.next();//переход к последнему индексу
      while(list.hasPrevious())
            /*вывод в обратном порядке */
      System.out.print(list.previous() + "  ");
      //методы, характерные для LinkedList
      aL.removeFirst();
      aL.removeLast();
      aL.removeLast();
      aL.addFirst("FIRST");
      aL.addLast("LAST");
      System.out.println("\n" + aL);
      }
}
```

Множества

Интерфейс **Set** объявляет поведение коллекции, не допускающей дублирования элементов. Интерфейс **SortedSet** наследует **Set** и объявляет поведение набора, отсортированного в возрастающем порядке с методами **first()/last()**, возвращающими соответственно первый и последний элементы.

Класс **HashSet** наследуется от абстрактного суперкласса **AbstractSet** и реализует интерфейс **Set**, используя хэш-таблицу для хранения коллекции. Ключ (хэш-код) используется вместо индекса для доступа к данным, что значительно ускоряет поиск определенного элемента. Скорость поиска существенна для коллекций с очень большим количеством элементов. Все элементы такого множества упорядочены посредством хэш-таблицы, в которой хранятся хэш-коды элементов.

Конструкторы класса:

```
HashSet()
HashSet(Collection c)
HashSet(int capacity)
```

`HashSet(int capacity, float fillRatio)`, где `capacity` – число ячеек для хранения хэш-кодов.

```
/* пример # 4 : использование множества для вывода
всех уникальных слов из файла :
DemoHashSet.java */
import java.util.*;
import java.io.*;
class DemoHashSet {
     public static void main(String[] args) {
          Set words = new HashSet(100);
// использовать коллекции LinkedHashSet или TreeSet
     long callTime = System.currentTimeMillis();
          try {
               BufferedReader in =
new BufferedReader(new FileReader(
                                   "c://pushkin.txt"));
//в конце файла должна быть строка END
               String line = "";
     while (!(line = in.readLine()).equals("END")) {
     StringTokenizer tokenizer =
               new StringTokenizer(line);
     while (tokenizer.hasMoreTokens()) {
               String word = tokenizer.nextToken();
               words.add(word);
                    }
               }
```

```
            } catch (IOException e) {
                  System.out.println(e);
            }
            Iterator it = words.iterator();
            while (it.hasNext())
                  System.out.println(it.next());
            long totalTime =
            System.currentTimeMillis()- callTime;
            System.out.println("различных слов: " +
words.size() + ", " + totalTime + " миллисекунд");
      }
}
```

Класс **TreeSet** для хранения объектов использует бинарное дерево, главным отличием которого является сортировка его элементов. При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что все добавляемые элементы предположительно реализуют интерфейс **Comparable**. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Класс **TreeSet** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **first()** и **last()**. Методы **SortedSet subSet(Object from, Object to)**, **SortedSet tailSet(Object from)** и **SortedSet headSet(Object to)** предназначены для извлечения определенной части множества.

```
/* пример # 5 : создание множества из списка и его
методы : DemoTreeSet.java */
import java.util.*;
public class DemoTreeSet {
      public static void main(String[] args) {
            Collection c = new ArrayList();
            boolean b;
            for (int i = 0; i < 6; i++)
      c.add(Integer.toString(
            (int) (Math.random() * 90)) + 'Y');
            System.out.println(c + "список");
            TreeSet set = new TreeSet(c);
            System.out.println(set + "множество");
b = set.add("5 Element"); //добавление(b=true)
b = set.add("5 Element"); //добавление(b=false)
```

```
            //после добавления
            System.out.println(set + "add");
            Iterator it = set.iterator();
            while (it.hasNext()) {
                  if (it.next() == "5 Element")
                        it.remove();
            }
            //после удаления
            System.out.println(set + "delete");
      //извлечение наибольшего и наименьшего элементов
            System.out.println(set.last() + " "
                  + set.first());
      }
}
```

В результате может быть выведено:

```
[42Y, 61Y, 55Y, 3Y, 4Y, 55Y]список
[3Y, 42Y, 4Y, 55Y, 61Y]множество
[3Y, 42Y, 4Y, 5  Element, 55Y, 61Y]add
[3Y, 42Y, 4Y, 55Y, 61Y]delete
 61Y 3Y
```

Множество инициализируется списком и сортируется сразу же в процессе создания. После добавления нового элемента производится неудачная попытка добавить его повторно. С помощью итератора элемент может быть найден и удален из множества.

Карты отображений

Карта отображений – это объект, который хранит пару "ключ-значение". Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по его уникальному ключу. Если элемент с указанным ключом отсутствует в карте, то возвращается значение **null**.

Классы карт отображений:

AbstractMap – реализует интерфейс **Map**;

HashMap – расширяет **AbstractMap**, используя хэш-таблицу, в которой ключи отсортированы относительно значений их хэш-кодов;

TreeMap – расширяет **AbstractMap**, используя дерево, где ключи расположены в виде дерева поиска в строгом порядке.

Интерфейсы карт:

Map – отображает уникальные ключи и значения;

Map.Entry – описывает пару "ключ-значение";

SortedMap – содержит отсортированные ключи.

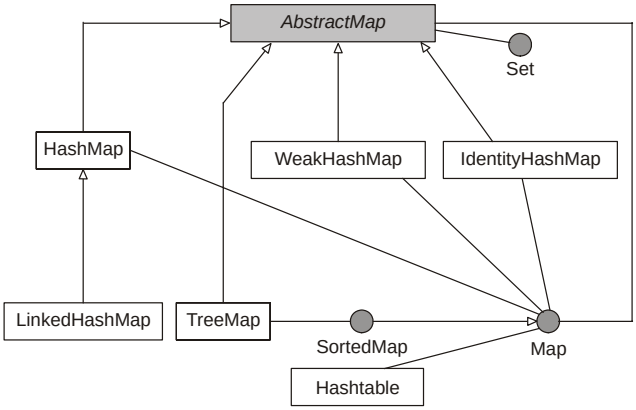

Рис. 10.3. Иерархия наследования карт

Интерфейс **Map** содержит следующие методы:

void clear() – удаляет все пары из вызываемой карты;

boolean containsKey(Object obj) – возвращает **true**, если вызывающая карта содержит **obj** как ключ;

Set entrySet() – возвращает множество, содержащее значения карты;

Set keySet() – возвращает множество ключей;

Object get(Object obj) – возвращает значение, связанное с ключом **obj**;

Object put(Object obj1, Object obj2) – помещает ключ **obj1** и значение **obj2** в вызывающую карту. При добавлении в карту элемента с существующим ключом произойдет замена текущего элемента новым. При этом метод возвратит заменяемый элемент;

Collection values() – возвращает коллекцию, содержащую значения карты.

Интерфейс **Map.Entry** содержит следующие методы:

Object getKey() – возвращает ключ текущего входа;

Object getValue() – возвращает значение текущего входа;

Object setValue(Object obj) – устанавливает значение объекта **obj** в текущем входе.

В примере показаны способы создания хэш-карты и доступа к ее элементам.

```
/* пример # 6 : создание хэш-карты и замена элемента
по ключу : DemoHashMap.java */
import java.util.*;
public class DemoHashMap {
     public static void main(String[] args){
```

```
            Map hm = new HashMap(5);
            for (int i = 1; i < 10; i++)
      hm.put(Integer.toString(i), i + " element");
            hm.put("14s", new Double(1.01f));
            System.out.println(hm);
            hm.put("5", "NEW");
System.out.println(hm + "с заменой элемента ");
            Object a = hm.get("5");
System.out.println(a + " - найден по ключу '5'");
/* вывод хэш-таблицы с помощью методов интерфейса
Map.Entry */
            Set set = hm.entrySet();
            Iterator i = set.iterator();
            while(i.hasNext()){
            Map.Entry me = (Map.Entry)i.next();
            System.out.print(me.getKey()+" : ");
            System.out.println(me.getValue());
            }
      }
}
```

Ниже приведен фрагмент кода корпоративной системы, где продемонстрированы возможности класса **HashMap** и интерфейса **Map.Entry** при определении прав пользователей.

```
/* пример # 7 : применение коллекций при проверке
доступа в систему : DemoSecurity.java */
import java.util.*;
public class DemoSecurity {
      public static void main(String[] args) {
      CheckRight.startUsing("2041", "Bill G.");
      CheckRight.startUsing("2420", "George B.");
/*добавление еще одного пользователя и проверка его
на возможность доступа */
      CheckRight.startUsing("2437", "Phillip K.");
      CheckRight.startUsing("2041", "Bill G.");
      }
}
class CheckRight {
      private static HashMap map = new HashMap();
      public static void startUsing(
                        String id, String name) {
            if (canUse(id)){
                  map.put(id, name);
            System.out.println("доступ разрешен");
            }
```

```
            else {
            System.out.println("в доступе отказано");
            }
      }
      public static boolean canUse(String id) {
            final int MAX_NUM = 2;//заменить 2 на 3
            int currNum = 0;
            if (!map.containsKey(id))
                  currNum = map.size();
            return currNum < MAX_NUM;
            }
      }
```

В результате будет выведено:

```
доступ разрешен
доступ разрешен
в доступе отказано
доступ разрешен,
```

так как доступ в систему разрешен одновременно только для двух пользователей. Если в коде изменить значение константы **MAX_NUM** на большее, чем 2, то новый пользователь получит права доступа.

Класс **WeakHashMap** позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости программы.

Класс **LinkedHashMap** запоминает порядок добавления объектов в карту и образует при этом дважды связанный список. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

Начиная с версии языка Java 1.4 добавлен класс **IdentityHashMap**, хэш-коды объектов-ключей которого вычисляются методом **System.identityHashCode()** по адресу объекта в памяти, в отличие от обычного значения **hashCode()**, вычисляемого сугубо по содержимому самого объекта.

Унаследованные коллекции

В ряде технологий, например в сервлетах, до сих пор применяются коллекции, существовавшие в языке Java с момента его создания, а именно карта **Hashtable** и перечисление **Enumeration**.

```
/* пример # 8 : создание хэш-таблицы и поиск элемента
по ключу : HashTableDemo.java  */
import java.util.*;
import java.io.*;
```

```
public class HashTableDemo {
      public static void main(String[] args) {
            Hashtable capitals = new Hashtable();
            showAll(capitals);
            capitals.put("Индия", "Дели");
            capitals.put("Китай", "Пекин");
            capitals.put("Беларусь", "Минск");
            showAll(capitals);
            //поиск по ключу
      System.out.print("введите название страны: ");
            BufferedReader br =
               new BufferedReader(
                  new InputStreamReader(System.in));
            String name;
            try {
                  name = br.readLine();
                  showCapital(capitals, name);
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
      private static void showAll(Hashtable capitals){
            Enumeration countries;
            if (capitals.isEmpty())
                  System.out.println("таблица пуста");
            else {
                  countries = capitals.keys();
                  String country;
            while (countries.hasMoreElements()) {
      country = (String) countries.nextElement();
                  System.out.print(country + " - ");
            System.out.println(capitals.get(country));
                  }
            }
      }
      private static void showCapital(
                  Hashtable cap, String country) {
            if (cap.get(country) != null) {
                  System.out.print(country + " - ");
                  System.out.println(cap.get(country));
            } else
            System.out.println("запись отсутствует");
      }
}
```

В результате в консоль будет выведено:

```
таблица пуста
Беларусь - Минск
Китай - Пекин
Индия - Дели
введите название страны: Индия
Индия – Дели
```

Принципы работы с коллекциями, в отличие от их структуры, со сменой версий языка существенно не изменились.

Параметризация коллекций в J2SE 5.0

Предлагается более удобный механизм при работе с коллекциями, а именно:

- отсутствие необходимости постоянно преобразовывать возвращаемые объекты (тип **Object**) к требуемому типу;
- предварительное сообщение компилятору о типе объектов, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции.

```
/* пример # 9 : создание параметризованной коллекции :
DemoGenerics.java  */
import java.util.*;
public class DemoGenerics {
  public static void main(String args[]) {
    Map <String, Integer> map =
                new HashMap <String, Integer> ();
    map.put("Key 1", 1);
    int res = map.get("Key 1");/* компилятор "знает"
тип значения */
Character ch = new Character('2');
//  map.put(ch, 2);//ошибка компиляции
//компилятор не позволит добавить "посторонний" тип
  }
}
```

В данной ситуации не создается новый класс для каждого конкретного типа и сама коллекция не меняется, просто компилятор снабжается информацией о типе элементов, которые могут храниться в **map**. При этом параметром коллекции не может быть базовый тип.

Следует отметить, что указывать тип следует при создании ссылки, иначе будет позволено добавлять объекты всех типов.

```
// пример # 10 : параметризация : UncheckCheck.java
import java.util.*;
public class UncheckCheck {
  public static void main(String args[]) {
```

```
    Collection c1 = new HashSet <String> ();
    c1.add("Java");
    c1.add(5);  //нет ошибок: c1 не параметризована
    for(Object ob : c1)
      System.out.print(ob);
    Collection <String> c2 = new HashSet<String>();
    c2.add("A");
//  c2.add(5);
//ошибка компиляции: так как c2 параметризована
    }
}
```

В результате будет выведено:

Java5

Чтобы параметризация коллекции была полной, необходимо указывать параметр и при объявлении ссылки, и при создании объекта.

Существуют уже готовые библиотеки, в которых никаких проверок нет, следовательно, при их использовании нельзя гарантировать, что в коллекцию не будет помещен "посторонний" объект. Для этого в класс **Collections** был добавлен новый метод – **checkedCollection()**:

```
public static < E > Collection < E >
checkedCollection(Collection< E > c, Class< E > type)
```

Этот метод создает коллекцию, проверяемую на этапе выполнения, то есть в случае добавления "постороннего" объекта генерируется исключение **ClassCastException**:

```
/* пример # 11 : проверяемая коллекция :
SafeCollection.java  */
import java.util.*;
public class SafeCollection{
  public static void main(String args[]) {
     Collection c = Collections.checkedCollection(
          new HashSet <String>(), String.class);
     c.add("Java");
     c.add(5.0);     //ошибка времени выполнения
  }
}
```

В этот же класс добавлен целый ряд методов, специализированных для проверки конкретных типов коллекций, а именно: **checkedList()**, **checkedSortedMap()**, **checkedMap()**, **checkedSortedSet()**, **checkedSet()**.

В Java 5.0 добавлен ряд новых классов и интерфейсов, таких как **EnumSet**, **EnumMap**, **PriorityQueue** и др. В качестве иллюстрации возможностей можно рассмотреть один из них – интерфейс **Queue**:

public interface Queue < E > **extends** Collection < E >

Методы интерфейса **Queue**:

E element() − возвращает, но не удаляет головной элемент очереди;

boolean offer(E o) − вставляет элемент в очередь, если возможно (например: ограничены размеры);

E peek() − возвращает, но не удаляет головной элемент очереди, возвращает **null,** если очередь пуста;

E poll() – возвращает и удаляет головной элемент очереди, возвращает **null,** если очередь пуста;

E remove() − возвращает и удаляет головной элемент очереди.

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение, если очередь пуста.

Примечательно, что ранее существовавший класс **LinkedList** теперь помимо интерфейса **List <E>** реализует и **Queue**:

```
/* пример # 12 : проверяемая коллекция :
DemoQueue.java */
import java.util.*;
public class DemoQueue {
      public static void main(String args[]) {
      LinkedList <Integer> c =
                  new LinkedList <Integer> ();
      //добавление десяти элементов
            for (int i = 0; i < 10; i++)
                  c.add(i);
            Queue <Integer> queue = c;
      for (int i : queue) //вывод элементов
                  System.out.print(i + " ");
                  System.out.println(" :size= "
                        + queue.size());
      //удаление девяти элементов
      for (int i = 0; i < 9; i++) {
                  int res = queue.poll();
                        }
            System.out.print("size= " + queue.size());
      }
}
```

В результате выполнения будет выведено:

```
0 1 2 3 4 5 6 7 8 9 :size=10
size=1
```

Обработка массивов

В пакете **java.util** находится класс **Arrays**, который содержит методы манипулирования содержимым массива, а именно для поиска, заполнения, сравнения, преобразования в коллекцию:

int binarySearch(параметры**)** – перегруженный метод организации бинарного поиска значения в массивах примитивных и объектных типов. Возвращает позицию первого совпадения;

void fill(параметры**)** – перегруженный метод для заполнения массивов значениями различных типов и примитивами;

void sort(параметры**)** – перегруженный метод сортировки массива или его части с использованием интерфейса **Comparator** и без него;

List asList(Object [] a) – метод, копирующий элементы массива в объект типа **List**.

В качестве простого примера применения указанных методов можно привести следующий код.

```
/* пример # 13 : методы класса Arrays :
ArraysEqualDemo.java */
import java.util.*;
public class ArraysEqualDemo {
      public static void main(String[] args) {
      char m1[] = new char[3],
           m2[] = { 'a', 'b', 'c' }, i;
            Arrays.fill(m1, 'a');
            System.out.print("массив m1:");
            for (i = 0; i < 3; i++)
                  System.out.print(" " + m1[i]);
            m1[1] = 'b';
            m1[2] = 'c';
//m1[2]='x'; //приведет к другому результату
      if (Arrays.equals(m1, m2))
      System.out.print("\nm1 и m2 эквивалентны");
      else
      System.out.print("\nm1 и m2 не эквивалентны");
            m1[0] = 'z';
            Arrays.sort(m1);
            System.out.print("\nмассив m1:");
            for (i = 0; i < 3; i++)
                  System.out.print(" " + m1[i]);
      System.out.print(
      "\n значение 'c' находится в позиции-"
      + Arrays.binarySearch(m1, 'c'));
      }
}
```

В результате компиляции и запуска будет выведено:

```
массив m1: a a a
m1 и m2 эквивалентны
массив m1: b c z
значение 'c' находится в позиции 1
```

Задания к главе 10

Вариант А

1. Ввести строки из файла, записать в список. Вывести строки в файл в обратном порядке.
2. Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.
3. Создать в стеке индексный массив для быстрого доступа к записям в бинарном файле.
4. Создать список из элементов каталога и его подкаталогов.
5. Создать стек из номеров записи. Организовать прямой доступ к элементам записи.
6. Организовать вычисления в виде стека.
7. Занести стихотворения одного автора в список. Провести сортировку по возрастанию размера.
8. Задать два стека, поменять информацию местами.
9. Определить класс **MyStack**. Объявить объект класса. Ввести последовательность символов и вывести ее в обратном порядке.
10. Создать класс **MyQueue** и объект класса. Ввести массив строк и определить, имеется ли стринг-образец в данной очереди.
11. Определить класс **MySet** на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.
12. Программа получает n параметров вызова, которые являются элементами вектора. Построить массив типа **`double`**, а на базе этого массива – объект класса **DoubleVector**, который необходимо сохранить в карте.
13. Списки (стеки, очереди) I(1..n) и U(1..n) содержат результаты n-измерений тока и напряжения на неизвестном сопротивлении R. Найти приближенное число R методом наименьших квадратов.
14. С использованием множества выполнить попарное суммирование произвольного конечного ряда чисел по следующим правилам: на первом этапе суммируются попарно рядом стоящие числа, на втором этапе суммируются результаты первого этапа и т.д. до тех пор, пока не останется одно число.

15. Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте **`HashMap`**.
16. Умножить два многочлена заданной степени, если коэффициенты многочленов хранятся в различных списках.
17. Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные – в начало этого списка.
18. Ввести строки из файла, записать в список **`ArrayList`**. Выполнить сортировку строк, используя метод **`sort()`** из класса **`Collections`**.

Вариант B

Для заданий варианта B главы 4 предусмотреть хранение, сортировку и обработку информации с использованием возможностей коллекций.

Вариант C

1. Во входном файле хранятся две разреженные матрицы A и B. Построить циклически связанные списки CA и CB, содержащие ненулевые элементы соответственно матриц A и B. Просматривая списки, вычислить: а) сумму S = A + B; б) произведение P = A * B.
2. Во входном файле хранятся наименования некоторых объектов. Построить список C1, элементы которого содержат наименования и шифры данных объектов, причем элементы списка должны быть упорядочены по возрастанию шифров. Затем “сжать” список C1, удаляя дублирующие наименования объектов.
3. Во входном файле расположены два набора положительных чисел; между наборами стоит отрицательное число. Построить два списка C1 и C2, элементы которых содержат соответственно числа 1-го и 2-го набора таким образом, чтобы внутри одного списка числа были упорядочены по возрастанию. Затем объединить списки C1 и C2 в один упорядоченный список, изменяя только значения полей ссылочного типа.
4. Во входном файле хранится информация о системе главных автодорог, связывающих г. Минск с другими городами Беларуси. Используя эту информацию, постройте дерево, отображающее систему дорог республики, а затем, продвигаясь по дереву, определить минимальный по длине путь из г. Минска в другой заданный город. Предусмотреть возможность для последующего сохранения дерева в виртуальной памяти.
5. Элементы списка циклически сдвинуть на n позиций.

6. Заданы два множества точек на плоскости. Определить пересечение и разность этих множеств.
7. На плоскости заданы n множеств по m точек в каждом. Среди точек первого множества найти такую точку, которая принадлежит наибольшему количеству множеств.
8. Выбрать три разные точки заданного на плоскости множества точек, составляющие треугольник наибольшего периметра.
9. Найти такую точку заданного на плоскости множества точек, сумма расстояний от которой до остальных минимальна.
10. Выпуклый многоугольник задан на плоскости перечислением координат вершин, сохраненных в списке, в порядке обхода его границы. Определить площадь многоугольника.
11. Из множества точек на плоскости, заданных координатами, выбрать точки, наиболее и наименее удаленные от прямой $Ax + By + C = 0$.
12. На плоскости тройками (x_i, y_i, r_i) задано множество n окружностей, где (x_i, y_i) – координаты центров, r_i – радиусы окружностей. Установить, пересекаются окружности, касаются или не имеют общих точек.
13. Для каждого из множества заданных координатами вершин треугольников определить его тип: разносторонний, равнобедренный, равносторонний или прямоугольный.

Тестовые задания к главе 10

Вопрос 10.1.

Какой интерфейс наиболее пригоден для создания класса, содержащего несортированные уникальные объекты?

1) `Set;`
2) `List;`
3) `Map;`
4) `Vector;`
5) нет правильного ответа.

Вопрос 10.2.

Какие из фрагментов кода создадут объект класса **`ArrayList`** и добавят элемент?

1) `ArrayList a = new ArrayList(); a.add("0");`
2) `ArrayList a = new ArrayList(); a[0]="0";`
3) `List a = new List(); a.add("0");`
4) `List a = new ArrayList(10); a.add("0");`

Вопрос 10.3.

Какой интерфейс реализует класс **Hashtable**?

1) Set;
2) Vector;
3) AbstractMap;
4) List;
5) Map.

Вопрос 10.4.

Дан код:

```
import java.util.*;
class Quest4 {
   public static void main (String args[]) {
      Object ob = new HashSet();
      System.out.print((ob instanceof Set) + ", ");
      System.out.print(ob instanceof SortedSet);
}}
```

Что будет выведено при попытке компиляции и запуска программы?

1) true, false;
2) true, true;
3) false, true;
4) false, false;
5) ничего из перечисленного.

Вопрос 10.5.

Какие из приведенных ниже названий являются именами интерфейсов пакета **java.util**?

1) SortedMap;
2) HashMap;
3) HashSet;
4) SortedSet;
5) Stack;
6) AbstractMap.

Глава 11
ГРАФИЧЕСКИЕ ИНТЕРФЕЙСЫ ПОЛЬЗОВАТЕЛЯ

Основы оконной графики

Для поддержки создания пользовательских интерфейсов Java 2 содержит библиотеки классов, позволяющих создавать и поддерживать окна, использовать элементы управления (кнопки, меню, полосы прокрутки и др.), применять инструменты для создания графических приложений. Графические инструменты в языке Java реализованы с помощью двух библиотек:

- Пакет AWT (загружается **`java.awt`**) содержит набор классов, позволяющих выполнять графические операции и создавать элементы управления.
- Пакет Swing (загружается **`javax.swing`**, имя **`javax`** обозначает, что пакет не является основным, а только расширением языка) содержит новые классы, по большей части аналогичные AWT. К именам этих классов добавляется **`J`** (**`JButton`**, **`JLabel`** и т. д.). Пакет является частью библиотеки JFC (Java Foundation Class), которая содержит большой набор компонентов JavaBeans, предназначенных для создания пользовательских интерфейсов.

Работа с окнами и графикой в Java осуществляется в апплетах и графических приложениях. Апплеты – это небольшие программы, встраиваемые в Web-документ и использующие для своей визуализации средства Web-браузера. Графические приложения сами отвечают за свою прорисовку.

Библиотека Swing, в отличие от AWT, более полно реализует парадигму объектно-ориентированного программирования. Однако библиотека Swing, хотя и является более современной, но полностью не заменяет собой AWT. В частности, обработка событий остается неизменной. К преимуществам библиотеки Swing следует отнести повышение надежности, расширение возможностей пользовательского интерфейса, а также независимость от платформы. Кроме того, эту библиотеку легче использовать и она визуально более привлекательна.

Апплеты используют окна, производные от класса **`Panel`**, графические приложения используют окна, производные от класса **`Frame`**, порожденного от класса **`Window`**.

Иерархия классов AWT, применяемых для построения визуальных приложений, приведена на рисунке 11.1.

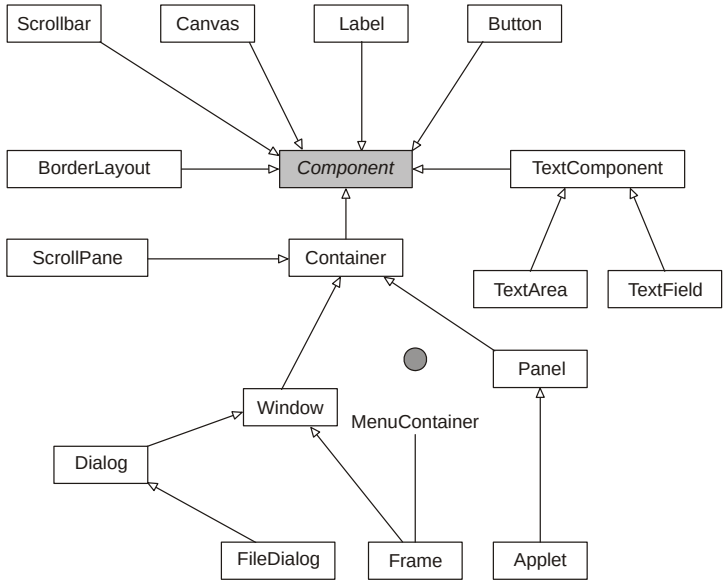

Рис. 11.1. Иерархия классов основных графических компонентов AWT

Суперкласс **`Component`** является абстрактным классом, инкапсулирующим все атрибуты визуального компонента. Порожденный от него подкласс **`Container`** содержит методы, которые позволяют вкладывать в него другие компоненты (объекты) и отвечает за размещение любых компонентов, которые он содержит. Этот класс порождает подклассы **`Panel`** и **`Window`**.

Класс **`Panel`** используется апплетом, графический вывод которого рисуется на поверхности объекта **`Panel`** – окна, не содержащего области заголовка, строки меню и обрамления.

Графические приложения используют класс **`Window`** (окно верхнего уровня), который сам непосредственно для вывода не используется. Для этого применяется его подкласс **`Frame`**. С помощью объекта типа **`Frame`** создается стандартное окно со строкой заголовка, меню, размерами.

Еще один используемый для вывода класс **`Canvas`** (пустое окно, в котором можно рисовать) не является подклассом **`Container`**.

Апплеты

Графические интерфейсы, рисунки и изображения могут быть реализованы в апплетах. Апплеты представляют собой классы языка Java, которые размещаются на серверах Internet, транспортируются клиенту по сети,

автоматически устанавливаются и запускаются браузером как часть документа WWW. Апплеты позволяют вставлять в документы поля, содержание которых меняется во времени, организовывать "бегущие строки", мультипликацию, производить вычисления. Апплеты являются наследниками класса **Applet** из пакета **java.applet** или его подкласса **JApplet** из пакета Swing. Класс **Applet**, в свою очередь, является наследником класса **Panel**.

Есть несколько методов класса **Applet**, которые управляют созданием и выполнением апплета на Web-странице. Апплету не нужен метод **main()**, код запуска помещается в методе **init()**. Перегружаемый метод **init()** автоматически вызывается для выполнения начальной инициализации апплета. Метод **start()** вызывается каждый раз, когда апплет переносится в поле зрения Web-браузера, чтобы начать операции. Метод **stop()** вызывается каждый раз, когда апплет выходит из поля зрения Web-браузера, чтобы позволить апплету завершить операции. Метод **destroy()** вызывается, когда апплет начинает выгружаться со страницы для выполнения финального освобождения ресурсов. Кроме этих методов автоматически загружаемым является метод **paint()** класса **Component**. Метод **paint()** не вызывается явно, а только из других методов, например **repaint()**.

Рассмотрим пример апплета, в котором используются методы **init()**, **paint()**, метод **setColor()** установки цвета символов, метод **drawString()** рисования строк**.**

```
/* пример # 1 : вывод строк разными цветами:
ColorConst.java */
import java.applet.*;
import java.awt.*;
public class ColorConst extends Applet {
   public void paint(Graphics g)  {
       g.setColor(Color.yellow);
      g.drawString("<Yellow>", 5, 30);//желтым цветом
       g.setColor(Color.blue);
      g.drawString("<Blue>", 5, 60);//синим цветом
       g.setColor(Color.green);
      g.drawString("<Green>", 5, 90);//зеленым цветом
     }
}
```

В результате рисуется строка желтым, синим и зеленым цветами в позиции, указанной в методе **drawString()**. Цвет выводимых символов устанавливается с помощью полей суперкласса **Color**.

После выполнения компиляции имя класса, содержащего байт-код апплета, помещается в тег **<applet** параметры**> </applet>** документа HTML. Например:

```
<html>
<applet code = ColorConst.class width = 50 height = 50>
</applet></html>
```

Исполнителем HTML-документа является браузер или специальная программа **appletviewer.exe**.

Большинство используемых в апплетах графических методов, как и использованные в примере методы **setColor()**, **drawString()**, – методы абстрактного базового класса **Graphics** из пакета **java.awt.** Методы апплета получают объект класса **Graphics** (графический контекст) в качестве параметра и вместе с ним – текущий цвет, шрифт, положение курсора. Установку контекста обычно осуществляют методы **update()** или **paint()**.

Ниже перечислены некоторые методы класса **Graphics**:

drawLine(int x1, int y1, int x2, int y2) – рисует линию;

drawRect(int x, int y, int width, int height) и **fillRect(int x, int y, int width, int height)** – рисуют прямоугольник и заполненный прямоугольник;

draw3DRect(int x, int y, int width, int height, boolean raised) – рисует трехмерный прямоугольник;

drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight) – рисует округленный прямоугольник;

drawOval(int x, int y, int width, int height) – рисует овал;

drawPolygon(int[] xPoints, int[] yPoints, int nPoints) – рисует полигон (многоугольник), заданный массивами координат **x** и **y**;

drawPolygon(Polygon p) – рисует полигон, заданный объектом **Polygon**;

drawPolyline(int[] xPoints, int[] yPoints, int nPoints) – рисует последовательность связных линий, заданных массивами **x** и **y**;

drawArc(int x, int y, int width, int height, int startAngle, int arcAngle) – рисует дугу окружности;

drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer) – вставляет изображение;

drawString(String str, int x, int y) – рисует строку;

setColor(Color c), **getColor()** – устанавливает и возвращает текущий цвет;

getFont() – возвращает текущий шрифт;

setFont(Font font) – устанавливает новый шрифт.

В примерах 2–4, приведенных ниже, демонстрируется использование методов класса **Graphics** для вывода графических изображений в окно апплета.

```
// пример # 2 : отображение полигона : DrawPoly.java
import java.applet.*;
import java.awt.*;
public class DrawPoly extends Applet {
   int poly1_x[] = {40,  80,   0,  40};
   int poly1_y[] = {5,  45,  45,   5};
   int poly2_x[] =
          {140, 180, 180, 140, 100, 100, 140};
   int poly2_y[] = {5,  25,  45,  65,  45,  25,   5};
   int poly3_x[] = {240, 260, 220, 260, 220, 240};
   int poly3_y[] = {5,  65,  85,  25,  25,   5};
   public void paint(Graphics g)     {
        g.drawPolygon(
           poly1_x, poly1_y, poly1_x.length);
        g.drawPolygon(
           poly2_x, poly2_y, poly2_x.length);
        g.drawPolygon(
           poly3_x, poly3_y, poly3_x.length);
     }
 }
```

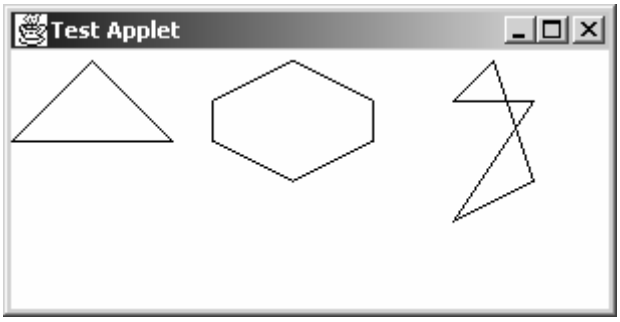

Рис. 11.2. Пример изображения полигонов

```
/* пример # 3 : трехмерные прямоугольники :
ThreeDRect.java */
import java.applet.*;
import java.awt.*;
public class ThreeDRect extends Applet {
   public void draw3DRect(Graphics g, int x, int y,
      int width, int height, boolean raised)     {
    g.draw3DRect(
```

```
      x, y, width - 1, height - 1, raised);
    g.draw3DRect(
      x + 1, y + 1, width - 3, height - 3, raised);
    g.draw3DRect(
      x + 2, y + 2, width - 5, height - 5, raised);
      }
   public void fill3DRect(Graphics g, int x, int y,
      int width, int height, boolean raised){
    g.draw3DRect(x, y, width-1, height-1, raised);
    g.draw3DRect(
      x + 1, y + 1, width - 3, height - 3, raised);
    g.draw3DRect(
      x + 2, y + 2, width - 5, height - 5, raised);
     g.fillRect(x + 3, y + 3, width - 6, height - 6);
      }
   public void paint(Graphics g){
       g.setColor(Color.gray);
       draw3DRect(g, 10, 5, 80, 40, true);
       draw3DRect(g, 130, 5, 80, 40, false);
       fill3DRect(g, 10, 55, 80, 40, true);
       fill3DRect(g, 130, 55, 80, 40, false);
     }
}
```

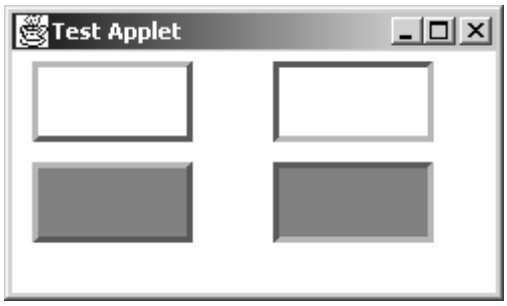

Рис 11.3. Трехмерные прямоугольники

```
// пример # 4 : вывод GIF-изображения : DrawImage.java
import java.applet.*;
import java.awt.*;
public class DrawImage extends Applet {
   Image img;
   public void init() {
       img = getImage(getCodeBase(), "cow.gif");
     }
```

```
   public void paint(Graphics g){
      g.drawImage(img, 0, 0, this);
      }
}
```

При использовании свойств тега **<applet>** существует возможность передать параметры из HTML-документа в код апплета. Пусть HTML-документ имеет вид:

```
<html><head><title>Параметры апплета</title></head>
<body>
<applet code=test.com.ReadParam.class
width=250 height= 300>
<param name = bNumber value = 4>
<param name = state value = true>
</applet></body> </html>
```

Тогда для чтения и обработки параметров **bNumber** и **state** апплет должен выглядеть следующим образом:

```
/* пример # 5 : передача параметров апплету :
ReadParam.java */
package test.com;
import java.awt.*;
import java.applet.*;
public class ReadParam extends Applet {
  int bNum;
  boolean state;
public void start() {//чтение параметров
        String param = getParameter("state");
      if(param != null)
        state = Boolean.valueOf(param).booleanValue();
      try {
           param = getParameter("bNumber");
      if(param != null)
           bNum = Integer.parseInt(param);
      } catch(NumberFormatException e) {
        bNum = 0;
        state = false;
      }
}
public void paint(Graphics g) {
      double d = 0;
  if (state)  d = Math.pow(bNum, 2);
  else g.drawString("Error Parameter", 0, 11);
      g.drawString("Statement: " + state, 0, 28);
```

```
        g.drawString("Value b: " + bNum, 0, 45);
        g.drawString("b power 2: " + d, 0, 62);
        }
}
```

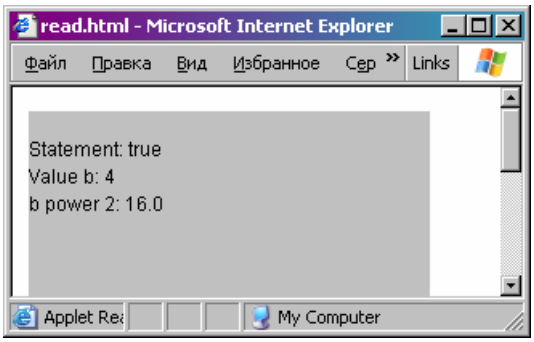

Рис. 11.4. Передача параметров в апплет

Если параметр недоступен, **`getParameter()`** возвращает **`null`**.

В настоящее время при применении управляющих компонентов в апплетах принято использовать интерфейсные классы, в которых компонент на экране создается средствами Java и в минимальной степени зависит от платформы и оборудования. Такого рода классы-компоненты были объединены в библиотеку под названием Swing.

```
// пример # 6 : апплет с компонентом : MyJApplet.java
import javax.swing.*;
import java.awt.*;
public class MyJApplet extends JApplet {
        JLabel lbl = new JLabel("Swing-applet!");
  public void init() {
     Container c = getContentPane();
        c.add(lbl);
  }
}
```

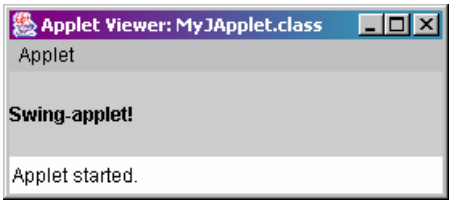

Рис. 11.5. Апплет с меткой

В этой программе производится помещение текстовой метки **`JLabel`** на форму в апплете. Конструктор класса **`JLabel`** принимает объект **`String`** и использует его значение для создания метки. Автоматически

вызываемый при загрузке апплета метод **init()** обычно отвечает за инициализацию полей и размещение компонента на форму. Для этого вызывается метод **add()** класса **Container,** который помещает компонент в контейнер. Метод **add()** сразу не вызывается, как в библиотеке AWT. Пакет **Swing** требует, чтобы все компоненты добавлялись в "панель содержания" формы **ContentPane**, так что требуется сначала вызывать метод **getContentPane()** класса **JApplet** для создания ссылки на объект, как часть процесса **add()**.

```
// пример # 7 : жизненный цикл апплета : DemoLC.java
import java.awt.*;
import javax.swing.*;
public class DemoLC extends JApplet {
  private int i;
  private String msg = null;
  private Color c;
public void init() {
     c = new Color(0, 0, 255);
     this.setBackground(c);
     this.setForeground(Color.white);
  setFont(new java.awt.Font("Courier", 1, 14));
  msg = " initialization";
  i = 1;
    }
public void start() {
 int j = i * 25;
 if (j < 255){
 c = new Color (j, j, 255 - j);
          }
 setBackground(c);
 String str = Integer.toString(i);
 msg += " " + str;
     }
public void paint(Graphics g){
     g.drawString(msg, 30, 30);
          }
public void stop() {
   i++;
   msg = "start() -  stop()";
     }
}
```

При работе со шрифтами можно узнать, какие из них доступны на компьютере, и использовать их. Для получения этой информации применяется метод

getAvailableFontFamilyNames() класса **GraphicsEnvironment**. Метод возвращает массив строк, содержащий имена всех доступных семейств шрифтов, зарегистрированных на данном компьютере.

```
// пример # 8 : доступ к шрифтам ОС : FontDemo.java
import javax.swing.*;
import java.awt.*;
public class FontDemo extends JApplet {
  private int i;
  private String msg = null;
  private String[] fonts;
      public void init() {
       GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
       fonts = ge.getAvailableFontFamilyNames();
      }
      public void start() {
      int j = i;
          if (j > fonts.length) i = 0;
          else setFont(new Font(fonts[j], 1, 14));
         String str = Integer.toString(i);
         msg = fonts[j] + " " + str;
         i++;
      }
      public void paint(Graphics g) {
      g.drawString(msg, 30, 30);
      }
}
```

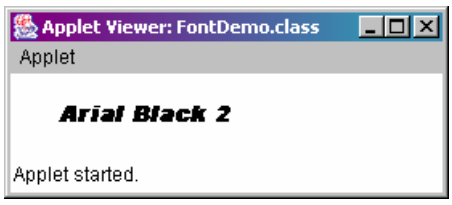

Рис. 11.6. Доступ к списку шрифтов

Библиотека Swing, так же, как и библиотека AWT, поддерживает набор графических методов класса **Graphics**. Вся графика рисуется относительно окна. Вывод в окно выполняется через графический контекст с помощью методов **paint()** или **update()**. Класс **Graphics**, объект которого передается в контекст, определяет функции рисования.

Фреймы

В отличие от апплетов графические приложения, расширяющие класс **java.awt.Frame** или его подкласс **jawax.swing.JFrame**, не нуждаются в браузере. Для создания графического интерфейса приложения необходимо предоставить ему объект **Frame** или **JFrame**, в который будут помещаться используемые приложением компоненты GUI. Большинство своих методов класс **Frame** наследует по иерархии от классов **Component**, **Container** и **Window**. Класс **JFrame** из библиотеки Swing является подклассом класса **Frame.**

Такое приложение запускается с помощью метода **main()** и само отвечает за свое отображение в окне.

```
/* пример # 9 : текст, линия и овалы : App.java */
import java.awt.*;
public class App extends Frame{
   String msg = "My-Window-Application";
   int x1 = 30, y1 = 50, x2 = 200, y2 = 50;
      public void paint(Graphics g){
            //вывод строки с позиции x=30 y=40
      g.drawString(msg, 30, 40);
      g.drawLine(x1, y1, x2, y2);//вывод линии
      int x = 30, y = 60, width = 150, height = 100;
            // установка синего цвета
      Color c = new Color(100, 100, 255);
      g.setColor(c);
      g.drawOval( x, y, width, height);//овал
      //сектор
      g.drawArc (
x + 100, y + 50, width - 50, height, 0, 360);
      }
      public static void main(String args[]){
     App fr = new App();
/*устанавливается размер окна*/
      fr.setSize(new Dimension(300, 250));
//заголовок
      fr.setTitle("Window-Application");
/*установка видимости. Обязательно! */
      fr.setVisible(true);
//перерисовка - вызов paint()
      fr.repaint();
      }
  }
```

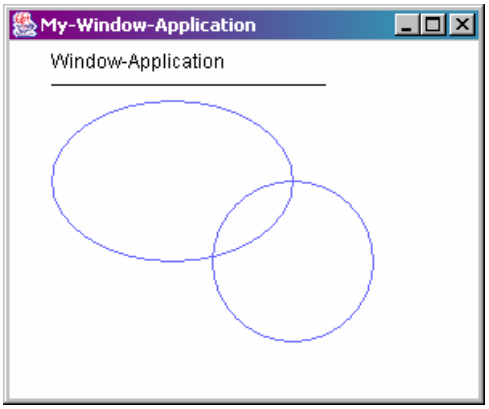

Рис. 11.7. Простейшее графическое приложение

Метод `main()` вызывает методы установки параметров окна и метод перерисовки окна `repaint()`. Фреймы активно используются для создания распределенных систем, эксплуатируемых в локальных и закрытых сетях.

Задания к главе 11

В следующих заданиях выполнить рисунок в окне апплета или фрейма.

1. Создать классы **Point** и **Line**. Объявить массив из n объектов класса **Point**. Для объекта класса **Line** определить, какие из объектов **Point** лежат на одной стороне от прямой линии и какие на другой. Реализовать ввод данных для объекта **Line** и случайное задание данных для объекта **Point**.
2. Создать классы **Point** и **Line**. Объявить массив из n объектов класса **Point** и определить в методе, какая из точек находится дальше всех от прямой линии.
3. Создать класс **Triangle** и класс **Point**. Объявить массив из n объектов класса **Point**, написать функцию, определяющую, какая из точек лежит внутри, а какая – снаружи треугольника.
4. Определить класс **Rectangle** и класс **Point**. Объявить массив из n объектов класса **Point**. Написать функцию, определяющую, какая из точек лежит снаружи, а какая – внутри прямоугольника.
5. Реализовать полиморфизм на основе абстрактного класса **AnyFigure** и его методов. Вывести координаты точки, треугольника, тетраэдра.
6. Определить класс **Line** для прямых линий, проходящих через точки $A(x_1, y_1)$ и $B(x_2, y_2)$. Создать массив объектов класса **Line**. Определить, используя функции, какие из прямых линий пересекаются, а какие совпадают. Нарисовать все пересекающиеся прямые.

7. Определить классы **Triangle** и **NAngle**. Определить, какой из m-введенных n-угольников имеет наибольшую площадь:
$$S_{\text{треуг.}} = \frac{1}{2}\left|(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)\right|.$$
8. Задать движение по экрану строк (одна за другой) из массива строк. Направление движения по апплету и значение каждой строки выбираются случайным образом.
9. Задать составление строки из символов, появляющихся из разных углов апплета и выстраивающихся друг за другом. Процесс должен циклически повторяться.
10. Задать движение окружности по апплету так, чтобы при касании границы окружность отражалась от нее с эффектом упругого сжатия.
11. Изобразить в апплете приближающийся издали шар, удаляющийся шар. Шар должен двигаться с постоянной скоростью.
12. Изобразить в окне приложения (апплета) отрезок, вращающийся в плоскости экрана вокруг одной из своих концевых точек. Цвет прямой должен изменяться при переходе от одного положения к другому.
13. Изобразить в окне приложения (апплета) отрезок, вращающийся в плоскости фрейма вокруг точки, движущейся по отрезку.
14. Изобразить четырехугольник, вращающийся в плоскости апплета вокруг своего центра тяжести.
15. Изобразить прямоугольник, вращающийся в плоскости фрейма вокруг одной из своих вершин.
16. Изобразить разносторонний треугольник, вращающийся в плоскости апплета вокруг своего центра тяжести.

Тестовые задания к главе 11

Вопрос 11.1.

Дан код:

```
<applet code=MyApplet.class width=200 height=200>
<param name=count value=5>
</applet>
```

Какой код читает параметр **count** в переменную **i**?

1) `int i = new Integer(getParameter("count")).intValue();`
2) `int i = getIntParameter("count");`
3) `int i = getParameter("count");`
4) `int i = new Integer(getIntParameter("count")).intValue();`
5) `int i = new Integer(getParameter("count"));.`

Вопрос 11.2.

В пользовательском методе **show()** был изменен цвет фона апплета. Какой метод должен быть вызван, чтобы это было визуализировано?

1) setbgcolor();
2) draw();
3) start();
4) repaint();
5) setColor().

Вопрос 11.3.

Какие из следующих классов наследуются от класса **Container**?

1) Window;
2) List;
3) Choice;
4) Component;
5) Panel;
6) Applet;
7) MenuComponent.

Вопрос 11.4.

Какие из следующих классов могут быть добавлены к объекту класса **Container**, используя его метод **add()**? (выберите 2)

1) Button;
2) CheckboxMenuItem;
3) Menu;
4) Canvas.

Вопрос 11.5.

Что будет результатом компиляции и выполнения следующего кода?

```
import java.awt.*;
class Quest5 {
  public static void main(String[] args) {
         Component b = new Button("Кнопка");
  System.out.print(((Button) b).getLabel());
  } }
```

1) ничего не будет выведено;
2) кнопка;
3) ошибка компиляции: класс **Quest5** должен наследоваться от класса **Applet;**
4) ошибка компиляции: ссылка на **Component** не может быть инициализирована объектом **Button;**
5) ошибка времени выполнения.

Глава 12
КЛАССЫ СОБЫТИЙ

События и их обработка

Подход к обработке событий в Java 2 основан на модели делегирования событий. В этой модели имеется блок прослушивания события (**`Listener`**), который ждет поступления события от источника, после чего обрабатывает его и возвращает управление. Источник – это объект, который генерирует событие, если изменяется его внутреннее состояние. Блоки прослушивания представляют собой объекты классов, которые создаются путем реализации интерфейсов прослушивания событий, определенных пакетом **`java.awt.event`**. Соответствующие методы, объявленные в используемых интерфейсах, необходимо явно реализовать при создании собственных классов прослушивания. Этот метод является обработчиком события.

Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются и принимают копию объекта события. Источник вызывает метод – обработчик события объекта, являющегося блоком прослушивания, и передает методу объект события в качестве параметра. В качестве блоков прослушивания могут использоваться внутренние классы и даже блок, являющийся источником. В этом случае в методе, регистрирующем блок прослушивания в качестве параметра, используется указатель **`this`**.

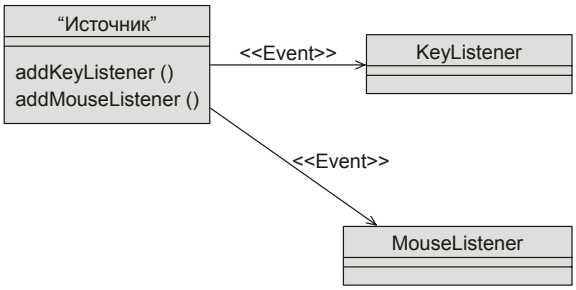

Рис. 12.1. Регистрация и обработка событий

В таблице приведены некоторые интерфейсы и их методы, которые должны быть реализованы в блоке прослушивания событий:

Интерфейсы	События
ActionListener	**actionPerformed(ActionEvent e)**
AdjustmentListener	**adjustmentValueChanged(AdjustmentEvent e)**
ComponentListener	**componentResized(ComponentEvent e)** **componentMoved(ComponentEvent e)** **componentShown(ComponentEvent e)** **componentHidden(ComponentEvent e)**
ContainerListener	**componentAdded(ContainerEvent e)** **componentRemoved(ContainerEvent e)**
FocusListener	**focusGained(FocusEvent e)** **focusLost(FocusEvent e)**
ItemListener	**itemStateChanged(ItemEvent e)**
KeyListener	**keyPressed(KeyEvent e)** **keyReleased(KeyEvent e)** **keyTyped(KeyEvent e)**
MouseListener	**mouseClicked(MouseEvent e)** **mousePressed(MouseEvent e)** **mouseReleased(MouseEvent e)** **mouseEntered(MouseEvent e)** **mouseExited(MouseEvent e)**
MouseMotionListener	**mouseDragged(MouseEvent e)** **mouseMoved(MouseEvent e)**
TextListener	**textValueChanged(TextEvent e)**
WindowListener	**windowOpened(WindowEvent e)** **windowClosing(WindowEvent e)** **windowClosed(WindowEvent e)** **windowIconified(WindowEvent e)** **windowDeiconified(WindowEvent e)** **windowActivated(WindowEvent e)**

Событие, которое генерируется в случае возникновения ситуации и затем передается блоку прослушивания для обработки, – это объект класса событий. В корне иерархии классов событий находится суперкласс **EventObject** из пакета **java.util**. Этот класс содержит два метода: **getSource()**, возвращающий источник событий, и **toString()**, воз-

вращающий строчный эквивалент события. Подкласс **AWTEvent** из пакета **java.awt** является суперклассом всех AWT-событий. Метод **getID()** определяет тип события, возникающего вследствие действий пользователя в визуальном приложении, а именно:

ActionEvent – генерируется: при нажатии кнопки; двойном щелчке клавишей мыши по элементам списка; при выборе пункта меню;

AdjustmentEvent – генерируется при изменении полосы прокрутки;

ComponentEvent – генерируется, если компонент скрыт, перемещен, изменен в размере или становится видимым;

FocusEvent – генерируется, если компонент получает или теряет фокус ввода и т.д.

Класс **InputEvent** является абстрактным суперклассом событий ввода (клавиатуры или мыши). Для регистрации события приложение-источник должно вызвать метод, регистрирующий блок прослушивания этого события. Форма метода:

public void addТип**Listener(**Тип**Listener** el**)**

Здесь **el** – ссылка на блок прослушивания события, получающая уведомление о событии, Тип – имя события, например: **addKeyListener()** – прослушивание событий клавиатуры.

Чтобы рассмотреть события, связанные с обработкой событий клавиатуры, необходимо реализовать интерфейс **KeyListener,** т.е. определить три метода, объявленные в этом интерфейсе. При нажатии клавиши генерируется событие со значением **KEY_PRESSED**. Это приводит к запросу обработчика событий **keyPressed()**. Когда клавиша отпускается, генерируется событие со значением **KEY_RELEASED** и выполняется обработчик **keyReleased()**. Если нажатием клавиши сгенерирован символ, то посылается уведомление о событии со значением **KEY_TYPED** и вызывается обработчик **keyTyped()**.

```
/* пример # 1 : обработка событий клавиатуры :
MyKey.java */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class MyKey extends Applet
                  implements KeyListener {
           String msg = " ";
      int x = 0, y = 20; //координаты вывода
      public void init() {
/*регистрация данного класса в качестве блока прослу-
шивания */
      addKeyListener(this);
```

```
      requestFocus(); //запрос фокуса ввода
      }
//реализация всех трех методов интерфейса KeyListener
      public void keyPressed(KeyEvent e) {
            showStatus("Key Down");
      } //отображение в строке состояния
      public void keyReleased(KeyEvent e) {
            showStatus("Key Up");
      } //отображение в строке состояния
      public void keyTyped(KeyEvent e) {
            msg += e.getKeyChar();
            repaint();//перерисовать
      }
      public void paint(Graphics g) {
      //значение клавиши в позиции вывода
      g.drawString(msg, x, y);
      }
}
```

Коды специальных клавиш (перемещение курсора, функциональных клавиш) недоступны через **keyTyped()**, для обработки нажатия этих клавиш используется метод **keyPressed()**.

В следующем апплете проверяется принадлежность точки нажатия мыши прямоугольнику.

```
/* пример # 2 : события нажатия клавиши мыши :
MyRect.java */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MyRect extends JApplet
            implements MouseListener {
   Rectangle a = new Rectangle(20, 20, 100, 60);
      public void init() {
             setBackground(Color.yellow);
/* регистрация данного класса в качестве блока про-
слушивания */
       addMouseListener(this);
      }
/* реализация всех пяти методов интерфейса
MouseListener */
      public void mouseClicked(MouseEvent me) {
   int x = me.getX();
   int y = me.getY();
```

```
      if (a.contains(x,y))
System.out.println("клик в зеленом прямоугольнике");
      else
System.out.println("клик в желтом фоне");
     }
//реализация остальных методов пустая
     public void mouseEntered(MouseEvent e) {}
     public void mouseExited(MouseEvent e) {}
     public void mousePressed(MouseEvent e) {}
     public void mouseReleased(MouseEvent e) {}
     public void paint(Graphics g) {
     g.setColor(Color.green);
         g.fillRect(a.x, a.y, a.width, a.height);
  }
}
```

Способ обработки событий в компонентах Swing – это интерфейс (графические компоненты) и реализация (код, который запускается при возникновении события). Каждое событие содержит сообщение, которое может быть обработано в разделе реализации.

При использовании компонента **JButton** определяется событие, связанное с нажатием кнопки. Для регистрации заинтересованности блока прослушивания в этом событии вызывается метод **addActionListener()** класса **JButton**. Этот метод ожидает аргумент, являющийся объектом, реализующим интерфейс **ActionListener**. Интерфейс содержит единственный метод **actionPerformed()**, который нужно реализовать.

```
/* пример # 3 : регистрация, генерация и обработка
ActionEvent : MyButton.java */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MyButton extends JApplet
          implements ActionListener {
     String msg;
     JButton yes, no;
public void init() {
 yes = new JButton ("yes");
 no = new JButton ("no");
     Container c = getContentPane();
     c.setLayout(new FlowLayout());
     c.add(yes);
     c.add(no);
  yes.addActionListener(this);//регистрация события
```

```
  no.addActionListener(this); //регистрация события
  }
public void actionPerformed(ActionEvent ae) {
   String str = ae.getActionCommand();
 if (str.equals("yes"))
     msg = "Button <yes> is pressed";
 else
     msg = "Button <no> is pressed";
          showStatus(msg);
    }
  }
```

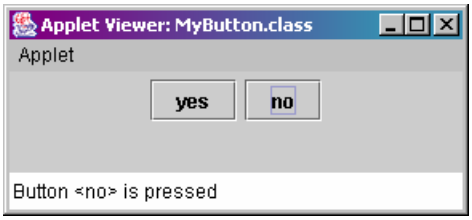

Рис. 12.2. Результат нажатия кнопки отображен в строке состояния

При создании кнопки вызывается конструктор **JButton** со строкой, которую нужно поместить на кнопке. **JButton** – это компонент, который автоматически заботится о своей перерисовке. Размещение кнопки на форме обычно производится внутри метода **init()** вызовом метода **add()** класса **Container**.

Классы-адаптеры

Для того чтобы не реализовывать все методы из соответствующих интерфейсов при создании классов – блоков прослушивания событий, используются классы-адаптеры. Такой класс содержит пустую реализацию всех методов из интерфейсов прослушивания событий, который он расширяет. При этом определяется новый класс, действующий как блок прослушивания событий, который расширяет один из имеющихся адаптеров и реализует только те события, которые требуется обрабатывать.

Например, класс **MouseMotionAdapter** имеет два метода: **mouseDragged()** и **mouseMoved()**. Сигнатуры этих пустых методов точно такие же, как в интерфейсе **MouseMotionListener**. Если существует заинтересованность только в событиях перетаскивания мыши, то можно просто расширить адаптер **MouseMotionAdapter** и переопределить метод **mouseDragged()** в своем классе. Событие же перемещения мыши обрабатывала бы реализация метода **mouseMoved(),** которую можно оставить пустой.

Когда происходит событие, связанное с окном, вызываются обработчики событий, определенные для этого окна. При создании оконного приложения используется метод **main()**, создающий для него окно верхнего уровня. После этого программа будет функционировать как приложение GUI, а не консольная программа. Программа поддерживается в работоспособном состоянии, пока не закрыто окно.

Для создания графического интерфейса потребуется предоставить место (окно), в котором он будет отображаться. Если программа является приложением, подобные действия она должна выполнять самостоятельно. В самом общем смысле окно является контейнером, т.е. областью, на которой рисуется пользовательский интерфейс. Графический контекст инкапсулирован в классе и доступен двумя способами:

- при переопределении методов **paint(),update()**;
- через возвращаемое значение метода **getGraphics()** класса **Component**.

Событие **FocusEvent** предупреждает программу, что компонент получил или потерял фокус ввода. Класс **InputEvent** является суперклассом для классов **KeyEvent** и **MouseEvent**. Событие **WindowEvent** извещает программу, что был активизирован один из системных элементов управления окна.

В следующем примере приложение создает объект **MyMouseWithFrame** и передает ему управление сразу же в методе **main()**.

```
/* пример # 4 : применение адаптеров :
MyMouseWithFrame.java */
import java.awt.*;
import java.awt.event.*;
public class MyMouseWithFrame extends Frame
           implements ActionListener {
      private Button button = new Button("Button");
      public String msg = "none";
      int count;
   public MyMouseWithFrame() {
      addMouseListener(new MyMouseAdapter(this));
      setLayout(null);
      setBackground(new Color(255, 255, 255));
      setForeground(new Color(0, 0, 255));
      button.setBounds(100, 100, 50, 20);
      button.addActionListener(this);
      add(button);
         }
   public void paint(Graphics g) {
```

```
      g.drawString(msg, 80, 50);
      }
   public void actionPerformed(ActionEvent e) {
      msg = "Button is pressed " + ++count;
      repaint();
       }
   public static void main(String[] args) {
      MyMouseWithFrame myf = new MyMouseWithFrame();
  myf.addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
                      System.exit(0);
                }
           });
      myf.setSize(new Dimension(250, 200));
      myf.setTitle("Frame Application");
      myf.setVisible(true);
      }
}
class MyMouseAdapter extends MouseAdapter {
      public MyMouseWithFrame mym;
   public MyMouseAdapter(MyMouseWithFrame mym) {
      this.mym = mym;
      }
   public void mousePressed(MouseEvent me) {
      mym.msg = "Mouse button is pressed";
      mym.repaint();
      }
   public void mouseReleased(MouseEvent me){
      mym.msg = "Mouse button is released";
      mym.repaint();
      }
}
```

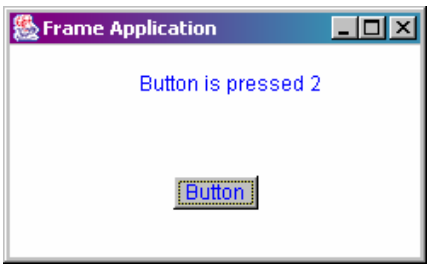

Рис. 12.3. Приложение с классами-адаптерами

Конструктор класса **MyMouseWithFrame** использует метод **addMouseListener(new MyMouseAdapter(this))** для регистрации событий мыши. При создании объекта класса **MyMouseWithFrame** этот метод сообщает объекту, что он заинтересован в обработке определенных событий. Однако вместо того, чтобы известить его об этом прямо, конструктор организует посылку ему предупреждения через объект класса **MyMouseAdapter**. Абстрактный класс **MouseAdapter** используется для обработки событий, связанных с мышью при создании блока прослушивания, и содержит следующие переопределяемые методы **mousePressed(MouseEvent e)**, **mouseReleased(MouseEvent e)**.

Класс **MyMouseWithFrame** также обрабатывает событие класса **WindowEvent**. Когда объект генерирует событие **WindowEvent**, объект **MyMouseWithFrame** анализирует, является ли оно событием **WindowClosing**. Если это не так, объект **MyMouseWithFrame** игнорирует его. Если получено ожидаемое событие, в программе запускается процесс завершения ее работы.

Абстрактный класс **WindowAdapter** используется для приема и обработки событий окна при создании объекта прослушивания. Класс содержит методы: **windowActivated(WindowEvent e)**, вызываемый при активизации окна; **windowClosing(WindowEvent e)**, вызываемый при закрытии окна, и др.

Задания к главе 12

1. Создать фрейм с областью для рисования “пером”. Создать меню для выбора цвета и толщины линии.
2. Создать апплет с областью для рисования “пером”. Создать меню для выбора цвета и толщины линии.
3. Создать фрейм с областью для рисования. Добавить кнопки для выбора цвета (каждому цвету соответствует своя кнопка), кнопку для очистки окна. Рисование на панели со скроллингом.
4. Создать апплет с областью для рисования. Добавить кнопки для выбора цвета (каждому цвету соответствует своя кнопка), кнопку для очистки окна. Рисование на панели со скроллингом.
5. Создать простой текстовый редактор, содержащий меню и использующий классы диалогов для открытия и сохранения файлов.
6. Создать апплет, содержащий (**JLabel**) с текстом “Простой апплет”, кнопку и текстовое поле (**JTextField**), в которое при каждом нажатии на кнопку выводится по одной строке из текстового файла.
7. Изменить апплет для предыдущей задачи таким образом, чтобы он мог работать и как апплет, и как приложение.

8. Составить программу для управления скоростью движения точки по апплету. Одна кнопка увеличивает скорость, другая – уменьшает. Каждый щелчок изменяет скорость на определенную величину.
9. Изобразить в окне гармонические колебания точки вдоль некоторого горизонтального отрезка. Если длина отрезка равна q, то расстояние от точки до левого конца в момент времени t можно считать равным q(1 + cos(wt))/2, где w – некоторая константа. Предусмотреть поля для ввода указанных величин и кнопку для остановки и пуска процесса.
10. Для предыдущей задачи предусмотреть возможность управления частотой колебаний с помощью двух кнопок. С помощью других двух кнопок (можно клавиш) управлять амплитудой, т.е. величиной q.
11. Построить в апплете ломаную линию по заданным вершинам. Координаты вершин вводятся через текстовое поле и фиксируются нажатием кнопки.
12. Нарисовать в апплете окружность с координатами центра и радиусом, вводимыми через текстовые поля.
13. Создать апплет со строкой, которая движется горизонтально, отражаясь от границ апплета и меняя при этом свой цвет на цвет, выбранный из выпадающего списка.
14. Создать апплет со строкой движущейся по диагонали. При достижении границ апплета все символы строки случайным образом меняют регистр. При этом шрифт меняется на шрифт, выбранный из списка.

Тестовые задания к главе 12

Вопрос 12.1.

Выбрать необходимое условие принадлежности класса к апплетам?

1) класс – наследник класса **Applet** при отсутствии метода **main();**
2) класс – наследник класса **Applet** или его подкласса;
3) класс – наследник класса **Applet** с переопределенным методом **paint();**
4) класс – наследник класса **Applet** с переопределенным методом **init();**
5) класс – наследник класса **Applet,** и все его методы объявлены со спецификатором доступа **public.**

Вопрос 12.2.

Дан код:

```
import java.awt.*;
public class Quest2 extends Frame{
public static void main(String[] args){
 Quest2 fr = new Quest2();
```

```
fr.setSize(222, 222);
fr.setVisible(true);
} }
```

Как сделать поверхность фрейма белой?

1) `fr.setBackground(Color.white);`
2) `fr.setColor(Color.white);`
3) `fr.Background(Color.white);`
4) `fr.color=Color.White;`
5) `fr.setColor(0,0,0).`

Вопрос 12.3.

Что произойдет при попытке компиляции и запуска следующего кода?

```
import java.awt.*;
import java.awt.event.*;
public class Quest3 extends Frame
      implements WindowListener {
       public Quest3(){
          setSize(300,300);
          setVisible(true);
          }
public void windowClosing(WindowEvent e){
                  System.exit(0);
                  }
public static void main(String args[]){
          Quest3 q = new Quest3();
          } }
```

1) ошибка компиляции;
2) компиляция и запуск с выводом пустого фрейма;
3) компиляция без запуска;
4) ошибка времени выполнения.

Вопрос 12.4.

Какие из приведенных классов являются классами-адаптерами?

1) `WindowAdapter;`
2) `WindowsAdapter;`
3) `AdjustmentAdapter;`
4) `ItemAdapter;`
5) `FocusAdapter.`

Вопрос 12.5.

Выберите из предложенных названий интерфейсы Event Listener.

1) `MouseMotionListener;`
2) `WindowListener;`
3) `KeyTypedListener;`
4) `ItemsListener.`

Глава 13
ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

В ранних (1.0.x) версиях Java использовались “тяжелые” компоненты AWT, связанные с аппаратными платформами и имеющие ограниченные возможности. Дальнейшее развертывание концепции “write once, run everywhere” (“написать однажды, запускать везде”) привело к тому, что в версии 1.1.x наметился переход к таким компонентам, которые бы не были завязаны на конкретные “железо” и операционные системы. Такого рода классы компонентов, написанные на Java, были объединены в библиотеку под названием Swing. Эти классы доступны разработчикам в составе как JDK, так и отдельного продукта JFC (Java Foundation Classes). Причем для совместимости со старыми версиями JDK старые компоненты из AWT остались нетронутыми, хотя компания JavaSoft, отвечающая за выпуск JDK, рекомендует не смешивать в одной и той же программе старые и новые компоненты. Кроме пакета Swing указанные библиотеки содержат большое число компонентов JavaBeans, которые могут использоваться как для ручной, так и для визуальной разработки пользовательских интерфейсов.

Менеджеры размещения

Перед использованием управляющих компонентов (например кнопок) их надо расположить на форме в нужном порядке. Java для этого использует менеджеры размещения. Они определяют способ, который панель использует, для задания порядка размещения управляющего элемента на форме. Менеджеры размещения контролируют, как выполняется позиционирование компонентов, добавляемых в окна, а также их упорядочение. Если пользователь изменяет размер окна, менеджер размещения переупорядочивает компоненты в новой области так, чтобы они оставались видимыми и в то же время сохранили свои позиции относительно друг друга.

Менеджер размещения представляет собой интерфейс **`LayoutManager`**, реализованный в виде классов, например: **`FlowLayout`**, **`BorderLayout`**, **`GridLayout`**, **`CardLayout`**, **`BoxLayout`** и т. д.

`FlowLayout` – менеджер поточной компоновки. При этом компоненты размещаются от левого верхнего угла окна, слева направо и сверху вниз, как и обычный текст. Этот менеджер используется по умолчанию при добавлении компонентов в апплеты. При использовании библиотеки AWT менеджер **`FlowLayout`** представляет собой класс, объявленный следующим образом:

```
public class FlowLayout extends Object
 implements LayoutManager, Serializable { }
/* пример # 1 : поточная компоновка по центру:
FlowLayoutEx.java */
   import java.applet.*;
   import java.awt.*;
   public class FlowLayoutEx extends Applet {
      Component c[] = new Component[6];
      public void init() {
   String[] msg = {"Метка 1", "Метка 2", "Метка 3"};
   String[] str = { "--1--", "--2--", "--3--" };
            setLayout(new FlowLayout());//по умолчанию
      setBackground(Color.gray);
      setForeground(Color.getHSBColor(1f, 1f, 1f));
            for (int i = 0; i < c.length - 3; i++) {
                  c[i] = new Button(str[i]);
                  add(c[i]);
                  c[i + 3] = new Label(msg[i]);
                  add(c[i + 3]);
            }
            setSize(220, 150);
      }
   }
```

Метод **setLayout(LayoutManager mgr)** устанавливает менеджер размещения для данного контейнера. Результаты работы апплета приведены на рисунке.

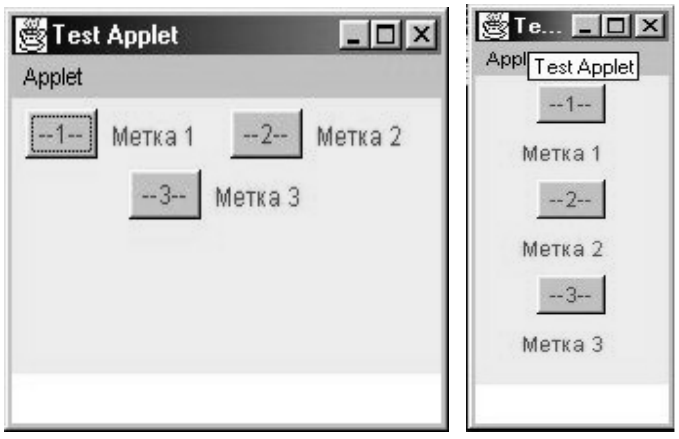

Рис. 13.1. Размещение компонентов **FlowLayout**

BorderLayout позволяет позиционировать элементы в областях фиксированного размера, граничащих со сторонами фрейма, которые обозначаются параметрами: **NORTH**, **SOUTH**, **EAST**, **WEST**. Остальное пространство обозначается как **CENTER**.

```
/* пример # 2 : фиксированная компоновка по областям:
BorderLayoutEx.java */
   import java.applet.*;
   import java.awt.*;
   public class BorderLayoutEx extends Applet {
      public void init() {
            setLayout(new BorderLayout());
      add(new Button("--0--"), BorderLayout.WEST);
      add(new Button("--1--"), BorderLayout.WEST);
//кнопка --1-- будет нарисована поверх кнопки --0--
      add(new Button("--2--"), BorderLayout.SOUTH);
      add(new Button("--3--"), BorderLayout.EAST);
      add(new Label("*BorderLayout*"),
            BorderLayout.CENTER);
      add(new Checkbox("Выбор"), BorderLayout.NORTH);
            setSize(220, 150);
      }
   }
```

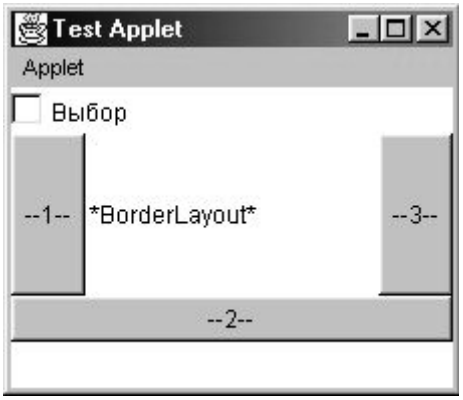

Рис. 13.2. Размещение компонентов **BorderLayout**

CardLayout создает вкладки, содержимое которых отображается при выполнении щелчка на заголовке. Каждое размещение можно представлять отдельной пронумерованной картой в колоде, которая может быть перетасована так, чтобы в данный момент наверху находилась любая карта.

Для работы с окнами используются классы **JFrame** и **JPanel** из библиотеки Swing вместо **Frame** и **Panel** из стандартного набора классов AWT. Панели класса **JPanel** должны добавляться в окна апплетов и фреймов исключительно методом **setContentPane()**, а не методом **add()**, как это делается в AWT.

```
/* пример # 3 : компоновка с вкладками :
CardLayoutEx.java */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class CardLayoutEx extends Applet implements
ActionListener {
     Button next, previous;
     Panel base;
     Checkbox ne1, ne2;
     TextField pr1;
     CardLayout cl;
     Label fornext, forprev;
  public void init() {
     previous = new Button("Предыдущий вопрос");
     next = new Button("Следующий вопрос");
           add(previous);
           add(next);
          cl = new CardLayout();
          this.setSize(600, 200);
           base = new Panel();
           base.setLayout(cl);
     forprev = new Label("1. Вычислить: 23|14");
     pr1 = new TextField(3);
     fornext = new Label(
     "2. Результат: new Object()==new Object()");
          ne1 = new Checkbox("true");
          ne2 = new Checkbox("false");
          Panel first = new Panel();
            first.add(forprev);
            first.add(pr1);
          Panel second = new Panel();
            second.add(fornext);
            second.add(ne1);
            second.add(ne2);
          base.add(first, "PREVIOUS");
          base.add(second, "NEXT");
            add(base);
```

```
                next.addActionListener(this);
                previous.addActionListener(this);
        }
    public void actionPerformed(ActionEvent ae) {
            if (ae.getSource() == next) {
                cl.show(base, "NEXT");
            } else {
                cl.show(base, "PREVIOUS");
            }
        }
}
```

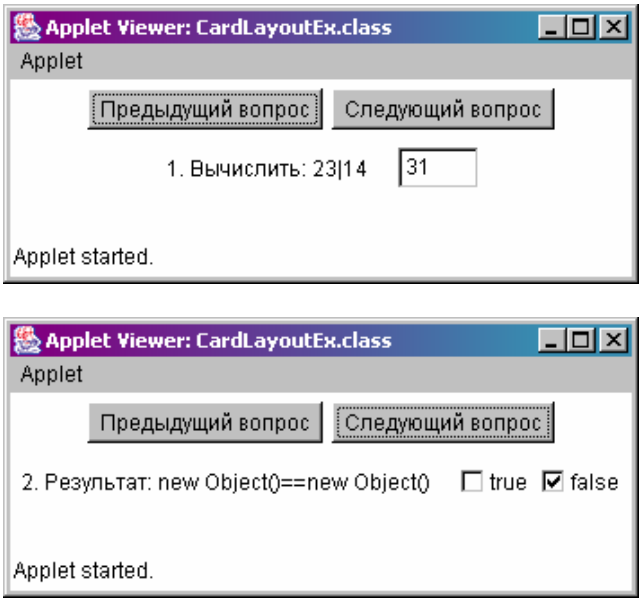

Рис. 13.3. Размещение компонентов **CardLayout**

GridLayout разделяет фрейм на заданное количество рядов и колонок. В отличие от него компоновка **BoxLayout** размещает некоторое количество компонентов по вертикали или горизонтали. На способ расположения компонентов изменение размеров фрейма не влияет.

```
/* пример # 4 : компоновка в табличном виде :
GridLayoutEx.java */
   import java.applet.Applet;
   import java.awt.*;
   public class GridLayoutEx extends Applet {
     Component b[]=new Component[7];
      public void init() {
            setLayout(new GridLayout(2, 4));
```

```
            for (int i = 0; i < b.length; i++)
            add((b[i] = new Button("(" + i +  ")")));
            }
   }
```

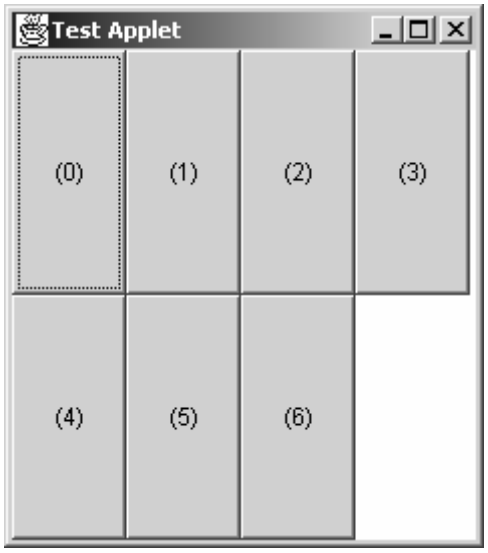

Рис. 13.4. Размещение компонентов **GridLayout**

Компоновка **BoxLayout** позволяет группировать элементы в подобластях фрейма в строки и столбцы. Возможности класса **Box** позволяют размещать компоненты в рамке, ориентированной горизонтально или вертикально.

```
/* пример # 5 : компоновка в группах с ориентацией :
BoxLayoutEx.java */
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
public class BoxLayoutEx extends JFrame {
      public BoxLayoutEx() {
            Container c = getContentPane();
            setBounds(20, 80, 300, 300);
            c.setLayout(new BorderLayout());
      //выбор ориентации рамки
            Box row = Box.createHorizontalBox();
            for (int j = 0; j < 4; j++) {
      JButton b = new JButton("Кн " + j);
      b.setFont(new Font("Tahoma", 1, 10 + j * 2));
                  row.add(b);
            }
```

```
            c.add(row, BorderLayout.SOUTH);
        //группировка компонентов и ориентация группы
            JPanel col = new JPanel();
            col.setLayout(
        new BoxLayout(col, BoxLayout.Y_AXIS));
            col.setBorder(new TitledBorder(
        new EtchedBorder(), "Столбец"));
            for (int j = 0; j < 4; j++) {
        JButton b = new JButton("Кнопка" + j);
        b.setFont(new Font("Tahoma", 1, 10 + j * 2));
                    col.add(b);
            }
            c.add(col, BorderLayout.WEST);
        }
        public static void main(String args[]) {
            BoxLayoutEx bl = new BoxLayoutEx();
            bl.setVisible(true);
        }
}
```

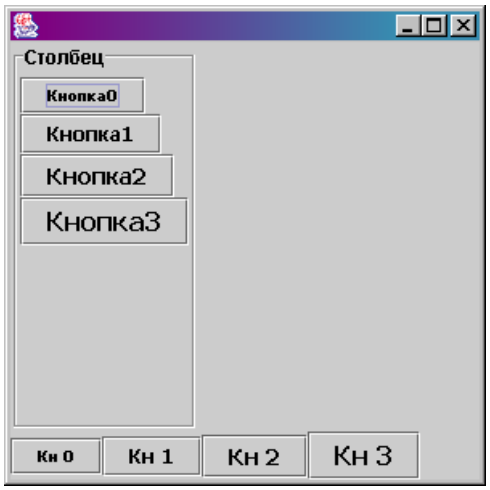

Рис. 13.5. Размещение компонентов **BoxLayout** и **Box**

Для того чтобы располагать компоненты в произвольных областях фрейма, следует установить для менеджера размещений значение **null** и воспользоваться методом **setBounds()**.

```
/* пример # 6 : произвольное размещение :
NullLayoutEx.java */
import java.awt.Container;
import javax.swing.*;
public class NullLayoutEx extends JFrame {
```

```
      public NullLayoutEx() {
            Container c = getContentPane();
//указание размеров фрейма
            this.setBounds(20, 80, 300, 300);
            c.setLayout(null);
            JButton jb = new JButton("Кнопка");
//указание координат и размеров кнопки
            jb.setBounds(200, 50, 90, 40);
            c.add(jb);
            JTextArea jta = new JTextArea();
//указание координат и размеров текстовой области
            jta.setBounds(10, 130, 180, 70);
            jta.setText("Здесь можно вводить текст");
            c.add(jta);
      }
      public static void main(String args[]) {
            NullLayoutEx nl = new NullLayoutEx();
            nl.setVisible(true);
      }
}
```

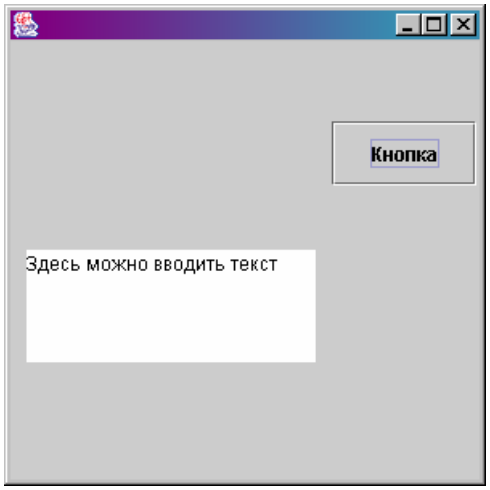

Рис. 13.6. Произвольное размещение компонентов

Элементы управления

Элементы управления из пакета AWT являются наследниками класса **`Component`**. При использовании пакета Swing компоненты наследуются от класса **`JComponent`**, производного от класса **`Container`**.

Текстовые метки **Label**, **JLabel** создаются с помощью конструкторов **Label()**, **Label(String str)**, **Label(String str,int alignment)**.

Списки – **List** и **JList**.

Полосы прокрутки – **ScrollBar** и **JScrollBar.**

Однострочная область ввода – **TextField** и **JTextField**.

Многострочная область ввода – **TextArea** и **JTextArea**.

Кнопки – **Button** и **JButton**.

Кнопки – **CheckBox** и **JCheckBox**, **RadioButton** и **JRadioButton**.

От **AbstractButton** наследуются два основных кнопочных класса: **JButton** и **JToggleButton**. Первый служит для создания обычных кнопок с разнообразными возможностями, а второй – для создания радиокнопок (класс **JRadioButton**) и отмечаемых кнопок (класс **JCheckBox**). Помимо названных от **AbstractButton** наследуется пара классов **JCheckBoxMenuItem** и **JRadioButtonMenuItem**, используемых для организации тех меню, пункты которых должны быть оснащены отмечаемыми радиокнопками.

Процесс создания кнопок достаточно прост: вызывается конструктор **JButton** с меткой, которую нужно поместить на кнопке. Класс **JButton** библиотеки Swing для создания обычных кнопок предлагает несколько различных конструкторов: **JButton()**, **JButton(String s)**, **JButton(Icon i)**, **JButton(String s, Icon i)**.

Если используется конструктор без параметров, то получится абсолютно пустая кнопка. Задав текстовую строку, получим кнопку с надписью. Для создания кнопки с рисунком конструктору передается ссылка на класс пиктограммы. Класс **JButton** содержит несколько десятков методов. **JButton** – это компонент, который автоматически перерисовывается как часть обновления. Это означает, что не нужно явно вызывать перерисовку кнопки, как и любого управляющего элемента; он просто помещается на форму и сам автоматически заботится о своей перерисовке. Чтобы поместить кнопку на форму, необходимо выполнить это в методе **init()**. Каждый раз, когда кнопка нажимается, генерируется action-событие. Оно посылается блокам прослушивания, зарегистрированным для приема события от этого компонента.

```
// пример # 7 : кнопка и ее методы : MyButtons.java
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class MyButtons extends JApplet
          implements ActionListener {
     String msg;
```

```
      JButton yes, maybe;
      JLabel lbl;
      JTextField txt;
      public void init() {
            yes = new JButton("yes");
yes.setActionCommand("yes"); //определение команды
            maybe = new JButton("may be");
            lbl = new JLabel("");
            txt = new JTextField(10);
      Container c = getContentPane();
            c.setLayout(new FlowLayout());
            c.add(yes);
            c.add(maybe);
            c.add(lbl);
            c.add(txt);
      yes.addActionListener(this);
      maybe.addActionListener(this);
      }
      public void actionPerformed(ActionEvent e) {
            String str =
                  ((JButton) e.getSource()).getText();
            if (str.equals("yes"))
                  msg = "Button <yes> is pressed”;
            else
                  msg = " Button <may be> is pressed ";
            lbl.setText(msg);
                  String name =
      e.getActionCommand(); // извлечение команды
            txt.setText(name);
      }
}
```

Рис. 13.7. Апплет с кнопками, меткой и текстовым полем

Метод **getSource()** возвращает ссылку на объект, явившийся источником события, который преобразуется в объект **JButton**. Метод **getText()** в виде строки извлекает текст, который изображен на кнопке, и помещает его с помощью метода **setText()** объекта **JLabel** в объект **lbl**. При этом определяется, какая из кнопок была нажата.

Команда, ассоциированная с кнопкой, возвращается вызовом метода **getActionCommand()** класса **ActionEvent**, экземпляр которого содержит всю информацию о событии. Для отображения результата нажатия кнопки использован компонент **JTextField**, представляющий собой поле, где может быть размещен и изменен текст. Хотя есть несколько способов создания **JTextField**, самым простым является сообщение конструктору нужной ширины текстового поля. Как только **JTextField** помещается на форму, можно изменять содержимое, используя метод **setText()**. Реализацию действий, ассоциированных с нажатием кнопки, лучше производить в потоке во избежание "зависания"

Класс **JComboBox** применяется для создания раскрывающегося списка элементов, из которых пользователем производится выбор. Таким образом, данный элемент управления имеет форму меню. В неактивном состоянии компонент типа **JComboBox** занимает столько места, чтобы показывать только текущий выбранный элемент. Для определения выбранного элемента можно вызвать метод **getSelectedItem()** или **getSelectedIndex()**.

```
// пример # 8 : выпадающий список : MyComboBox.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MyComboBox extends JApplet
           implements ItemListener{
    JLabel lbl;
    JComboBox cb;
public void init(){
    lbl = new JLabel("0");
    cb = new JComboBox();//создание объекта
  Container c = getContentPane();
  c.setLayout(new FlowLayout());
//добавление элементов в выпадающий список
           cb.addItem("11");
           cb.addItem("22");
           cb.addItem("33");
           cb.addItem("44");
           cb.addItemListener(this);
  c.add(cb); //размещение объекта в контейнер
  c.add(lbl);
}
public void itemStateChanged(ItemEvent ie) {
            int arg =
Integer.valueOf((String)ie.getItem()).intValue();
           //возведение в квадрат
      double res = Math.pow(arg, 2);
```

```
      lbl.setText(Double.toString(res));
       }
}
```

Рис. 13.8. Выпадающий список

При выборе элемента списка генерируется событие **ItemEvent** и посылается всем блокам прослушивания, зарегистрированным для приема уведомлений о событиях данного компонента. Каждый блок прослушивания реализует интерфейс **ItemListener**. Этот интерфейс определяет метод **itemStateChanged()**. Объект **ItemEvent** передается этому методу в качестве аргумента. Приведенная программа позволяет выбрать из списка число, возводит его в квадрат и выводит в объект **JLabel**. Здесь также следует обратить внимание на способы преобразования строковой информации в числовую и обратно.

В следующем примере рассмотрено отслеживание изменения состояния объекта **JCheckBox**.

```
/* пример # 9 : отслеживание изменения состояния
флажка : MyCheckBox.java */
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class MyCheckBox extends JApplet
           implements ItemListener{
   JCheckBox cb;
   JLabel lbl;
public void init() {
   cb = new JCheckBox();
   lbl = new JLabel("initialization");
     Container c = getContentPane();
    c.setLayout(new FlowLayout());
     cb.addItemListener(this);
    c.add(cb);
    c.add(lbl);
}
public void itemStateChanged(ItemEvent ie) {
        String msg;
      if(ie.getStateChange() == ItemEvent.SELECTED)
            msg = "True";
```

```
      else   msg = "False";
             lbl.setText(msg);
   }
}
```

Рис. 13.9. Элемент управления **JCheckBox**

Блок прослушивания событий в ответ на генерацию события **ItemEvent** вызвал метод **getStateChange()**, извлекающий из объекта класса **ItemEvent** константу состояния, в данном случае **SELECTED**, прослушиваемого объекта класса **JCheckBox**.

Если требуется "группа кнопок" с поведением вида "исключающее или", то объекты класса **RadioButton** необходимо добавить в группу типа **ButtonGroup**. Но, как показывает приведенный ниже пример, любая кнопка типа **AbstractButton** может быть добавлена в **ButtonGroup**.

Для предотвращения повтора большого количества кода этот пример использует рефлексию, которую обеспечивает применение классов пакета **java.lang.reflect** для генерации различных типов кнопок. Группа кнопок создается в методе **makePanel()** и помещается в объект класса **JPanel**.

```
/* пример # 10 : различные типы кнопок и флажков :
DiffButtons.java */
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
public class DiffButtons extends JApplet {
     static String[] str = {"-1-", "-2-", "-3-"};
public void init() {
   Container c = getContentPane();
   c.setLayout(new FlowLayout());
   c.add(makePanel(JToggleButton.class, str));
   c.add(makePanel(JCheckBox.class, str));
   c.add(makePanel(JRadioButton.class, str));
}
static JPanel makePanel(Class aCl, String[] str) {
   ButtonGroup b = new ButtonGroup();
```

```
   JPanel p = new JPanel();
   String title = aCl.getName();
   title = title.substring(title.lastIndexOf('.')
                + 1);
         p.setBorder(new TitledBorder(title));
for(int j = 0; j < str.length; j++) {
    AbstractButton abs = new JButton("ошибка");
try {
   Constructor cons = aCl.getConstructor(
     new Class[] { String.class });
   abs = (AbstractButton)cons.newInstance(
     new Object[]{str[j]});
} catch(Exception e) {
System.err.println(e + "/n нельзя создать: " + aCl);
           }
      b.add(abs);
      p.add(abs);
     }
return p;
   }
}
```

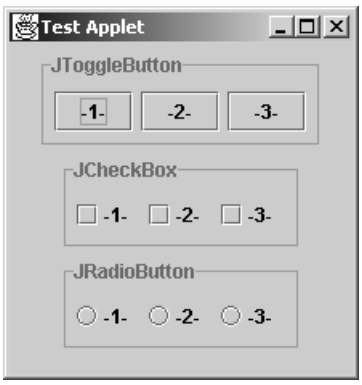

Рис. 13.10. Группы кнопок

Заголовок для бордюра берется из имени класса, от которого отсекается вся информация о пути. **AbstractButton** инициализируется с помощью **JButton**, которая имеет метку “ошибка”, так что если игнорируется сообщение исключения, то проблема видна на экране. Метод **getConstructor()** производит объект **Constructor**, который принимает массив аргументов типов в массиве **Class**, переданном **getConstructor()**. Затем вызывается метод **newInstance()**, создающий экземпляр кнопки заданного типа. После запуска программы видно, что все кнопки, объединенные в группы, показывают поведение вида “исключающее или”.

Для добавления в приложение различного вида всплывающих меню и диалоговых окон следует использовать обширные возможности класса **JOptionPane**. Эти возможности реализуются статическими методами класса вида **show**Имя**Dialog(**параметры**)**. Наиболее используемыми являются методы **showConfirmDialog()**, **showMessageDialog()**, **showInputDialog()** и **showOptionDialog()**.

Для подтверждения/отказа выполняемого в родительском окне действия применяется метод **showConfirmDialog()**.

```
/* пример # 11 : диалог Да/Нет : DemoConfirm.java */
import javax.swing.*;
public class DemoConfirm {
      public static void main(String[] args) {
            int result =
                  JOptionPane.showConfirmDialog(
                        null,
                        "Хотите продолжить?",
                        "Chooser",
                        JOptionPane.YES_NO_OPTION);
            if (result == 0)
                  System.out.println("You chose yes");
            else
                  System.out.println("You chose No");
      }
}
```

В качестве первого параметра метода указывается окно, к которому относится сообщение, но так как в данном случае здесь и далее используется консоль, то он равен **null**.

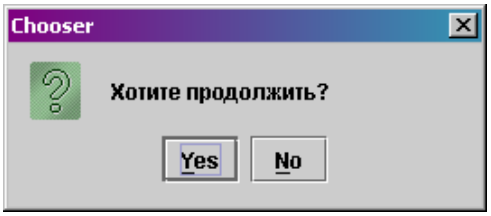

Рис. 13.11. Диалог выбора

Для получения вариаций указанного диалога можно использовать следующие константы: **YES_NO_CANCEL_OPTION**, **OK_CANCEL_OPTION**, **DEFAULT_OPTION** и некоторые другие.

Для показа сообщений (информационных, предупреждающих, вопросительных и т.д.) применяется метод **showMessageDialog()**.

```
/* пример # 12 : сообщение : DemoMessage.java */
import javax.swing.*;
public class DemoMessage {
```

```
      public static void main(String[] args) {
            JOptionPane.showMessageDialog(
                  null,
                  "Файл может быть удален!",
                  "Внимание!",
                  JOptionPane.WARNING_MESSAGE);
// ERROR_MESSAGE – сообщение об ошибке
// INFORMATION_MESSAGE - информационное сообщение
// WARNING_MESSAGE  - уведомление
// QUESTION_MESSAGE - вопрос
// PLAIN_MESSAGE - без иконки
      }
}
```

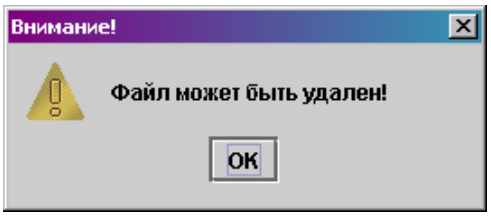

Рис. 13.12. Вывод сообщения

Если необходимо обязать пользователя приложения ввести какую-либо информацию на определенном этапе работы, то следует использовать возможности метода **showInputDialog()**. Причем простейший запрос создать очень легко.

```
/* пример # 13 : запрос на ввод : DemoInput.java */
import javax.swing.*;
public class DemoInput {
      public static void main(String[] args) {
            String str =
JOptionPane.showInputDialog("Please input a value");
          if (str != null)
            System.out.println("You input : " + str);
      }
}
```

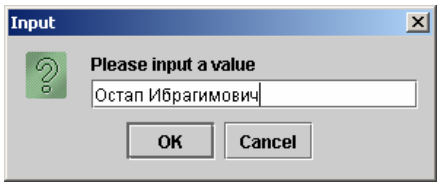

Рис. 13.13. Простейший запрос на ввод строки

Следующий пример предоставляет возможность выбора из заранее определенного списка значений.

```
/* пример # 14 : формирование запроса на выбор из
списка : DemoInputWithOptions.java */
import javax.swing.*;
public class DemoInputWithOptions {
      public static void main(String[] args) {
            Object[] possibleValues =
                  { "легкий", "средний", "трудный" };
            Object selectedValue =
                  JOptionPane.showInputDialog(
                        null,
                        "Выберите уровень",
                        "Input",
                        JOptionPane.INFORMATION_MESSAGE,
                        null,
                        possibleValues,
                        possibleValues[0]);
            // possibleValues[1] - элемент для фокуса
            // второй null – иконка по умолчанию
            if (selectedValue != null)
System.out.println("You input : " + selectedValue);
      }
}
```

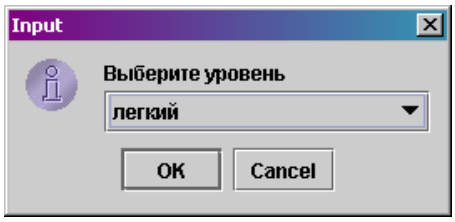

Рис. 13.14. Запрос на выбор из списка

Другим способом создания диалога с пользователем является применение возможностей класса **JDailog**. В этом случае можно создавать диалоги с произвольным набором компонентов.

```
// пример # 15 : произвольный диалог : DemoJDialog.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class MyDialog extends JDialog
                  implements ActionListener {
      JButton cancel = new JButton("Cancel");
      JButton ok = new JButton("Ok");
```

```
      JFrame parent;
      public MyDialog(JFrame parent, String name) {
            super(parent, name, true);
            this.parent = parent;
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            c.add(new JLabel("Exit ?"));
            ok.addActionListener(this);
            cancel.addActionListener(this);
            c.add(ok);
            c.add(cancel);
            setSize(200, 100);
            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setVisible(true);
      }
      public void actionPerformed(ActionEvent e) {
                  dispose();
            if(e.getActionCommand().equals("Ok"))
                  parent.dispose();
      }
}
public class DemoJDialog extends JFrame {
      private JButton jButton = new JButton("Dialog");
      DemoJDialog () {
            super("My DialogFrame");
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            jButton.addActionListener(
                        new ActionListener() {
        public void actionPerformed(ActionEvent e){
            JDialog jDialog =
      new MyDialog(JDialogDemo.this, "MyDialog");
                  }
            });
            c.add(jButton);
      }
      public static void main(String[] args) {
            DemoJDialog f = new DemoJDialog();
            f.setDefaultCloseOperation(EXIT_ON_CLOSE);
            f.setSize(200, 120);
            f.setVisible(true);
      }
}
```

Рис. 13.15. Произвольный диалог

Для создания пользовательского меню следует воспользоваться возможностями классов **JMenu**, **JMenuBar** и **JMenuItem**.

```
/* пример # 16 : создание меню : SimpleMenu.java */
import javax.swing.*;
import java.awt.event.*;
public class SimpleMenu extends JApplet
                        implements ActionListener {
      JMenu menu;
      JMenuItem item1, item2;
      public void init() {
            JMenuBar menubar = new JMenuBar();
            setJMenuBar(menubar);
            menu = new JMenu("Main");
            item1 = new JMenuItem("About");
            item2 = new JMenuItem("Exit");
            item1.addActionListener(this);
            item2.addActionListener(this);
            menu.add(item1);
            menu.add(item2);
            menubar.add(menu);
      }
      public void actionPerformed(ActionEvent e) {
            JMenuItem jmi = (JMenuItem)e.getSource();
            if (jmi.equals(item2)) System.exit(0);
            else  showStatus("My Simple Menu");
      }
}
```

Рис. 13.16. Простейшее меню

Задания к главе 13

Вариант А

1. Создать апплет. Поместить на него текстовое поле **JTextField**, кнопку **JButton** и метку **JLabel**. В метке отображать все введенные символы, разделяя их пробелами.
2. Поместить в апплет две панели **JPanel** и кнопку. Первая панель содержит поле ввода и метку "Поле ввода"; вторая – поле вывода и метку "Поле вывода". Для размещения в окне двух панелей и кнопки "Скопировать" использовать менеджер размещения **BorderLayout**.
3. Изменить задачу 2 так, чтобы при нажатии на кнопку "Скопировать" текст из поля ввода переносится в поле вывода, а поле ввода очищается.
4. Задача 2 модифицируется так, что при копировании поля ввода нужно, кроме собственно копирования, организовать занесение строки из поля ввода во внутренний список. При решении использовать коллекцию, в частности **ArrayList**.
5. К условию задачи 2 добавляется еще одна кнопка с надписью "Печать". При нажатии на данную кнопку весь сохраненный список должен быть выведен в консоль. При решении использовать коллекцию, в частности **TreeSet**.
6. Написать программу для построения таблицы значений функции $y = a\sqrt{x} * \cos(ax)$. Использовать метку **JLabel**, содержащую текст "Функция: $y = a\sqrt{x} * \cos(ax)$"; панель, включающую три текстовых поля **JTextField**, содержащих значения параметра, шага (например, 0.1) и количества точек. Начальное значение x=0. С каждым текстовым полем связана метка, содержащая его название. В приложении должно находиться текстовое поле со скроллингом, содержащее полученную таблицу.
7. Создать форму с набором кнопок так, чтобы надпись на первой кнопке при ее нажатии передавалась на следующую, и т.д.
8. Создать форму с выпадающим списком так, чтобы при выборе элемента списка на экране появлялись GIF-изображения, двигающиеся в случайно выбранном направлении по апплету.
9. В апплете изобразить прямоугольник (окружность, эллипс, линию). Направление движения объекта по экрану изменяется на противоположное щелчком по клавише мыши. При этом каждый второй щелчок меняет цвет фона.
10. Создать фрейм с изображением окружности. Длина дуги окружности изменяется нажатием клавиш от 1 до 9.
11. Создать фрейм с кнопками. Кнопки "вверх", "вниз", "вправо", "влево" двигают в соответствующем направлении линию. При достижении границ фрейма линия появляется с противоположной стороны.

12. Создать фрейм и разместить на нем окружность (одну или несколько). Объект должен “убегать” от указателя мыши. При приближении на некоторое расстояние объект появляется в другом месте фрейма.
13. Создать фрейм/апплет с изображением графического объекта. Объект на экране движется к указателю мыши, когда последний находится в границах фрейма/апплета.
14. Изменить задачу 12 так, чтобы количество объектов зависело от размеров апплета и изменялось при “перетягивании” границы в любом направлении.
15. Промоделировать в апплете вращение спутника вокруг планеты по эллиптической орбите. Когда спутник скрывается за планетой, то не он виден.
16. Промоделировать в апплете аналоговые часы (со стрелками) с кнопками для увеличения/уменьшения времени на час/минуту.

Вариант В

Для заданий варианта В главы 4 создать графический интерфейс для занесения информации при инициализации объекта класса, для выполнения действий, предусмотренных заданием и для отправки сообщений другому пользователю системы.

Тестовые задания к главе 13

Вопрос 13.1.

Какой менеджер размещения использует таблицу с ячейками равного размера?

1) `FlowLayout;`
2) `GridLayout;`
3) `BorderLayout;`
4) `CardLayout.`

Вопрос 13.2.

Дан код:

```
import java.awt.*;
public class Quest2 extends Frame{
      Quest2(){
      Button yes = new Button("YES");
      Button no = new Button("NO");
      add(yes);
      add(no);
      setSize(100, 100);
      setVisible(true);
    }
    public static void main(String[] args){
```

```
    Quest2 q = new Quest2();
  } }
```

В результате будет выведено:

1) две кнопки, занимающие весь фрейм, YES – слева и NO – справа;
2) одна кнопка YES, занимающая целый фрейм;
3) одна кнопка NO, занимающая целый фрейм;
4) две кнопки наверху фрейма – YES и NO.

Вопрос 13.3.

Какое выравнивание устанавливается по умолчанию для менеджера размещений **`FlowLayout`**?

1) `FlowLayout.RIGHT;`
2) `FlowLayout.LEFT;`
3) `FlowLayout.CENTER;`
4) `FlowLayout.LEADING;`
5) указывается явно.

Вопрос 13.4.

Сколько кнопок будет размещено в приведенном ниже апплете?

```
import java.awt.*;
public class Quest4 extends java.applet.Applet{
  Button b = new Button("YES");
  public void init(){
  add(b);
  add(b);
  add(new Button("NO"));
  add(new Button("NO"));
  }}
```

1) одна кнопка с YES и одна кнопка NO;
2) одна кнопка с YES и две кнопки NO;
3) две кнопки с YES и одна кнопка NO;
4) две кнопки с YES и две кнопки NO.

Вопрос 13.5.

Объект **`JCheckBox`** объявлен следующим образом:

```
JCheckBox ob = new JCheckBox();
```

Какая из следующих команд зарегистрирует его в блоке прослушивания событий?

1) `ob.addItemListener();`
2) `ob.addItemListener(this);`
3) `addItemListener(this);`
4) `addItemListener();`
5) ни одна из приведенных.

Глава 14
ПОТОКИ И МНОГОПОТОЧНОСТЬ

К большинству современных Web-приложений выдвигаются требования одновременной поддержки многих пользователей и разделения информационных ресурсов. Потоки – средство, которое поможет организовать одновременное выполнение нескольких задач с помощью многопоточности – использования нескольких потоков управления в одной программе. Это особенно важно при разработке распределенных информационных систем. Существуют два способа запуска класса в потоке: расширение класса **Thread** и реализация интерфейса **Runnable**.

```
// пример # 1 : расширение класса Thread : Talk.java
class Talk extends Thread {
  public void run() {
      for (int i = 0; i < 8; i++) {
       System.out.println("Talking");
      try {
            //остановка на 400 миллисекунд
            Thread.sleep(400);
      } catch (InterruptedException e) {
            System.out.print(e);
                  }
               }
        }
}
```

При реализации интерфейса **Runnable** необходимо определить его единственный абстрактный метод **run()**. Например:

```
/* пример # 2 : реализация интерфейса Runnable :
TalkWalk.java */
class Walk implements Runnable {
    public void run() {
      for (int  i = 0; i < 8; i++) {
             System.out.println("Walking");
      try {
            Thread.sleep(300);
      } catch (InterruptedException e) {
            System.out.println(e);
```

```
                }
            }
        }
}
public class TalkWalk {
      public static void main(String[] args) {
      //новые потоки
      Talk talk  = new Talk();
      Thread walk = new Thread(new Walk());
      Walk w = new Walk();//просто объект
         talk.start();
         walk.start();
//w.run();//выполнится метод, но поток не запустится!
      }
}
```

Использование двух потоков для объектов классов **Talk** и **Walk** приводит к выводу строк: `Talking Walking`.

Жизненный цикл потока

При выполнении программы объект класса **Thread** может быть в одном из четырех состояний: “новый”, “работоспособный”, “неработоспособный” и “пассивный”. При создании потока он получает состояние “новый” и не выполняется. Для перевода потока из состояния “новый” в состояние “работоспособный” следует выполнить метод **start()**, который вызывает метод **run()** – основное тело потока. Интерфейс **Runnable** не имеет метода **start()**, а только единственный метод **run()**. Поэтому для запуска такого потока, как **Walk**, следует создать объект класса **Thread** и передать объект **Walk** его конструктору, однако при прямом вызове метода **run()** поток не запустится, выполнится только тело самого метода:

```
new Walk().run();
```

Поток переходит в состояние “неработоспособный” вызовом методов **suspend()**, **wait()** или методов ввода/вывода, которые предполагают задержку. Вернуть потоку работоспособность после вызова метода **suspend()** можно методом **resume()**. Поток переходит в “пассивное” состояние, если вызваны методы **interrupt()**, **stop()** или метод **run()** завершил выполнение. После этого, чтобы выполнить поток еще раз, необходимо создать новую копию потока. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания с помощью метода **sleep()**, при выполнении которого может генерироваться прерывание **InterruptedException**.

```
/* пример # 3 : жизненный цикл потока :
LiveCycle.java */
import java.awt.*;
class MyThread extends Thread {
  public void run(){
      while (true) {
          try {
           sleep(400);
           } catch (InterruptedException e) {
          System.out.println(e.toString());
           }
          System.out.println("Thread работает!");
        }
    }
 }
public class LiveCycle extends java.applet.Applet {
   MyThread myThread = new MyThread ();
   public void init(){
      myThread.start();
           System.out.println("поток стартовал");
      }
   public boolean mouseDown(Event e, int x, int y) {
      myThread.suspend();
           System.out.println("поток остановлен");
       return(true);
      }//метод из Java 1.1
   public boolean mouseUp(Event e, int x, int y) {
           System.out.println("поток возобновлен");
     myThread.resume();
        return(true);
     }//метод из Java 1.1
   public void destroy() {
           System.out.println("уничтожение потока");
     myThread.resume();
     myThread.stop();
    }
}
```

Метод **suspend()** позволяет приостановить выполнение потока, пока не возникло какое-либо событие, например, пока не нажата кнопка. Выполнение возобновляется при отпускании кнопки мыши и вызове метода **resume()**.

Методы **suspend()**, **resume()** и **stop()** являются deprecated-методами, так как они не являются в полной мере “потокобезопасными”.

Управление приоритетами и ThreadGroup

Потоку можно назначить приоритет от 1 (константа **MIN_PRIORITY**) до 10 (**MAX_PRIORITY**) с помощью метода **setPriority()**, получить значение приоритета можно с помощью метода **getPriority()**.

```
/* пример # 4 : установка приоритета :
ThreadPriority.java */
class MyThread extends Thread {
      public MyThread(String name){
        super(name);
        }
      public void run(){
        for (int i = 0; i < 4; i++){
        System.out.println(getName() + " " + i);
       try {
            sleep(10);
      } catch (InterruptedException e) {
            System.out.print("Error" + e);
                  }
          }
      }
 }
public class ThreadPriority {
      public static void main(String[] args) {
   MyThread min_thr = new MyThread ("Thread Min");
   MyThread max_thr = new MyThread ("Thread Max");
   MyThread norm_thr = new MyThread ("Thread Norm");
       min_thr.setPriority(Thread.MIN_PRIORITY);
       max_thr.setPriority(Thread.MAX_PRIORITY);
       norm_thr.setPriority(Thread.NORM_PRIORITY);
      min_thr.start();
      norm_thr.start();
      max_thr.start();
      }
 }
```

Поток с более высоким приоритетом в данном случае может монополизировать вывод на консоль.

Потоки объединяются в группы потоков. После создания потока нельзя изменить его принадлежность к группе.

```
ThreadGroup tg = new ThreadGroup("Группа потоков 1");
Thread t0 = new Thread(tg, "поток 0");
```

Все потоки, объединенные группой, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод **getThreadGroup()**.

Управление потоками

Приостановить выполнение потока можно с помощью метода **sleep()** класса **Thread**. Менее надежный альтернативный способ состоит в вызове метода **yield()**, который может сделать некоторую паузу и позволяет другим потокам начать выполнение своей задачи. Метод **join()** блокирует работу всех других потоков до тех пор, пока не будет закончено выполнение вызывающего метод потока.

```
// пример # 5 : задержка потока : JoinDemo.java
class Th extends Thread {
      public Th(String str) {
            super();
            setName(str);
      }
      public void run() {
            String nameT = getName();
         System.out.println("Старт потока " + nameT);
      if (nameT.compareTo("First") == 0) {
                  try {
                        sleep(1000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
System.out.println("завершение потока " + nameT);
      } else if (nameT.compareTo("Second") == 0) {
System.out.println("завершение потока " + nameT);
            }
      }
}
public class JoinDemo {
      public static void main(String[] args)
                  throws InterruptedException {
            Th tr1 = new Th("First");
            Th tr2 = new Th("Second");
            tr1.start();
            tr1.join();
/* join() не дает стартовать другому потоку до окончания текущего */
            tr2.start();
      }
}
```

Будет выведено:

```
Старт потока First
завершение потока First
Старт потока Second
завершение потока Second
```

Несмотря на вызов метода **sleep()** при запуске первого потока, второй поток не стартует до тех пор, пока первый не завершит свою работу. Это определяет вызов метода **join()** после старта первого потока.

Вызов метода **yield()** для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, для того чтобы другие потоки могли выполнять свои действия. Однако если требуется надежная остановка потока, то следует использовать его крайне осторожно или вообще применить другой способ.

```
// пример # 6 : задержка потока : YieldDemo.java
public class YieldDemo {
     public static void main(String[] args) {
          new Thread() {
               public void run() {
          System.out.println("старт потока t1");
          Thread.yield();
          System.out.println("завершение t1");
               }
          }.start();
          new Thread() {
               public void run() {
          System.out.println("старт потока t2");
          System.out.println("завершение t2");
               }
          }.start();
      }
  }
```

Скорее всего, в результате будет выведено:

```
старт потока t1
старт потока t2
завершение t2
завершение t1
```

Активизация метода **yield()** в коде метода **run()** первого объекта потока приведет к тому, что, скорее всего, первый поток будет остановлен на некоторый квант времени, что даст возможность другому потоку запуститься и выполнить свой код.

Потоки-демоны

Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon()**, вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

```
/* пример # 7 : запуск и выполнение потока-демона :
DemoDaemonThread.java */
class T extends Thread {
      public void run() {
            try {
            if (isDaemon()){
            System.out.println("старт потока-демона");
            sleep(1000); //заменить параметр на 1
            } else {
      System.out.println("старт обычного потока");
                        sleep(10);
                  }
            } catch (InterruptedException e) {
                  System.out.print("Error" + e);
            } finally {
                  if (!isDaemon())
                  System.out.println(
      "завершение работы обычного потока");
                  else
                  System.out.println(
      "завершение работы потока-демона");
            }
      }
}
public class DemoDaemonThread {
      public static void main(String[] args)
                  throws InterruptedException {
            T tr = new T();
            T trdaemon = new T();
            trdaemon.setDaemon(true);
            trdaemon.start();
            tr.start();
      }
}
```

В результате компиляции и запуска будет выведено:

```
старт потока-демона
старт обычного потока
завершение работы обычного потока
```

Поток-демон (из-за вызова метода **sleep(1000)**) не успел завершить выполнение своего кода до завершения основного потока приложения, связанного с методом **main()**. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода **main()**, не обращая внимания на то, что поток-демон еще работает. Если уменьшать время задержки потока-демона, то он может успеть завершить свое выполнение до окончания работы основного потока.

Потоки в апплетах

Добавить анимацию в апплет можно при использовании потоков. Поток, ассоциированный с апплетом, следует запускать тогда, когда апплет становится видимым, и останавливать при сворачивании браузера. В этом случае метод **repaint()** обновляет экран, в то время как программа выполняется. Поток создает анимационный эффект повторением вызова метода **paint()** и отображением вывода в новой позиции.

```
/* пример # 8 : освобождение ресурсов апплетом :
ApplThread.java */
import java.awt.*;
public class ApplThread extends java.applet.Applet
          implements Runnable {
      Font font = new Font("Arial", Font.BOLD, 36);
      String msg = "Java 2    ";
      Thread t = null;
      int i, len = msg.length() - 1;
      boolean stop;
      public void start() {
            t = new Thread(this);
            stop = false;
            t.start();
      }
      public void run() {
            char ch;
            while (true) {
                  repaint();
                  try {
                        Thread.sleep(300);
                        ch = msg.charAt(len);
```

```
                    msg = msg.substring(0, len);
                    msg = ch + msg;
                    i++;
                    if (stop)  break;
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        public void paint(Graphics g) {
            g.setFont(font);
            g.drawString(msg, 20 + i, 20 + i);
        }
        public void stop() {
            stop = true;
            i = 0;
            t = null;
        }
}
```

При вызове метода **stop()** апплета поток перестает существовать, так как ссылка на него устанавливается в **null** и освобождает ресурсы. Для следующего запуска потока необходимо вновь инициализировать ссылку и вызвать метод **start()** потока.

Методы synchronized

Очень часто возникает ситуация, когда много потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу. Например, когда два потока записывают информацию в файл/объект/поток. Для предотвращения такой ситуации может использоваться ключевое слово **synchronized**.

Рассмотрим класс **SyncroThreads**, в котором создается два потока. В этом же классе создается экземпляр класса **Synchro**, содержащий переменную типа **String**. Экземпляр **Synchro** передается в качестве параметра обоим потокам. Первый поток записывает экземпляр класса **Synchro** в файл. Второй поток также пытается сделать запись объекта **Synchro** в тот же самый файл. Для избежания одновременной записи такие методы объявляются **synchronized**. Синхронизированный метод изолирует объект, содержащий этот метод, после чего объект становится недоступным для других потоков. Изоляция снимается, когда поток полностью выполнит соответствующий метод. Другой способ снятия изоляции – вызов метода **wait()** из изолированного метода.

В примере продемонстрирован вариант синхронизации файла для защиты от одновременной записи информации в файл двумя различными потоками.

```
/* пример # 9 : синхронизация доступа к файлу :
SynchroThreads.java */
import java.io.*;
public class SynchroThreads {
      public static void main(String[] args) {
            Synchro s = new Synchro();
            T t1 = new T("First", s);
            T t2 = new T("Second", s);
            t1.start();
            t2.start();
      }
}
class T extends Thread {
      String str;
      Synchro s;
      public T(String str, Synchro s) {
            this.str = str;
            this.s = s;
      }
      public void run() {
            for (int i = 0; i < 5; i++) {
                  s.writing(str, i);
            }
      }
}
class Synchro {
      File f = new File("c:\\data.txt");
      public Synchro() {
            try {
                  f.delete();
                  f.createNewFile();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
public synchronized void writing(String str, int i) {
            try {
            RandomAccessFile raf =
                  new RandomAccessFile(f, "rw");
                  raf.seek(raf.length());
```

```
                System.out.print(str);
                raf.writeBytes(str);
            Thread.sleep((long) (Math.random() * 15));
                raf.seek(raf.length());
                System.out.print("->" + i + " ");
                raf.writeBytes("->" + i + " ");
            } catch (IOException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            notify();
        }
}
```

Код построен таким образом, что при отключении синхронизации метода **writing()** при его вызове одним потоком другой поток может вклиниться и произвести запись своей информации, несмотря на то, что метод не завершил запись, инициированную первым потоком.

Инструкция synchronized

Синхронизировать объект можно не только при помощи методов с соответствующим модификатором, но и при помощи синхронизированного блока кода. В этом случае происходит блокировка объекта, указанного в инструкции **synchronized**, и он становится недоступным для других синхронизированных методов и блоков. Обычные методы на синхронизацию внимания не обращают, поэтому ответственность за грамотную блокировку объектов ложится на программиста.

```
/* пример # 10 : блокировка объекта потоком :
TwoThread.java */
public class TwoThread {
      public static void main(String args[]) {
            final StringBuffer s = new StringBuffer();
            new Thread() {
                  public void run() {
                        int i = 0;
                  synchronized (s) {
                              while (i++ < 3) {
                                    s.append("A");
                  try {
                        sleep(15);
                  } catch (InterruptedException e) {
                        System.out.print("ошибка" + e);
```

```
                                    }
                              System.out.println(s);
                                    }
                              }
                        }
                  }.start();
                  new Thread() {
                        public void run() {
                              int j = 0;
                        synchronized (s) {
                                    while (j++ < 3) {
                                          s.append("B");
                        try {
                              sleep(10);
                        } catch (InterruptedException e) {
                              System.out.print("ошибка" + e);
                                    }
                              System.out.println(s);
                                    }
                              }
                        }
                  }.start();
      }
}
```

В результате компиляции и запуска будет, скорее всего (так как и второй поток может заблокировать объект первым), выведено:

```
A
AA
AAA
AAAB
AAABB
AAABBB
```

Один из потоков блокирует объект, и до тех пор, пока он не закончит выполнение блока синхронизации, в котором производится изменение значения объекта, ни один другой поток не может вызвать синхронизированный блок для этого объекта.

Если в коде убрать синхронизацию объекта **s**, то вывод будет другим, так как другой поток сможет получить доступ к объекту и изменить его раньше, чем первый закончит выполнение цикла.

В следующем примере рассмотрено взаимодействие методов **wait()** и **notify()** при освобождении и возврате блокировки в **synchronized** блоке.

Метод **wait()**, вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченные объекты, в частности объект **lock**. Возвратить блокировку объекта (объектов) можно вызовом метода **notify()** для конкретного объекта или **notifyAll()** для всех объектов. Это может быть выполнено из другого потока, заблокировавшего в свою очередь, указанные объекты.

```
/* пример # 12 : взаимодействие wait() и notify() :
DemoWait.java */
public class DemoWait {
      private BaseThread base = new BaseThread();
      private Blocked lock = new Blocked();
      class BaseThread extends Thread {
            public void run() {
                  lock.doWait();
            }
      }
      class Blocked {
            void doWait() {
                  synchronized (lock) {//или this
                        try {
      System.out.println("освобождение блокировки");
                        wait();//освобождение блокировки
            System.out.println("возврат блокировки");
            Thread.sleep(100);
                  } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
            System.out.println("doWait() закончен");
                  }
            }
      }
      void demoStart() {
            base.start();
            try {
                  Thread.sleep(50);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            synchronized (lock) {
            System.out.println("извещение doWait()");
                  lock.notify();//извещение doWait()
            }
```

```
            synchronized (lock) {
        System.out.println("блокировка после doWait()");
            }
        }
        public static void main(String[] args) {
            new DemoWait().demoStart();
        }
}
```

В результате компиляции и запуска будут выведены следующие сообщения:

```
освобождение блокировки
извещение doWait()
возврат блокировки
doWait() закончен
блокировка после doWait()
```

Задержки потоков методом **sleep()** используются для точной демонстрации последовательности действий, выполняемых потоками.

Потоки в J2SE 5.0

Добавлены пакеты классов **java.util.concurrent.locks**, **java.util.concurrent**, **java.util.concurrent.atomic,** возможности которых обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков параллельных (concurrent) классов, вызов утилит синхронизации, использование семафоров, ключей и atomic-переменных.

```
// пример # 13 : применение семафора : Sort.java
import java.util.concurrent.*;
public class Sort {
      public static final int ITEMS_COUNT = 15;
      public static double items[];
   /* семафор, контролирующий разрешение на доступ к
элементам массива */
      public static Semaphore sortSemaphore =
            new Semaphore(0, true);
      public static void main(String [] args) {
            items = new double[ITEMS_COUNT];
            for(int i = 0 ; i < items.length ; ++i)
            items[i] = Math.random();
      new Thread(new ArraySort(items)).start();
            for(int i = 0 ; i < items.length ; ++i) {
            /* при проверке доступности элемента сорти-
руемого массива происходит блокировка главного потока
до освобождения семафора */
```

```
            sortSemaphore.acquireUninterruptibly();
            System.out.println(items[i]);
      }
   }
}
class ArraySort implements Runnable {
   private double items[];
   public ArraySort(double items[]) {
      this.items = items;
   }
   public void run(){
      for(int i = 0 ; i < items.length - 1 ; ++i) {
      for(int j = i + 1 ; j < items.length ; ++j) {
                  if( items[i] < items[j] ) {
                        double tmp = items[i];
                        items[i] = items[j];
                        items[j] = tmp;
                  }
            }
            // освобождение семафора
            Sort.sortSemaphore.release();
            try {
                  Thread.sleep(555);
            } catch (InterruptedException e) {
                  System.out.print(e);
            }
      }
      Sort.sortSemaphore.release();
   }
}
```

Задания к главе 14

Вариант А

1. Создать апплет, используя поток: строка движется горизонтально, отражаясь от границ апплета и меняя при этом случайным образом свой цвет.
2. Создать апплет, используя поток: строка движется по диагонали. При достижении границ апплета все символы строки случайным образом меняют регистр.
3. Организовать сортировку массива методами Шелла, Хоара, пузырька, на основе бинарного дерева в разных потоках.
4. Реализовать сортировку графических объектов, используя алгоритмы из задания 3.

5. Создать апплет с точкой, движущейся по окружности с постоянной угловой скоростью. Сворачивание браузера должно приводить к изменению угловой скорости движения точки для следующего запуска потока.
6. Изобразить точку, пересекающую с постоянной скоростью окно слева направо (справа налево) параллельно горизонтальной оси. Как только точка доходит до границы окна, в этот момент от левого (правого) края с вертикальной координатной *y*, выбранной с помощью датчика случайных чисел, начинает свое движение другая точка и т.д. Цвет точки также можно выбирать с помощью датчика случайных чисел. Для каждой точки создается собственный поток.
7. Изобразить в приложении правильные треугольники, вращающиеся в плоскости экрана вокруг своего центра. Каждому объекту соответствует поток с заданным приоритетом.
8. Условия предыдущих задач изменяются таким образом, что центр вращения перемещается от одного края окна до другого с постоянной скоростью параллельно горизонтальной оси.
9. Создать фрейм с тремя шариками, одновременно летающими в окне. С каждым шариком связан свой поток со своим приоритетом.
10. Два изображения выводятся в окно. Затем они постепенно исчезают с различной скоростью в различных потоках (случайным образом выбираются точки изображения, и их цвет устанавливается в цвет фона).
11. Условие предыдущей задачи изменить на применение эффекта постепенного “проявления” двух изображений.

Вариант В

Для заданий варианта В главы 4 организовать синхронизированный доступ к ресурсам (файлам). Для каждого процесса создать отдельный поток выполнения.

Тестовые задания к главе 14

Вопрос 14.1.

Дан код:

```
class Q implements Runnable{
      int i = 0;
            public int run(){
             System.out.println("i = "+ ++i);
             return i;
       }}
public class Quest1 {
      public static void main(String[] args) {
            Q ob = new Q();
```

```
            ob.run();
}}
```

При попытке компиляции и запуска будет выведено:

1) i = 0;
2) i = 1;
3) ошибка компиляции: создать объект потока можно только с помощью инициализации объекта класса **Thread** или его наследников;
4) ошибка компиляции: неправильно реализован метод **run();**
5) ошибка времени выполнения: поток должен запускаться методом **start().**

Вопрос 14.2.

Дан код:

```
Thread t1=new Thread();
      t1.setPriority(7);
ThreadGroup tg=new ThreadGroup("TG");
      tg.setMaxPriority(8);
Thread t2=new Thread(tg,"t2");
      System.out.print("приоритет t1="
            + t1.getPriority());
      System.out.print(", приоритет t2="
            + t2.getPriority());
```

В результате компиляции и выполнения будет выведено:

1) приоритет t1 = 7, приоритет t2 = 5;
2) приоритет t1 = 7, приоритет t2 = 8;
3) приоритет t1 = 10, приоритет t2 = 8;
4) нет правильного.

Вопрос 14.3.

Дан код:

```
class T1 implements Runnable{
public void run(){
System.out.print("t1 ");
      } }
class T2 extends Thread{
public void run(){
System.out.print("t2 ");
      } }
public class Quest3 {
      public static void main(String[] args) {
            T1 t1 = new T1();
            T2 t2 = new T2(t1);
      t1.start();
```

```
t2.start();
} }
```

В результате компиляции и запуска будет выведено:

1) t1 t2;
2) t2 t1;
3) ошибка компиляции: метод **start()** не определен в классе **T1;**
4) ошибка компиляции: в классе **T2** не определен конструктор, принимающий в качестве параметра объект **Thread;**
5) ничего из перечисленного.

Вопрос 14.4.

Какое из указанных действий приведет к тому, что поток переходит в состояние “пассивный”? (выберите 2)

1) вызов метода **sleep()** без параметра;
2) вызов метода **stop();**
3) окончание выполнения метода **run();**
4) вызов метода **notifyAll();**
5) вызов метода **wait()** с параметром **null.**

Вопрос 14.5.

Дан код:

```
class Quest5 extends Thread {
      Quest5 () { }
      Quest5 (Runnable r) { super(r); }
         public void run() {
             System.out.print("thread ");
         }
         public static void main(String[] args) {
            Runnable r = new Quest5();  //1
            Quest5 t = new Quest5(r);    //2
             t.start();
         }   }
```

При попытке компиляции и запуска будет выведено:

1) ошибка компиляции в строке //1;
2) ошибка компиляции в строке //2;
3) thread;
4) thread thread;
5) код будет откомпилирован, но ничего выведено не будет.

Глава 15
СЕТЕВЫЕ ПРОГРАММЫ

Поддержка Интернет

Java делает сетевое программирование простым благодаря наличию специальных средств и классов. Большинство этих классов находится в пакете **java.net**. Сетевые приложения включают Internet-приложения, к которым относятся Web-браузер, e-mail, сетевые новости, передача файлов. Основной используемый протокол – TCP/IP.

Приложения клиент/сервер используют компьютер, выполняющий специальную программу – сервер, которая предоставляет услуги другим программам – клиентам. Клиент – это программа, получающая услуги от сервера. Клиент устанавливает соединение с сервером и пересылает серверу запрос. Сервер осуществляет прослушивание клиентов, получает и выполняет запрос после установки соединения. Результат выполнения запроса может быть возвращен сервером клиенту. На протоколе TCP/IP основаны следующие протоколы:

HTTP – Hypertext Transfer Protocol (WWW);

NNTP – Network News Transfer Protocol (группы новостей);

SMTP – Simple Mail Transfer Protocol (посылка почты);

POP3 – Post Office Protocol (чтение почты с сервера);

FTP – File Transfer Protocol (протокол передачи файлов).

Каждый компьютер по протоколу TCP/IP имеет уникальный IP-адрес. Это 32-битовое число, обычно записываемое как четыре числа, разделенные точками, каждое из которых изменяется от **0** до **255**. IP-адрес может быть временным и выделяться динамически для каждого подключения или быть постоянным, как для сервера. IP-адреса используются во внутренних сетевых системах. Обычно при подключении к Internet вместо числового IP-адреса используются символьные имена (например: `www.bsu.iba.by`), называемые именами домена. Специальная программа DNS (Domain Name Server), располагаемая на отдельном сервере, преобразует имя домена в числовой IP-адрес. Получить IP-адрес в программе можно с помощью объекта класса **InetAddress** из пакета **java.net**.

```
/* пример # 1 : вывод IP-адреса компьютера, подклю-
ченного к локальной сети : MyLocal.java */
import java.net.*;
```

```
 public class MyLocal {
    public static void main(String[] args){
           InetAddress myIP = null;
    try {
     myIP = InetAddress.getLocalHost();
    } catch (UnknownHostException e) {
     System.out.println("ошибка доступа ->" + e);
           }
     System.out.println("Мой IP ->" + myIP);
     }
}
```

В результате будет выведено, например:

```
Мой IP ->bsuiba_lab05/172.17.16.14
```

Метод **getLocalHost()** класса **InetAddress** создает объект **myIP** и возвращает IP-адрес компьютера.

Следующая программа демонстрирует при наличии Internet-соединения, как получить IP-адрес из имени домена с помощью сервера имен доменов (DNS), к которому обращается метод **getByName()** класса **InetAddress**.

```
/* пример # 2 : извлечение IP-адреса из имени домена
: IPfromDNS.java */
import java.net.*;
public class IPfromDNS {
  public static void main(String[] args) {
     InetAddress bsu_iba = null;
   try {
     bsu_iba =
InetAddress.getByName("www.bsu.iba.by");
      } catch (UnknownHostException e) {
     System.out.println("ошибка доступа ->" + e);
           }
   System.out.println("IP-адрес ->" + bsu_iba);
   }
}
```

Будет выведено:

```
IP-адрес ->www.bsu.iba.by/66.98.178.7
```

Если в качестве сервера используется этот же компьютер без сетевого подключения, в качестве IP-адреса указывается **127.0.0.1** или **localhost**. Для явной идентификации услуг к IP-адресу присоединяется номер порта через двоеточие, например **149.21.8.2:8443**. Здесь указан номер порта **8443**. Номера портов от **1** до **1024** могут быть заняты для внутреннего использования, например, если порт явно не указан, браузер

воспользуется значением по умолчанию: **20** – **FTP**-данные, **21** – **FTP**-управление, **53** – **DNS**, **80** – **HTTP**, **110** – **POP3**, **119** – **NNTP**. К серверу можно подключиться с помощью различных портов. Каждый порт предоставляет определенную услугу.

Для доступа к сети Internet в браузере указывается адрес URL. Адрес URL (Universal Resource Locator) состоит из двух частей – префикса протокола (http, https, ftp и т.д.) и URI (Universal Resource Identifier). URI содержит Internet-адрес, необязательный номер порта и путь к каталогу, содержащему файл, например:

```
http://bsu.iba.by/cgi-bin/news.pl
```

URI не может содержать такие специальные символы, как пробелы, табуляции, возврат каретки. Их можно задавать через шестнадцатеричные коды. Например: %20 обозначает пробел. Другие зарезервированные символы: символ & – разделитель аргументов, символ ? – следует перед аргументами запросов, символ + – пробел, символ # – ссылки внутри страницы (имя_страницы#имя_ссылки).

Можно создать объект класса **URL**, указывающий на ресурсы в Internet. В следующем примере объект **URL** используется для доступа к HTML-файлу, на который он указывает, и отображает его в окне браузера с помощью метода **showDocument()**.

```
/* пример # 3 : вывод документа в браузер :
MyShowDocument.java */
import java.net.*;
import java.applet.*;
import java.awt.*;
public class MyShowDocument extends Applet {
      URL bsu_iba = null;
      public void init() {
          try {
String url = "http://bsu.iba.by/cgi-bin/news.pl";
            bsu_iba = new URL(url);
         } catch (MalformedURLException e) {
       System.out.print("ошибка: " + e.getMessage());
  }
}
public boolean mouseDown(Event evt, int x, int y) {
/* при щелчке кнопки мыши в апплете происходит пере-
ход к странице www.bsu.iba.by */
 getAppletContext().showDocument(bsu_iba, "_blank");
    return true;
   }
}
```

Метод **showDocument()** может содержать параметры для отображения страницы различными способами: "**_self**" – выводит документ в текущий фрейм, "**_blank**" – в новое окно, "**_top**" – на все окно, "**_parent**" – в родительском окне, "**имя окна**" – в окне с указанным именем. Для корректной работы данного примера апплет следует запускать с помощью браузера, а не с помощью appletviewer.

В следующем примере методы **getDocumentBase()** и **getCodeBase()** используются для получения URL страницы апплета и URL апплета.

```
/* пример # 4 : получение URL-ов :
MyDocumentBase.java */
import java.applet.*;
import java.net.*;
import java.awt.*;
public class MyDocumentBase extends Applet {
   public void init() {
     URL html = getDocumentBase();
     URL codebase = getCodeBase();
   System.out.println("URL страницы : " + html);
   System.out.println("URL апплета : " + codebase);
    }
}
```

В следующей программе читается содержимое HTML-файла с сервера и выводится в окно консоли.

```
// пример # 5 : чтение HTML-файла : MyURLTest.java
import java.net.*;
import java.io.*;
public class MyURLTest {
   public static void main(String[] args) {
      try {
       URL bsu_iba = new URL("http://www.bsu.iba.by");
InputStreamReader isr =
      new InputStreamReader(bsu_iba.openStream());
       BufferedReader d = new BufferedReader(isr);
       String line = new String();
        while (line != null) {
            line = d.readLine();
            System.out.println(line);
                    }
            }
      catch (IOException e) {
      System.out.print("ошибка: " + e.getMessage());
            }
      }
}
```

Сокеты и сокетные соединения

Сокеты – это сетевые разъемы, через которые осуществляются двунаправленные поточные соединения между компьютерами. Сокет определяется номером порта и IP-адресом. При этом IP-адрес используется для идентификации компьютера, номер порта – для идентификации процесса, работающего на компьютере. Когда одно приложение знает сокет другого, создается сокетное соединение. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер ждет, пока клиент не свяжется с ним. Первое сообщение, посылаемое клиентом на сервер, содержит сокет клиента. Сервер в свою очередь создает сокет, который будет использоваться для связи с клиентом, и посылает его клиенту с первым сообщением. После этого устанавливается коммуникационное соединение.

Сокетное соединение с сервером создается с помощью объекта класса **Socket**. При этом указывается IP-адрес сервера и номер порта (80 для HTTP). Если указано имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу:

```
try {
     Socket socket = new Socket("localhost", 8030);
} catch (IOException e) {
     System.out.println("ошибка: " + e);
     }
```

Сервер ожидает сообщения клиента и должен быть запущен с указанием определенного порта. Объект класса **ServerSocket** создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода **accept()**, который возвращает сокет клиента:

```
 try {
     Socket s = null;
     ServerSocket server = new ServerSocket(8030);
     s = server.accept();
} catch (IOException e) {
     System.out.println("ошибка: " + e);
      }
```

Клиент и сервер после установления сокетного соединения могут получать данные из потока ввода и записывать данные в поток вывода с помощью методов **getInputStream()** и **getOutputStream()** или к **PrintStream** для того, чтобы программа могла трактовать поток как выходные файлы.

В следующем примере для посылки клиенту строки "привет!" сервер вызывает метод **getOutputStream()** класса **Socket**. Клиент получает данные от сервера с помощью метода **getInputStream()**. Для

разъединения клиента и сервера после завершения работы сокет закрывается с помощью метода **close()** класса **Socket**. В данном примере сервер посылает клиенту строку "привет!", после чего разрывает связь.

```
/* пример # 6 : передача клиенту строки :
MyServerSocket.java */
import java.io.*;
import java.net.*;
public class MyServerSocket{
     public static void main(String[] args)
          throws Exception {
     Socket s = null;
  try { //посылка строки клиенту
    ServerSocket server = new ServerSocket(8030);
     s = server.accept();
     PrintStream ps =
          new PrintStream(s.getOutputStream());
         ps.println("привет!");
         ps.flush();
    s.close(); // разрыв соединения
  } catch (IOException e) {
     System.out.println("ошибка: " + e);
          }
      }
}
/* пример # 7 : получение клиентом строки :
MyClientSocket.java */
import java.io.*;
import java.net.*;
public class MyClientSocket {
  public static void main(String[] args) {
     Socket socket = null;
  try {//получение строки клиентом
     socket = new Socket("имя_компьютера", 8030);
     BufferedReader dis = new BufferedReader(
      new InputStreamReader(socket.getInputStream()));
    String msg = dis.readLine();
    System.out.println(msg);
  } catch (IOException e) {
     System.out.println("ошибка: " + e);
        }
    }
}
```

Аналогично клиент может послать данные серверу через поток вывода с помощью метода **getOutputStream()**, а сервер может получать данные с помощью метода **getInputStream()**.

Если необходимо протестировать подобный пример на одном компьютере, можно выступать одновременно в роли клиента и сервера, используя статические методы **getLocalHost()** класса **InetAddress** для получения динамического IP-адреса компьютера, который выделяется при входе в сеть Интернет.

Многопоточность

Сервер должен поддерживать многопоточность, иначе он будет не в состоянии обрабатывать несколько соединений одновременно. Сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток. В следующем примере создается класс **NetServerThread**, расширяющий класс **Thread**.

```
/* пример # 8 : сервер для множества клиентов :
NetServerThread.java */
import java.net.*;
import java.io.*;
  public class NetServerThread extends Thread {
      Socket socket;
      int i;
      PrintStream ps;
  public NetServerThread(Socket s) {
      socket = s;
   try {
      ps = new PrintStream(s.getOutputStream());
      } catch (IOException e) {
            System.out.println("ошибка: " + e);
       }
}
public static void main(String[] args) {
           try {
      ServerSocket server = new ServerSocket(8030);
           while (true) {
      Socket s = null;
      s = server.accept();
      NetServerThread nst = new NetServerThread(s);
           nst.start();
                     }
           } catch (IOException e) {
                 System.out.println("ошибка: " + e);
```

```
                }
        }
        public void run() {
         while (true) {
              String  msg = "сообщение: " + i++;
              sendMessage(msg);
              }
        }
        public void sendMessage(String msg) {
              ps.println(msg);
              System.out.println(msg + "<передача>");
              ps.flush();
        }
 }
```

Сервер передает сообщение, посылаемое клиенту. Для клиентских приложений поддержка многопоточности также необходима. Например, один поток ожидает выполнения операции ввода/вывода, а другие потоки выполняют свои функции.

```
/* пример # 9 : получение сообщения клиентом в потоке
: NetClientThread.java */
import java.net.*;
import java.io.*;
public class NetClientThread extends Thread {
  BufferedReader br = null;
  Socket s = null;
      public NetClientThread() {
        try {//соединение с кольцевым адресом
          s = new Socket("127.0.0.1", 8030);
             InputStreamReader isr =
      new InputStreamReader (s.getInputStream());
             br = new BufferedReader(isr);
         } catch (IOException e) {
             System.out.println("ошибка: " + e);
              }
      }
      public static void main(String[] args) {
      NetClientThread nct = new NetClientThread();
             nct.start();
             }
      public void run() {
             while (true) {
```

```
            try {
             String msg = br.readLine();
                if (msg == null) break;
                else System.out.println(msg);
            } catch (IOException e) {
              System.out.println("ошибка: " + e);
                }
            }
      }
}
```

Сервер должен быть инициализирован до того, как клиент попытается осуществить сокетное соединение. При этом может быть использован IP-адрес локального компьютера.

Задания к главе 15

Вариант А

Создать на основе сокетов клиент/серверное визуальное приложение:

1. Клиент посылает через сервер сообщение другому клиенту.
2. Клиент посылает через сервер сообщение другому клиенту, выбранному из списка.
3. Чат. Клиент посылает через сервер сообщение, которое получают все клиенты. Список клиентов хранится на сервере в файле.
4. Клиент при обращении к серверу получает случайно выбранный сонет Шекспира из файла.
5. Сервер рассылает сообщения выбранным из списка клиентам. Список хранится в файле.
6. Сервер рассылает сообщения в определенное время определенным клиентам.
7. Сервер рассылает сообщения только тем клиентам, которые в настоящий момент находятся в on-line.
8. Чат. Сервер рассылает всем клиентам информацию о клиентах, вошедших в чат и покинувших его.
9. Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.
10. Игра по сети в “Морской бой”.
11. Игра по сети в “21”.

12. Игра по сети “Го”. Крестики-нолики на безразмерном (большом) поле. Для победы необходимо выстроить пять в один ряд.

Вариант В

Для заданий варианта В главы 4 на базе сокетных соединений разработать сетевой вариант системы. Для каждого пользователя должно быть создано клиентское приложение, соединяющееся с сервером.

Тестовые задания к главе 15

Вопрос 15.1.

Каким способом будет подключен объект **socket**, если он объявлен следующим образом

```
Socket socket = new Socket("host", 23);
```

1) POP3;
2) FTP;
3) TCP/IP;
4) IPX;
5) UDP.

Вопрос 15.2.

Как получить содержимое страницы, используя его URL при следующем объявлении?

```
String   url = new String("http://bsu.iba.by");
```

1)
```
Socket content = new
Socket(new URL(url)).connect();
```
2)
```
Object content = new URL(url).getContent();
```
3)
```
String content = new
URLHttp(url).getString();
```
4)
```
Object content = new
URLConnection(url).getContent();
```
5)
```
String content = new
URLConnection(url).connect().
```

Вопрос 15.3.

С помощью какого метода можно получить содержимое страницы по определенному адресу в сети Интернет?

1) `getDocumentBase();`
2) `getCodeBase();`
3) `getURLAddress();`

4) `getCodeAddress();`
5) `getURLBase()`.

Вопрос 15.4.

Какие исключительные ситуации возможны при открытии сокетного соединения вида:

```
Socket s = new Socket("bsu.iba.by", 8080);
```

1) `IOException;`
2) `MalformedURLException;`
3) `UnknownHostException;`
4) `UnknownURLException;`
5) `UnknownPortException`.

Вопрос 15.5.

Дан код:

```
Socket s = null;
    ServerSocket server = new ServerSocket(8080);
     s = server.accept();
PrintStream p = new PrintStream(s.getOutputStream());
         p.print("привет!");
```

Как поместить сообщение "`привет!`" в сокет и дать указание закрыть сокетное соединение после передачи информации?

1) `p.flush();`
2) `p.close();`
3) `s.flush();`
4) `s.close();`
5) нет правильного.

Часть 3. Технологии разработки Web-приложений

В третьей части даны основы программировании распределенных информационных систем с применением сервлетов, JSP и баз данных, а также даны основные принципы создания собственных библиотек тегов.

Глава 16

СЕРВЛЕТЫ

Согласно заявлению Sun Microsystems, на настоящий момент более 90% корпоративных систем поддерживают Java 2 Enterprise Edition. Эта платформа позволяет быстро и без особых издержек объединить возможности сети Интернет и корпоративных информационных систем.

Сервлеты – это компоненты приложений Java 2 Platform Enterprise Edition (J2EE), выполняющиеся на стороне сервера, способные обрабатывать клиентские запросы и динамически генерировать ответы на них. Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP.

Сервлет может применяться, например, для создания серверного приложения, получающего от клиента запрос, анализирующего его и делающего выборку данных из базы, а также пересылающего клиенту страницу HTML, сгенерированную с помощью JSP на основе полученных данных.

Преимуществом сервлетов перед CGI или ASP можно считать быстродействие и то, что сервлеты, являющиеся переносимыми на различные платформы, пишутся на объектно-ориентированном языке высокого уровня Java, который расширяется большим числом классов и программных интерфейсов.

В настоящее время сервлеты поддерживаются большинством Web-серверов и являются частью платформы J2EE.

Все сервлеты реализуют общий интерфейс **`Servlet`**. Для обработки HTTP-запросов можно воспользоваться в качестве базового класса

абстрактным классом **HttpServlet**. Базовая часть классов JSDK помещена в пакет **javax.servlet**. Однако класс **HttpServlet** и все, что с ним связано, располагаются на один уровень ниже в пакете **javax.servlet.http**.

Жизненный цикл сервлета начинается с его загрузки в память контейнером сервлетов при старте либо в ответ на первый запрос. Далее происходят инициализация, обслуживание запросов и завершение существования.

Первым вызывается метод **init()**. Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе программисты помещают код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом **service()** сервлета, а все параметры запроса упаковываются в объект **ServletRequest**, который передается в качестве первого параметра методу **service()**. Второй параметр метода – объект **ServletResponse**. В этот объект упаковываются выходные данные в процессе формирования ответа клиенту. Каждый новый запрос приводит к новому вызову метода **service()**. В соответствии со спецификацией JSDK, метод **service()** должен уметь обрабатывать сразу несколько запросов, т.е. быть синхронизирован для выполнения в многопоточных средах. Если же нужно избежать множественных запросов, сервлет должен реализовать интерфейс **SingleThreadModel**, который не содержит ни одного метода и только указывает серверу об однопоточной природе сервлета. При обращении к такому сервлету каждый новый запрос будет ожидать в очереди, пока не завершится обработка предыдущего запроса.

После завершения выполнения сервлета контейнер сервлетов вызывает метод **destroy()**, в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Интерфейсом **Servlet** предусмотрена реализация еще двух методов: **getServletConfig()** и **getServletInfo()**. Первый возвращает объект типа **ServletConfig**, содержащий параметры конфигурации сервлета, а второй – строку, описывающую назначение сервлета.

При разработке сервлетов в качестве базового класса в большинстве случаев используют не интерфейс **Servlet**, а класс **HttpServlet**, отвечающий за обработку запросов HTTP.

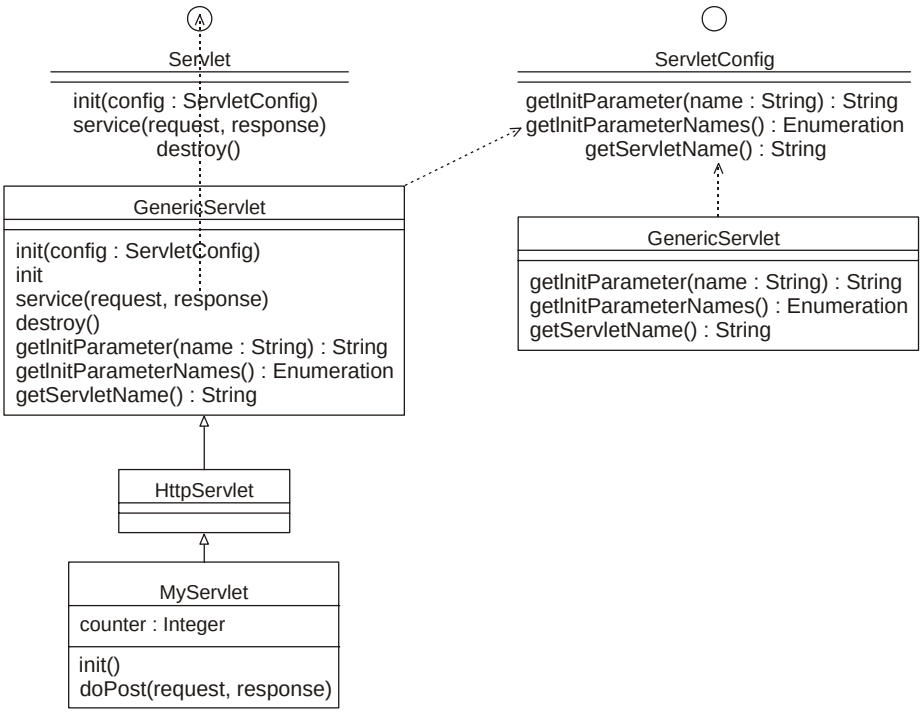

Рис. 16.1. Иерархия наследования сервлетов

Класс **`HttpServlet`** имеет реализованный метод **`service()`**, служащий диспетчером для других методов, каждый из которых обрабатывает методы доступа к ресурсам. В спецификации HTTP определены следующие методы: **`GET`**, **`HEAD`**, **`POST`**, **`PUT`**, **`DELETE`**, **`OPTIONS`** и **`TRACE`**. Наиболее часто употребляются методы **`GET`** и **`POST`**, с помощью которых на сервер передаются запросы, а также параметры для их выполнения.

При использовании метода **`GET`** (по умолчанию) параметры передаются как часть URL, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер. При использовании метода **`POST`** (**`method=POST`**) параметры (поля формы) передаются в содержимом HTTP-запроса и упакованы согласно полю заголовка **`Content-Type`**. По умолчанию в формате: `<имя>=<значение>&<имя>=<значение>&...`
Однако форматы упаковки параметров могут быть самые разные, например: в случае передачи файлов с использованием формы
`enctype="multipart/form-data"`.

В задачу метода **service()** класса **HttpServlet** входит анализ полученного через запрос метода доступа к ресурсам и вызов метода, имя которого сходно с названием метода доступа к ресурсам, но перед именем добавляется префикс **do**: **doGet()** или **doPost()**. Кроме этих методов могут использоваться методы: **doHead()**, **doPut()**, **doDelete()**, **doOptions()** и **doTrace()**. Разработчик должен переопределить нужный метод, разместив в нем функциональную логику.

В следующем примере приведен готовый к выполнению "шаблон" сервлета:

```
// пример # 1 : шаблон сервлета : MyServlet.java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class MyServlet extends HttpServlet {
      public MyServlet() {
            super();
            // Put your code here
      }
      public void init() throws ServletException {
            // Put your code here
      }
      public void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
            response.setContentType("text/html");
            this.preventCaching(request, response);
            PrintWriter out = response.getWriter();
            out.print("This is ");
            out.print(this.getClass().getName());
            out.print(", using the GET method");
            }
      public void doPost(
            HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
            response.setContentType("text/html");
            this.preventCaching(request, response);
            PrintWriter out = response.getWriter();
```

```
            out.print("This is ");
            out.print(this.getClass().getName());
            out.print(", using the POST method");
            }
/** Prevents navigator from caching data */
     protected void preventCaching(HttpServletRequest
request,HttpServletResponse response) {
/* see http:
//www.w3.org/Protocols/rfc2616/rfc2616-sec14.html */
            String protocol = request.getProtocol();
if ("HTTP/1.0".equalsIgnoreCase(protocol)) {
            response.setHeader("Pragma", "no-cache");
} else if ("HTTP/1.1".equalsIgnoreCase(protocol)) {
     response.setHeader("Cache-Control", "no-cache");
                  }
            response.setDateHeader("Expires", 0);
     }
     public void destroy() {
super.destroy(); // Just puts "destroy" string in log
            // Put your code here
     }
}
```

Практика включения HTML-кода в код сервлета не считается хорошей, так как эти действия "уводят" сервлет от его основной роли – контроллера приложения. Это приводит к разрастанию размеров сервлета, которое на определенном этапе становится неконтролируемым и реализует вследствие этого анти-шаблон "Волшебный сервлет". Даже приведенный выше маленький сервлет имеет признаки анти-шаблона, так как содержит метод **print()**, используемый для формирования кода HTML. Сервлет должен использоваться только для реализации бизнес-логики приложения и обязан быть отделен как от непосредственного формирования ответа на запрос, так и от данных, необходимых для этого. Обычно для формирования ответа на запрос применяются страницы JSP. Признаки наличия анти-шаблонов все же будут встречаться ниже, но это отступление сделано только с точки зрения компактности примеров.

Интерфейсы ServletRequest и HttpServletRequest

Поток данных поступает от клиента в виде закодированного и упакованного запроса. Вызывая методы интерфейса **ServletRequest**, можно получать определенный набор данных, посланных клиентом. Так, метод **getCharacterEncoding()** определяет символьную кодировку запроса, а методы **getContentType()** и **getProtocol()** – MIME-тип пришед-

шего запроса, а также название и версию протокола соответственно. Информацию об имени сервера, принявшего запрос, и порте, на котором запрос был "услышан" сервером, выдают методы **getServerName()** и **getServerPort()**. Можно узнать данные и о клиенте, от имени которого пришел запрос. Его IP-адрес возвращается методом **getRemoteAddr()**, а его имя – методом **getRemoteHost()**.

Если необходим прямой доступ к содержимому полученного запроса, надежный способ получить его – вызвать метод **getInputStream()** или **getReader()**. Первый возвращает ссылку на объект класса **ServletInputStream**, а второй – на **BufferedReader**. После этого можно читать любой байт из полученного запроса, используя технику работы с потоками Java.

Если, обращаясь к серверу, клиент помимо универсального адреса задал параметры, сервлету может понадобиться узнать их значение. Примером может служить электронная анкета, посылаемая по методу **GET** и выполненная в виде Web-страницы с формой ввода, значения полей и кнопок которой автоматически преобразуются в параметры URL. Три специальных метода в интерфейсе **ServletRequest** занимаются разбором параметров и выдачей их значений. Первый из них, **getParameter()**, возвращает значение параметра по его имени или **null**, если параметра с таким именем нет. Похожий метод, **getParameterValues()**, возвращает массив строк, а именно все значения параметра по его имени, причем параметр может иметь несколько значений. И еще один метод, **getParameterNames()**, возвращает объект типа **Enumeration**, позволяющий узнать имена всех присланных параметров.

Интерфейс **HttpServletRequest** является производным от интерфейса **ServletRequest** и используется для получения информации в HTTP-сервлетах. В интерфейсе **HttpServletRequest** имеются дополнительные методы, обеспечивающие программисту доступ к деталям протокола HTTP. Так, можно запросить массив cookies, полученный с запросом и содержащий информацию о клиенте, используя метод **getCookies()**. Файл cookies – это маленький файл, сохраняемый приложением на стороне клиента. Узнать о методе доступа к ресурсам, на основе которого построен запрос, можно с помощью вызова **getMethod()**. Строку запроса HTTP можно получить методом **getQueryString()**. Метод **getRemoteUser()** используется для получения имени пользователя, выполнившего запрос.

В следующем примере рассматривается применение некоторых методов интерфейса **HttpServletRequest** для получения данных о запросе, посылаемом по методу **GET**. Сервлет является компонентом Web-приложения **FirstProject** и в данном случае помещен в папке

/WEB-INF/classes проекта. Процесс запуска Web-сервера, организации и размещения проекта приведен ниже в разделе “Запуск Web-сервера и размещение проекта”.

```
/* пример # 2 : извлечение информации о запросе :
RequestClass.java */
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestClass extends HttpServlet {
     public void doGet(HttpServletRequest req,
          HttpServletResponse resp)
     throws IOException, ServletException {
          // установка MIME-типа содержания ответа
          resp.setContentType("text/html");
          // поток для данных ответа
          PrintWriter out = resp.getWriter();
          out.println("<HTML><HEAD>");
out.println("<TITLE>Request Information </TITLE>");
          out.println("</HEAD><BODY>");
     out.println("<H3>Request Information </H3>");
     out.println("<BR>Method: " + req.getMethod());
     out.println("<BR>Request URI: " +
req.getRequestURI());
     out.println("<BR>Protocol: " +
req.getProtocol());
     out.println("<BR>PathInfo: " +
req.getPathInfo());
     out.println("<BR>Remote Address: " +
req.getRemoteAddr());
     out.println("</BODY></HTML>");
          // закрытие потока
          out.close();
     }
}
```

В результате выполнения в браузер будет выведено:

Request Information

Method: GET

Request URI: /FirstProject/RequestClass

Protocol: HTTP/1.1

PathInfo: null

Remote Address: 127.0.0.1

Интерфейсы ServletResponse и HttpServletResponse

Генерируемые сервлетами данные пересылаются серверу-контейнеру с помощью объектов, реализующих интерфейс **ServletResponse**, а сервер, в свою очередь, пересылает ответ клиенту, инициировавшему запрос. Чаще всего приходится задавать MIME-тип генерируемых документов методом **setContentType()** и находить ссылки на потоки вывода двумя другими методами: **getOutputStream()** возвращает ссылку на поток **ServletOutputStream**, а метод **getWriter()** вернет ссылку на поток типа **PrintWriter**.

В интерфейсе **HttpServletResponse**, наследующем интерфейс **ServletResponse**, есть еще несколько полезных методов. Например, можно переслать cookie на клиентскую станцию, вызвав метод **addCookie()**. О возникших ошибках сообщается вызовом метода **sendError(int sc, String msg)**, которому в качестве параметра передается код ошибки и при необходимости текстовое сообщение. Кроме того, по мере надобности в заголовок ответа можно добавлять параметры, для чего служит метод **setDateHeader()**. С помощью метода **setAttribute(String name, Object ob)** устанавливаются значения атрибутов компонентов, являющиеся внутренними параметрами для передачи информации между компонентами приложения, например от сервлета к странице JSP или другому сервлету. Извлечь переданную компонентом информацию позволяют методы **getAttributeNames()** и **getAttribute(String name)**.

В следующем примере рассматривается применение методов интерфейсов **HttpServletRequest** и **HttpServletResponse** для получения header-информации из запроса, передаваемого по методу **POST** и генерации ответа клиенту.

```
/* пример # 3 : извлечение header-информации из клиентского запроса : RequestHeader.java */
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestHeader extends HttpServlet {
     public void doPost(HttpServletRequest request,
HttpServletResponse response)
     throws IOException, ServletException {
          response.setContentType("text/html");
          PrintWriter out = response.getWriter();
          Enumeration e = request.getHeaderNames();
```

```
            while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = request.getHeader(name);
            out.println(name + " = " + value);
            }
      }
}
```

Приведенный выше сервлет вызывается нажатием кнопки "Execute" формы из документа **index.html** по адресу URL – **RequestHeader**

```
<!--DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"-->
<HTML>
  <HEAD><TITLE>index.html</TITLE></HEAD>
<H2>ЗАПУСК СЕРВЛЕТА</H2>
  <BODY>
    <FORM action="RequestHeader" method="POST">
      <INPUT type="submit"  value="Execute">
    </FORM>
  </BODY>
</HTML>
```

В результате выполнения в браузер будет возвращено:

accept = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, application/x-shockwave-flash, */*
referer = http://localhost:8080/FirstProject/index.html
accept-language = en-us
content-type = application/x-www-form-urlencoded
accept-encoding = gzip, deflate
user-agent = Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
host = localhost:8080
content-length = 0
connection = Keep-Alive
cache-control = no-cache

Интерфейс ServletConfig

Ранее уже упоминался метод **getServletConfig()**, но не было сказано об интерфейсе **ServletConfig**, у которого имеются полезные методы. Имя сервлета можно получить, вызвав метод **getServletName()**. Инсталлируя сервлет на сервере, можно задавать параметры инициализации. Имена этих параметров можно получить через объект **Enumeration**,

возвращаемый методом **getInitParameterNames()**. Значение же конкретного параметра получают вызовом **getInitParameter(String n)**.

Также важен метод получения объекта контекста сервлета **getServletContext()**, возвращающий ссылку на **ServletContext**, используя которую можно узнать практически всю информацию о среде, в которой запущен и выполняется сервлет.

Контекст выполнения сервлета дает средства для общения с сервером. Пусть для выполнения задачи требуется узнать тип MIME (Multipurpose Internet Mail Extension) того или иного файла или документа. Для выполнения этой задачи из запроса тип извлекается методом **getContentType()**, а из контекста – методом **getMimeType()**. По умолчанию MIME-типом для сервлетов является **text/plain**, но используется обычно **text/html**.

Если нужно узнать истинный маршрут файла относительно каталога, в котором сервер хранит документы, то для этого предназначен метод **getRealPath()**. Информация о самом сервере предоставляется по вызову **getServerInfo()**.

Простой сервлет

Когда клиент переходит по адресу URL, который обрабатывается сервлетом, контейнер сервлета перехватывает запрос и вызывает метод **doGet()** или **doPost()**. Эти методы вызываются после конфигурации объектов, наследующих интерфейсы **HttpServletRequest**, **HttpServletResponse**. Задача методов **doGet()** и **doPost()** – взаимодействие с HTTP-запросом клиента и создание HTTP-ответа, основанного на данных запроса. Метод **getWriter()** объекта-ответа возвращает поток **PrintWriter**, который используется для записи символьных данных ответа.

Сервлет **FirstServlet** следует попробовать вызывать с различных компьютеров локальной сети или запустить несколько раз браузер и вызывать попеременно:

```
/* пример # 4 : счетчик посещений или "кликов" :
FirstServlet.java */
package test.com;
import java.io.*;
import java.util.Locale;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
     public class FirstServlet extends HttpServlet {
  // счетчик подключений к сервлету
```

```
  private volatile int count;
  public void init() throws ServletException {
    super.init();
    count = 0;
  }
  public void doGet(
    HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    performTask(req, res);
  }
  public void doPost(
  HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    performTask(req, res);
  }
  public void performTask(
  HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
      String title = "First Servlet";
      PrintWriter out = res.getWriter();
      res.setContentType("text/html; charset=Cp1251");
      out.println("<HTML><HEAD><TITLE>"
       + title + "</TITLE>"
       + "</HEAD><BODY><H2>This page is generated "
       + "by FirstServlet<H2><H3>This is its "
       + ++count
       + " -th execution</H3></BODY></HTML>");
      out.close();
   }
}
```

Запуск Web-сервера и размещение проекта

Сервер Apache Tomcat применяется в качестве обработчика страниц JSP и сервлетов. Последняя версия может быть загружена с сайта **jakarta.apache.org**.

Ниже приведены необходимые действия по выполнению сервлета из предыдущего примера с помощью контейнера сервлетов Web-сервера Tomcat 4.1. Пусть Web-сервер установлен в каталоге **/Apache Group/Tomcat4.1**. В этом же каталоге разместятся следующие подкаталоги:

/bin – содержит **startup**, **shutdown** и другие исполняемые файлы;

/conf – содержит конфигурационные файлы, в частности конфигурационный файл контейнера сервлетов **server.xml**;

/server – помещаются классы;

/logs – помещаются log-файлы;

/webapps – в этот каталог помещаются папки, содержащие сервлеты и другие компоненты приложения.

Пусть необходимо поместить в каталог **/webapps** папку **/FirstProject** с вложенным в нее сервлетом **FirstServlet** и определить, что требуется поместить в эту папку. Папка **/FirstProject** должна содержать каталог **/WEB-INF**, в котором помещаются подкаталоги:

/classes – содержит класс сервлета **FirstServlet.class**;

/lib – содержит библиотеки классов (если они есть), упакованные в JAR-файлы (архивы java);

/src – содержит исходный файл сервлета **FirstServlet.java**;

а также **web.xml** – конфигурационный файл приложения.

В файле **web.xml** необходимо прописать имя и путь к сервлету. Этот файл можно сконфигурировать так, что имя сервлета в браузере не будет совпадать с истинным именем сервлета. Например:

```
<web-app>
<servlet>
  <servlet-name>FirstServletname</servlet-name>
  <display-name>FirstServletdisplay</display-name>
  <servlet-class>test.com.FirstServlet
</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>FirstServletname</servlet-name>
  <url-pattern>/FirstServlettest</url-pattern>
</servlet-mapping>
</web-app>
```

Здесь указано имя сервлета **FirstServletname**, имя класса сервлета **FirstServlet.class**, URL-имя сервлета, по которому происходит его вызов **Servtest**.

Таким образом, требуется выполнить следующие действия:

1. Компиляцию сервлета: **javac FirstServlet.java**;
2. Полученный файл класса **FirstServlet.class** поместить в папку **/FirstProject/WEB-INF/classes/test/com**;
3. В эту же папку **/FirstProject/WEB-INF** поместить файл конфигурации **web.xml**;
4. Переместить папку **/FirstProject** в каталог **/webapps** сервера Tomcat;
5. Стартовать Tomcat;
6. Вызвать браузер и набрать строку:

http://localhost:8080/FirstProject/FirstServlettest

Рис. 16.2. Результат работы сервлета **FirstServlet**

7. Если вызывать сервлет из **index.html**, то тег **FORM** должен выглядеть следующим образом:

```
<FORM action="FirstServlettest">
      <INPUT type="submit" value="Execute">
</FORM>
```

Файл **index.html** помещается в папку **/webapps/FirstProject** и в браузере набирается строка:

```
http://localhost:8080/FirstProject/index.html
```

Сервлет будет вызван из HTML-документа по URL-имени **FirstServlettest**.

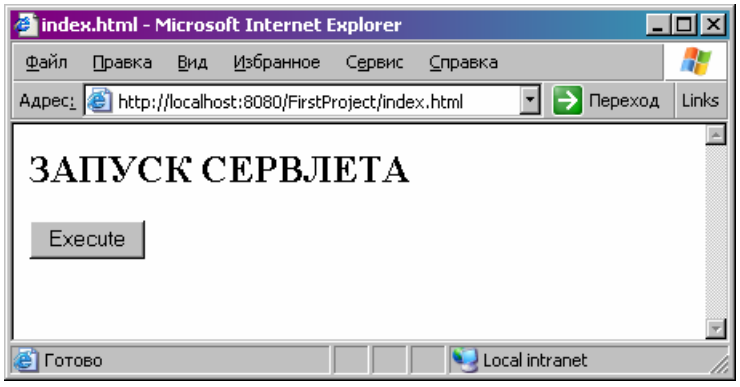

Рис. 16.3. Форма вызова сервлета

Извлечение информации из запроса

Распределенное приложение может быть эффективным только в случае, если оно способно принимать информацию от физически удаленных клиентов. В следующем примере сервлет извлекает данные пользовательской формы, переданные вместе с запросом по методу **GET**.

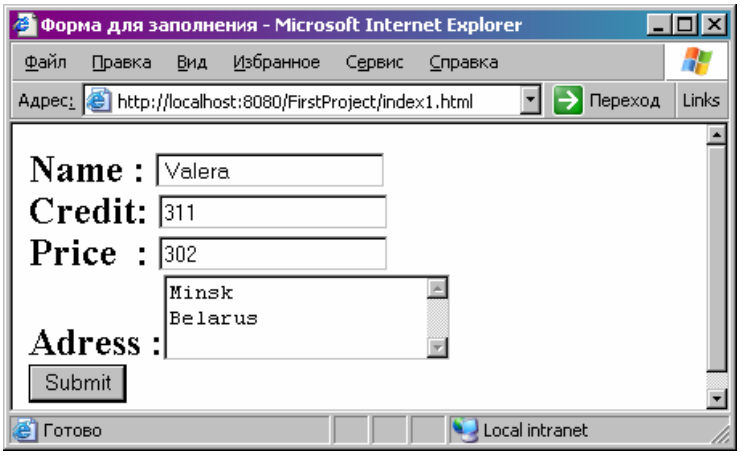

Рис. 16.4. Форма вызова сервлета с передачей информации

Приведенная форма является результатом отображения документа **index1.html**, находящейся в папке **/webapps/FirstProject**:

```
<HTML>
  <BODY>
     <TITLE>Форма для заполнения</TITLE>
    <FORM action="testform" >
      <H2> Name :
        <INPUT type="text" name="p0" value=" "> <BR>
Credit: <INPUT type="text" name="p1" value="0"> <BR>
Price: <INPUT type="text" name="p2" value="0"> <BR>
Adress:<TEXTAREA  name="Adress" rows=3 cols=20>
</TEXTAREA> <BR>
       <INPUT type="submit" value="Submit">
      <BR>
    </FORM>
  </BODY>
</HTML>
```

Для обработки данных, полученных из полей формы, используется сервлет, приведенный ниже.

```
/* пример # 5 : обработка запроса клиента :
FormRequest.java */
package test.com;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FormRequest extends HttpServlet {
     public void doGet(HttpServletRequest req,
HttpServletResponse resp)
          throws ServletException, IOException {
          performTask(req, resp);
     }
     public void performTask(HttpServletRequest req,
HttpServletResponse resp) {
          try {
               String val[] = new String[3];
               int rest;
               for (int i = 0; i < val.length; i++)
               val[i] = req.getParameter("p" + i);
     int c = Integer.valueOf(val[1]).intValue();
     int p = Integer.valueOf(val[2]).intValue();
               rest = c - p;
               PrintWriter out = resp.getWriter();
               resp.setContentType(
                    "text/html; charset=Cp1251");
          out.println("<HTML><HEAD>");
          out.println("<TITLE>FormRequest</TITLE>");
          out.println("</HEAD><BODY><BR>");
          out.println("<TABLE BORDER=3><TR><TD>");
               out.println(
"Name</TD><TD>Credit</TD><TD>Price</TD><TD> Rest ");
          out.println("</TD></TR><TR>");
          for (int i = 0; i < val.length; i++)
          out.println("<TD>" + val[i] + "</TD>");
          out.println("<TD>" + rest + "</TD></TR>");
          out.println("Adress:" +
               req.getParameter("Adress"));
          out.println("</TABLE></BODY></HTML>");
               out.close();
          } catch (Throwable e) {
               e.printStackTrace();
          }
     }
}
```

Класс сервлета относится к пакету **test.com**, поэтому файл **FormRequest.class** должен быть размещен в папке

/webapps/FirstProject/WEB-INF/classes/test/com

и обращение к этому классу, например из документа HTML, должно производиться как **test.com.FormRequest**. В файле **web.xml** должны находиться строки:

```
<servlet>
    <servlet-name>MyForm</servlet-name>
    <servlet-class>test.com.FormRequest
</servlet-class>
</servlet>
  <servlet-mapping>
    <servlet-name>MyForm</servlet-name>
    <url-pattern>/testform</url-pattern>
  </servlet-mapping>
```

Обращение к сервлету производится по его URL-имени **testform**. Результат выполнения:

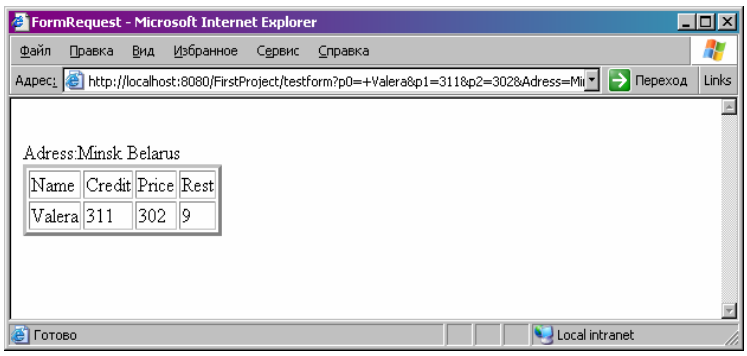

Рис. 16.5. Результат обработки запроса клиента

В следующем примере пользовательская форма в HTML-документе с именем **index2.html** содержит текстовое поле ввода и набор переключателей. При подтверждении из формы вызывается сервлет **InfoFromRequest**. Подтверждение происходит при помощи элемента ввода **submit** (кнопки), надпись на элементе – "Push".

В форме задан метод **POST**, при помощи которого происходит передача данных формы в виде отдельных заголовков. Если не задавать этот метод, по умолчанию будет использоваться метод **GET** и данные формы будут передаваться через универсальный запрос (URL), в который к адресу будут добавлены значения соответствующих элементов.

```
<!--DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01-->
Transitional//EN">
<HTML><HEAD>
<META http-equiv="Content-Type" content="text/html;
charset=CP1251">
```

```
  <BODY>
<FORM action="InfoFromRequesttest" method=POST>
   <H2>Название проекта:
<INPUT type="text" name="name" value=""></H2>
   <H3><BR>Технология исполнения<BR>
Sun<INPUT type="radio"
           name="company"
           value="Sun" checked>
        <BR>
Microsoft<INPUT type="radio"
           name="company"
           value="Microsoft">
        <BR>Язык программирования<BR>
Java <INPUT type="checkbox"
           name="language"
           value="Java"><BR>
C#<INPUT type="checkbox"
           name="language"
           value="C#"><BR></H3>
      <INPUT type="submit" value="Push"> <BR>
</FORM>
</BODY></HTML>
```

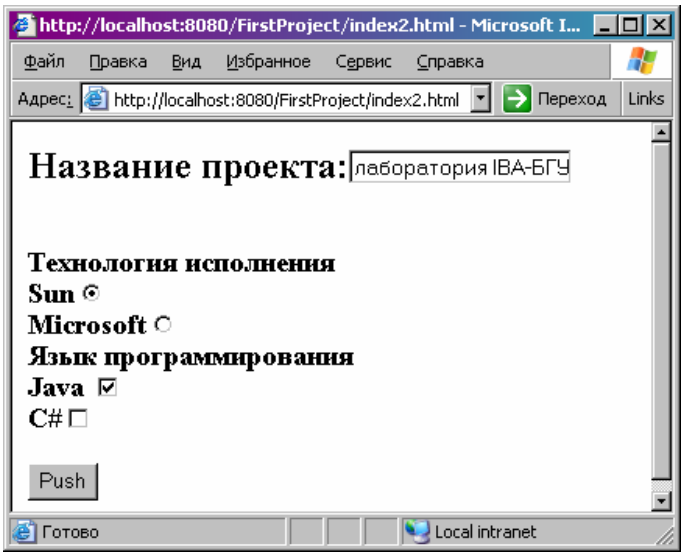

Рис. 16.6. HTML-форма

В форме имеется текстовое поле с именем **`name`** и значением по умолчанию `Название проекта`. Значение поля можно изменить на странице.

В форме заданы два элемента ввода типа **radio**, имеющих общее имя company. При наличии на странице нескольких полей, имеющих одно имя, спецификация этого поля позволяет выбрать только один из них. Им задаются соответствующие значения Sun и Microsoft, и при выборе одной из кнопок значение, заданное соответствующей кнопке, заносится в значение элемента company. Здесь по умолчанию задана выбранной кнопка со значением Sun при помощи свойства **checked**. Если не отмечена ни одна из радиокнопок, значение элемента будет не определено. Такая ситуация может возникнуть, если по умолчанию отмеченная кнопка не задана, а пользователь ничего не отметил.

Заданы также два поля типа **checkbox** с именем **language** и значениями Java и C#. При выборе этих элементов пользователем им присваиваются соответствующие значения.

В итоге пользователь может изменить значения текстового поля и радиокнопки и задать значения переменным **checkbox**. При нажатии кнопки типа **submit** происходит подтверждение формы и вызывается сервлет.

Сервлет получает и извлекает значения всех переменных формы и отображает их вместе с именами переменных.

```
/* пример # 6 : извлечение информации из запроса кли-
ента : InfoFromRequest.java */
package test.com;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class InfoFromRequest extends HttpServlet {
  public void doGet(
HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
      performTask(req, resp);
  }
  public void doPost(
HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
      performTask(req, resp);
  }
  public void performTask(
HttpServletRequest req, HttpServletResponse resp) {
```

```
      try {
        String name, value;
        PrintWriter out = resp.getWriter();
    resp.setContentType("text/html; charset=CP1251");
        out.println("<HTML><HEAD>");
        out.println("<TITLE>InfoFromRequest</TITLE>");
        out.println("</HEAD><BODY><BR>");
        out.println("<TABLE BORDER=3><TR>");
        out.println("<TD>NAME</TD><TD>VALUE</TD>");
        out.println("</TR>");
        Enumeration names = req.getParameterNames();
        while (names.hasMoreElements()) {
            name = (String) names.nextElement();
      /*    name = new String(
        name.getBytes("ISO-8859-1"), "CP1251"); */
            value = req.getParameterValues(name)[0];
      /* value = new String(
        value.getBytes("ISO-8859-1"), "CP1251"); */
            out.println("<TR>");
            out.println("<TD>" + name + "</TD>");
            out.println("<TD>" + value + "</TD>");
            out.println("</TR>");
        }
        out.println("</TABLE></BODY></HTML>");
        out.close();
      } catch (Throwable theException) {
        theException.printStackTrace();
      }
    }
}
```

В сервлете в объекте **resp** задается тип содержимого **text/html** и кодировка **CP1251,** если нужно отобразить кириллицу. После этого объект **out** устанавливается в выходной поток **resp.getWriter()**, в который будут помещаться данные. Из запроса **HttpServletRequest req** извлекается объект типа **Enumeration** с текстовыми значениями имен переменных формы. Далее, итерируя по элементам этого объекта, последовательно извлекаются все параметры. Для каждого имени переменной можно при необходимости (если не указана кодовая страница) произвести перекодировку: вначале извлекается объект итерации в кодировке, в которой он передается, а именно **ISO-8859-1**, после создается новая строка с необходимой кодировкой, в данном случае **CP1251**. Для каждой из переменных извлекаются из запроса соответствующие им зна-

чения при помощи метода **getParameterValues(name)**. Тем же способом их кодировка может быть изменена и добавлена в выходной поток.

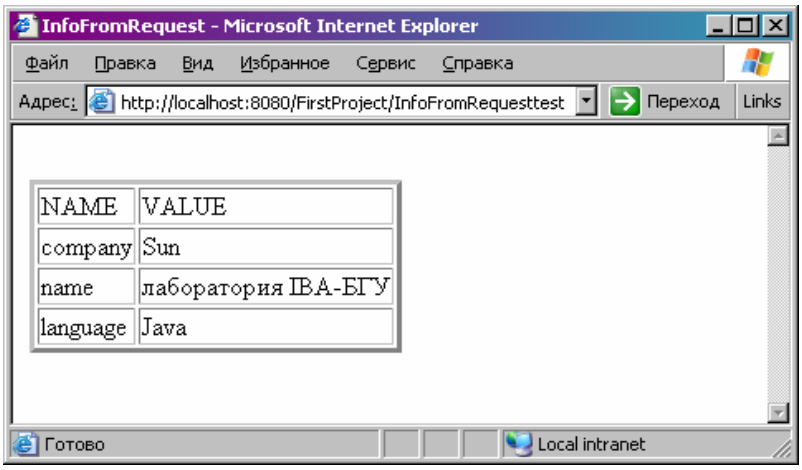

Рис. 16.7. Результат выполнения запроса

Метод **getParameterValues()** возвращает значения любой переменной формы по имени этой переменной. Массив возвращается потому, что некоторые переменные формы могут иметь несколько значений, например: группа флажков или радиокнопок. Другой метод доступа не предполагает предварительного знания их имен. Метод **getParameterNames()** возвращает объект **Enumeration**, в котором содержатся все имена переменных формы.

Многозадачность

Контейнер сервлетов будет иметь несколько потоков выполнения, распределяемых согласно запросам клиентов. Вероятна ситуация, когда два клиента одновременно вызовут методы **doGet()** или **doPost()**. Метод **service()** должен быть написан с учетом вопросов многозадачности. Любой доступ к разделяемым ресурсам, которыми могут быть файлы, объекты, необходимо защитить ключевым словом **synchronized**. Ниже приведен пример посимвольного вывода строки сервлетом с паузой между выводом символов в 500 миллисекунд, что позволяет другим клиентам, вызвавшим сервлет, успеть вклиниться в процесс вывода при отсутствии синхронизации.

```
/* пример # 5 : доступ к синхронизированным ресурсам
: ServletSynchronization.java */
package test.com;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
public class ServletSynchronization
          extends HttpServlet {
```

```
// оригинал строки
private final String SYNCHRO = "SYNCHRONIZATION";
// синхронизируемый объект
private StringBuffer lockedString =
         new StringBuffer();
public void doGet(
  HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {
  performTask(req, res);
}
public void doPost(
  HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException {
  performTask(req, res);
}
protected void performTask(
  HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
  try {
    Writer out = res.getWriter();
    out.write(
      "<HTML><HEAD>"
      + "<TITLE>SynchronizationDemo</TITLE>"
      + "</HEAD><BODY>");
    out.write(createString());
    out.write("</H1></BODY></HTML>");
    out.flush();
    out.close();
  } catch (Throwable e) {
    throw new RuntimeException(
      "Failed to handle request: " + e);
  }
}
protected String createString() {
  synchronized (lockedString) {
    try {
      for (int i = 0; i < SYNCHRO.length(); i++) {
        lockedString.append(SYNCHRO.charAt(i));
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    String result = lockedString.toString();
    lockedString = new StringBuffer();
```

```
            return result;
        }
    }
}
```

Результаты работы сервлета при наличии и отсутствии синхронизации представлены на рисунках.

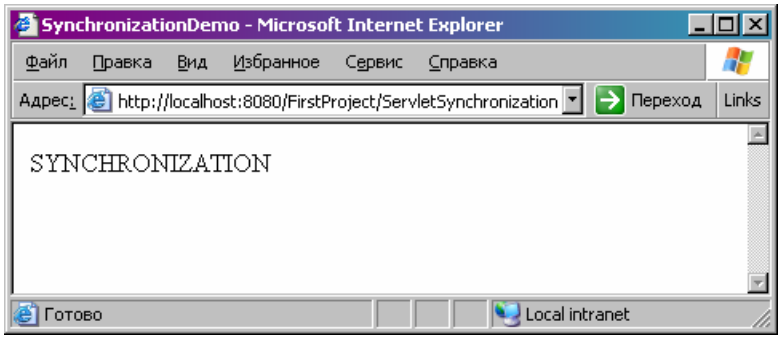

Рис. 16.8. Результат работы сервлета **Synchronization** с блоком синхронизации

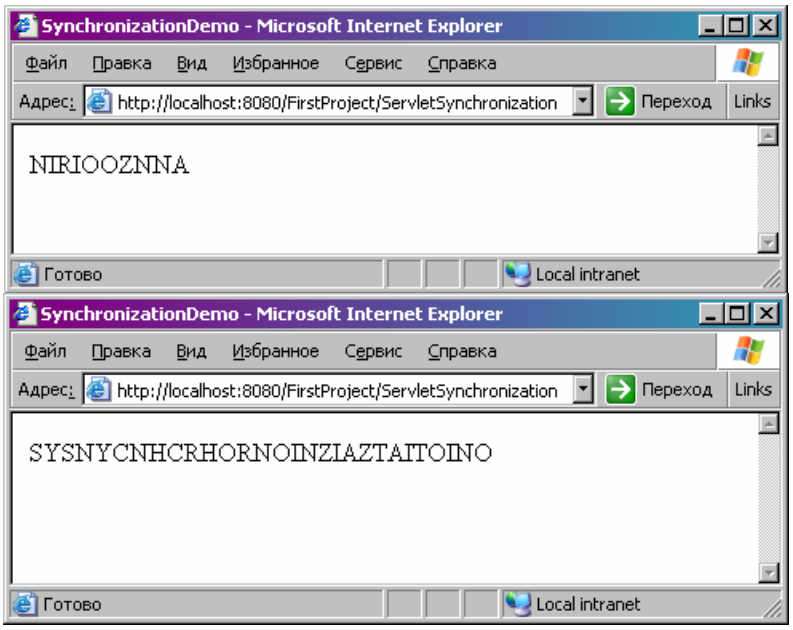

Рис. 16.9. Результат работы сервлета **Synchronization** без синхронизации

Можно синхронизировать и весь сервлет целиком, но причиной, почему это не делается, является возможность нахождения критической секции вне основного пути выполнения программы.

Задания к главе 16

Вариант А

Создать сервлет и взаимодействующие с ним пакеты Java-классов и HTML-документов, выполняющие следующие действия:

1. Генерация таблиц по переданным параметрам: заголовок, количество строк и столбцов, цвет фона.
2. Вычисление тригонометрических функций в градусах и радианах с указанной точностью. Выбор функций должен осуществляться через выпадающий список.
3. Поиск слова, введенного пользователем. Поиск и определение частоты встречаемости осуществляется в текстовом файле, расположенном на сервере.
4. Вычисление объемов тел (параллелепипед, куб, сфера, тетраэдр, тор, шар, эллипсоид и т.д.) с точностью и параметрами, указываемыми пользователем.
5. Поиск и (или) замена информации в коллекции по ключу (значению).
6. Выбор текстового файла из архива файлов по разделам (поэзия, проза, фантастика и т.д.) и его отображение.
7. Выбор изображения по тематике (природа, автомобили, дети и т.д.) и его отображение.
8. Информация о среднесуточной температуре воздуха за месяц задана в виде списка, хранящегося в файле. Определить: а) среднемесячную температуру воздуха; б) количество дней, когда температура была выше среднемесячной; в) количество дней, когда температура опускалась ниже $0^{O}C$; г) три самых теплых дня.
9. Игра с сервером в “21”.
10. Реализация адаптивного теста из цепочки в 3–4 вопроса.
11. Определение значения полинома в заданной точке. Степень полинома и его коэффициенты вводятся пользователем.
12. Вывод фрагментов текстов шрифтами различного размера. Размер шрифта и количество строк задается на стороне клиента.
13. Информация о точках на плоскости хранится в файле. Выбрать все точки, наиболее приближенные к заданной прямой. Параметры прямой и максимальное расстояние от точки до прямой вводятся на стороне клиента.
14. Осуществить сортировку введенного пользователем массива целых чисел. Числа вводятся через запятую.
15. Реализовать игру с сервером в крестики-нолики.
16. Осуществить форматирование выбранного пользователем текстового файла, так чтобы все абзацы имели отступ ровно 3 пробела, а длина каждой строки была ровно 80 символов и не имела начальными и конечными символами пробел.

Вариант B

Для заданий варианта B главы 4 на основе сервлетов разработать механизм аутентификации и авторизации пользователя. Сервлет должен сгенерировать приветствие с указанием имени, роли пользователя, а также указать текущую дату и IP-адрес компьютера пользователя.

Тестовые задания к главе 16

Вопрос 16.1.

Каким образом в методе **`init()`** сервлета получить параметр инициализации сервлета с именем "URL"? (выберите 2)

1) `ServletConfig.getInitParameter("URL");`
2) `getServletConfig().getInitParameter("URL");`
3) `this.getInitParameter("URL");`
4) `HttpSerlet.getInitParameter("URL");`
5) `ServletContext.getInitParameter("URL").`

Вопрос 16.2.

Какой метод сервлета **`FirstServlet`** будет вызван при активизации ссылки следующего HTML-документа?

```
<html><body>
  <a href="/FirstProject/FirstServlettest">OK!</a>
</body><html>
```

Соответствующий сервлету тег **`<url-pattern>`** в файле **`web.xml`** имеет вид:

```
<url-pattern>/FirstServlettest</url-pattern>
```

1) `doGet();`
2) `doGET();`
3) `performTask();`
4) `doPost();`
5) `doPOST().`

Вопрос 16.3.

Контейнер вызывает метод **`init()`** экземпляра сервлета…

1) при каждом запросе к сервлету;
2) при каждом запросе к сервлету, при котором создается новая сессия;
3) при каждом запросе к сервлету, при котором создается новый поток;
4) только один раз за жизненный цикл экземпляра;
5) когда сервлет создается впервые;
6) если время жизни сессии пользователя, от которого пришел запрос, истекло.

Вопрос 16.4.

Каковы типы возвращаемых значений методов **`getResource()`** и **`getResourceAsStream()`** интерфейса **`ServletContext`**?

1) `ServletContext` не имеет таких методов;
2) `String` и `InputStream`;
3) `URL` и `InputStream`;
4) `URL` и `StreamReader`.

Вопрос 16.5.

Какие интерфейсы находятся в пакете **`javax.servlet`**?

1) `ServletRequest`;
2) `ServletOutputStream`;
3) `PageContext`;
4) `Servlet`;
5) `ServletContextEvent`;
6) ни один из перечисленных.

Вопрос 16.6.

Как можно получить всю информацию из запроса, посланного следующей формой? (выберите два варианта ответа)

```
<HTML><BODY>
  <FORM action="/com/MyServlet">
    <INPUT type="file" name="filename">
    <INPUT type="submit" value="Submit">
  </FORM></BODY></HTML>
```

1) `request.getParameterValues("filename");`
2) `request.getAttribute("filename");`
3) `request.getInputStream();`
4) `request.getReader();`
5) `request.getFileInputStream()`.

Глава 17
ПОЛЬЗОВАТЕЛЬСКИЕ СЕССИИ

Сеанс (сессия)

Когда клиент заходит на Web-ресурс и выполняет варианты использования один за другим, то контекстная информация о клиенте не хранится. В протоколе HTTP нет возможностей для сохранения и изменения подобной информации. При этом возникают проблемы в распределенных системах с различными уровнями доступа для разных пользователей. Действия, которые может делать администратор системы, не может выполнять гость. В данном случае необходима проверка прав пользователя при переходе с одной страницы на другую. Существует несколько способов хранения текущей информации.

Сеанс (сессия) – соединение между клиентом и сервером, устанавливаемое на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеансы используются для обеспечения хранения данных во время нескольких запросов Web-страницы или на обработку информации, введенной в пользовательскую форму. Как правило, имея дело с сессией, возникают следующие проблемы:

- проблема поддержки распределенной сессии (синхронизация/репликация данных, уникальность идентификаторов и т.д.);
- проблема безопасности (подбор чужого идентификатора сессии);
- проблема инвалидации сессии (expiration), предупреждение пользователя об уничтожении сессии и возможность ее продления (watchdog).

Чтобы открыть новый сеанс, используется метод **`getSession()`** интерфейса **`HttpServletRequest`**. Система извлекает из переданных в сервлет запросов информацию о пользователе, а его права в приложении, как правило, извлекаются из БД и добавляются в объект **`HttpSession`**, соответствующий данному пользователю. Кроме того, сессия содержит информацию о дате и времени создания и последнего обращения к сессии, которая извлекается с помощью методов **`getCreationTime()`** и **`getLastAccessedTime()`**.

Если для метода **`getSession()`** входной параметр равен **`true`**, то сервлет-контейнер проверяет наличие активного сеанса, установленного с

данным клиентом. В случае успеха метод возвращает дескриптор этого сеанса. В противном случае метод устанавливает новый сеанс:

```
HttpSession session = request.getSession(true);
```

Чтобы сохранить значения переменной в текущем сеансе, используется метод **putValue()**, чтобы прочесть – **getValue()**. В настоящий момент эти методы являются устаревшими (deprecated) и вместо них используются соответственно **setAttribute()**, **getAttribute()**, а вместо **removeValue()** используют **removeAttribute()**. Список имен всех переменных, сохраненных в текущем сеансе, можно получить, используя метод **String[] getValueNames()**. Обычно используется более надежный метод **Enumeration getAttributeNames(),** работающий так же, как и соответствующий метод интерфейса **HttpServletRequest.**

Метод **String getId()** возвращает уникальный идентификатор, который получает каждый сеанс при создании. Метод **isNew()** возвращает **false** для уже существующего сеанса и **true** – для только что созданного.

Методы **getRequestURL()** и **getRequestURI()** позволяют извлечь из запроса URL и URI клиента, обратившегося к сервлету, и, если необходимо, добавить его в сессию.

Если требуется сохранить для использования в будущем одну из переменных сеанса, представляющего собой целое число, то:

```
Integer amount = new Integer(1251);
session.setAttribute("credit", amount);
```

После этого любой подключившийся к текущему сеансу сервлет сможет прочесть значение переменной **credit** следующим образом:

```
Integer val =
     (Integer)session.getAttribute("credit");
```

Завершить сеанс можно методом **invalidate()**. Сеанс уничтожает все связи с объектами и данные, сохраненные в старом сеансе, будут потеряны для всех сервлетов.

```
/* пример # 1 : добавление информации в сессию :
SessionServlet.java */
package test.com;
import java.io.*;
import javax.servlet.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
public class SessionServlet extends HttpServlet {
     public void doGet(
          HttpServletRequest req,
          HttpServletResponse resp)
```

```
            throws ServletException, IOException {
            performTask(req, resp);
      }
      public void doPost(
            HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException, IOException {
            performTask(req, resp);
      }
      protected void performTask(
            HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException, IOException {
            try {
/* возвращается ссылка на сессию для текущего пользо-
вателя (если сессия еще не существует, то она при
этом создается) */
      HttpSession session = req.getSession(true);
      Writer out = resp.getWriter();
      StringBuffer url = req.getRequestURL();
      session.setAttribute("URL", url);
                  out.write(
      "<html><head><title>My Session Counter &
other</title></head><body><h2>My session counter: ");
/* количество запросов, которые были сделаны к дан-
ному сервлету текущим пользователем в рамках текущей
пользовательской сессии (следует приводить значение
к строковому виду для корректного отображения в ре-
зультате) */
      out.write(
String.valueOf(prepareMySessionCounter(session)));
      out.write("<br> Creation Time: "
                  + new
java.util.Date(session.getCreationTime()));
            out.write("<br> Time of last access: "
            + new
java.util.Date(session.getLastAccessedTime()));
            out.write("<br> sesion ID : "
                  + session.getId());
            out.write("<br> Your URL: " + url);
                  int timeLive = 60;
            session.setMaxInactiveInterval(timeLive);
```

```
out.write("</h2><br>Set max inactive interval : "
                + timeLive);
                out.write("</body></html>");
                out.flush();
                out.close();
          } catch (Throwable e) {
                e.printStackTrace();
      throw new RuntimeException("Failed : " + e);
          }
      }
      /* увеличивает счетчик обращений к сервлету и
добавляет его в сессию */
      protected int prepareMySessionCounter(
                HttpSession session) {
      Integer counter =
          (Integer) session.getAttribute("counter");
          if (counter == null) {
      session.setAttribute("counter", new Integer(1));
                return 1;
          } else {
      int counterValue = counter.intValue();

      session.setAttribute(counterValue++;
          "counter", new Integer(counterValue));
                return counterValue;
          }
      }
}
```

В качестве данных сеанса выступает объект типа **Integer**. В ответ на пользовательский запрос сервлет **SessionServlet** возвращает страницу HTML, на которой отображается значение сеансовой переменной типа **Integer**. Это значение соответствует числу, которое присвоил данному клиенту сервлет **SessionServlet** в текущем сеансе. Сеанс устанавливается непосредственно между клиентом и Web-сервером. Каждый клиент устанавливает с сервером свой собственный сеанс.

Cookie

Cookie – это небольшие блоки текстовой информации, которые сервер посылает клиенту. Браузер возвращает эту информацию обратно на сервер как часть заголовка HTTP, когда клиент повторно заходит на тот же Web-ресурс. Cookies могут быть ассоциированы не только с сервером, но и также с доменом – в этом случае браузер посылает их на все сервера ука-

занного домена. Этот принцип лежит в основе одного из протоколов обеспечения единой идентификации пользователя (Single Signon), где сервера одного домена обмениваются идентификационными маркерами (token) с помощью общих cookies.

Cookie были созданы в компании Netscape как средства отладки, но теперь используются повсеместно. Файл cookie – это файл небольшого размера для хранения информации, который создается серверным приложением и размещается на компьютере пользователя. Браузеры накладывают ограничения на размер файла cookie и общее количество cookie, которые могут быть установлены на пользовательском компьютере приложениями одного Web-сервера.

Чтобы послать cookie клиенту, сервлет должен создать cookie, указав конструктору имя и значение блока и добавить их в объект-ответ. Конструктор использует имя блока в качестве первого параметра, а его значение – в качестве второго.

```
Cookie cookie = new Cookie("OREO","2005");
response.addCookie(cookie);
```

Извлечь информацию cookie из запроса можно с помощью метода **getCookie()** объекта **HttpServletRequest**, который возвращает массив объектов, составляющих этот файл.

```
Cookie[] cookies = request.getCookies();
```

После этого для каждого объекта можно вызвать метод **getValue()**, который возвращает строку **String** с содержимым блока cookie. В данном случае метод **getValue()** вернет значение "2005".

Объект cookie имеет целый ряд параметров: путь, домен, номер версии, время жизни, комментарий. Одним из важнейших является срок жизни в секундах от момента первой отправки клиенту. Если параметр не указан, то cookie существует только до момента первого закрытия браузера.

```
/* пример # 2 : создание и чтение cookie :
CookieShow.java */
package test.com;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieShow extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
     throws IOException, ServletException {
      String name = "MycookieName";
      String value = "newValue";
      Cookie c = new Cookie(name, value);//set cookie
```

```
            c.setMaxAge(-1);//заменить на 3600*24
            c.setPath("/");
            response.addCookie(c);
      performTask(request, response);
      }
            // print out cookies
      protected void performTask(
            HttpServletRequest request,
            HttpServletResponse response)
      throws ServletException, IOException {
            try {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            Cookie[] cookies = request.getCookies();
         if(cookies != null){
         for (int i = 0; i < cookies.length; i++) {
            Cookie c1 = cookies[i];
            String name1 = c1.getName();
            String value1 = c1.getValue();
            int ver = c1.getVersion();
            String path = c1.getPath();
            int maxage = c1.getMaxAge();
      out.println("Path:" + path + "<BR>");
      out.println(name1 + " = " + value1 + "<BR>");
      out.println("Version:" + ver + "<BR>");
      out.println("MaxAge" + maxage + "<BR>");
                                    }
                              }
            out.close();
                  }
            catch (Throwable e) {
                  e.printStackTrace();
      throw new RuntimeException(e.toString());
            }
    }
}
```

В результате в файле cookie будет содержаться следующая информация:

Path:null
MycookieName=newValue
Version:0
MaxAge-1

Файл cookie можно изменять. В следующем примере приведена программа модификации cookie, хранимого на компьютере клиента.

```
/* пример # 3 : создание cookie и чтение количества
вызовов сервлета из cookie : CookieServlet.java */
package test.com;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieServlet extends HttpServlet
          implements Servlet {
public static final int MAX_AGE_COOKIE = 3600*24*30;
     public void doGet(HttpServletRequest request,
                HttpServletResponse response)
      throws ServletException, IOException {
          performTask(request, response);
     }
     public void doPost(HttpServletRequest request,
                HttpServletResponse response)
      throws ServletException, IOException {
          performTask(request, response);
     }
     protected void performTask(HttpServletRequest
          request, HttpServletResponse response)
      throws ServletException, IOException {
          try {
          Writer out = response.getWriter();
          out.write("My session counter: ");
/* устанавливает счетчик количества вызовов сервлета
пользователем */
out.write(String.valueOf(
     prepareMyCookieCounter(request, response)));
                out.flush();
                out.close();
          } catch (Throwable e) {
                e.printStackTrace();
          throw new RuntimeException(
"Failed to handle request: " + e.toString());
          }
     }
/* обновляет в cookie счетчик обращений пользователя
к сервлету */
     protected int prepareMyCookieCounter(
HttpServletRequest req, HttpServletResponse resp) {
```

```
            Cookie[] cookies = req.getCookies();
            if (cookies != null) {
            for (int i = 0; i < cookies.length; i++) {
      if ("counter".equals(cookies[i].getName())) {
            String counterStr = cookies[i].getValue();
                        int counterValue;
                              try {
      counterValue = Integer.parseInt(counterStr);
                  } catch (NumberFormatException e) {
                                    counterValue = 0;
                              }
                              counterValue++;
      Cookie counterCookie =
new Cookie("counter", String.valueOf(counterValue));
            counterCookie.setMaxAge(MAX_AGE_COOKIE);
            resp.addCookie(counterCookie);
                        return counterValue;
                        }
                  }
            }
Cookie counterCookie = new Cookie("counter", "1");
      counterCookie.setMaxAge(MAX_AGE_COOKIE);
      response.addCookie(counterCookie);
            return 1;
      }
}
```

В результате в файле cookie будет содержаться следующая информация:

My session counter: 9

Задания к главе 17

Вариант А

Для всех заданий использовать авторизованный вход в приложение. Параметры авторизации, дату входа в приложение и время работы сохранять в сессии.

1. В тексте, хранящемся в файле, определить длину содержащейся в нем максимальной серии символов, отличных от букв. Все такие серии символов с найденной длиной сохранить в cookie.
2. В файле хранится текст. Для каждого из слов, которые вводятся в текстовые поля HTML-документа, вывести в файл cookie, сколько раз они встречаются в тексте.
3. В файле хранятся несколько стихотворений, которые разделяются строкой, состоящей из одних звездочек. В каком из стихо-

творений больше всего восклицательных предложений? Результат сохранить в файле cookie.

4. Записать в файл cookie все вопросительные предложения текста, которые хранятся в текстовом файле.
5. Код программы хранится в файле. Посчитать количество операторов этой программы и записать результаты поиска в файл cookie, перечислив при этом все найденные операторы.
6. Код программы хранится в файле. Сформировать файл cookie, записи которого дополнительно слева содержат уровень вложенности циклов. Ограничения на входные данные: а) ключевые слова используются только для обозначения операторов; б) операторы цикла записываются первыми в строке.
7. Посчитать, сколько раз в исходном тексте программы, хранящейся на диске, встречается оператор, который вводится с терминала. Сохранить в файле cookie также номера строк, в которых этот оператор записан. Ограничения: ключевые слова используются только для обозначения операторов.
8. Сохранить в cookie информацию, введенную пользователем, и восстановить ее при следующем обращении к странице.
9. Выбрать из текстового файла все числа-полиндромы и их количество. Результат сохранить в файл cookie.
10. В файле хранится текст. Найти три предложения, содержащие наибольшее количество знаков препинания, и сохранить их в файл cookie.
11. Подсчитать количество различных слов в файле и сохранить информацию в файл cookie.
12. В файле хранится код программы. Удалить из текста все комментарии и записать измененный файл в файл cookie.
13. В файле хранится HTML-документ. Проверить его на правильность и записать в файл cookie первую строку и позицию (если они есть), нарушающую правильность документа.
14. В файле хранится HTML-документ. Найти и вывести все незакрытые теги с указанием строки и позиции начала в файл cookie. При выполнении задания учесть возможность присутствия тегов, которые не требуется закрывать. Например:
.
15. В файле хранится HTML-документ с незакрытыми тегами. Закрыть все незакрытые теги так, чтобы документ HTML стал правильным, и записать измененный файл в файл cookie. При выполнении задания учесть возможность присутствия тегов, которых не требуется закрывать. Например:
.
16. В файле хранятся слова русского языка и их эквивалент на английском языке. Осуществить перевод введенного пользователем текста и записать его в файл cookie.

17. Выбрать из файла все адреса электронной почты и сохранить их в файл cookie.
18. Выбрать из файла имена зон (*.by, *.ru и т.д.) вводимые пользователем и сохранить их в файл cookie.
19. Выбрать из файла все заголовки разделов и подразделов (оглавление) и записать их в файл cookie.
20. При работе приложения сохранять в сессии имена всех файлов, к которым обращался пользователь.

Вариант В

Для заданий варианта В главы 4 каждому пользователю должен быть поставлен в соответствие объект сессии. В файл cookie должна быть занесена информация о времени и дате последнего сеанса пользователя и информация о количестве посещений ресурса и роли пользователя.

Тестовые задания к главе 17

Вопрос 17.1.

Каким образом можно явно удалить объект сессии?

1) нельзя, так как сессия может быть удалена только после истечения времени жизни;
2) вызовом метода **`invalidate()`** объекта сессии;
3) вызовом метода **`remove()`** объекта сессии;
4) вызовом метода **`delete()`** объекта сессии;
5) вызовом метода **`finalize()`** объекта сессии.

Вопрос 17.2.

Какие методы могут быть использованы объектом сессии?

1) `setAttribute(String name, Object value);`
2) `removeAttribute();`
3) `deleteAttribute();`
4) `setValue(String name, Object value);`
5) `getAttributeNames();`
6) `getInactiveTime().`

Вопрос 17.3.

Каким образом можно получить объект-сеанс из ассоциированного с ним объекта-запроса **`HttpServletRequest req`** ?

1) `HttpSession session = req.getSession();`
2) `HttpSession session = req.createHttpSession();`
3) `Session session = req.createSession();`
4) `Session session = req.getSession();`
5) `HttpSession session = req.getHttpSession();`
6) `HttpSession session = req.createSession();`
7) `HttpSession session = req.getSession(true).`

Вопрос 17.4.

Какие из следующих утверждений относительно объекта **`Cookie`** являются верными?

1) имя файла передается конструктору в качестве параметра при создании объекта **`Cookie`** и далее не может быть изменено;
2) имя файла может быть изменено с помощью вызова метода **`Cookie.setName(String name);`**
3) значение объекта может быть изменено с помощью вызова метода **`setValue(String value);`**
4) браузер ограничивает размер одного файла cookie до 4096 байт;
5) браузер не ограничивает общее число файлов cookie;
6) максимальное время существования файла cookie в днях устанавливается вызовом метода **`Cookie.setMaxAge(int day).`**

Вопрос 17.5.

Какие из следующих объявлений объекта класса **`Cookie`** верны?

1) `Cookie c1 = new Cookie ();`
2) `Cookie c2 = new Cookie ("cookie2");`
3) `Cookie c3 = new Cookie ("$cookie3", "value3");`
4) `Cookie c4 = new Cookie ("cookie  4", "value4");`
5) `Cookie c5 = new Cookie ("cookie5", "value5");`
6) `Cookie c6 = new Cookie ("6cookie", "value6");`
7) `Cookie c7 = new Cookie ("c7,8", "value7").`

Вопрос 17.6.

Каким образом файлы cookie присоединяются к объекту-ответу **`HttpServletResponse resp`**?

1) `resp.setCookie(Cookie cookie);`
2) `resp.addCookie(Cookie cookie);`
3) `resp.createCookie(Cookie cookie);`
4) `resp.putCookie(Cookie cookie);`
5) `resp.setCookies(Cookie cookie).`

Глава 18

JDBC

Драйвера, соединения и запросы

JDBC – это стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД. Это взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В JDBC определяются четыре типа драйверов:

1. Тип 1 – драйвер, использующий другой прикладной интерфейс, в частности ODBC, для работы с СУБД (так называемый JDBC-ODBC – мост). Стандартный драйвер первого типа **`sun.jdbc.odbc.JdbcOdbcDriver`** входит в JSDK.
2. Тип 2 – драйвер, работающий через нативные библиотеки (т.е. клиента) СУБД.
3. Тип 3 – драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером, который, в свою очередь, подключается к нужной СУБД.
4. Тип 4 – сетевой драйвер, работающий напрямую с нужной СУБД и не требующий установки native-библиотек.

Предпочтение естественным образом отдается второму типу, однако если приложение выполняется на машине, на которой не предполагается установка клиента СУБД, то выбор производится между третьим и четвертым типами. Причем четвертый тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер четвертого типа будет более эффективным по сравнению с третьим типом с точки зрения производительности. Первый же тип, как правило, используется редко, т.е. в тех случаях, когда у СУБД нет своего драйвера JDBC, зато есть драйвер ODBC.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Строго говоря, JDBC не имеет прямого отношения к J2EE, но так как взаимодействие с СУБД является неотъемлемой частью Web-приложений, то эта технология рассматривается в данном контексте.

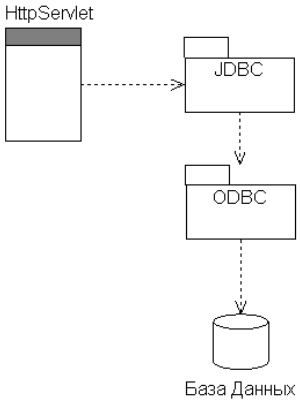

Рис. 18.1. Доступ к БД с помощью JDBC-ODBC-моста

Обычная последовательность действий:

1. *Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса*

```
String driverName = "org.gjt.mm.mysql.Driver";
```

Для СУБД MySQL или

```
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
```

для СУБД MSAccess и т.д.

После этого выполняется собственно загрузка драйвера в память:

```
Class.forName(driverName);
```

и становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно:

```
import COM.ibm.db2.jdbc.net.DB2Driver;
```

а затем,

```
new DB2Driver();
```

для СУБД DB2.

2. *Установка соединения с БД в виде*

```
Connection cn = DriverManager.getConnection(
"jdbc:mysql://localhost/my_db", "login", "password");
```

В результате будет возвращен объект **`Connection`** и будет одно установленное соединение с БД **`my_db`**. Класс **`DriverManager`** предоставляет средства для управления набором драйверов баз данных. Методу **`getConnection()`** необходимо передать тип и физическое месторасположение БД, а также логин и пароль для доступа. С помощью метода **`registerDriver()`** драйвера регистрируются, а методом **`getDrivers()`** можно получить список всех драйверов.

3. *Создание объекта для передачи запросов*

```
Statement st = cn.createStatement();
```

Объект класса **Statement** используется для выполнения запроса без его предварительной подготовки и команд SQL. Могут применяться также операторы для выполнения подготовленных запросов и хранимых процедур **PreparedStatement** и **CallableStatement**. Созданный объект можно использовать для выполнения запроса.

4. *Выполнение запроса*

Результаты выполнения запроса помещаются в объект **ResultSet**:

```
ResultSet rs = st.executeQuery(
   "SELECT * FROM my_table");
```

Для добавления или изменения информации в таблице вместо метода **executeQuery()** запрос помещается в метод **executeUpdate()**.

5. *Обработка результатов выполнения запроса* производится методами интерфейса **ResultSet**, где самыми распространенными являются **next()** и **getString()** и аналогичные методы, начинающиеся с **get**Тип**()** и **update**Тип**()**. Среди них следует выделить методы **getClob()** и **getBlob()**, позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его позиции в строке.

При первом вызове метода **next()** указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение **false**.

6. *Закрытие соединения*

```
cn.close();
```

После того как база больше не нужна, соединение закрывается.

Для того чтобы правильно пользоваться приведенными методами, программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально.

СУБД MySQL

СУБД MySQL совместима с JDBC и будет применяться для создания экспериментальных БД. Последняя версия может быть загружена с сайта **www.mysql.com**. Для корректной установки необходимо следовать инструкциям мастера установки. Каталог лучше выбирать по умолчанию, например: **c:/mysql**. По окончании установки СУБД автоматически создастся администратор СУБД с именем `root` и пустым паролем. Если

планируется разворачивать реально работающее приложение, то необходимо исключить тривиальных пользователей сервера (иначе злоумышленники могут получить полный доступ к БД). Для запуска следует использовать команду из папки **/mysql/bin**:

```
mysqld-nt  -standalone
```

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания БД и таблиц используются команды языка SQL.

Дополнительно требуется подключить библиотеку, содержащую драйвер MySQL

```
mysql-connector-java-3.0.10-stable-bin.jar,
```

и разместить ее в каталоге **/WEB-INF/lib** проекта.

Простое соединение и простой запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем **db2** и одной таблицей **users**. Таблица должна содержать два поля: символьное – **name** и числовое – **phone** и несколько занесенных записей. Сервлет, осуществляющий простейший запрос на выбор всей информации из таблицы, выглядит следующим образом.

```
/* пример # 1 : соединение с базой :
ServletToBase.java */
package test.com;
import java.io.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletToBase extends HttpServlet {
     public void doGet(HttpServletRequest req,
HttpServletResponse resp)
          throws ServletException, IOException {
          performTask(req, resp);
     }
     public void doPost(HttpServletRequest req,
HttpServletResponse resp)
          throws ServletException, IOException {
          performTask(req, resp);
     }
     public void performTask(HttpServletRequest req,
HttpServletResponse resp) {
```

```
          try {
               PrintWriter out = null;
               try {
resp.setContentType("text/html; charset=Cp1251");
          out = resp.getWriter();
          String sql;
          String url = "jdbc:mysql://localhost/db2";
          Class.forName("org.gjt.mm.mysql.Driver");
/* можно создать БД в MSAccess и настроить ODBC, но в
этом случае */
     //String url = "jdbc:odbc:db2";
     //Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                    Connection cn = null;
                    try {
     cn = DriverManager.getConnection(url,"root","");
               Statement st = null;
                         try {
                         st = cn.createStatement();
                              ResultSet rs = null;
                              try {
                    sql = "SELECT * FROM users";
                    rs = st.executeQuery(sql);
                    out.println("<B>From DataBase");
                              while (rs.next()) {
                              out.println(
          "<br>Name:-> "  + rs.getString(1)
          + " Phone:-> "  + rs.getInt(2));
                                        }
                              } finally {
/* закрыть ResultSet, если он был открыт и ошибка
произошла во время чтения данных */
          //проверка - успел ли создаться ResultSet
          if (rs != null)
               rs.close();
     else out.println(
     "ошибка произошла во время чтения данных");
                              }
                         } finally {
/*закрыть Statement, если он был открыт и ошибка про-
изошла во время чтения данных или создания
ResultSet*/
          //проверка - успел ли создаться Statement
```

```
            if (st != null)
                  st.close();
            else out.println("Statement не создан");
                              }
                        } finally {
/*закрыть Connection, если он был открыт и ошибка
произошла во время чтения данных или создания
ResultSet, или создания и использования Statement*/
            //проверка - успел ли создаться Connection
            if (cn != null)
                  cn.close();
            else out.println("Connection не создан");
                        }
                  } finally {
/*закрыть PrintWriter, если он был инициализирован и
ошибка произошла во время работы с БД*/
//проверка – успел ли инициализироваться PrintWriter
            if (out != null)
                  out.close();
            else out.println("PrintWriter not init");
                  }
            } catch (Throwable e) {
                  /* Handle exception here */
                  System.out.print(e);
            }
      }
}
```

В несложном приложении достаточно контролировать закрытие соединения, так как “провисшее” соединение снижает быстродействие системы.

Еще один способ соединения с базой данных возможен с использованием файла ресурсов **database.properties**, в котором хранятся, как правило, путь к БД, логин и пароль доступа. Например:

```
url=jdbc:mysql://localhost/my_db/?useUnicode=true&
characterEncoding=Cp1251
user=root
password=pass
```

В этом случае соединение создается в классе бизнес-логики, отвечающем за взаимодействие с базой данных, с помощью следующего кода:

```
public Connection getConnection()
      throws java.sql.SQLException {
            try {
Class.forName(
```

```
      "org.gjt.mm.mysql.Driver").newInstance();
            } catch (ClassNotFoundException e) {
throw new SQLException("can't load jdbc driver ");
            }
      ResourceBundle resource =
            ResourceBundle.getBundle("database");
            String url = resource.getString("url");
            String user = resource.getString("user");
            String password =
                  resource.getString("password");
      return
DriverManager.getConnection(url, user, password);
}
```

Объект класса **ResourceBundle**, содержащий ссылки на все внешние ресурсы проекта, создается с помощью вызова статического метода **getBundle()**, с параметром в виде имени необходимого файла ресурсов. Если требуемый файл отсутствует, то генерируется исключительная ситуация **MissingResourceException**. Для чтения из объекта ресурсов используется метод **getString()**, извлекающий информацию по указанному в параметре ключу. В классе **ResourceBundle** определен ряд полезных методов, в том числе метод **getKeys(),** возвращающий объект **Enumeration**, который применяется для последовательного обращения к элементам. Методы **getObject()** и **getStringArray()** извлекают соответственно объект и массив строк по передаваемому ключу.

Метаданные

Существует целый ряд методов интерфейсов **ResultSetMetaData** и **DatabaseMetaData** для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для строк подобных методов нет.

Получить объект **ResultSetMetaData** можно следующим образом:

```
ResultSetMetaData rsMetaData = rs.getMetaData();
```

Некоторые методы интерфейса **ResultSetMetaData**:

int getColumnCount() – возвращает число столбцов набора результатов объекта **ResultSet**;

String getColumnName(int column) – возвращает имя указанного столбца объекта **ResultSet**;

int getColumnType(int column) – возвращает тип данных указанного столбца объекта **ResultSet** и т.д.

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

Некоторые методы весьма обширного интерфейса **DatabaseMetaData**:

String getDatabaseProductName() – возвращает название СУБД;

String getDatabaseProductVersion() – возвращает номер версии СУБД;

String getDriverName() – возвращает имя драйвера JDBC;

String getUserName() – возвращает имя пользователя БД;

String getURL() – возвращает местонахождение источника данных;

ResultSet getTables() – возвращает набор типов таблиц, доступных для данной БД, и т.д.

Подготовленные запросы и хранимые процедуры

Для представления запросов существуют еще два типа объектов **PreparedStatement** и **CallableStatement**. Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных. Второй интерфейс используется для выполнения хранимых процедур.

Для подготовки SQL-запроса, в котором отсутствуют конкретные значения, используется метод **prepareStatement(String sql)**, возвращающий объект **PreparedStatement**. Установка входных значений конкретных параметров производится с помощью методов **setString()**, **setInt()** и подобных им. После чего и осуществляется непосредственное выполнение запроса методами **executeUpdate()**, **executeQuery()**. Так как данный оператор предварительно подготовлен, то он выполняется быстрее обычных операторов, ему соответствующих. Оценить преимущества во времени можно, выполнив большое число повторяемых запросов с предварительной подготовкой запроса и без нее.

```
/* пример # 2 : создание и выполнение подготовленного
запроса : PreparedStatementServlet.java */
package test.com;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
class Rec {
     static void insert(PreparedStatement ps, int id,
String name, String surname, int salary)
            throws SQLException {
     //   установка входных параметров
```

```
            ps.setInt(1, id);
            ps.setString(2, name);
            ps.setString(3, surname);
            ps.setInt(4, salary);
      //    выполнение подготовленного запроса
            ps.executeUpdate();
      }
}
public class PreparedStatementServlet
            extends HttpServlet {
      protected void doGet(HttpServletRequest req,
                        HttpServletResponse resp)
            throws ServletException, IOException {
            performTask(req, resp);
      }
      protected void doPost(HttpServletRequest req,
                        HttpServletResponse resp)
            throws ServletException, IOException {
            performTask(req, resp);
      }
      protected void performTask(HttpServletRequest req,
      HttpServletResponse resp)
            throws ServletException, IOException {
            resp.setContentType("text/html");
            PrintWriter out = resp.getWriter();
            try {
      Class.forName("org.gjt.mm.mysql.Driver");
            Connection cn = null;
                  try {
            cn = DriverManager.getConnection(
"jdbc:mysql://localhost/db3","root","");
                  PreparedStatement ps = null;
                  String sql =
"INSERT INTO emp(id,name,surname,salary) VALUES(?,?,?,?)";
      //    компиляция (подготовка) запроса
            ps = cn.prepareStatement(sql);
      Rec.insert(ps, 2203, "Иван", "Петров", 230);
      Rec.insert(ps, 2308, "John", "Black", 450);
      Rec.insert(ps, 2505, "Mike", "Call", 620);
            out.println("COMPLETE");
                  } finally {
                  if (cn != null) cn.close();
                  }
            } catch (Exception e) {
```

```
                e.printStackTrace();
            }
            out.close();
        }
}
```

Результатом выполнения данной программы будет добавление в базу данных **db3** трех записей и вывод в окно браузера слова COMPLETE.

Интерфейс **CallableStatement** расширяет возможности интерфейса **PreparedStatement** и обеспечивает выполнение хранимых процедур.

Хранимая процедура – это в общем случае любой код, выполняющийся в адресном пространстве процессов СУБД, который можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае хранимая процедура будет рассматриваться в более узком смысле как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных. Для создания объекта **CallableStatement** вызывается метод **prepareCall()** объекта **Connection**.

Хранимая процедура может иметь три типа параметров: входные, выходные и смешанные. Тип любого выходного параметра должен быть зарегистрирован методом **registerOutParameter()**. После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**, например:

```
String sql = "{ call infoFromMyTable(?,?,?,?) }";
CallableStatement cs = cn.prepareCall(sql);
...
ResultSet rs = cs.executeQuery();
```

Полученный в результате выполнения запроса объект **ResultSet** обрабатывается обычным способом.

Транзакции

При проектировании распределенных систем часто возникают ситуации, когда сбой в системе или какой-либо ее периферийной части может привести к потере информации или, что еще хуже, к финансовым потерям. Простейшим примером может служить пример с перечислением денег с одного счета на другой. Если сбой произошел в тот момент, когда операция снятия денег с одного счета уже произведена, а операция зачисления на другой счет еще не произведена, то система, позволяющая такие ситуации, должна быть признана не отвечающей требованиям заказчика. Или

должны выполняться обе операции, или не выполняться вовсе. В этом случае такие две операции трактуют как одну и называют транзакцией.

Транзакции (деловые операции) определяют как единицу работы, обладающую свойствами ACID:

- Атомарность – две или более операций выполняются все или не выполняется ни одна. Успешно завершенные транзакции фиксируются, в случае неудачного завершения происходит откат всей транзакции.
- Согласованность – если происходит сбой, то система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности проверяет успешное завершение всех операций транзакции.
- Изолированность – во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.
- Долговечность – все изменения, произведенные с данными во время транзакции, сохраняются, например, в базе данных. Это позволяет восстанавливать систему.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор `COMMIT`. В JDBC эта операция выполняется по умолчанию после каждого вызова методов **`executeQuery()`** и **`executeUpdate()`**, но для снятия блокировок БД следует выполнять фиксацию и после запросов на чтение. Если же необходимо сгруппировать запросы и только после этого выполнить операцию `COMMIT`, то сначала вызывается метод **`setAutoCommit()`** интерфейса **`Connection`** с параметром **`false`**, в результате выполнения которого текущее соединение с БД переходит в режим неавтоматического подтверждения операций. После этого выполнение любого запроса на изменение информации в таблицах базы данных не приведет к необратимым последствиям, пока операция `COMMIT` не будет выполнена непосредственно. Подтверждает выполнение SQL-запросов метод **`commit()`** интерфейса **`Connection`**, в результате действия которого все изменения таблицы производятся как одно логическое действие. Если же транзакция не выполнена, то методом **`rollback()`** отменяются действия всех запросов SQL, начиная от последнего вызова **`commit()`**. В следующем примере информация добавляется в таблицу в режиме действия транзакции, подтвердить или отменить действия которой можно, снимая или добавляя комментарий в строках вызова методов **`commit()`** и **`rollback().`**

```
/* пример # 3 : выполнение транзакции :
SQLTransactionServlet.java */
package test.com;
```

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
class ItemsEntries extends HashMap {
      HashMap map = new HashMap();
      ItemsEntries()  {
//для простоты пусть информация дана в виде
            map.put("FDD", new Integer(33));
            map.put("Mouse", new Integer(85));
            map.put("Modem", new Integer(23));
      }
}
public class SQLTransactionServlet
            extends HttpServlet {
      public void doGet(HttpServletRequest req,
                        HttpServletResponse resp)
            throws ServletException, IOException {
                  taskPerform(req,resp);
      }
      public void doPost(HttpServletRequest req,
                        HttpServletResponse resp)
            throws ServletException, IOException {
                  taskPerform(req,resp);
      }
      public void taskPerform(HttpServletRequest req,
                              HttpServletResponse res){
            try {
                  PrintWriter out = null;
      res.setContentType("text/html; charset=Cp1251");
      out = res.getWriter();
      String sql, url = "jdbc:mysql://localhost/db1";
      Class.forName("org.gjt.mm.mysql.Driver");
                  Connection cn = null;
                        try {
cn = DriverManager.getConnection(url, "root", "");
                  cn.setAutoCommit(false);
                  Statement st = null;
                        st = cn.createStatement();
                        ResultSet rs = null;
      String query = "SELECT COUNT (*) FROM items";
                        rs = st.executeQuery(query);
```

```
                    rs.next();
                out.println(
        "исходное количество записей в таблице: "
             + rs.getString(1) + "<BR>");
                    rs.close();
                    int total = 0;
             ItemsEntries items = new ItemsEntries();
             Iterator iter =
                items.map.keySet().iterator();
                    while(iter.hasNext()) {
                String mark = (String)iter.next();
                int count =
             ((Integer)items.map.get(mark)).intValue();
                out.println(mark + "  " + count);
        String upd =
             "INSERT INTO items(count, mark) VALUES("
             + count + ",'" + mark + "')";
                int tot = st.executeUpdate(upd);
        out.println("всего занесено строк: "
                + (total += tot) + "<BR>");
                                }
                    rs = st.executeQuery(query);
                    rs.next();
        out.println("количество записей в таблице"
        + "перед выполнением commit() или rollback(): "
        + rs.getString(1) + "<BR>");
                    rs.close();
        cn.commit();//подтверждение
        //cn.rollback();//откат
                    rs = st.executeQuery(query);
                    rs.next();
        out.println("количество записей"
        + "в таблице после подтверждения изменений: "
        + rs.getString(1) + "<BR>");
/*out.println("количество записей"
+ "после выполнения 'отката': " + rs.getString(1));*/
             } finally {
             if (cn != null) cn.close();
                    }
             } catch (Exception e) {
             e.printStackTrace();
             }
        }
```

```
}
```

Если таблица `items` базы данных **`db1`** до изменения выглядела, например, следующим образом:

	mark	count
☐	CD-ROM	15
☐	Flash-Usb	12
☐	HDD	55

Рис. 18.2. Таблица до выполнения запроса

то после внесения изменений и их подтверждения она примет вид:

	mark	count
☐	CD-ROM	15
☐	Flash-Usb	12
☐	HDD	55
☐	FDD	33
☐	Mouse	85
☐	Modem	23

Рис. 18.3. Таблица после подтверждения выполнения запроса

Приведенный пример в полной мере не отражает принципы транзакции, но демонстрирует способы ее поддержки методами языка Java.

Пул соединений

При большом количестве клиентов приложения к БД этого приложения выполняется большое количество запросов. Соединение с БД является дорогостоящей (по требуемым ресурсам) операцией. Эффективным способом решения данной проблемы является организация пула (pool) используемых соединений, которые не закрываются физически, а хранятся в очереди и предоставляются повторно для других запросов.

Пул соединений – это одна из стратегий предоставления соединений приложению (не единственная, да и самих стратегий организации пула существует несколько).

Разделяемый доступ к источнику данных можно организовать, например, путем объявления статической переменной типа **`DataSource`** из пакета **`javax.sql`**, однако в J2EE принято использовать для этих целей каталог. Источник данных типа **`DataSource`** – это компонент, предоставляющий соединение с приложением СУБД.

Класс **`InitialContext`**, как часть JNDI API, обеспечивает работу с каталогом именованных объектов. В этом каталоге можно связать объект источника данных **`DataSource`** с некоторым именем (не только с именем БД, но и вообще с любым), предварительно создав объект **`DataSource`**.

Затем созданный объект можно получить с помощью метода **lookup()** по его имени. Методу **lookup()** передается имя, всегда начинающееся с имени корневого контекста.

```
javax.naming.Context ct =
   new javax.naming.InitialContext();
DataSource ds =
(DataSource)ct.lookup("java:jdbc/mybd");
Connection cn = ds.getConnection("name", "pass");
```

После выполнения запроса соединение завершается и его объект возвращается обратно в пул вызовом:

```
cn.close();
```

Некоторые производители СУБД для облегчения создания пула соединений определяют собственный класс на основе интерфейса **DataSource**. В этом случае пул соединений может быть создан, например, следующим образом:

```
import COM.ibm.db2.jdbc.DB2DataSource;
    ...
   DB2DataSource ds = new DB2DataSource();
   ds.setServerName("//localhost:6061/mybd");
Connection cn = ds.getConnection("db2adm", "pass");
```

Драйвер определяется автоматически в объекте **DB2DataSource**.

Задания к главе 18

Вариант А

В каждом из заданий необходимо выполнить следующие действия:

- Организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение.
- Создать БД. Привести таблицы к одной из нормированных форм.
- Создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов.
- Создать класс на добавление информации.
- Создать HTML-документ с полями для формирования запроса.
- Результаты выполнения запроса передать клиенту в виде HTML-документа.

1. **Файловая система.** В БД хранится информация о дереве каталогов файловой системы – каталоги, подкаталоги, файлы.

   Для каталогов необходимо хранить:
   – родительский каталог;
   – название.
   Для файлов необходимо хранить:
   – родительский каталог;

– название;
– место, занимаемое на диске.

- Определить полный путь заданного файла (каталога).
- Подсчитать количество файлов в заданном каталоге, включая вложенные файлы и каталоги.
- Подсчитать место, занимаемое на диске содержимым заданного каталога.
- Найти в базе файлы по заданной маске с выдачей полного пути.
- Переместить файлы и подкаталоги из одного каталога в другой.
- Удалить файлы и каталоги заданного каталога.

2. **Видеотека.** В БД хранится информация о домашней видеотеке – фильмы, актеры, режиссеры.

Для фильмов необходимо хранить:
– название;
– актеров;
– дату выхода;
– страну, в которой выпущен фильм.
Для актеров и режиссеров необходимо хранить:
– ФИО;
– дату рождения.

- Найти все фильмы, вышедшие на экран в текущем и прошлом году.
- Вывести информацию об актерах, снимавшихся в заданном фильме.
- Вывести информацию об актерах, снимавшихся как минимум в 2 фильмах.
- Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.
- Удалить все фильмы, дата выхода которых была более 2 лет назад.

3. **Расписание занятий.** В БД хранится информация о преподавателях и проводимых ими занятиях.

Для предметов необходимо хранить:
– название;
– время проведения (день недели);
– аудитории, в которых проводятся занятия.
Для преподавателей необходимо хранить:
– ФИО;
– предметы, которые он ведет;
– количество пар в неделю по каждому предмету;
– количество студентов, занимающихся на каждой паре.

- Вывести информацию о преподавателях, работающих в заданный день недели в заданной аудитории.
- Вывести информацию о преподавателях, которые не ведут занятия в заданный день недели.
- Вывести дни недели, в которых проводится наименьшее количество занятий.
- Вывести дни недели, в которых занято наименьшее количество аудиторий.
- Перенести первые занятия заданных дней недели на последнее место.

4. **Письма.** В БД хранится информация о письмах и отправляющих их людях.

   Для людей необходимо хранить:
   – ФИО;
   – дату рождения.
   Для писем необходимо хранить:
   – отправителя;
   – получателя;
   – тему письма;
   – текст письма;
   – дату отправки.
   - Найти пользователя, длина писем которого наименьшая.
   - Вывести информацию о пользователях, а также количестве полученных и отправленных ими письмах.
   - Вывести информацию о пользователях, которые получили хотя бы одно сообщение с заданной темой.
   - Вывести информацию о пользователях, которые не получали сообщения с заданной темой.
   - Направить письмо заданного человека с заданной темой всем людям.

5. **Сувениры.** В БД хранится информация о сувенирах и их производителях.

   Для сувениров необходимо хранить:
   – название;
   – производителя;
   – дату выпуска;
   – цену.
   Для производителей необходимо хранить:
   – название;
   – страну.
   - Вывести информацию о сувенирах заданного производителя.

- Вывести информацию о сувенирах, произведенных в заданной стране.
- Вывести информацию о производителях, чьи цены на сувениры меньше 1000.
- Вывести информацию о производителях заданного сувенира, произведенных в прошлом году.
- Удалить заданного производителя и его сувениры.

6. **Заказ.** В БД хранится информация о заказах магазина и товарах в них.
   Для заказа необходимо хранить:
   – номер заказа;
   – товары в заказе;
   – дату поступления.
   Для товаров в заказе необходимо хранить:
   – товар;
   – количество.
   Для товара необходимо хранить:
   – название;
   – описание;
   – цену.
   - Вывести полную информацию о заданном заказе.
   - Вывести номера заказов, сумма которых не превосходит 100 и количество различных товаров равно 1.
   - Вывести номера заказов, содержащие заданный товар.
   - Вывести номера заказов, не содержащие заданный товар и поступившие в течение текущего дня.
   - Сформировать новый заказ, состоящий из товаров, заказанных в текущий день.
   - Удалить все заказы, в которых присутствует заданное количество заданного товара.
7. **Продукция.** В БД хранится информация о продукции компании.
   Для продукции необходимо хранить:
   – название;
   – группу продукции (телефоны, телевизоры и др.);
   – описание;
   – дату выпуска;
   – значения параметров.
   Для групп продукции необходимо хранить:
   – название;
   – перечень групп параметров (размеры и др.).
   Для групп параметров необходимо хранить:
   – название;

– перечень параметров.

Для параметров необходимо хранить:

– название;

– единицу измерения.

- Вывести перечень параметров для заданной группы продукции.
- Вывести перечень продукции, не содержащий заданного параметра.
- Вывести информацию о продукции для заданной группы.
- Вывести информацию о продукции и всех ее параметрах со значениями.
- Удалить из базы продукцию, содержащую заданные параметры.
- Переместить группу параметров из одной группы товаров в другую.

8. **Погода.** В БД хранится информация о погоде в различных регионах.

Для погоды необходимо хранить:

– регион;

– дату;

– температуру;

– осадки.

Для регионов необходимо хранить:

– название;

– площадь;

– тип жителей.

Для типов жителей необходимо хранить:

– название;

– язык общения.

- Вывести сведения о погоде в заданном регионе.
- Вывести даты, когда в заданном регионе шел снег, и температура была ниже –10.
- Вывести информацию о погоде за прошедшую неделю в регионах, жители которых общаются на заданном языке.
- Вывести среднюю температуру за прошедшую неделю в регионах с площадью более 1000.

9. **Магазин часов.** В БД хранится информация о часах магазина.

Для часов необходимо хранить:

– марку;

– тип (кварцевые, механические);

– цену;

– количество;

– производителя.

Для производителей необходимо хранить:

– название;

– страну.

- Вывести марки механических часов.

- Вывести информацию о механических часах, цена на которые не превосходит 1000.
- Вывести марки часов, изготовленных в заданной стране.
- Вывести производителей, общая сумма часов которых в магазине не превышает 100.

10. **Города.** В БД хранится информация о городах и их жителях.
    Для городов необходимо хранить:
    – название;
    – год создания;
    – площадь;
    – количество населения для каждого типа жителей.
    Для типов жителей необходимо хранить:
    – город проживания;
    – название;
    – язык общения.
    - Вывести информацию обо всех жителях заданного города, разговаривающих на заданном языке.
    - Вывести информацию обо всех городах, в которых проживают жители выбранного типа.
    - Вывести информацию о городе с максимальным количеством населения и всех типах жителей, в нем проживающих.
    - Вывести информацию о самом древнем типе жителей.
11. **Планеты.** В БД хранится информация о планетах, их спутниках и галактиках.
    Для планет необходимо хранить:
    – название;
    – радиус;
    – температуру ядра;
    – наличие атмосферы;
    – наличие жизни;
    – спутники.
    Для спутников необходимо хранить:
    – название;
    – радиус;
    – расстояние до планеты.
    Для галактик необходимо хранить:
    – название;
    – планеты.
    - Вывести информацию обо всех планетах, на которых присутствует жизнь, и их спутниках в заданной галактике.

- Вывести информацию о планетах и их спутниках, имеющих наименьший радиус и наибольшее количество спутников.
- Вывести информацию о планете, галактике, в которой она находится, и ее спутниках, имеющей максимальное количество спутников, но с наименьшим общим объемом этих спутников.
- Найти галактику, сумма ядерных температур планет которой наибольшая.

12. **Точки.** В БД хранится некоторое конечное множество точек с их координатами.
    - Вывести точку из множества, наиболее приближенную к заданной.
    - Вывести точку из множества, наиболее удаленную от заданной.
    - Вывести точки из множества, лежащие на одной прямой с заданной прямой.
13. **Треугольники.** В БД хранятся треугольники и координаты их точек на плоскости.
    - Вывести треугольник, площадь которого наиболее приближается к заданной.
    - Вывести треугольники, сумма площадей которых наиболее приближается к заданной.
    - Вывести треугольники, помещающиеся в окружность заданного радиуса.
14. **Словарь.** В БД хранится англо-русский словарь, в котором для одного английского слова может быть указано несколько его значений и наоборот. Со стороны клиента вводятся последовательно английские (русские) слова. Для каждого из них вывести на консоль все русские (английские) значения слова.
15. **Словари.** В двух различных базах данных хранятся два словаря: русско-белорусский и белорусско-русский. Клиент вводит слово и выбирает язык. Вывести перевод этого слова.
16. **Стихотворения.** В БД хранятся несколько стихотворений с указанием автора и года создания. Для хранения стихотворений использовать объекты типа Blob. Клиент выбирает автора и критерий поиска.
    - В каком из стихотворений больше всего восклицательных предложений?
    - В каком из стихотворений меньше всего повествовательных предложений?
    - Есть ли среди стихотворений сонеты и сколько их?
17. **Четырехугольники.** В БД хранятся координаты вершин выпуклых четырехугольников на плоскости.
    - Вывести координаты вершин параллелограммов.
    - Вывести координаты вершин трапеций.

18. **Треугольники.** В БД хранятся координаты вершин треугольников на плоскости.
    - Вывести все равнобедренные треугольники.
    - Вывести все равносторонние треугольники.
    - Вывести все прямоугольные треугольники.
    - Вывести все тупоугольные треугольники с площадью больше заданной.

Вариант В

Для заданий варианта В главы 4 создать базу данных для хранения информации. Определить класс для организации соединения (пула соединений). Создать классы для выполнения соответствующих заданию запросов в БД.

Тестовые задания к главе 18

Вопрос 18.1.

Объекты каких классов позволяют загрузить и зарегистрировать необходимый JDBC-драйвер и получить соединение с базой данных или получить доступ к БД через пространство имен?

1) `java.sql.DriverManager;`
2) `javax.sql.DataSource;`
3) `java.sql.Statement;`
4) `java.sql.ResultSet;`
5) `java.sql.Connection.`

Вопрос 18.2.

Какой интерфейс из пакета **`java.sql`** должен реализовывать каждый драйвер JDBC?

1) `Driver;`
2) `DriverManager;`
3) `Connection;`
4) `DriverPropertyInfo;`
5) `ResultSet.`

Вопрос 18.3.

С помощью какого метода интерфейса **`Connection`** можно получить сведения о базе данных, с которой установлено соединение?

1) `getMetaData();`
2) `getDatabaseInfo();`
3) `getInfo();`
4) `getMetaInfo();`
5) `getDatabaseMetaData().`

Вопрос 18.4.

Какой интерфейс пакета **`java.sql`** используется, когда запрос к источнику данных является обращением к хранимой процедуре?

1) `Statement;`
2) `PreparedStatement;`
3) `StoredStatement;`
4) `CallableStatement;`
5) `StoredProcedure.`

Вопрос 18.5.

Какой метод интерфейса **`Statement`** необходимо использовать при выполнении SQL-оператора SELECT, который возвращает объект **`ResulSet`**?

1) `execute();`
2) `executeQuery();`
3) `executeUpdate();`
4) `executeBatch();`
5) `executeSelect();`
6) `executeSQL().`

Глава 19
JAVA SERVER PAGES

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems, чтобы облегчить создание страниц с динамическим содержанием. Страница JSP обеспечивает разделение динамической и статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание. Это свойство используется при разработке и поддержке страниц, так как дизайнерам нет необходимости знать, как работать с динамическими данными. Один и тот же функционал может иметь представления HTML для браузера и WML для сотового телефона с WAP, так называемый "Device Independence". Например, пусть существуют страницы **`*.jsp`** HTML-формата в папке **`/iexplorer`** и те же **`*.jsp`** WML-формата в папке **`/wap`**, а из сервлета запрос перенаправляется на нужный диалог в зависимости от типа устройства, т.е. имя **`JSP = <device type>/<dialog name>.jsp`**, таким образом, в качестве login-диалога для WAP-устройства будет вызван **`/wap/login.jsp`**.

Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности в его **`service()`**-метод.

Под терминами "динамическое/статическое содержание" обычно понимаются не части JSP, а динамическое/статическое содержание Web-приложения, а именно:

- динамические ресурсы – изменяемые в процессе работы: это и сервлеты, и JSP;
- статические ресурсы – не изменяемые в процессе работы, например HTML, JavaScript, картинки и т.д.

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера, в то время как динамические нуждаются в движке (Servlet Engine) и в большинстве случаев в доступе к уровню данных. Поэтому, имея разделенный на три зоны контент в стандартной инфраструктуре (внешний и внутренний firewall, DMZ, связка "HTTP-сервер – сервер приложений"), можно размещать статические ресурсы перед firewall, тем самым повышая производительность в несколько, раз не нарушая требований безопасности.

Рекомендуется разрабатывать параллельно две части: Web-приложение, состоящее только из динамических ресурсов, и Web-приложение, состоящее только из статических ресурсов.

Чтобы облегчить внедрение динамической структуры, JSP использует ряд тегов, которые дают возможность проектировщику страницы вставить значение полей объекта JavaBean в файл JSP.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

- *Разделение динамического и статического содержания.*
  Возможность разделить логику приложения и дизайн Web-страницы снижает сложность разработки Web-сайтов и упрощает их поддержку.
- *Независимость от платформы.*
  Так как JSP-технология, основанная на языке программирования Java, не зависит от платформы, то JSP могут выполняться практически на любом Web-сервере. Разрабатывать JSP можно на любой платформе.
- *Многократное использование компонентов.*
  Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание Web-сайтов.
- *Скрипты и теги.*
  Спецификация JSP объявляет собственные теги, кроме того, JSP поддерживают как JavaScript так и HTML-теги. JavaScript обычно используется, чтобы добавить функциональные возможности на уровне HTML-страницы. Теги обеспечивают возможность использования JavaBean и выполнение обычных функций.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером. Этот процесс ответствен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP. Об архитектуре сайтов, использующих JSP/Servlet-технологии, часто говорят как о thin-client (использование ресурсов клиента незначительно), потому что большая часть логики выполняется на сервере.

Процессы, выполняемые с файлом JSP при первом вызове или при его изменении:

1. Браузер делает запрос к странице JSP.
2. JSP-engine анализирует содержание файла JSP.
3. JSP-engine создает временный сервлет с кодом, основанным на исходном тексте файла JSP, при этом контейнер транслирует операторы Java в метод **`_jspService()`**. Если нет ошибок компиляции, то этот метод вызывается для непосредственной обработки запроса. Полученный сервлет ответствен за исполнение статических элементов JSP, определенных во время разработки в дополнение к созданию динамических элементов.

4. Полученный текст компилируется в файл ***.class**.
5. Вызываются методы **init()** и **service()** (**doGet()** или **doPost()**), и сервлет логически исполняется.
6. Сервлет установлен. Комбинация статического HTML и графики вместе с динамическими элементами, определенными в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа **ServletResponse**.

Последующие вызовы файла JSP просто вызовут сервисный метод сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен и сервлет не будет выгружен вручную либо пока не будет изменен файл JSP.

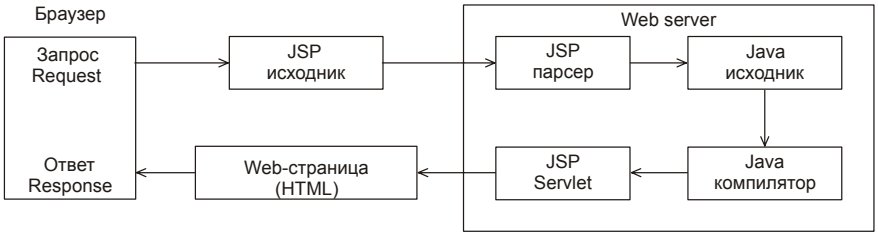

Рис. 19.1. Рабочий цикл JSP

JSP-код Java заключается в специальные теги, которые указывают контейнеру, чтобы он использовал этот код для генерации сервлета или его части. Таким образом поддерживается документ, который одновременно содержит и страницу, и код Java, который управляет этой страницей. Статические части HTML-страниц посылаются в виде строк в метод **write()**. Динамические части включаются прямо в код сервлета. С этого момента страница ведет себя как обычная HTML-страница с ассоциированным сервлетом.

JSP составляется из стандартных HTML-тегов, JSP-тегов и пользовательских JSP-тегов. В спецификации JSP 1.1 существует шесть основных тегов:

<%@ *директива* **%>**

<%! *объявление* **%>**

<% *скриптлет* **%>**

<%= *вычисляемое выражение* **%>**

<%-- *JSP-комментарий* **--%>**

<!-- *HTML-комментарий* **-->**

Директивы

Директивы используются для установки параметров серверной страницы JSP и имеют общий вид:

```
<%@ директива имя=значение %>
```

Например:

```
<%@ page
language="java"
contentType="text/html; charset=Cp1251"
pageEncoding="Cp1251"
errorPage="errorjsp.jsp"
info="директива info"
import="java.util.*"
%>
```

Параметр **language** директивы **page** определяет используемый язык, пока он только один. В параметр **info** можно помещать информацию о данной странице, которую можно получить, используя метод **getServletInfo()**. Параметр **import** описывает пакеты и типы, доступные среде выполнения сценариев. Параметр **contentType** специфицирует декодирование символов и MIME-тип JSP-ответа. Директива **taglib** подключает библиотеки пользовательских тегов. Директива **include** позволяет включать в код данной страницы JSP другие документы допустимых типов.

```
<!--пример # 1 : включение в код содержимого другой
страницы JSP : jsp01.jsp -->
<HTML>
<HEAD><TITLE>jsp01.jsp</TITLE></HEAD>
<BODY><H1>First JSP</H1>
<%@ include file="jsp02.jsp"%>
</BODY></HTML>
```

Включаемая страница JSP может иметь следующий простейший вид:

```
<!-- пример # 2 : код включаемой JSP : jsp02.jsp -->
<HTML><HEAD>
<TITLE>jsp02.jsp</TITLE>
</HEAD>
<BODY>
<P> jsp02.jsp was included in  jsp01.jsp</P>
</BODY></HTML>
```

Запуск JSP:

```
http://127.0.0.1:8080/FirstProject/jsp01.jsp
```

Параметр **errorPage** указывает на страницу, переход к которой будет осуществлен в случае возникновения ошибки в текущей странице.

```
<!--пример # 3 : генерация ошибки : jsp03.jsp -->
<HTML><HEAD>
<%@ page
language="java"
contentType="text/html; charset=Cp1251"
pageEncoding="Cp1251"
errorPage="errorjsp.jsp"
%>
</HEAD>
<BODY><FORM>
<INPUT type="checkbox"
           name="checkB" value="yesError">
<INPUT type="submit"
           name="error" value="ERROR"></FORM>
<%
if("yesError".equals(request.getParameter("checkB")))
     throw new Exception("yesError");
%>
</BODY></HTML>
```

Страница, вызываемая при ошибках, может иметь статический вид, но при необходимости сообщает о типе и месте возникшего исключения в понятной для клиента приложения форме.

```
<!--пример # 4 : ERROR PAGE : errorjsp.jsp -->
<HTML><HEAD>
<TITLE>errorjsp.jsp</TITLE>
</HEAD>
<BODY><P>Exception Generated!</P></BODY>
</HTML>
```

Роль простейшей JSP может сыграть обычная HTML-страница, переименованная с расширением **.jsp**.

Объявления

Блок объявлений содержит переменные Java и методы, которые вызываются в expression-блоке. Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптлетах и выражениях. Например:

```
<%!
     private int getCount = 0;
     private String getString(){return "СТРОКА";}
     String text = new String("Слово");
%>
```

Скриптлеты

JSP поддерживает вживление Java-кода в скриптлет-блок. Скриптлеты обычно используют маленькие блоки кода и выполняются во время обработки запроса клиента. Когда все скриптлеты собираются воедино в том порядке, в котором они записаны на странице, они должны представлять собой правильный код языка программирования. Контейнер помещает код Java в метод **_jspServlet()** на этапе трансляции.

```
<!--пример # 5 : jsp со скриптлетами : simple.jsp -->
<HTML><HEAD>
<%@ page
language="java"
contentType="text/html; charset=Cp1251"
pageEncoding="ISO-8859-5"
%>
</HEAD>
<BODY><FORM>
11 * 11 - 9 =
<INPUT type="text" name="text" size="2">
<br><br>
<INPUT type="submit" name="submit" value="Ответ">
</FORM>
<% if(request.getParameter("text") != null) {
     if ("112".equals(request.getParameter("text"))){
%>
<br> Верно!
<% } else { %>
<br> ОШИБКА!
<% } }%>
</BODY></HTML>
```

Но такие конструкции, как правило, не используются. JSP все больше походит на Java-программы, а теги JSP встречаются лишь изредка.

Выражения

В качестве выражений используются операторы языка Java, которые вычисляются, после чего результат вычисления преобразуется в строку **String** и посылается в поток **out**, как в случае

```
<%= text + new String(" 1") %>
<%= Runtime.getRuntime().totalMemory() %>
```

Первое выражение к строке **text** присоединяет вновь созданную строку и отправляет результат в поток **out**. Второе выражение определяет количество свободной памяти.

```
<!--пример # 6 : jsp с выражениями, объявлениями :
simplecount.jsp -->
<HTML><HEAD>
<%@ page
language="java"
contentType="text/html; charset=Cp1251"
pageEncoding="Cp1251"
import="java.util.*"
%>
<TITLE>simplecount.jsp</TITLE>
</HEAD>
<%!  long localTime = System.currentTimeMillis();
     Date localDate = new Date();
     int hitCount = 0; %>
<BODY>
<H2> Дата загрузки <%= localDate %> </H2>
<H2> Сегодня <%= new Date()%> </H2>
<H3> Страница работает
  <%=(System.currentTimeMillis() - localTime)/1000 %>
секунд </H3>
<H3> Страницу посетили <%= ++hitCount %> раз, начиная
с <%= localDate %> </H3>
<% System.out.println(
     "Страницу посетили (на консоль): " + hitCount);
     System.out.println("до свидания"); %>
</BODY></HTML>
```

Неявные объекты

JSP-страница всегда имеет доступ ко многим функциональным возможностям сервлета, создаваемым Web-контейнером по умолчанию. Неявный объект:

- **request** – представляет запрос клиента. Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletRequest**. Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletRequest**. Область видимости в пределах страницы.
- **response** – представляет ответ клиенту. Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletResponse**. Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletResponse**. Область видимости в пределах страницы.

- **pageContext** – определяет контекст JSP-страницы и предоставляет доступ к неявным объектам. Объект класса **javax.servlet.jsp.PageContext**. Область видимости в пределах страницы.
- **session** – создается контейнером для протокола HTTP и является экземпляром класса **javax.servlet.http.HttpSession**, предоставляет информацию о сессии клиента, если такая была создана. Область видимости в пределах сессии.
- **application** – контейнер, в котором исполняется JSP-страница, является экземпляром класса **javax.servlet.ServletContext**. Область видимости в пределах приложения.
- **out** – содержит выходной поток сервлета. Информация, посылаемая в этот поток, передается клиенту. Объект является экземпляром класса **javax.servlet.jsp.JspWriter**. Область видимости в пределах страницы.
- **config** – содержит параметры конфигурации сервлета и является экземпляром класса **javax.servlet.ServletConfig**. Область видимости в пределах страницы.
- **page** – ссылка **this** для текущего экземпляра данной страницы является объектом **java.lang.Object**. Область видимости в пределах страницы.
- **exception** – представляет собой исключение одного из подклассов класса **java.lang.Throwable**, которое передается странице сообщения об ошибках и доступно только на ней.

Стандартные элементы action

Однако теги, объявленные выше, применяются не так уж часто. Наиболее используемыми являются стандартные теги версии JSP 2.0:

- **jsp:useBean** – позволяет использовать экземпляр компонента **JavaBean**. Если экземпляр с указанным идентификатором не существует, то он будет создан с областью видимости **page** (страница), **request** (запрос), **session** (сессия) или **application** (приложение). Объявляется, как правило, с атрибутами **id** (имя объекта), **scope** (область видимости), **class** (полное имя класса), **type** (по умолчанию **class**).

```
<jsp:useBean id="ob"
             scope="session"
      class="test.MyBean" />
```

Создан объект **ob** класса **MyBean**, и в дальнейшем через это имя можно вызывать доступные методы класса.

```
package test;
public class MyBean {
      private String info = "нет информации";
      public String getInfo() {
      return info;
      }
      public void setInfo(String s) {
      info = s;
      }
}
```

- **jsp:setProperty** – позволяет устанавливать значения полей указанного в атрибуте **name** объекта:

```
<jsp:setProperty name="ob"
                        property="info"
                        value="привет" />
```

- **jsp:getProperty** – получает значения полей указанного объекта, преобразует его в строку и отправляет в неявный объект **out**:

```
<jsp:getProperty name="ob" property="info" />
```

- **jsp:include** – позволяет включать файлы в генерируемую страницу при запросе страницы:

```
<jsp:include page="относительный URL"
                      flush="true"/>
```

- **jsp:forward** – позволяет передать запрос другой странице:

```
<jsp:forward page="относительный URL"/>
```

- **jsp:plugin** – замещается тэгом **<OBJECT>** или **<EMBED>**, в зависимости от типа броузера, в котором будет выполняться подключаемый апплет или Java Bean.
- **jsp:params** – группирует параметры внутри тега **jsp:plugin**.
- **jsp:param** – добавляет параметры в объект запроса, например в элементах **forward**, **include**, **plugin**.
- **jsp:fallback** – указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься. Используется внутри элемента **plugin**.
- **jsp:text** – содержит текстовую информацию.

В качестве примера можно привести следующий фрагмент:

```
<jsp:plugin type="bean | applet"
                  code="test.com.ReadParam"
                  width="250"
                  height="250">
```

```
        <jsp:params>
           <jsp:param name="bNumber" value="7" />
           <jsp:param name="state" value="true" />
        </jsp:params>
        <jsp:fallback>
             <p> unable to start plugin </p>
        </jsp:fallback>
    </jsp:plugin>
```

Код апплета находится в примере 5 главы 10, и пакет, в котором он объявлен, должен быть расположен в корне папки **/WEB-INF**, а не в папке **/classes**.

Элементы **<jsp:attribute>**, **<jsp:body>**, **<jsp:invoke>**, **<jsp:doBody>**, **<jsp:element>**, **<jsp:output>** используются в основном при включении в страницу пользовательских тегов.

Извлечение полей и значений

В приведенном ниже примере рассматривается JSP, с помощью которой можно отправить на сервер заполненную клиентом форму. При посылке формы с заполненными полями по адресу URL страница сама определяет поля и выводит их на экран. Это позволяет совместить в одном файле и страницу, содержащую форму для заполнения, и код ответа, обрабатывающий приходящие данные, а именно в **index.jsp**.

```
<!--пример # 7 : считывание информации и генерация
ответа : index.jsp -->
<HTML><HEAD>
<%@ page
language="java"
contentType="text/html; charset=Cp1251"
pageEncoding="Cp1251"
%>
<META http-equiv="Content-Type" content="text/html;
charset=Cp1251">
<TITLE>index.jsp</TITLE>
<%@ page import="java.util.*" %>
<BODY>
<H1>Форма для заполнения</H1>
<H3><%
  request.setCharacterEncoding("Cp1251");
  Enumeration flds = request.getParameterNames();
  if(!flds.hasMoreElements()) { %>
<FORM method="POST" action="index.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
```

```
Field<%=i%>: <INPUT type="text"
                      size="20" name="Поле<%=i%>"
                      value="Значение<%=i%>"><BR>
<%  } %>
<INPUT TYPE=submit name=submit
                   value="Переслать данные">
</FORM>
<%} else {
     while(flds.hasMoreElements()) {
     String field = (String)flds.nextElement();
     String value = request.getParameter(field);
%>
<LI><%= field %> = <%= value %></LI>
<%  }  } %></H3>
</BODY></HTML>
```

JSP + Servlet + JSP

В большинстве случаев используются не сервлеты или JSP, а их сочетание. В JSP представляется, как будут выглядеть результаты запроса, а сервлет отвечает за вызов классов бизнес-логики и передачу результатов выполнения бизнес-логики в соответствующие JSP и их вызов. Т.е. сервлеты не генерируют ответа сами, а только выступают в роли контроллера запросов. Такая архитектура построения приложений носит название MVC (Model/View/Controller). Model – классы бизнес-логики и длительного хранения, View – страницы JSP, Controller – сервлет.

В следующем примере JSP-страница **login.jsp** содержит форму для ввода логина и пароля для аутентификации в системе:

```
<!--пример # 8 : форма ввода информации и вызов контроллера : login.jsp -->
<%@ page errorPage="error.jsp" %>
<HTML><HEAD><TITLE>Login</TITLE></HEAD>
<BODY><H3>Login</H3>
<HR>
<FORM name="loginForm"
     method="POST"
     action="/FirstProject/serv/controller">
     <INPUT type="hidden" name="cmd" value="login">
     Login:<BR>
     <INPUT type="text"
            name="login"
            value=""><BR>
     Password:<BR>
     <INPUT type="password"
```

```
            name="password"
            value=""><BR>
      <INPUT type="submit" value="Enter">
</FORM>
<HR>
</BODY></HTML>
```

Код сервлета-контроллера **SimpleController**:

```
/* пример # 9 : контроллер запросов :
SimpleController.java */
package test.com;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.*;
public class SimpleController extends HttpServlet {
  public void doGet(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, IOException {
    performTask(request, response);
  }
  public void doPost(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, IOException {
      performTask(request, response);
}
protected void performTask(HttpServletRequest
            request, HttpServletResponse response)
      throws ServletException, IOException {
      try {
      String cmd = request.getParameter("cmd");
      //проверка команды
            if ("login".equals(cmd)) {
            //команда на логин в систему
      String login = request.getParameter("login");
String password = request.getParameter("password");
      LoginLogic loginLogic = new LoginLogic();
            //проверка пароля
      if (loginLogic.checkLogin(login, password)) {
            request.setAttribute("user", login);
            jump("main.jsp", request, response);
      } else {
             jumpError(
"Incorrect login or password", request, response);
```

```
            }
            } else {
            jump("login.jsp", request, response);
            }
      } catch (Throwable e) {
            e.printStackTrace();
      jumpError(e.toString(), request, response);
      }
}
//переход на указанную JSP-страницу
protected void jump(String url, HttpServletRequest
request, HttpServletResponse response)
      throws ServletException, IOException {
      RequestDispatcher rd =
      getServletContext().getRequestDispatcher(url);
      rd.forward(request, response);
      }
//переход на страницу ошибки
protected void jumpError(String errorMessage,
      HttpServletRequest request,
      HttpServletResponse response)
       throws ServletException, IOException {
      request.setAttribute("errorMessage",
                        errorMessage);
      jump("error.jsp", request, response);
      }
}
```

Ниже приведен код класса бизнес-логики **LoginLogic**, выполняющий проверку правильности введенных логина и пароля с помощью запроса в БД:

```
/* пример # 10 : бизнес-класс проверки :
LoginLogic.java */
package test.com;
import java.sql.*;
public class LoginLogic {
  public boolean checkLogin(
            String login, String password) {
   //проверка логина и пароля
         try {
       String url = "jdbc:mysql://localhost/db4";
       Class.forName("org.gjt.mm.mysql.Driver");
       Connection cn = null;
```

```
      try {
   cn = DriverManager.getConnection(url, "root", "");
       PreparedStatement st = null;
        try {
     st = cn.prepareStatement(
"SELECT * FROM USERS WHERE LOGIN = ? AND PASSWORD = ?");
          st.setString(1, login);
          st.setString(2, password);
          ResultSet rs = null;
          try {
            rs = st.executeQuery();
        /* проверка, что есть хотя бы один пользова-
тель с указанным логином и паролем */
           return rs.next();
          } finally {
            if (rs != null) rs.close();
                    }
        } finally {
          if (st != null) st.close();
                  }
      } finally {
        if (cn != null) cn.close();
              }
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }
}
```

Страница **main.jsp** показывается пользователю в случае успешной аутентификации в приложении:

```
<!--пример # 11 : сообщение о входе : main.jsp -->
<%@ page errorPage="error.jsp" %>
<HTML><HEAD>
     <TITLE>Welcome</TITLE>
</HEAD>
<BODY>
<H3>Welcome:
<%=(String) request.getAttribute("user")%></H3>
</BODY></HTML>
```

Страница **error.jsp** показывается пользователю в случае возникновения ошибок (например, если неправильно введены логин и пароль):

```
<!-- пример # 12 : страница ошибок, предлагающая по-
вторить процедуру ввода информации : error.jsp -->
<%@ page isErrorPage="true" %>
<html><head>
     <title>Error</title>
</head>
<body>
<h3>Error</h3>
<hr>
<%=(request.getAttribute("errorMessage") != null) ?
(String) request.getAttribute("errorMessage") :
((exception != null)
? exception.toString() : "unknown error")%>
<hr>
<a href=
"http://localhost:8080/FirstProject/serv/controller">
Return to login page</a>
</body></html>
```

И последнее, что надо сделать в приложении, – это настроить файл **web.xml**, чтобы можно было обращаться к сервлету-контроллеру по имени **serv/controller,** т.е. необходимо настроить mapping (маппинг).

```
<!--пример # 13 : имя сервлета и путь к нему :
web.xml -->
     <servlet>
     <servlet-name>controller</servlet-name>
     <display-name>FirstProject</display-name>
     <servlet-class>test.com.SimpleController
</servlet-class>
     </servlet>
<servlet-mapping>
     <servlet-name>controller</servlet-name>
     <url-pattern>/serv/controller</url-pattern>
</servlet-mapping>
```

В данном случае в поле **<servlet-name>** было занесено имя **controller,** а в поле **<url-pattern>** – соответственно **/serv/controller**. По умолчанию эти поля выглядели бы следующим образом:

```
<servlet-name>SimpleController</servlet-name>
<url-pattern>/SimpleController</url-pattern>
```

Запуск примера производится из командной строки Web-браузера при запущенном Web-сервере Tomcat 4.1, например в виде:

```
http://localhost:8080/FirstProject/login.jsp
```

В этом случае при вызове сервлета в браузере будет отображен путь и имя в виде:

```
http://localhost:8080/FirstProject/serv/controller
```

Задания к главе 19

Вариант А

Реализовать приложение, используя технологию взаимодействия JSP и сервлетов. Вся информация должна храниться в базе данных.

1. Осуществить перевод денег с одного счета на другой с указанием реквизитов: Банк, Номер счета, Тип счета, Сумма. Таблицы должны находиться в различных базах данных. Подтверждение о выполнении операции должно выводиться в JSP с указанием суммы и времени перевода.
2. Регистрация пользователя в системе. Должны быть заполнены поля: Имя, Фамилия, Дата рождения, Телефон, Город, Адрес. Система должна присваивать уникальный ID, генерируя его. При регистрации пользователя должна производиться проверка на несовпадение ID. Результаты регистрации с датой регистрации должны выводиться в JSP. При совпадении имени и фамилии регистрация не должна производиться.
3. Телефонный справочник. Таблица должна содержать Фамилию, Адрес, Номер телефона. Поиск должен производиться по части фамилии или по части номера. Результаты должны выводиться вместе с датой выполнения в JSP.
4. Управление складом. Заполняются поля Товар и Количество. Система выводит промежуточную информацию о существующем количестве товара и запрашивает подтверждение на добавление. При отсутствии такого товара на складе добавляется новая запись.
5. Словарь. Ввод слова. Системой производится поиск одного или нескольких совпадений и осуществляется вывод результатов в JSP. Перевод может осуществляться в обе стороны. Одному слову может соответствовать несколько значений.
6. Каталог библиотеки. Выдается список книг, наличествующих в библиотеке. Запрос на заказ отправляется пометкой требуемой книги. Система проверяет в БД, свободна книга или занята. В случае занятости возвращается информация о сроках возвращения книги в фонд.

7. Голосование. Выводится вопрос и варианты ответа. Пользователь имеет возможность проголосовать и просмотреть результаты голосования по данному вопросу. БД должна хранить дату и время каждого голосования и выводить при необходимости соответствующую статистику по датам подачи голоса.

Вариант В

Для заданий варианта В главы 4 создать распределенное приложение, использующее страницы JSP на стороне клиента, сервлет в качестве контроллера и БД для хранения информации.

Тестовые задания к главе 19

Вопрос 19.1.

Как правильно объявить и проинициализировать переменную **j** типа **int** в тексте JSP?

1) `<%! int j = 1 %>;`
2) `<%@ int j = 2 %>;`
3) `<%! int j = 3; %>;`
4) `<%= int j = 4 %>;`
5) `<%= int j = 5; %>.`

Вопрос 19.2.

Какие из перечисленных переменных можно использовать в выражениях и скриптлетах JSP без предварительного объявления?

1) `error;`
2) `page;`
3) `this;`
4) `exception;`
5) `context.`

Вопрос 19.3.

Какой из следующих интерфейсов объявляет метод **_jspService()**?

1) `javax.servlet.jsp.Jsp;`
2) `javax.servlet.jsp.JspServlet;`
3) `javax.servlet.jsp.JspPage;`
4) `javax.servlet.jsp.HttpJspPage;`
5) `javax.servlet.jsp.HttpJspServlet.`

Вопрос 19.4.

Тег **jsp:useBean** объявлен как

```
<jsp:useBean   id="appJsp"
                   class="main.ApplicationJSP"
                   scope="application" />
```

В объекте какого типа должен быть сохранен созданный экземпляр?

1) `ServletConfig;`
2) `HttpApplication;`
3) `ServletContext;`
4) `ServletConfig;`
5) `ApplicationContext.`

Вопрос 19.5.

Какой тег JSP используется для извлечения значения поля экземпляра JavaBean в виде строки?

1) `jsp:useBean.toString;`
2) `jsp:param.property;`
3) `jsp:propertyType;`
4) `jsp:getProperty;`
5) `jsp:propertyToString;`

Глава 20
ПОЛЬЗОВАТЕЛЬСКИЕ ТЕГИ

Одной из основных причин использования пользовательских тегов можно назвать упрощение жизни Web-дизайнерам, которым гораздо привычнее использовать теги, а не код на языке Java.

Для создания пользовательских тегов необходимо определить класс обработчика тега, определяющий поведение тега, а также дескрипторный файл библиотеки тегов (файл **.tld**), в которой описываются один или несколько тегов, устанавливающий соответствия между именами XML-элементов и реализацией тегов. Для того чтобы библиотека была доступна для данного проекта, необходимо зарегистрировать ее URI в данном приложении.

При определении нового тега создается класс Java, который должен реализовывать интерфейс **javax.servlet.jsp.tagext.Tag**. В этом случае наследуется класс **TagSupport** или **BodyTagSupport** (для тегов без тела и с телом соответственно). Эти классы реализуют интерфейс **Tag** и содержат стандартные функции, необходимые для базовых тегов. Поскольку нужно использовать и другие классы, класс для тега должен также импортировать классы из пакетов **javax.servlet.jsp** и, если необходима передача информации в поток вывода, то **java.io**.

Простой тег

Все, что требуется сделать для тега без атрибутов или тела, – это переопределить метод **doStartTag()**, определяющий код, который вызывается во время запроса, если обнаруживается начальный элемент тега.

В качестве примера можно привести следующий класс.

```
/ пример # 1 : простейший тег без тела и атрибутов :
GetInfoTag.java */
package test.mytag;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
import java.io.IOException;
// класс бизнес-логики (см. пример #7 текущей главы)
import test.my.Rows;
public class GetInfoTag extends TagSupport {
     public int doStartTag() throws JspException {
//получение информации, передаваемой на страницу
```

```
int info = Integer.parseInt(new Rows().getRows());
   String str = "var value =<B>( " + info + " )</B>";
           try {
                 pageContext.getOut().write(str);
           } catch (IOException e) {
                 throw
           new JspException(e.getMessage());
           }
           return SKIP_BODY;
     }
}
```

Если в теге отсутствует тело, метод **doStartTag()** должен возвратить константу **SKIP_BODY**, дающую указание системе игнорировать любое содержимое между начальными и конечными элементами создаваемого тега.

Метод **write()** класса **JspWriter** (объект которого возвращает метод **getOut()**) выводит на страницу содержимое объекта **str**. Объект **pageContext** класса **PageContext** есть поле, унаследованное от класса **TagSupport**, обладающее доступом ко всей области имен, ассоциированной со страницей JSP. С помощью методов этого объекта можно получить:

getRequest() – объект запроса;
getResponse() – объект ответа;
getServletContext() – объект **ServletContext**;
getServletConfig() – объект конфигурации сервлета;
getSession() – объект сессии;
ErrorData getErrorData() – информацию об ошибках;

а также:

– с помощью метода **forward(String relativeUrlPath)** сделать перенаправление на другую страницу или action-класс;

– с помощью метода **include()** включить в поток выполнения текущие ресурсы **ServletRequest** или **ServletResponse**, определяемый относительным адресом.

Следующей задачей является идентификация для сервера и связывание его с конкретным именем XML-тега. Эта задача выполняется в формате XML с помощью дескрипторного файла библиотеки тега.

Файл дескриптора пользовательских тегов должен содержать корневой элемент **<taglib>**, содержащий список описаний тегов в элементах **<tag>**. Каждый из элементов определяет имя тега, под которым к нему можно обращаться на странице JSP, и идентифицирует класс, который обрабатывает тег. Для идентификации используется полное имя класса, например: **test.mytag.GetInfoTag**. Также в данном примере присутствует стандартный заголовок XML-файла с указанием версии и адреса ресурса DTD, который определяет допустимый формат тега **<taglib>**.

Перед списком тегов, сразу после открывающего тега **<taglib>,** указываются следующие параметры:

- **tlibversion** – версия пользовательской библиотеки тегов;
- **jspversion** – версия стандарта JSP, необходимая для данной библиотеки;
- **shortname** – краткое имя. В качестве него принято указывать рекомендуемое сокращение для использования в JSP-страницах;
- **uri** – уникальный идентификатор ресурса, идентифицирующий данную библиотеку. Параметр необязательный, но если его не указать, то необходимо регистрировать библиотеку в каждом новом приложении через файл **web.xml**;
- **info** – указывается область применения данной библиотеки.

В элементе **tag** между его начальным **<tag>** и конечным **</tag>** тегами должны находиться четыре составляющих элемента:

- **name** – тело этого элемента определяет имя базового тега, к которому будет присоединяться префикс директивы **taglib**;
- **tagclass** – полное имя класса – обработчика тега;
- **info** – краткое описание тега;
- **bogycontent** – имеет значение **empty**, если теги не имеют тела. Теги с телом, содержимое которого может интерпретироваться как обычный JSP-код, используют значение **jsp**, а редко используемые теги, тела которых полностью обрабатывают, используют значение **tagdependent**.

Вся эта информация помещается в файл **mytaglib.tld**, который имеет вид:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems,
Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-
jsptaglibrary_1_1.dtd">
<taglib>
     <tlibversion>1.0</tlibversion>
     <jspversion>1.1</jspversion>
     <shortname>mytag</shortname>
     <uri>/WEB-INF/mytaglib.tld</uri>
     <info>my library tag</info>
     <tag>
          <name>getinfo</name>
          <tagclass>test.mytag.GetInfoTag</tagclass>
          <bodycontent>empty</bodycontent>
```

```
            <info>First Example TAG</info>
      </tag>
</taglib>
```

Зарегистрировать URI библиотеки пользовательских тегов для приложения можно двумя способами:

1. Указать доступ к ней в файле **web.xml**.

   Библиотека **mytaglib.tld** должна быть зарегистрирована в файле **web.xml** в виде:

```
   <taglib>
      <taglib-uri>/WEB-INF/mytaglib.tld</taglib-uri>
      <taglib-location>/WEB-INF/mytaglib.tld
</taglib-location>
   </taglib>
```

2. Прописать URI библиотеки в файле-описании (**.tld**) библиотеки и поместить этот файл в папку **/WEB-INF** проекта. В таком случае в файле **web.xml** ничего прописывать не требуется. Преимуществом данного способа является то, что так можно использовать библиотеку во многих приложениях под одним и тем же URI.

Непосредственное использование в странице JSP созданного и зарегистрированного простейшего тега выглядит следующим образом:

```
<!-- пример # 2 : вызов простого тега : DemoTag1.jsp
-->
<HTML><HEAD>
<%@ taglib uri="/WEB-INF/mytaglib.tld"
            prefix="mytag" %>
      </HEAD>
            <BODY>
                  <mytag:getinfo/>
            </BODY>
</HTML>
```

В результате выполнения тега клиент в браузере получит следующую информацию:

var value = (3)

Тег с атрибутами

Тег может принимать параметры и передавать их для обработки в соответствующий ему класс. Для этого при описании тега в **.tld** файле используются атрибуты, которые должны объявляться внутри элемента **tag** с помощью элемента **attribute**. Внутри элемента **attribute** между тегами **<attribute>** и **</attribute>** должны находиться следующие обязательные вложенные элементы:

- **name** – имя атрибута;
- **required** – указывает на то, всегда ли должен присутствовать данный атрибут при использовании тега, который принимает значение **true** или **false**;
- **rtexprvalue** (необязательный) – показывает, может ли значение атрибута быть JSP-выражением вида **<%=**expression**%>** (значение **true**) или оно должно задаваться строкой данных (значение **false**). По умолчанию устанавливается **false**, поэтому этот элемент обычно опускается, если не требуется задавать значения атрибутов во время запроса.

Соответственно для каждого из атрибутов тега класс, его реализующий, должен содержать метод **set**ИмяАтрибута**()**, как описано в спецификации JavaBeans.

В следующем примере рассматривается простейший тег с атрибутом **name**, который выводит пользователю сообщение:

```
// пример # 3 : тег с атрибутом : HelloTag.java
package test.mytag;
import javax.servlet.jsp.tagext.TagSupport;
import java.io.IOException;
 public class HelloTag extends TagSupport {
    private Object name;
     public void setName(Object name) {
        this.name = name;
    }
     public int doStartTag(){
        try {
        pageContext.getOut().write("Hello, " + name);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return SKIP_BODY;
    }
}
```

В файл **mytaglib.tld** должна быть помещена следующая информация о теге:

```
 <tag>
    <name>hello</name>
    <tagclass>test.mytag.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Hello tag with name attribute</info>
    <attribute>
```

```
            <name>name</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>
```

Следует обратить внимание на то, что метод **void setName(Object ob)** для атрибута **name** принимает **Object** в качестве параметра, потому что элемент **rtexpvalue** установлен в **true**. Если бы он был **false**, то в этом случае метод **set**Параметр**()** должен принимать объект типа **String** в качестве параметра.

Использовать созданный тег в файле **DemoTag2.jsp** можно следующим образом:

```
<!--пример # 4 : вызов тега с передачей ему значения
: DemoTag1.jsp -->
<HTML>
<%@taglib uri="/WEB-INF/mytaglib.tld"
                prefix="mytag"%>
    <HEAD>
    <%@ page
          language="java"
          contentType="text/html; charset=CP1251"
          pageEncoding="CP1251"
    %>
    <TITLE>DemoTag2.jsp</TITLE>
    </HEAD>
          <BODY>
    <% String str = request.getParameter("name"); %>
    <mytag:hello name= "<%= str %>" />
          </BODY>
</HTML>
```

При обращении по адресу:

```
http://localhost:8080/FirstProject/DemoTag2.jsp?name=
Бендер.
```

в браузер будет выведено:

Hello, Бендер.

Тег с телом

Как и в обычных тегах, между открывающим и закрывающим пользовательскими тегами может находиться какая-то информация, которая называется телом тега, или **body**. Тип этой информации указывается в элементе **body.** На данный момент поддерживаются следующие значения для **body**:

- **empty** – пустое тело;
- **jsp** – тело состоит из всего того, что может находиться в JSP-файле. Используется для расширения функциональности JSP-страницы;
- **tagdependent** – тело интерпретируется классом, реализующим данный тег. Используется в очень частных случаях.

Когда разрабатывается пользовательский тег с телом, то лучше наследовать класс тега от класса **BodyTagSupport**, реализующего в свою очередь интерфейс **BodyTag**. Кроме методов класса **TagSupport** (суперкласс для **BodyTagSupport**) он имеет методы, среди которых следует выделить:

void doInitBody() – вызывается один раз перед первой обработкой тела, после вызова метода **doStartTag()** и перед вызовом **doAfterBody()**;

int doAfterBody() – вызывается после каждой обработки тела. Если вернуть в нем константу **EVAL_BODY_AGAIN**, то **doAfterBody()** будет вызван еще раз. Если **SKIP_BODY** – то обработка тела будет завершена;

int doEndTag() – вызывается один раз, когда обработаны все остальные методы.

Для того чтобы тело было обработано, метод **doStartTag()** должен вернуть **EVAL_BODY_INCLUDE** или **EVAL_BODY_BUFFERED**; если будет возвращено **SKIP_BODY**, то метод **doInitBody()** не вызывается.

Для примера рассматривается следующий класс обработки тега, который получает значения атрибута **num** (в данном случае методом установки значения для атрибута **num** будет метод **setNum(String num)**) и формирует таблицу с указанным количеством строк, куда заносятся значения из тела тега:

```
// пример # 5 : тег с телом : AttrTag.java
package test.mytag;
import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class AttrTag extends BodyTagSupport {
      private int num;
      public void setNum(String num) {
      this.num = Integer.valueOf(num).intValue();
      }
      public int doStartTag() throws JspTagException {
            try {
            pageContext.getOut().write(
            "<TABLE BORDER=\"3\" WIDTH=\"100%\">");
```

```
      pageContext.getOut().write("<TR><TD>");
            } catch (IOException e) {
                  throw
            new JspTagException(e.getMessage());
            }
            return EVAL_BODY_INCLUDE;
      }
      public int doAfterBody()
             throws JspTagException {
            if (num > 1) {
                  num--;
                  try {
pageContext.getOut().write("</TR></TD><TR><TD>");
                  } catch (IOException e) {
                  throw
            new JspTagException(e.getMessage());
                  }
                  return EVAL_BODY_AGAIN;
            } else {
                  return SKIP_BODY;
            }
      }
      public int doEndTag() throws JspTagException {
            try {
            pageContext.getOut().write("</TR></TD>");
            pageContext.getOut().write("</TABLE>");
            } catch (IOException e) {
                  throw
                  new JspTagException(e.getMessage());
            }
            return SKIP_BODY;
      }
}
```

В файл **.tld** следует вставить информацию о теге в виде:

```
      <tag>
            <name>bodyattr</name>
            <tagclass>test.mytag.AttrTag</tagclass>
            <bodycontent>jsp</bodycontent>
            <info>Attribute and body tag</info>
            <attribute>
                  <name>num</name>
                  <required>false</required>
                  <rtexprvalue>true</rtexprvalue>
```

```
            </attribute>
      </tag>
```

При использовании в файле JSP тег **attr** может вызываться с параметрами и без них:

```
<!-- пример # 6 : тег с телом : DemoTag3.jsp -->
<HTML><HEAD>
<META http-equiv="Content-Type" content="text/html;
charset=Windows-1251">
      <%@ taglib uri="/WEB-INF/mytaglib.tld"
                  prefix="mytag" %>
      <%@ page import="test.my.Rows" %>
            <TITLE>Example</TITLE>
</HEAD><BODY>
      <%! Rows rw = new Rows();%>
      <mytag:bodyattr num="<%= rw.getRows()%>">
            <%= rw.getSet()%>
      </mytag:bodyattr>
      <mytag:bodyattr> Просто текст </mytag:bodyattr>
</BODY></HTML>
```

В результате запуска этой JSP клиенту будет возвращено:

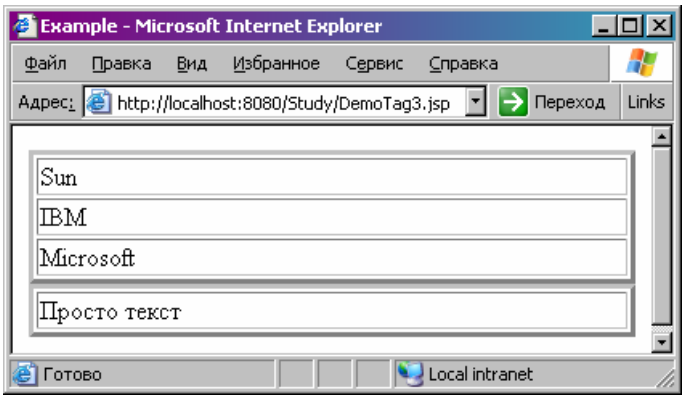

Рис. 20.1. Выполнение тега с телом

В примерах данной главы были использованы методы класса **Rows**, который приведен ниже:

```
/* пример # 7 : примитивный класс бизнес-логики :
Rows.java */
package test.my;
public class Rows extends java.util.HashSet{
      private java.util.Iterator it;
      public Rows(){
//переписать этот класс на чтение информации из БД
```

```
            this.add("Sun");
            this.add("Microsoft");
            this.add("IBM");
      }
      public String getRows() {
            it = this.iterator();
            return Integer.toString(this.size());
      }
      public String getSet(){
            return it.next().toString();
      }
}
```

В этой книге были рассмотрены основные возможности языка Java применительно к созданию распределенных приложений. Ряд дополнительных возможностей остался вне поля зрения, поэтому читайте спецификации, которые можно загрузить по адресу:

```
java.sun.com/products/servlet/index.jsp
java.sun.com/products/jsp/index.jsp
java.sun.com/products/jdbc/index.jsp
java.sun.com/j2ee/1.4/index.jsp
```

Задания к главе 20

Вариант А

Создать классы пользовательских тегов, формирующих нужное количество элементов (строк, ячеек и др.) для размещения результатов выполнения запроса.

1. Элемент массива называют локальным максимумом, если у него нет соседа большего, чем он сам. Аналогично определяется локальный минимум. Определить количество локальных максимумов и локальных минимумов в заданном строкой массиве чисел. Массив задает клиент. Возвратить все максимумы и минимумы пользователю.
2. В неубывающей последовательности, заданной клиентом, найти количество различных элементов и количество элементов, меньших, чем заданное число, и вернуть ему результат.
3. Дана числовая последовательность a_1, a_2,…, a_n . Вычислить суммы вида $S_i = a_i+a_{i+1} +…+ a_j$ для всех $1 \le i \le j \le N$ и среди этих сумм определить максимальную. Последовательность и число N задает клиент.
4. Точка A и некоторое конечное множество точек в пространстве заданы своими координатами и хранятся в базе данных. Найти N точек из множества, ближайших к точке A. Число N задает клиент.

5. В базе данных хранится список студентов и их оценок по предметам за сессию по 100-балльной системе. Выбрать без повторений все оценки и соответствующие им записи, встречающиеся более одного раза.
6. Получить упорядоченный по возрастанию массив C, состоящий из k элементов, путем слияния упорядоченных по возрастанию массивов A и B, содержащих n и m элементов соответственно, k = n + m. Элементы массивов хранятся в базе данных, а значения n и m задает клиент.
7. В матрице A найти сумму элементов, расположенных в строках с отрицательным элементом на главной диагонали, и произведение элементов, расположенных в строках с положительным элементом в первом столбце. Матрица размерности n хранится в базе данных. Клиент задает размерность m < n матрицы, для которой будет произведен расчет.
8. В программе, хранящейся в текстовом файле, удалить строки с № 1 до № 2, где № 1 и № 2 вводятся клиентом. Удаляемые строки возвратить клиенту. Предусмотреть случаи, когда, например, № 1 меньше номера первой строки, № 1 = № 2, № 2 больше номера последней строки, и другие исключительные ситуации.
9. После n-ой строки программы, которая хранится в файле, вставить m строк. Числа n, m и вставляемые строки вводятся пользователем. Новый набор данных сохранить на диске и вернуть клиенту.
10. В БД хранятся координаты множества m точек трехмерного пространства. Найти такую точку, чтобы шар заданного радиуса с центром в этой точке содержал максимальное число точек. Координаты найденных точек возвратить клиенту.
11. Из заданного множества точек на плоскости, координаты которых хранятся в базе данных, выбрать две различные точки так, чтобы окружности заданного пользователем радиуса с центрами в этих точках содержали внутри себя одинаковое количество заданных точек. Полученные множества возвратить клиенту.
12. В базе данных хранятся координаты конечного множества точек плоскости. Пользователем вводятся координаты центра и радиусы 5 концентрических окружностей. Между какими окружностями (1 и 2, 2 и 3, ..., 4 и 5) больше всего точек заданного множества? Полученное множество точек возвратить клиенту.
13. В базе данных хранятся координаты вершин выпуклых четырехугольников на плоскости. Сформировать ответ клиенту, содержащий координаты всех вершин трапеций, которые можно сформировать из данных точек.

14. В базе данных хранятся координаты вершин треугольников на плоскости. Для прямоугольных треугольников возвратить клиенту координаты вершин прямого угла, площадь и координаты вершин (одной или двух), ближайших к оси OX.
15. В базе данных хранятся координаты множества точек плоскости A и коэффициенты уравнений множества прямых в этой же плоскости. Передать клиенту набор из пар различных точек таких, что проходящая через них прямая параллельна прямой из множества B.

Вариант B

Для заданий варианта B предыдущей главы использовать пользовательские теги для визуализации работы приложения.

Тестовые задания к главе 20

Вопрос 20.1.

Какой элемент тега **<attribute>** определяет имя атрибута, которое должно быть передано обработчику тегов?

1) <attribute-name>;
2) <name>;
3) <attributename>;
4) <param-name>.

Вопрос 20.2.

Обработчик тега реализует интерфейс **BodyTag**. Сколько раз может быть в нем вызван метод **doAfterBody()**?

1) класс **BodyTag** не поддерживает метод **doAfterBody()**;
2) 0;
3) 1;
4) 0 или 1;
5) сколько угодно раз.

Вопрос 20.3.

Какой метод обработчика тега будет вызван, если метод **doStartTag()** вернет значение **Tag.SKIP_BODY**?

1) doAfterBody();
2) doBody();
3) skipBody();
4) doEndTag();.
5) нет правильного.

Вопрос 20.4.

Какой из следующих элементов необходим для корректности тега **<taglib>** в файле **web.xml**?

1) <uri-tag>;
2) <tag-uri>;
3) <uri-name>;
4) <uri-location>;
5) <taglib-uri>.

Вопрос 20.5.

Какие элементы описывают характеристики пользовательского тега в файле **.tld**?

1) value;
2) name;
3) rtexprvalue;
4) class.

Вопрос 20.6.

Какие утверждения верны относительно метода **doInitBody()** класса **BodyTagSupport**?

1) используется контейнером и не может быть переопределен;
2) он может быть переопределен;
3) может возвращать или константы **SKIP_BODY**, или **EVAL_BODY_INCLUDE;**
4) его возвращаемое значение имеет тип **void**.

Вопрос 20.7.

Что нужно сделать в файле **.tld** для этого тега, чтобы в теле тега использовать скриптлеты?

1) в **body-content** должно быть выставлено значение **jsp;**
2) в **script-enabled** должно быть выставлено **true;**
3) ничего, так как скриптлеты используются по умолчанию.

УКАЗАНИЯ И ОТВЕТЫ

Глава 1

Вопрос 1.1.

Для того чтобы быть запускаемым приложением, класс может быть объявлен как **`public`**, так и friendly (без спецификатора). Метод **`main()`** не может быть объявлен как **`private`**, так как в этом случае он будет недоступен для выполнения. Для запуска приложения метод **`main()`** должен получить в качестве параметра массив строк, а не строку, иначе он будет являться просто перегруженным методом.

Ответ: 2, 3.

Вопрос 1.2.

Слово **`goto`** в Java не может быть использовано даже в качестве идентификатора, но оно является резервированным словом так же, как и **`const`**. Выход на метку осуществляется с помощью **`break`** или **`continue`**.

Методы создаются только в классах. Множественное, как и циклическое, наследование запрещено.

Ответ: 4, 5.

Вопрос 1.3.

Если переменная объявлена в методе, то до своего использования она должна быть проинициализирована, иначе компилятор сообщает об ошибке.

При инициализированном значении переменной **`args`** в методе **`print()`** ошибка не возникает, так как оператор сложения инициирует действия по преобразованию в строку всех участвующих в конкатенации объектов.

Параметр **`String[] args`** в **`main()`** – просто соглашение, и может быть использован любой приемлемый идентификатор для массива строк.

Ответ: 2.

Вопрос 1.4.

Все методы, производящие чтение из потока, являются потенциальными источниками возникновения ошибок ввода (**`IOException`**), которые должны быть обработаны в случае их появления.

Длина считываемых данных может не совпадать с длиной объявленного массива. Лишние данные будут просто утеряны.

Метод **read(byte[] b)** помещает считываемую информацию в массив, передаваемый ему в качестве параметра.

Ответ: 2.

Вопрос 1.5.

Методы **main()** являются корректно перегруженными, поэтому никаких ошибок не произойдет, а будет вызван только метод, запускающий приложение.

Ответ: 4.

Ответы:

1.1. 2), 3)
1.2. 4), 5)
1.3. 2)
1.4. 2)
1.5. 4)

Глава 2

Вопрос 2.1.

Строка **float f=7.0;** не скомпилируется, поскольку встроенный тип числа компонента с плавающей точкой – double. Следует компилировать

float f=(float) 7.0 или **float f=7.0f**;

Строка **char c="z"**; не скомпилируется, поскольку **char** должен определяться одинарными кавычками. Следует компилировать

char c='z';

Строка **byte b=255**; не скомпилируется, поскольку байт может определить числа между **–128 и +127**;

значение типа **boolean** может быть только **true** или **false**.

Ответ: 5, 6.

Вопрос 2.2.

Условие может находиться только в круглых скобках, после которых может располагаться один оператор либо блок кода, заключенный в фигурные скобки.

Ответ: 1,5.

Вопрос 2.4.

Идентификатор может начинаться с буквы, или знака доллара '**$**', или подчеркивания '**_**'. Идентификатор не может начинаться с других символов, как, например, цифра или '**#**', причем последний не может находиться и внутри него.

Ответ: 3, 4, 5.

Вопрос 2.5.

Метод **floor()** возвращает ближайшее к аргументу –1.51 меньшее значение, а именно –2.0. Метод **round()** возвращает округленное целое значение аргумента, а именно –2.0. Метод **ceil()** возвращает ближайшее к аргументу большее значение, а именно –1.0. Методы **min()** и **max()** должны иметь два параметра, поэтому такая запись приведет к ошибке компиляции.

Ответ: 3.

Вопрос 2.6.

Индексация массивов начинается с индекса 0, соответственно в данном случае элемент с индексом 1 будет иметь значение 2.

Ответ: 2.

Вопрос 2.7.

Объявление **a1** корректно, длина такого массива равна нулю. Объявление **a5** некорректно, так как одновременно задано количество элементов для выделения памяти и определяются сами элементы, что является избыточной информацией.

Ответ: 1, 2, 4.

Вопрос 2.8.

Ошибка компиляции не возникнет, так как тип **char** преобразуется к большему по размеру типу **int**. Значение локальной переменной **i** не изменится, так как она передается в метод по значению. В то же время метод может изменять значения переменных класса.

Ответ: 3.

Ответы:

2.1. 5), 6)
2.2. 1), 5)
2.3. 1), 3), 5)
2.4. 3), 4), 5)
2.5. 3)
2.6. 2)
2.7. 1), 2), 4)
2.8. 3)

Глава 3

Вопрос 3.1.

Поля класса всегда инициализированы. Если инициализация не произведена явно, то поле инициализируется значением по умолчанию, предусмотренному для базового типа или значением **null** для объектного типа.

Ответ: 4.

Вопрос 3.2.

Конструктор может быть объявлен только со спецификаторами **public**, **private**, **protected** или без спецификатора. В случае использования **private** и **protected** объект класса может быть создан с помощью статического метода класса, который вызывает такой конструктор.

Применение **final** или **abstract** при объявлении конструктора не имеет смысла, потому что он не участвует в наследовании.

Спецификатор **volatile** применяется только к полям классов, используемых потоками (**Thread**).

Применение **native** с конструктором не предусматривается спецификацией языка.

Ответ: 1, 5.

Вопрос 3.3.

Метод или логический блок могут вызывать конструктор только с помощью оператора **new**.

Ответ: 3.

Вопрос 3.4.

Статический метод может быть вызван из нестатического, обратное неверно, так как статический метод может быть вызван без создания экземпляра класса и имеет доступ только к статическим полям и методам класса. Использование спецификатора **private** не ограничивает видимость поля или метода внутри объявления класса. Ничто не мешает методу быть объявленным **final static**.

Ответ: 1.

Вопрос 3.5.

При запуске приложения выполняются только статические логические блоки класса приложения и тело метода **main()**.

Ответ: 4.

Ответы:

3.1. 4)
3.2. 1), 5)
3.3. 3)
3.4. 1)
3.5. 4)

Глава 4

Вопрос 4.1.

Во время выполнения при попытке преобразования с сужением типа будет сгенерировано исключение **ClassCastException**.

Ответ: 3.

Вопрос 4.2.

Класс **Object** при наследовании может быть указан явно только в том случае, если класс не наследует другой класс. В данной ситуации для корректности кода необходимо убрать один из двух классов после **extends**, причем неважно какой.

Ответ: 3.

Вопрос 4.3.

Компилятор создаст конструктор по умолчанию для класса **B**, который для создания объекта попытается вызвать несуществующий конструктор без параметров класса **A**. В итоге будет сгенерирована ошибка компиляции в строке 2.

Ответ: 2, 4.

Вопрос 4.4.

В одном файле не может быть двух **public** классов.

Ответ: 1.

Вопрос 4.5.

Методы объявленные как **private**, не наследуются, поэтому на них не распространяются принципы полиморфизма. Так что метод с такой же сигнатурой, объявленный в подклассе, не имеет никакой связи с методом из суперкласса. В таком случае при вызове через ссылку на суперкласс происходит попытка вызвать его **private**-метод, что приводит к ошибке компиляции.

Ответ: 1.

Вопрос 4.6.

В строке 1 ошибки не будет, так как происходит безопасное преобразование вверх. Во второй строке вызывается полиморфный метод. Ошибка компиляции произойдет при попытке вызова метода, принадлежащего только подклассу, через ссылку на суперкласс, через которую он просто недоступен.

Ответ: 5.

Вопрос 4.7.

Вместо первого и третьего комментариев вызовы конструкторов ставить нельзя, так как явный вызов конструктора может осуществляться только с помощью **super()** или **this()** соответственно только из конструкторов подкласса или текущего класса.

Ответ: 3.

Ответы:

4.1. 3)
4.2. 3)
4.3. 2), 4)
4.4. 1)
4.5. 1)
4.6. 5)
4.7. 3)

Глава 5

Вопрос 5.1.

Объявление пакета должно предшествовать любому другому коду, причем оно должно быть единственным. Комментарии могут находиться везде.

Ответ: 2, 3.

Вопрос 5.2.

Код не будет откомпилирован, так как подкласс наследует абстрактный класс и при этом не реализует его абстрактный метод и сам не объявлен как абстрактный.

Ответ: 3.

Вопрос 5.3.

Интерфейсы не могут включать реализованные методы и неинициализированные поля. Все поля интерфейса трактуются как константы. Абстрактный метод не может быть статическим из-за нарушения принципов полиморфизма, также он не может быть **`protected`** и **`private`** из-за того, что не может быть использован и переопределен. Атрибуты по умолчанию перед полями и методами интерфейса можно записать в явном виде.

Ответ: 2, 4.

Вопрос 5.4.

Методы, объявленные как **`public`** и **`protected`**, могут быть переопределены в классе другого пакета, наследующего данный класс. Методы **`private`** и без спецификатора доступа в этом случае недоступны и поэтому попытка их переопределения вызывает ошибку компиляции. Причем вложенность одного пакета в другой никак не влияет на видимость классов и методов.

Ответ: 4, 5.

Ответы:

5.1. 2), 3)
5.2. 3)
5.3. 2), 4)
5.4. 4), 5)
5.5. 1), 2), 5)

Глава 6

Вопрос 6.1.

Нестатические методы nested-класса не имеют доступа к нестатическим полям и методам своего внешнего класса.

Ответ: 5.

Вопрос 6.2.

Ошибки компиляции не возникнет, так как анонимный класс реализует абстрактный метод абстрактного класса. При первом вызове метода **show()** будет возвращено значение 2, но значение поля **i** после этого будет увеличено на единицу. При втором вызове метода поле **i** сначала будет увеличено на единицу, а уже потом сработает оператор **return**, и таким образом метод возвратит значение 4.

Ответ: 2.

Вопрос 6.3.

Объявить объект внутреннего (нестатического) класса можно, только предварительно создав объект внешнего класса. Конструкторы обоих классов должны вызываться так же, как и для всех других классов, т.е. с помощью оператора **new**.

Ответ: 4.

Вопрос 6.4.

В результате выполнения кода **Owner ob=new Owner()** будет создан объект **Owner**. Его метод **meth()** создаст объект типа **Inner** в результате выполнения кода **Abstract abs=ob.meth()**. При его выполнении ничего выведено на консоль не будет, так как метод **meth()** класса **Inner**, выводящий на консоль строку **inner**, будет вызван только один раз командой **abs.meth()**.

Ответ: 1.

Вопрос 6.5.

В первой строке объявляется поле, во второй – метод, в третьей – внутренний класс. Все они могут иметь одинаковое имя, что не мешает компилятору различать их.

Ответ: 4.

Ответы:

6.1. 5)
6.2. 2)
6.3. 4)
6.4. 1)
6.5. 4)

Глава 7

Вопрос 7.1.

Метод **substring(i,j)** извлекает подстроку из вызывающей строки, начиная с символа в позиции **i** и заканчивая символом в позиции **j**, не включая его. Первый символ строки находится в позиции 0.

Ответ: 2.

Вопрос 7.3.

Java не допускает перегрузки оператора, как в C++, но для удобства оператор **+** переопределен для строк и преобразует объекты любых типов в его строковый эквивалент.

Ответ: 1, 2.

Вопрос 7.4.

Ошибок компиляции не возникнет, так как, во-первых, **ch** получит соответствующее коду 0H74 значение '**t**', и, во-вторых, сложение символа со строкой в результате даст строку "**tava**".

Ответ: 6.

Вопрос 7.5.

Метод **insert()** вставляет строку в указанную позицию вызывающего объекта класса **StringBuffer** и сохраняет в нем изменения.

Ответ: 1.

Ответы:

7.1. 2)
7.2. 3)
7.3. 1), 2)
7.4. 6)
7.5. 1)

Глава 8

Вопрос 8.1.

Методы класса **File** могут создавать, удалять, изменять имя каталога, но изменять корневой каталог можно только через переменные окружения.

Вопрос 8.3.

Методы класса **File** могут создавать файл, удалять его, изменять его имя, но к информации, содержащейся в файле, доступа не имеют.

Вопрос 8.5.

Использование **transient** указывает на отказ от сохранения значения помеченного поля объекта при записи объекта в поток.

Ответ: 3.

Ответы:

8.1. 4)
8.2. 2)
8.3. 2), 3)
8.4. 4)
8.5. 3)

Глава 9

Вопрос 9.1.

Блок try может завершаться инструкцией **`catch`** или **`finally`**. В данном случае во избежание ошибки компиляции необходимо поставить инструкцию **`catch(java.io.IOException e)`**, т.к. метод **`write()`** способен генерировать исключение, которое сам не обрабатывает. Метод **`inc()`** возвращает значение, поэтому необходимо завершить код метода инструкцией **`return counter`**. Так как в вопросе предлагалось выбрать два правильных ответа, то возможное добавление в код инструкции **`finally`** не представляется возможным.

Ответ: 2, 5.

Вопрос 9.2.

При вызове метода **`meth()`** с параметром 5 переменная *y* последовательно будет принимать следующие значения: в строке 1 в десятичной системе счисления будет 8; в строке 2 будет значение 13 (8 + 5); строка 3 генерирует исключение, поэтому строка 4 будет пропущена, а в результате выполнения инструкции **`catch`** значение будет уменьшено на единицу. Если бы в строке 3 исключение генерировалось без использования оператора **`if`**, то возникла бы ошибка компиляции из-за принципиальной невозможности выполнения строки 4.

Ответ: 1.

Вопрос 9.3.

При генерации исключения последовательно выполняются блоки **`catch`** и **`finally`**, причем возвращаемое методом значение переменной **`count`** будет взято из инструкции return блока **`finally`**.

Ответ: 4.

Вопрос 9.4.

Варианты 1 и 4 не скомпилируются, т.к. они включают классы **`IOException`** и **`Exception`**, не обработанные в базовом классе. Поскольку в случаях 2 и 3 в качестве параметра выступают типы **`long`** и **`short`**, то эти методы являются перегруженными, для которых таких ограничений не существует.

Ответ: 2, 3.

Вопрос 9.5.

При подстановке варианта 3 будет скомпилирована ошибка, т.к. **`IOException`** является проверяемым исключением и в блоке **`try`** должна быть предусмотрена возможность его генерации. При использовании варианта 4 ошибка компиляции возникнет вследствие того, что исключе-

нию типа **Exception** не предоставлен соответствующий блок **catch**. Вариант 2 содержит синтаксическую ошибку.

Ответ: 1.

Ответы:

9.1. 2), 5)
9.2. 1)
9.3. 4)
9.4. 2), 3)
9.5. 1)

Глава 10

Вопрос 10.1.

Интерфейсы **List**, **Vector** допускают наличие одинаковых элементов в своих реализациях. Коллекция **Map** запрещает наличие одинаковых ключей, но не значений. Множество **Set** по определению не может содержать одинаковых элементов.

Ответ: 1.

Вопрос 10.2.

Объект, не являющийся коллекцией, может быть добавлен в коллекцию только при помощи метода **add()**. Класс **ArrayList** содержит конструкторы вида **ArrayList()**, **ArrayList(int capacity)** и **ArrayList(Collection c)**. Интерфейс **List** конструктора не имеет по определению.

Ответ: 1, 4.

Вопрос 10.3.

Класс **Hashtable** реализует интерфейс **Map** и наследует абстрактный класс **AbstractMap**.

Ответ: 5.

Вопрос 10.4.

Класс **HashSet** реализует интерфейс **Set**. Интерфейс **SortedSet** реализует класс **TreeSet**. Проверка **instanceof** проводится не по ссылке, а по объекту.

Ответ: 1.

Вопрос 10.5.

Stack, **HashMap** и **HashSet** являются классами, а **AbstractMap** – абстрактный класс. Интерфейсами являются **SortedSet** и **SortedMap**.

Ответ: 1, 4.

Ответы:

10.1. 1)
10.2. 1), 4)
10.3. 5)
10.4. 1)
10.5. 1), 4)

Глава 11

Вопрос 11.1.

Правильным вариантом является следующий код:

```
int i =
new Integer(getParameter("count")).intValue();
```

Метод **getParameter()** извлекает из формы значение параметра **count** в виде строки, которая используется для инициализации объекта класса **Integer**. Метод **intValue()** используется для преобразования к базовому типу.

Ответ: 1.

Вопрос 11.2.

Для того чтобы изменения цвета фона стали видны пользователю, требуется перерисовка всего апплета вызовом метода **paint()**. Это действие можно выполнить, вызвав методы **repaint()** или **update()**.

Ответ: 4.

Вопрос 11.5.

Объекты из пакета AWT могут объявляться и вызывать свои методы в любых приложениях.

Ответ: 2.

Ответы:

11.1. 1)
11.2. 4)
11.3. 5), 6)
11.4. 1), 4)
11.5. 2)

Глава 12

Вопрос 12.1.

Чтобы класс был апплетом, достаточно, чтобы его суперклассом был класс **Applet**. Переопределение методов производится при необходимости закрепления за апплетом каких-либо действий.

Ответ: 2.

Вопрос 12.3.

Попытка компилировать данный код приведет к ошибке вследствие того, что часть методов интерфейса **`WindowListener`** не реализована в классе **`Quest3`**.

Ответ: 1.

Ответы:

12.1. 2)
12.2. 1)
12.3. 1)
12.4. 1), 5)
12.5. 1), 2)

Глава 13

Вопрос 13.2.

По умолчанию фреймы используют менеджер размещений **`BorderLayout`**, поэтому если при добавлении элемента на фрейм не было указано его месторасположение, то элемент займет весь фрейм. Следующий добавленный таким же образом элемент будет прорисован поверх предыдущего.

Ответ: 3.

Вопрос 13.4.

Команда **`add(b)`**, вызванная во второй раз, пытается добавить на апплет уже существующий там объект. Команда **`add(new Button("NO"))`** каждый раз добавляет новый объект.

Ответ: 2.

Вопрос 13.5.

Метод всегда вызывается объектом, который необходимо зарегистрировать. В качестве параметра должен передаваться объект приложения или апплета, в котором размещается данный компонент.

Ответ: 2.

Ответы:

13.1 2)
13.2 3)
13.3 3)
13.4 2)
13.5 2)

Глава 14

Вопрос 14.1.

Объект потока создается только после вызова конструктора класса **`Thread`** или его подкласса, но к ошибке компиляции создание такого объ-

екта, как в примере, не приведет. Поток всегда запускается вызовом метода **start()**. Результатом же вызова метода **run()** будет выполнение кода метода **run(),** никак не связанное с потоком. В данной ситуации ошибка компиляции произойдет из-за того, что сигнатура метода **run()** в интерфейсе **Runnable** не совпадает с его реализацией в классе **Q**, т.е. метод не реализован и класс **Q** должен быть объявлен как **abstract**.

Вопрос 14.2.

Поток **t1** не входит ни в одну группу, поэтому его приоритет останется неизменным, т.е. 7. Вызов метода **setMaxPriority()** для группы потоков с параметром 8 большим, чем 5, приведет к тому, что приоритет группы потоков, а следовательно, и потока **t2** будет установлен как **NORMAL_PRIORITY**.

Ответ: 1.

Вопрос 14.3.

Поток **t1** не может быть создан, т.к. класс **T1** не имеет метода **start()**, но создать его можно, например, командой

```
Thread t1 = new Thread(new T1());
```

Объект **t2** не может быть создан, т.к. у класса **T2** нет конструктора, способного принимать параметры.

Ответ: 3, 4.

Вопрос 14.4.

Методы **sleep()**, **wait()** приводят к временной остановке и переходу в состояние "неработоспособный". Методы **notify()** и **notifyAll()** не имеют отношения к изменению состояния потоков, они лишь уведомляют другие потоки о снятии изоляции с синхронизированных ресурсов. Метод **stop()** и завершение выполнения метода **run()** приводят поток в состояние "пассивный", из которого запуск потока с тем же именем возможен только после инициализации ссылки.

Ответ: 2, 3.

Вопрос 14.5.

При запуске приложения будет создано два потока **r** и **t**, но стартует только второй. Поток **t** инициализирован с использованием ссылки на первый поток. Это обстоятельство в данном контексте не оказывает влияния на выполнение второго потока. В итоге метод **run()** будет вызван только один раз.

Ответ: 3.

Ответы:

14.1. 4)
14.2. 1)
14.3. 3), 4)
14.4. 2), 3)
14.5. 3)

Глава 15

Вопрос 15.1.

Класс **`Socket`** поддерживает TCP-соединения. Порт 23 зарезервирован для протокола Telnet, являющегося подчиненным протоколом TCP/IP. Для UDP-соединений существует класс **`DatagramSocket`**.

Ответ: 3.

Вопрос 15.2.

Для получения содержимого страницы сначала создается объект URL, затем вызывается метод **`getContent().`**

Ответ: 2.

Вопрос 15.4.

Соответствующий конструктор класса **`Socket`** имеет вид:

```
public Socket(String host, int port)
    throws UnknownHostException, IOException
```

Ответ: 1, 3.

Вопрос 15.5.

Команда **`p.flush()`** поместит сообщение в поток, ассоциированный с сокетом, а команда **`s.close()`** закроет сокет после обмена информацией с клиентом.

Ответ: 1, 4.

Ответы:

15.1. 3)
15.2. 2)
15.3. 1)
15.4. 1), 3)
15.5. 1), 4)

Глава 16

Вопрос 16.1.

Вызов **`getServletConfig()`**, как правило, осуществляется из метода **`init()`** и возвращает объект **`ServletConfig`**, соответствующий этому сервлету, а вызов метода **`getInitParameter(String str)`**

класса **ServletConfig** возвращает значение требуемого параметра. Объект класса **HttpServlet** также может вызвать этот метод. Параметры инициализации хранятся в XML-файле.

Ответ: 2, 3.

Вопрос 16.2.

Щелчок по ссылке посылает **GET** запрос по умолчанию, который обрабатывается методом **doGet()** сервлета. Чтобы вызвать метод **doPost()**, тип запроса нужно указывать явно.

Ответ: 1.

Вопрос 16.3.

Перед обработкой самого первого запроса контейнер сервлетов вызывает метод **init()** сервлета. После того как метод выполнился, сервлет может отвечать на запросы пользователей. При этом не имеет значения, отслеживалась сессия пользователя или нет, создавался новый поток выполнения или нет.

Ответ: 4, 5.

Вопрос 16.5.

ServletOutputStream и **ServletContextEvent** – классы пакета **javax.servlet**. **PageContext** – класс пакета **javax.servlet.jsp**. Интерфейсами указанного пакета являются **ServletRequest** и **Servlet**.

Ответ: 1, 4.

Вопрос 16.6.

Следует обратить внимание на тип тега <**input**> на форме. На самом деле форма посылает файл. Данные из файла могут быть получены в сервлете из объектов **ServletInputStream** (для бинарных файлов) или **Reader** (для текстовых файлов), извлеченных из запроса.

Ответ: 3, 4.

Ответы:

16.1. 2), 3)
16.2. 1)
16.3. 4), 5)
16.4. 3)
16.5. 1), 4)
16.6. 3), 4)

Глава 17

Вопрос 17.3.

У метода **getSession()** объекта-запроса есть две разновидности: без параметров и форма **getSession(boolean create)**. Вызов

`getSession(true)` указывает на необходимость создания объекта-сессии, если он не существует. Других способов извлечения сессии из объекта-запроса нет.

Ответ: 1, 7.

Вопрос 17.4.

Имя файла cookie передается конструктору и далее не может быть изменено. Метода **`setName(String name)`** у файла cookie не существует. В то же время значение файла передается конструктору и впоследствии может быть изменено при помощи вызова метода **`setValue(String value)`**. Браузер накладывает ограничение на размер каждого файла cookie (не более 4 Kb) и общее количество cookie (не более 20 cookie для одного Web-сервера и всего не более 300). Максимальное время существования файла cookie устанавливается с помощью метода **`setMaxAge(int expiry)`**, но значение параметра должно быть задано в секундах.

Ответ: 1, 3, 4.

Вопрос 17.5.

Конструктор **`Cookie(String name, String value)`** использует два параметра: имя файла в качестве первого, а его значение – в качестве второго. Имя не должно начинаться с символа '**`$`**' и содержать запятых, точек с запятой, пробелов. Подобные требования накладываются и на значение cookie, т.е. оно не должно содержать круглых и фигурных скобок, пробелов, знака равенства, запятых, двойных кавычек, слэшей, двоеточия, точки с запятой и т.д.

Ответ: 5, 6.

Вопрос 17.6.

Объекты, представляющие cookies, присоединяются к объекту-ответу **`HttpServletResponse req`** только при помощи метода **`addCookie()`**.

Ответ: 2.

Ответы:

17.1. 2)
17.2. 1), 2), 5)
17.3. 1), 7)
17.4. 1), 3), 4)
17.5. 5), 6)
17.6. 2)

Глава 18

Вопрос 18.1.

Объект **`DriverManager`** для установки соединения с БД использует драйвер БД и ее **`URL`**. Объект **`DataSource`** использует имя для поиска объекта.

Ответ: 1, 2.

Вопрос 18.2.

Производитель СУБД должен создать и предоставить драйвер для соединения с БД. Все драйвера должны реализовывать интерфейс **`java.sql.Driver`**.

Ответ: 1.

Вопрос 18.3.

Метод **`getMetaData()`** извлекает из установленного соединения объект **`DatabaseMetaData`**, в котором определен целый ряд методов, позволяющих получить информацию о состоянии БД и ее таблиц.

Ответ: 1.

Вопрос 18.5.

Метод **`executeUpdate()`** используется для выполнения SQL-операторов, производящих изменения в БД. Метод **`execute()`** применяется, если неизвестен тип данных, возвращаемых оператором SQL. Метод **`executeBatch()`** применяется для выполнения группы команд SQL. Метод **`executeQuery()`** возвращает результат выполнения оператора `SELECT`, упакованный в объект **`ResultSet`**.

Ответ: 2.

Ответы:

18.1. 1), 2)
18.2. 1)
18.3. 1)
18.4. 4)
18.5. 2)

Глава 19

Вопрос 19.1.

Для объявления переменных предназначен тег **`<%!  %>`**, внутри которого должен находиться компилируемый без ошибок код Java, завершаемый символом '**`;`**' или заключенный в фигурные скобки.

Ответ: 3.

Вопрос 19.2.

Неявные переменные существуют на протяжении всего жизненного цикла сервлета и ограничены только областью видимости. Переменные **this** и **page** суть одна и та же переменная, представляющая текущий экземпляр JSP. Переменная **exception** создается только при возникновении ошибки на странице и доступна только на странице обработки ошибок.

Ответ: 2, 3, 4.

Вопрос 19.4.

Созданный экземпляр будет обладать областью видимости в пределах приложения и представляет собой контейнер для исполнения JSP типа **ServletContext**.

Ответ: 3.

Вопрос 19.5.

Для получения значения свойства компонента используется действие **jsp:getProperty** в виде:

```
<jsp:getProperty name="имяКом" property="имяСв" />
```

Ответ: 4.

Ответы:

19.1. 3)
19.2. 2), 3), 4)
19.3. 4)
19.4. 3)
19.5. 4)

Глава 20

Вопрос 20.2.

Если метод **doAfterBody()** вернет значение **EVAL_BODY_TAG**, то контейнер вызовет метод еще раз. Контейнер прекратит обработку, если будет возвращено значение **SKIP_BODY**.

Ответ: 5.

Вопрос 20.3.

Если метод **doStartTag()** вернет значение **SKIP_BODY**, то это значит, что тело тега не будет обработано и должен вызваться метод, завершающий работу тега, – **doEndTag()**.

Ответ: 4.

Вопрос 20.6.

Метод имеет сигнатуру

```
public void doInitBody() throws JSPException,
```

поэтому он не возвращает конкретных значений и может быть переопределен.

Ответ: 2, 4.

Ответы:

20.1. 2)
20.2. 5)
20.3. 4)
20.4. 5)
20.5. 2), 3)
20.6. 2), 4)
20.7. 1)

ПРИЛОЖЕНИЕ 1

Язык разметки гипертекстовых документов HTML

Язык HTML (HyperText Markup Language) позволяет публиковать в Internet документы с помощью заголовков, текста, списков, таблиц, получать доступ к документам с помощью гиперссылок, включать видеоклипы, звук и т.д. Страницы JSP, привнося свои теги, активно используют уже существующие HTML-теги. Все более широко используемый формат XML также использует теги. И наоборот, можно сказать, что HTML – это XML-документ с заданной структурой тегов.

HTML-документ создается с помощью текстового редактора или специализированных HTML-редакторов и конвертеров и сохраняется в виде текстового файла с расширением **html** или **htm**. Для просмотра HTML-документов используются Web-браузеры, интерпретирующие документы. HTML-документ состоит из вложенных друг в друга элементов (тегов). Теги HTML начинаются со стартового тега (“**< >**”) и заканчиваются завершающим тегом (“**</>**”). Сам документ – это один большой элемент вида:

```
<HTML>
<!--Содержание документа-->
</HTML>
```

HTML не реагирует на регистр символов, описывающих тег.

Заголовочная часть документа <HEAD>

Тег заголовочной части документа используется сразу после тега **<HTML>**. Данный тег содержит общее описание документа. Стартовый тег **<HEAD>** помещается непосредственно перед тегом **<TITLE>** и другими тегами, описывающими документ, а завершающий тег **</HEAD>** размещается сразу после окончания описания документа. Например:

```
<HTML><HEAD>
            <TITLE> Список сотрудников </TITLE>
</HEAD></HTML>
```

Заголовок документа <TITLE>

Большинство Web-браузеров отображают заголовок документа, ограниченный тегами **<TITLE>** и **</TITLE>** вверху экрана, отдельно от содержимого документа, и в файле закладок.

Тело документа <BODY>

Тело документа должно находиться между тегами **<BODY>** и **</BODY>**. Это та часть документа, которая отображается как текстовая и графическая информация документа. Технически стартовые и завершающие теги типа **<HTML>**, **<HEAD>** и **<BODY>** необязательны. Но настоятельно рекомендуется их использовать, поскольку использование данных тегов позволяет Web-браузеру разделить заголовочную часть документа и смысловую часть.

```
<!--пример 1: простой HTML документ-->
<HTML>
<HEAD><TITLE>Домашняя страница </TITLE></HEAD>
<BODY>
<!-- Это комментарий-->
<H1>Пример заголовка, размер 1</H1>
<H2>Пример заголовка, размер 2</H2>
<H6>Пример заголовка, размер 6</H6>
<ADDRESS>Романчик - e-mail:rom@bsu.by</ADDRESS> <P>
</BODY></HTML>
```

В этом примере использовались следующие теги гипертекста: **<TITLE>** – тег, использующийся для определения заголовка (названия) документа; **<P>** – тег нового абзаца. Тег **<ADDRESS>** позволяет сформировать информацию о связи с автором документа HTML и определяет вид сообщения. Как любой язык, HTML позволяет вставлять в тело документа комментарии, которые сохраняются при передаче документа по сети, но не отображаются браузером. Синтаксис комментария:

```
<!-- Это комментарий -->
```

В тексте HTML-документа структурно выделяются собственно текст, заголовки частей текста, заголовки более высокого уровня и т.д. Самый большой заголовок обозначается цифрой 1, следующий – 2, и т.д. до шести. Синтаксис заголовка уровня 1 следующий:

```
<H1> Заголовок первого уровня </H1>
```

Заголовки других уровней могут быть представлены в общем случае так: **<Hx>**. Заголовок x-го уровня **</Hx>**, где x – цифра от 1 до 6, определяющая уровень заголовка.

В языке описания гипертекстовых документов все теги – парные (start-тег и stop-тег). В конечном теге присутствует слэш, который сообщает обозревателю о завершении тега. Существуют исключения из этого правила пар, например: тег **<P>**, определяющий абзац, не требует завершающего тега. Не все теги совместимы с обозревателями. Если браузер не понимает тега, то он его просто пропускает.

Разбиение текста на смысловые группы

В HTML-документе игнорируются символы возврата каретки и перехода к новой строке. Браузер разделяет абзацы при наличии тега **<P>**, который вставляет новую строку и осуществляет переход к следующей строке.

Дополнительные параметры тега **<P>** позволяют выравнивать абзац по левому краю, центру и правому краю соответственно:

```
<P ALIGN=left|center|right>
```

Тег **
** извещает браузер о разрыве строки. Наилучший пример использования данного тега – последовательность строк, где браузер должен отображать их одну под другой.

Дополнительный параметр **CLEAR** позволяет расширить возможности тега **
**:

```
<BR CLEAR=left|right|all>
```

Параметр **CLEAR** позволяет не просто выполнить перевод строки, но и разместить следующую строку, начиная с чистой левой **(left)**, правой **(right)** или обеих **(all)** границ окна браузера. Например, если рядом с текстом слева встречается рисунок, то можно использовать тег **
** для смещения текста ниже рисунка:

```
<!--пример 2: Разбиение текста-->
<HTML><HEAD>
<TITLE>Домашняя страница </TITLE></HEAD>
<BODY>
<H1>Пример заголовка, размер 1</H1>
Шура Балаганов <P>  <!--Пропуск строки-->
Октябрьская улица, <BR> <!--Разрыв строки-->
10а, офис 326 <BR>
<p> данная строка демонстрирует <BR CLEAR=left>
<img src="fract.gif" align=baseline>
разрыв строки и вывод слева после рисунка </p>
<ADDRESS>Романчик - e-mail:rom@bsu.by</ADDRESS> <P>
</BODY></HTML>
```

Здесь тег **<img src="fract.gif" align=baseline>** включает рисунок.

Если не требуется, чтобы браузер автоматически переносил строку, то можно обозначить ее тегами **<NOBR>** и **</NOBR>**. В этом случае браузер не будет переносить строку, даже если она выходит за границы экрана. Вместо этого браузер позволит горизонтально прокручивать окно. Например: **<NOBR>**. Данная строка является самой длинной строкой документа, которая не допускает какого-либо разбиения где бы то ни было **</NOBR>.** Если вы хотите все же позволить разбиение данной строки на две, но в строго запланированном месте, то вставьте тег **<WBR>** в это место. Например: **<NOBR>**. Данная строка является самой длинной строкой документа **<WBR>**, которая не допускает какого-либо разбиения где бы то ни было **</NOBR>**.

Линии

Тег **<HR>** проводит контурную горизонтальную линию (опция **SHADE**). Например:

<HR NOSHADE> – горизонтальная линия с тенью;

<HR WIDTH=50%> – ширина линии, пол-экрана;

<HR WIDTH=75% ALIGN=LEFT|CENTER|RIGHT> – ширина 75%, выравнивание влево, по центру, вправо;

<HR SIZE=n> устанавливает толщину линии в **n** пикселей, где **n** от 1 до 175 (по умолчанию **n=2**).

```
<!--пример 3: Линии с разным выравниванием-->
<HTML><HEAD>
<TITLE>Примеры горизонтальных линий </TITLE></HEAD>
      <BODY>
<B> Стандартная линия, задаваемая тегом &LTHR&GT: </B>
<HR><P>
<B> Линия, заданная тегом &LTHR&GT с параметром NOSHADE:
</B>
<HR NOSHADE>
<B> Линия шириной 50% и с выравниванием по левому краю: </B>
<HR WIDTH=50% ALIGN=LEFT><P>
<B> Линия шириной 25% и с выравниванием по центру: </B>
<HR WIDTH=25% ALIGN=CENTER><P>
```

```
<B>Линия шириной 75% с выравниванием по правому краю:</B>
<HR WIDTH=75% ALIGN=RIGHT><P>
<B> Линия толщиной, равной 10, ширина 80, по центру:</B>
<HR WIDTH=80% ALIGN=CENTER SIZE=10 NOSHADE><P>
</BODY></HTML>
```

Предварительное форматирование

Дополнительные пробелы, символы табуляции и возврата каретки в исходном тексте HTML-документа будут проигнорированы Web-браузером при интерпретации документа. HTML-документ может включать вышеописанные элементы, только если они помещены внутрь тегов **<PRE>** и **</PRE>**. Эти теги используются, чтобы текст выглядел так, как набран, например при создании таблиц.

Стилевое оформление текста

Приведенные ниже теги для оформления стиля текста применяются в настоящее время крайне редко, так как получили широкое распространение таблицы стилей CSS. Можно центрировать все элементы документа в окне браузера с помощью тега **<CENTER>**. Например: **<CENTER><H1>**. Все элементы между тегами будут находиться в центре окна **</H1></CENTER>**. Гипертекстовый документ может быть оформлен с использованием следующих стилей:

<B> Полужирный **</B>**,
<I> *Курсив* **</I>**,
<TT> Моноширинный **</TT>**,
<BLINK> Мерцание, используется в Netscape Navigator **</BLINK>**,
<S> Зачеркнутый текст **</S>**,
<U> Подчеркнутый текст **</U>**,
<BIG> КРУПНЫЙ ТЕКСТ **</BIG>**,
<SMALL> мелкий текст **</SMALL>**,
_{** Нижний индекс **},
^{** Верхний индекс **}.

```
<!--пример 4: Различные стили форматирования-->
<HTML><HEAD>
<TITLE>Домашняя страница </TITLE></HEAD>
<BODY>
<CENTER> <H1>Добро пожаловать </H1>
<H2>Приглашаются все !</H2> </CENTER><HR>
Я рад <B>приветствовать Вас</B> на моей <I>странице</I>.
<P>
Вот что я <SUP>люблю</SUP> делать в <SUB>свободное</SUB>
время: <BR>
<U> - Читать книги </U><BR> <!—Подчеркивание текста -->
<S>- Исследовать Интернет</S><BR> <!—Зачеркивание текста --
>
<BLINK> -Путешествовать</BLINK><BR><HR>
<!--Blink B Internet Explorer Не работает -->
Фрагменты текста можно <TT>выделить</TT>
<BIG>крупным</BIG>шрифтом. Некоторые фрагменты бывает по-
лезно выделить <SMALL>мелким</SMALL> шрифтом.
```

```
      <ADDRESS>Романчик - rom@bsu.by</ADDRESS>
</BODY></HTML>
```

Логический стиль документа

Текст в документе HTML может быть логически выделен одним из следующих тегов:

<DFN> – определить слово. Как правило – курсив**;**

<EM> – усилить акцент. Как правило – курсив;

<CITE> – заголовок чего-то большого. Курсив;

<CODE> – компьютерный код. Моноширинный шрифт;

<KBD> – текст, введенный с клавиатуры. Жирный шрифт;

<SAMP> – сообщение программы. Моноширинный шрифт;

<STRONG> – очень важные участки. Жирный шрифт;

<VAR> – замена переменной на число. Курсив;

<BLOCKQUOTE> – позволяет включить цитату в объект.

```
<!--пример 5: Логический стиль документа-->
<HTML><HEAD>
<TITLE> Элементы содержания </TITLE></HEAD>
<BODY bgcolor="white" >
<CENTER><H5 > Элементы содержания </H5></center> <HR>
<P><BLOCKQUOTE> Это цитата (элемент BLOCKQUOTE)
</blockquote>
<P> <INS> Использование элемента INS </ins>
<P> <DEL> Использование элемента DEL </del>
<P> <Q> Использование элемента Q </q>
<P> <EM> Использование элемента EM </em>
<P> <STRONG> Использование элемента STRONG </strong>
<P> <CODE> Использование элемента CODE </code>
<P> <SAMP> Использование элемента SAMP </samp>
<P> <KBD> Использование элемента KBD </kbd>
<P> <VAR> Использование элемента VAR </var>
<P> <CITE> Использование элемента CITE </cite> <P>
<ADDRESS> Так выглядит формат адреса (элемент ADDRESS)
</ADDRESS>
</BODY></HTML>
```

Работа с тегами FONT

Теги **BIG**, **SMALL**, **B**, **I** весьма ограничены. Тег **<FONT>** позволяет установить вид, размер и цвет шрифта.

```
<FONT SIZE=n> размер шрифта n=1..7, стандартный размер n=3
</FONT>
<FONT SIZE= +3> относительный размер, 3+3=6 </FONT>
<FONT SIZE=-2> относительный размер, 1 </FONT>
Кроме размера могут устанавливаться цвет и тип шрифта,
например:
<FONT COLOR=RED SIZE=6> Пример шрифта </FONT>
<FONT FACE="Arial">Другой шрифт</FONT>
```

```
<FONT COLOR=#FF0000>красный</FONT>
<FONT COLOR=#00FF00>зеленый</FONT>
<FONT COLOR=#0000FF>синий</FONT>
```

```
<!--пример 6: Различные виды шрифтов-->
<HTML><HEAD>
<TITLE> Элементы форматирования текста</TITLE></HEAD>
<BODY>
<HR> <H3>Задание абсолютных размеров шрифтов</h3>
<P><FONT size=7> Шрифт размера 7</font>
<P><FONT size=5> Шрифт размера 5</font>
<P><FONT size=3> Шрифт размера 3</font>
<P><FONT size=1> Шрифт размера 1</font> <HR>
<H3>Задание относительных размеров шрифтов</h3>
<P><FONT size=+4> Шрифт размера +4</font>
<P><FONT size=+2> Шрифт размера +2</font>
<P><FONT size=+1> Шрифт размера +1</font>
<P><FONT size=+0> Шрифт размера +0</font>
<P><FONT size=-1> Шрифт размера -1</font>
<P><FONT size=-2> Шрифт размера -2</font> <HR>
<FONT color="green"> Задан зеленый цвет шрифта</font>
<P><FONT size=+1 face="Courier" color="red"> Шрифт Courier
</font>
</BODY></HTML>
```

Цвет символов на всей странице можно изменить с помощью аргумента **TEXT** тега **<BODY>**: **<BODY TEXT="цвет">...</BODY>** Аргумент **BGCOLOR="цвет"** изменяет цвет фона.

```
<<!-пример 7: Управление цветом текста-->
<HTML><HEAD>
<TITLE> Цветовое оформление </TITLE></HEAD>
<BODY bgcolor="white" TEXT="blue">
<CENTER>
<FONT size=6 color="red">Управление цветом текста</font>
<HR color="rede">
<FONT size=5 color="lime"><B>Стандартные цвета</b></font>
<TABLE border=3 >
<TR><TD>Аквамарин - aqua<TD bgcolor="aqua" width=200>
<TR><TD>Белый - white<TD bgcolor="white" width=200>
<TR><TD>Желтый - yellow<TD bgcolor="yellow" width=200>
<TR><TD>Зеленый - green<TD bgcolor="green" width=200>
<TR><TD>Золотистый - gold<TD bgcolor="gold" width=200>
<TR><TD>Индиго - indigo<TD bgcolor="indigo" width=200>
<TR><TD>Каштановый - maroon<TD bgcolor="maroon" width=200>
<TR><TD>Красный -red<TD bgcolor="red" width=200>
<TR><TD>Оливковый -olive<TD bgcolor="olive" width=200>
<TR><TD>Пурпурный -purple<TD bgcolor="purple" width=200>
<TR><TD>Светло-зеленый<TD bgcolor="lime" width=200>
```

```
<TR><TD>Серебристый -silver<TD bgcolor="silver" width=200>
<TR><TD>Серый - gray<TD bgcolor="gray" width=200>
<TR><TD>Синий - blue<TD bgcolor="blue" width=200>
<TR><TD>Ультрамарин - navy<TD bgcolor="navy" width=200>
<TR><TD>Фиолетовый - violet<TD bgcolor="violet" width=200>
<TR><TD>Фуксиновый - fuchsia<TD bgcolor="fuchsia"
width=200>
<TR><TD>Черный - black<TD bgcolor="black" width=200>
</table> </center> <HR color="red">
<FONT size=5 color="red"><B> красный</B></font><P>
<HR color="lime">
<FONT size=5 color="green"><B> зеленый</B></font><P>
<HR color="aqua">
<FONT size=5 color="blue"><B> синий</B></font><P>
<HR color=#FF8000">
<FONT size=5 color=#FF8000><B> оранжевый
цвет:</B></font><P>
<FONT size=5 color="gray"><B>Горизонтальная линия в качест-
ве прямоугольника:</b></font><P>
<HR color="red" size=15 width=195 align="right">
<HR color="yellow" size=15 width=285>
<HR color="lime" size=15 width=195 align="left">
<HR color="olive">
<FONT size=4 color=#008521>
<B>Необходим режим монитора
HighColor (16 разрядов) или TrueColor (24
разряда)</B></font><P>
</BODY></HTML>
```

Специальные символы

Символы, которые не могут быть введены в текст документа непосредственно через клавиатуру, называются специальными символами. Для таких символов существуют особые теги. Четыре символа – знак меньше <, знак больше >, амперсант & и двойные кавычки “” имеют специальное значение внутри HTML и, следовательно, не могут быть использованы в тексте в своем обычном значении. Скобки используются для обозначения начала и конца HTML-тегов, а амперсант используется для обозначения так называемой escape-последовательности. Для использования одного из этих символов введите одну из следующих escape-последовательностей:

```
< - &lt; > - &gt; & - &amp; " - &quote.
```

Списки и таблицы

Списки подразделяются на ненумерованные, создаваемые с тегом **<UL>**, и нумерованные, создаваемые с тегом **OL**:

```
<!--пример 8: Ненумерованные и нумерованные списки-->
<HTML><HEAD>
<TITLE>Использование списков </TITLE></HEAD>
<BODY>
```

```
<CENTER><H3>Домашняя страница </H3></CENTER>
<h4>Ненумерованный список: Мои интересы:</h4>
<UL>
<LH><B>Занятия в свободное время:</B></LH>
<LI> Компьютеры
<LI> Чтение книг
<LI> Бассейн
<LI> Отдых на природе
</UL> <HR>
<H4> Нумерованный (упорядоченный) список.</H4>
<OL TYPE=1>
<LH><B>Мое путешествие</B></LH>
<LI> Прибытие в Варшаву
<LI> Автобусом в Будапешт
<LI> Самолетом в Рим
</OL> <HR>
<OL TYPE=A>
<LH><B>Продолжение путешествия</B></LH>
<LI> Автобусом в Берлин
<LI> Поездом в Варшаву
<LI> Пешком в Минск
</OL> <HR>
</BODY></HTML>
```

При этом вид нумерации устанавливается аргументом **TYPE: TYPE=1** – стандартная нумерация **1,2,3,4...; TYPE=A** – прописные буквы **A, B, C, D...; TYPE=a** – строчные буквы **a,b,c,d...; TYPE=I** – римские цифры **I, II, III, IV...; TYPE=i** – строчные римские цифры **i,ii,iii,iv,v....**

Еще один вид списков – списки определений **DL** – состоит из пар элементов: определяемого **<DT>** и определения **<DD>**.

```
<!--пример 9: Списки определений-->
<HTML><HEAD>
<TITLE>Использование списков</TITLE></HEAD>
<BODY>
<CENTER><H2> Толковый словарь</H2></CENTER><HR>
<DL>
<LH><Big><B> Список терминов</B></Big></LH><HR>
      <DT><B>"Anchor"</B>
<DD><I>То, что образовывает гипертекстовую ссылку</I>
      <DT><B>" Anonymous "</B>
<DD><I>Один из методов получения доступа к информации</I>
      <DT><B>"Lamer"</B>
<DD><I> Юзер-идиот</I>
      <DT><B>"Bullet Маркер"</B>
<DD><I>Например: маленький круг, квадрат, звездочка.</I>
      <DT><B>"Cookies "</B>
```

```
<DD><I>Технология, позволяющая сохранять индивидуальную ин-
формацию о пользователе сети</I>
            </DL>
</BODY></HTML>
```

Таблицы можно создавать с помощью предварительного форматирования, например:

```
<PRE>
<B>   ИМЯ           КОМАНДА       ОЧКИ </B>
      Superman      Динамо          10
      BigMan        БАТЭ             7
      Small         Барселона        5
</PRE>
```

Для создания таблиц используются следующие теги **HTML**: **<TABLE>** и **</TABLE>** – охватывают таблицу. Для того чтобы была видна разделяющая строки и столбцы сетки, используется атрибут **BORDER** (например: **<TABLE BORDER=1>**). Текст в тегах **<CAPTION>** и **</CAPTION>** выводится в виде заголовка. В тегах **<TH>** и **</TH>** помещаются заголовки столбцов или строк. Теги **<TR>** и **</TR>** определяют каждую строку таблицы. Теги **<TD>** и **</TD>** определяют текст каждой ячейки таблицы.

```
<!--пример 10: Простая таблица-->
<HTML><HEAD>
<TITLE>Использование таблиц</TITLE>
</HEAD><BODY>
<TABLE BORDER>
<CAPTION ALIGN=top>Лучшие нападающие года</CAPTION>
      <TR>
            <TH>Имя</TH>
            <TH>Команда</TH>
            <TH>Очки</TH>
      </TR>
      <TR>
            <TD>Small   </TD>
            <TD> Барселона</TD>
            <TD>5</TD>
      </TR>
      <TR>
            <TD>Superman</TD>
            <TD>  Dinamo</TD>
            <TD>10</TD>
      </TR>
      <TR>
            <TD>BigMan  </TD>
            <TD>БАТЭ    </TD>
            <TD>7</TD>
      </TR>
</TABLE></BODY></HTML>
```

Чтобы ячейка занимала две строки вместо одной, можно заменить тег на следующий: **<TD ROWSPAN=2> </TD>**. Аналогично два столбца можно объеди-

нить с помощью тега **<TH COLSPAN=2>** или **<TD COLSPAN=2>**. Изменить цвет в таблице можно с помощью аргумента **BGCOLOR**, как в следующем примере:

```
<!--пример 11: изменение цвета-->
<HTML><HEAD>
<TITLE> Таблицы </TITLE></HEAD>
<BODY bgcolor="white">
<CENTER><FONT size=6>Примеры таблиц</font></center>
<HR color="blue">
<TABLE border=4 cellspacing=3>
<CAPTION><B>Стандартная таблица<B> </caption>
<TR><TH bgcolor="yellow">Заголовок 1
 <TH bgcolor="yellow">Заголовок 2
<TR><TD>Ячейка 1
 <TD>Ячейка 2
<TR><TD>Ячейка 3
 <TD>Ячейка 4
</TABLE>
<HR color="blue">
<TABLE border="4" cellspacing=3 background="fon01.gif">
<CAPTION>Таблица с объединенными ячейками и фоновым
       рисунком</caption>
<TR><TH rowspan="2"> <TH colspan="2">Заголовок 1
<TR><TH>Заголовок 1.1<TH>Заголовок 1.2
<TR><TH>Заголовок 2<TD>Ячейка 1<TD>Ячейка 2
<TR><TH>Заголовок 3<TD>Ячейка 3<TD>Ячейка 4
</TABLE>
<HR color="blue">
<H2>Использование элементов THEAD, TBODY и TFOOT</h2>
<TABLE border=2>
<THEAD>
<TR> <TD>Заголовок 1<TD>Заголовок 2
<TFOOT>
<TR> <TD>Нижний блок таблицы<TD> 
<TBODY>
<TR> <TD>Строка 1 <TD>Ячейка 1.2
<TR> <TD>Строка 2 <TD>Ячейка 2.2
<TBODY>
<TR> <TD>Строка 3 <TD>Ячейка 3.2
<TR> <TD>Строка 4 <TD>Ячейка 4.2
<TR> <TD>Строка 5 <TD>Ячейка 5.2
</TABLE>
<HR color="blue">
<H2>Объединение ячеек</h2>
<TABLE border=4 cellspacing=0 width=70%>
<TR><TD bgcolor="yellow"><B>Заголовок 1</b>
 <TD bgcolor="yellow"><B>Заголовок 2</b>
<TR><TD rowspan=3 bgcolor="lime">Ячейка 1
```

```
 <TD>Ячейка 2
<TR><TD>Ячейка 3
<TR><TD>Ячейка 4
<TR><TD colspan=2 bgcolor="aqua" align="center">Ячейка 5
</TABLE>
<HR color="blue">
<H2>Вложенная таблица</h2>
<TABLE border=4 cellspacing=2 width=75%>
<TR><TD bgcolor="yellow">Таблица 1
 <TD bgcolor="yellow">
<TABLE border=2 cellspacing=2>
<TR><TD bgcolor="red">Таблица 2
 <TD bgcolor="red">Ячейка 2-2
<TR><TD bgcolor="red">Ячейка 3-2
 <TD bgcolor="red">Ячейка 4-2
</TABLE>
<TR><TD bgcolor="yellow">Ячейка 3-1
 <TD bgcolor="yellow">Ячейка 4-1
</TABLE></BODY></HTML>
```

Ссылки

HTML позволяет связать текст или картинку с другими гипертекстовыми документами с помощью тегов **<A>** и **<LINK>**. Текст, как правило, выделяется цветом или оформляется подчеркиванием. Чтобы сформировать ссылку, следует набрать **<A, введите HREF= "filename">**, ввести текст ссылки, закрыть тег **</A>**. В следующем примере слово **Minsk** ссылается на документ **MinskAnapa.html**, образуя гипертекстовую ссылку:

<A HREF="MinskAnapa.html">Minsk</A>.

Если документ, формирующий ссылку, находится в другой директории, то подобная ссылка называется относительной. Например:

<A HREF="MinskAnapa/MinskMoscow.html">Minsk</A>

Ссылки можно формировать на основе URL, используя синтаксис: **protocol: //hostport/path**. Например:

```
<A  HREF="http://www.w3.org/TR/REC-html40">  Документ  HTML
</A>
```

```
<!--пример 12: Создание ссылок на html-файлы-->
<HTML><HEAD>
<TITLE>Ссылки на домашней странице</TITLE></HEAD>
<BODY>
<H1>Внутренние ссылки на части документа</H1></CENTER>
<FONT SIZE=+1>
<HR NOSHADE>
<H2>Вы можете:</H2>
<UL>
<LI>Посмотреть <A HREF="Pr11.htm">Простейший пример</A>
<LI>Посмотреть <A HREF="Pr12.htm">Второй пример</A>
```

```
<LI>Посмотреть <A HREF="Pr13.htm">разбиение текста</A>
<LI>Узнать <A HREF="Pr14.htm">О линиях</A>
</UL>
<HR NOSHADE>
Если вас интересует подробная информация, пишите по адресу
 <A HREF="mailto:Rom@Bsu.by">Rom@Bsu.by</A>
</FONT></BODY></HTML>
```

Якоря на Web-странице

Для того чтобы организовать ссылки на разделы документа, находящегося в одном файле, используются якоря **(anchors)**. При этом создается ссылка на якорь: **<A href="#**Имя якоря**">**Текст гиперссылки**</A>**.

Сам якорь с указанным именем помещается в то место документа, в которое осуществляется переход: **<A name="**Имя якоря**">** Текст **</A>**.

```
<!--пример 13: Ссылки на якоря-->
<HTML> <HEAD>
<TITLE>Якоря на домашней странице</TITLE>
</HEAD>
<BODY>
<!-- Создание ссылок на якоря -->
<UL><LH>Содержание</LH>
<LI><A href="#section1">Введение</A>
<LI><A href="#section2">Обзор</A>
<LI><A href="#section3">Подробное рассмотрение</A></UL><P>
<HR>
...тело документа...
<HR>
<H2><A name="section1">Введение</A></H2><HR>
...section 1...
<HR><!-- Установка якорей -->
<H2><A name="section2">Обзор</A></H2><HR>
...section 2...
<HR>
<H3><A name="section3">Подробное рассмотрение</A></H3><HR>
...section 3...
<HR>
 <A HREF="mailto:Romanchik@Bsu.by">
<ADDRESS>Романчик - e-mail:rom@bsu.by</ADDRESS></A>
<P></BODY> </HTML>
```

Такой же эффект можно получить, используя заглавия вместо якоря:

```
<!--пример 14: Заглавия вместо якоря-->
<HTML><HEAD>
<TITLE>Ссылки на заголовки</TITLE>
</HEAD>
<BODY>
<H1>Table of Contents</H1><P>
<A href="#section1">Introduction</A><BR>
```

```
<A href="#section2">Some background</A><BR>
<A href="#section3">The first part</A><BR>
...the rest of the table of contents...
<H2 id="section1">Introduction</H2> <HR>
...section 1...<HR>
<H2 id="section2">Some background</H2><HR>
...section 2...<HR>
<H3 id="section3">The first part</H3><HR>
...section 3...<HR>
...Продолжение документа...
</BODY></HTML>
```

Кроме элемента **A**, который помещается в теле HTML-документа **<BODY>**, для организации ссылок используется элемент **LINK**, который помещается в заголовке документа **<HEAD>**.

Установить цвет ссылки можно с помощью атрибута **LINK** тега **<BODY>**, цвет посещенной ссылки – с помощью атрибута **VLINK**, цвет активной ссылки – с помощью атрибута **ALINK**. Например:

```
<BODY TEXT=LIME LINK=RED VLINK=SILVER ALINK=BLUE>
```

Чтобы установить ссылку с помощью изображения, надо вместо текста ссылки поставить HTML-код для вывода изображения:

```
<A HREF=”sample.htm”> <IMG SRC=”image.gif”> </A>
```

Изображения на странице

Любая Web-страница должна иметь графические изображения. Рисунок с Web-страницы можно скопировать и сохранить, щелкнув правой кнопкой мыши по рисунку. В появившемся меню выбрать пункт `Save Picture As`. В сети Интернет много бесплатных рисунков, не защищенных авторскими правами.

Рисунки можно изготовить с помощью графических редакторов Paint Shop Pro, Photoshop, 3D-Studio, Corel Draw и др. Указанные редакторы позволяют конвертировать изображения из формата BMP в один из принятых в Web форматов GIF, JPEG или PNG.

Формат GIF (Graphical Interchange Format) был разработан фирмой CompuServe более 20 лет назад. Это формат упакованных растровых изображений, который стал широко использоваться в Web. Формат PNG является усовершенствованием GIF. На сегодняшний день графический формат GIF (Graphics Interchange Format) – основной в Internet. Это обусловлено прежде всего тем, что файлы картинок, сохраненные в формате GIF, весьма компактны. GIF очень плотно сжимает графический файл, при этом на качестве самой картинки подобное сжатие практически не отражается. GIF обладает еще одной необыкновенной возможностью. Этот графический формат позволяет определять в исходной картинке прозрачные цвета, т.е. части картинки при ее просмотре в обозревателе становятся невидимыми.

Формат JPEG – это формат, в котором изображения могут изменять размеры и занимают в 3–4 раза меньше места, лучше сохраняют цвета и детали. У формата JPEG имеется преимущество: JPEG способен работать не только с 256 цветами, как GIF, но и с 16 миллионами цветов. Итак, GIF – рисунки, а JPEG – фотографии.

Как поместить изображение на домашнюю страницу? Изображение можно вставить в любое место страницы с помощью тега **<IMG>**, вставляемого между

тегами **<BODY>** и **</BODY>**. Например, изображение **FRACT.GIF** загружается из текущего каталога: **<IMG SCR=FRACT.GIF>**.

Загрузка изображения из каталога уровнем выше:

```
<IMG SRC=../FRACT.GIF>
```

Загрузка изображения с другого диска:

```
<IMG SRC=FILE:///D:\FRACT.GIF>
```

Считается хорошим тоном вместе с изображением использовать альтернативный текст, который выводится, пока изображение загружается. Например:

```
<IMG SRC=FRACT.GIF ALT="Фрактал">
```

Если изображение не найдено, браузеры выводят на место изображения стандартную пиктограмму. Изображения можно выравнивать с помощью атрибута **ALIGN** по левому, правому краю, самому высокому элементу в строке, по середине:

```
ALIGN=LEFT| RIGHT |TOP |TEXTTOP| MIDDLE |ABSMIDDLE| BOTTOM|
BASELINE
```

В примере текст будет выводиться справа от изображения:

```
<IMG SRC=fract.gif ALT="Фрактал" ALIGN=LEFT>
```

Размеры изображения можно изменять с помощью атрибутов **HEIGTH** и **WIDTH**. При этом меняются не размеры изображения, а только вид на экране. Например:

```
<IMG SRC=tigers.gif ALT= "ТИГРЫ" ALIGN = LEFT WIDTH=240
HEIGHT=260>
```

На странице могут использоваться пиктограммы – маленькие изображения в форматах GIF или JPG. Примерами пиктограмм могут быть линии, маркеры, пиктограммы новинок.

Фон

Цвет фона устанавливается атрибутом **BGCOLOR** тега **<BODY>**. Например:

```
<BODY BGCOLOR=”RED”>
```

Фоновые изображения задаются атрибутом **BACKGROUND**. Например:

```
<BODY BACKGROUND="bg.jpg">
```

Фоновые рисунки похожи на обои, наклеиваемые из небольших периодических рисунков. Обычно цвет фона светлый, рисунок легкий.

Формат GIF позволяет задать один из 256 цветов, который при отображении браузером будет игнорироваться, вместо него устанавливается цвет фона, так что изображение будет прозрачным. Прозрачные изображения можно создать в формате GIF графическим редактором.

Наложение изображений

Ключевое слово **LOWSRC** позволяет сначала загрузить файл с низким разрешением, затем больший файл с высоким разрешением:

```
<IMG SRC="highcar.gif" LOWSRC="lowcar.gif">
```

В примере сначала загружается файл **lowcar.gif**, а затем **highcar.gif**.

Звуковоспроизведение

Для того чтобы вставить в Web-страницу звуковой файл, например midi-файл, следует использовать конструкцию:

```
<EMBED SRC="music.mid" WIDTH="140" HEIGHT="50"
ALIGN="MIDDLE" BORDER="0" AUTOSTART=TRUE>
```

В этом теге были применены следующие параметры: **WIDTH** – параметр, определяющий ширину midi-плеера, **HEIGHT** – параметр, определяющий высоту midi-плеера, **BORDER** – ширина рамки midi-плеера, **AUTOSTART** – запустить midi-плеер сразу после того как загрузится документ HTML.

Анимация

Анимационный GIF является обыкновенным графическим файлом. Дело в том, что подобный файл состоит из нескольких изображений, которые через браузер последовательно выводятся на экран. Чтобы создать анимационную картинку, необходимо сначала создать картинки, из которых будет состоять результирующий файл. Эти картинки можно сделать, например, в Adobe Photoshop.

Фреймы

Технология фреймирования в HTML позволяет просматривать в одном окне обозревателя несколько гипертекстовых документов. Один фрейм отображает только один гипертекстовый документ. Создание фрейма осуществляется с помощью тегов **<FRAMESET>** и **</FRAMESET>.** Тег **<BODY>** в этом случае не используется. Если браузер не поддерживает фреймы, то между тегом **<NOFRAMES>** и тегом **</NOFRAMES>** заносится текст, который распознает браузер.

Тег **<FRAME SRC="Name1">** позволяет описать первый фрейм, т.е. присвоить имя гипертекстовому документу. Второй фрейм описывается тегом **<FRAME SRC="Name2" NAME="Main">**.

Тег **<FRAMESET COLS=n>** позволяет определить количество фреймов и задать размер фреймов в процентах от размера окна обозревателя или зафиксировать эти размеры в пикселях.

Тег **<FRAME>**, имеющий самое большое количество атрибутов, позволяет настроить свойства фрейма. Ниже описываются эти атрибуты:

NAME= – имя фрейма.

MARGINWIDTH= – горизонтальный отступ (от 1 до 6) между фреймом и его границей.

MARGINHEIGHT= – вертикальный отступ (от 1 до 6) между фреймом и его границей.

SCROLLING=YES|NO|AUTO – позволяет создать/не создавать полосы прокрутки.

SCROLLING=AUTO – позволяет отображать полосы прокрутки в зависимости от свойств обозревателя.

NORESIZE – фиксированный размер фрейма.

SRC= – задать гипертекстовый документ для этого фрейма.

TARGET=Name – открыть ссылку во фрейме с именем **Name**.

```
<!--пример 16: Фреймы-->
<HTML><HEAD>
```

```
<TITLE> Пример фреймов </TITLE></HEAD>
<FRAMESET rows="20%,60%,20%">
<FRAME src="fr1.htm" noresize scrolling="no">
<FRAMESET cols="22%,78%">
<FRAME src="fr2.htm">
<FRAME   src="fr3.htm"   scrolling="yes"   marginwidth="10"
marginheight="75">
</frameset>
<FRAME src="fr4.htm">
</frameset>
<NOFRAMES>
<CENTER><FONT size=6>Фреймы</font></center>
<HR color="blue">
Этот браузер не может воспроизводить фреймы
</noframes>
</frameset></HTML>
```

Формы HTML

Пользователь заполняет форму и передает информацию из нее для обработки программе, работающей на сервере. Эта программа может быть написана по технологии CGI, ASP или Servlet/JSP.

Теги **<FORM>...</FORM>** используются для обозначения документа как формы. Внутри элемента **<FORM>** определяется последовательность элементов **<INPUT>**, которые представляют поля для ввода информации.

<INPUT TYPE=TEXT> помещает в форму текстовое поле данных.

Если **<INPUT>** используется с атрибутом **TYPE=TEXT**, устанавливаемым по умолчанию, то могут быть использованы еще три атрибута. Атрибут **MAXLENGTH** устанавливает максимальное число вводимых символов. Атрибут **SIZE** определяет размер видимой на экране области, занимаемой текущим полем. Атрибут **VALUE** устанавливает начальное значение поля.

<INPUT TYPE=CHECKBOX> позволяет определить флажок для протокола передачи. Тип элемента ввода **CHECKBOX** позволяет получить ответы пользователя типа ДА/НЕТ. Элемент **INPUT** при установке атрибута **TYPE=CHECKBOX** использует также атрибуты **NAME="имя" VALUE="значение"**. Элемент **<INPUT TYPE=RADIO>** позволяет определить кнопку переключения и используется, если надо выбрать одно из нескольких значений.

```
<!--пример 17: Простая форма и элементы checkbox и radio-->
<HTML><HEAD>
<TITLE>Простая форма, checkbox и radio </TITLE>
</HEAD><BODY>
<FORM><H2>Простая форма</H2>
<P>My street:<INPUT NAME="street"><BR>
City: <INPUT NAME="city" SIZE="20" MAXLENGTH="20"
VALUE="Minsk"> <BR>
Zip: <INPUT NAME="zip" SIZE="5" MAXLENGTH="5"
VALUE="99999"><BR>
</FORM> <HR>
```

```
<P><H2>Ваша любимая команда</H2>
<FORM><!--Выбор одной или нескольких команд -->
<INPUT TYPE="checkbox" NAME="team" VALUE="шахтеры">
шахтеры<BR>
<INPUT TYPE="checkbox" NAME="team" VALUE="ковбои"> ковбои
<BR> <INPUT TYPE="checkbox" NAME="team" VALUE="викинги">
викинги<BR>
</FORM> <HR>
<P><H2>Какая из команд самая любимая?</H2>
<FORM><!--Выбор только одной из нескольких команд -->
<INPUT TYPE="radio" NAME="team" VALUE="шахтеры">
шахтеры <BR>
<INPUT TYPE="radio" NAME="team" VALUE=" ковбои"> ковбои
<BR>
<INPUT TYPE="radio" NAME="team" VALUE="викинги"> викинги
<BR>
</FORM><HR>
</BODY></HTML>
```

Элемент ввода **SELECT** позволяет использовать при вводе списки с прокруткой и выпадающее меню. Для определения списка пунктов используется элемент **<OPTION>** и необязательные атрибуты **MULTIPLE**, **NAME, SIZE**.

Атрибут **SELECTED** устанавливает значение элемента для первоначального выбора. Атрибут **VALUE** указывает на значение, возвращаемое формой после выбора данного пункта.

```
<!--пример 18: Формы. Элемент SELECT-->
<HTML>
<FORM><SELECT NAME="flower">
<OPTION>chocolate
<OPTION>vanila
<OPTION VALUE="Banana">Banana
<OPTION SELECTED>cherry
</SELECT> </FORM>
</HTML>
```

Элемент **<INPUT TYPE=RESET>** используется для создания кнопки `Reset`, по которой можно щелкнуть мышкой и вернуться к начальным значениям полей.

Элемент **<INPUT TYPE=SUBMIT>** используется для создания кнопки, по которой можно щелкнуть и отправить введенные данные в виде сообщения по указанному адресу. Дополнительный атрибут **NAME** устанавливает название кнопки **submit**. Атрибут **VALUE** хранит значение переменной поля формы.

```
<!--пример 19 : Формы. Элемент SELECT-->
<HTML>
<HEAD>
<TITLE> Формы </TITLE>
</HEAD>
<BODY bgcolor=#C0C0C0>
<CENTER><FONT size=6>Элементы диалога </font></center><P>
```

```
<HR color="blue">
<H2>Пример формы</h2>
<TABLE border=3 bgcolor="pink">
<FORM action="сервер.домен.путь" method="post">
<TR>
<TD><LABEL for="imya1">Имя: </label>
<TD><INPUT type="text" id="imya1">
<TR>
<TD><LABEL for="familiya1">Фамилия: </label>
<TD><INPUT type="text" id="familiya1">
<TR>
<TD><LABEL for="telefon1">Телефон: </label>
<TD><INPUT type="text" id="telefon1">
<TR>
<TD>Пол:   
<INPUT type="radio" name="pol1" value="Male"> M 
<INPUT type="radio" name="pol1" value="Female"> Ж
<TD align="bottom">
<INPUT align="bottom" type="submit" value="Отослать">
<INPUT type="reset">
</form>
</table>
<HR color="blue">
<H2>Элемент ISINDEX</h2>
<ISINDEX prompt="Строка для ввода критерия поиска">
<HR color="blue">
<H2>Элементы INPUT</h2>
<H3> Ввод текстовой строки </h3>
<INPUT type="text" size=50>
<H3> Ввод пароля </h3>
<INPUT type="password">
<H3> Флажки </h3>
<INPUT type="checkbox" name="F001" checked>
<INPUT type="checkbox" name="F001" checked>
<H3> Переключатели </h3>
<INPUT type="radio" name="S001" value="Первый">
<INPUT type="radio" name="S001" value="Второй">
<INPUT type="radio" name="S001" value="Третий" checked>
<H3> Кнопка подтверждения ввода </h3>
<INPUT type="submit" value="Подтверждение">
<H3> Кнопка с изображением </h3>
<INPUT type="image" src="knopka1.gif">
<H3> Кнопка очистки формы </h3>
<INPUT type="reset" value="Очистка">
<H3> Файл </h3>
<INPUT type="file" name="photo" accept="image/*">
<HR color="blue">
<H2>Элемент SELECT </h2>
```

```
<SELECT multiple>
<OPTION value=a>Первый
<OPTION value=b>Второй
<OPTION value=c>Третий
<OPTION value=d>Четвертый
</select>
<HR color="blue">
<H2>Элемент TEXTAREA
<TEXTAREA rows=5 cols=30>
Область для ввода текста
</textarea></h2>
<HR color="blue">
<H2>Кнопка с надписью и рисунком</h2>
<BUTTON name="submit" value="submit" type="submit">
Надпись<IMG src="gif1.gif" alt="Рисунок"></button>
<HR color="blue">
<H2>Группа полей</h2>
<FIELDSET>
<LEGEND>Заголовок группы</legend>
Имя: <INPUT name="imya2" type="text">
Фамилия: <INPUT name="familiya2" type="text"><BR>
Телефон: <INPUT name="telefon2" type="text"><BR>
Текст подсказки
</fieldset>
<HR color="blue">
</BODY></HTML>
```

Метатеги

Любой метатег размещается в заголовке HTML-документа между тегами **<HEAD>** и **</HEAD>** и состоит из следующих атрибутов:

```
<META HTTP-EQUIV="имя" CONTENT="содержимое">
<META NAME="имя" CONTENT="содержимое">
```

Метатеги с атрибутом **HTTP-EQUIV** управляют действиями браузеров и могут быть использованы для совершенствования информации, выдаваемой обычными HTTP-заголовками. В качестве параметра **"имя"** могут быть использованы следующие аргументы:

Expires – Дата устаревания: если указанная дата прошла, то запрос этого документа вызывает повторный сетевой запрос, а не подгрузку документа из кэша. Дата со значением "**0**" заставляет браузер каждый раз при запросе проверять – изменялся ли этот документ. Например:

```
<META HTTP-EQUIV="expires" CONTENT="Sun, 3 April 2005
05:45:15 GMT">
```

Pragma – контроль кэширования. Значением должно быть "no-cache".

Content-Type – указание типа документа. Может быть расширено указанием браузеру кодировки страницы (charset). Например:

```
<META HTTP-EQUIV="Content-type" CONTENT="text/html;
charset=windows-1251>
```

Content-language – указание языка документа. Комбинация поля **Accept-Language** (посылаемого браузером) с содержимым **Content-language** может быть условием выбора сервером того или иного языка.

```
<META HTTP-EQUIV="Content-language" CONTENT="en-GB">
```

Refresh – определение задержки в секундах, после которой браузер автоматически обновляет документ. Дополнительная возможность – автоматическая загрузка другого документа.

```
<META HTTP-EQUIV="Refresh" Content="3,
URL=http://www.bsu.iba.by/cgi-bin/news.pl">
```

Window-target – определяет окно текущей страницы; может быть использован для прекращения появления новых окон браузера при применении фреймовых структур.

```
<META HTTP-EQUIV="Window-target" CONTENT="_top">
```

Ext-cache – определяет имя альтернативного кэша

```
<META HTTP-EQUIV="Ext-cache" CONTENT=
"name=/some/path/index.db; istructions=User nstructions">
```

PICS-Label – **Platform-Independent Content rating Scheme**. Обычно используется для определения рейтинга "взрослости" содержания **(sex)**, однако может использоваться для других целей.

Управление индексацией страницы для поисковых роботов осуществляется с использованием атрибута **NAME**.

```
<META NAME="Robots" CONTENT="NOINDEX,FOLLOW">
```

Возможные значения: **ALL**, **NONE**, **INDEX**, **NOINDEX**, **FOLLOW**, **NOFOLLOW**.

Description – краткая аннотация содержания документа. Используется поисковыми системами для описания документа.

```
<META NAME="Description" CONTENT= "Документ содержит сло-
варь METAтегов">
```

Keywords – используется поисковыми системами для индексирования документа. Обычно здесь указываются синонимы к словам в заголовке **title** или альтернативный заголовок.

```
<META NAME="Keywords" CONTENT="теги, метаданные, список">
```

Document-state – управление индексацией страницы для поисковых роботов. Определяет частоту индексации – или один раз индексировать, или реиндексировать документ регулярно.

```
<META NAME="Document-state" CONTENT="Static">
```

Возможные значения: **Static**, **Dynamic**

URL – управление индексацией страницы для поисковых роботов. Определяет частоту индексации – или один раз индексировать, или реиндексировать документ регулярно.

```
<META NAME="URL" CONTENT="absolute_url">
```

Author – обычно имя автора, формат произвольный.

Generator – обычно название и версия редактора, с помощью которого создана эта страница.

Copyright – обычно описание авторских прав на документ.

Resource-type – текущее состояние данного файла. Важен для поисковых систем: если его значение **document**, то поисковая система приступает к индексированию.

Каскадные таблицы стилей

Таблицы стилей (CSS) позволили отделить содержание документа от его форматирования и отображения. HTML-документы могут содержать внедренные стили непосредственно внутри документа или импортировать стили из связанных таблиц стилей, находящихся в файлах с расширением CSS. Элемент **META** указывает тип документа в виде:

```
<META http-equiv="Content-Style-Type" content="text/css">
```

При использовании внедренных стилей для установки стиля отдельного элемента HTML в этом элементе используется атрибут **style.** В следующем примере устанавливается цвет и размер шрифта для отдельного параграфа и заголовка:

```
   <P style="font-size: 12pt; color: fuchsia">Размер шрифта
12 пикселей </P>
   <H1 style="text-decoration:underline"> Текст с подчерки-
ванием </H1>
   <H2 style="color: red"> Текст заголовка красным цве-
том</H2>
```

Объявление значений свойств имеет вид **"name: value"**.

Для того чтобы стили относились к нескольким элементам документа, например к одному или всем **P**-элементам, **H1**-элементам, гиперссылкам, используется блок **STYLE**. Блок **STYLE** размещается в секции **HEAD** документа. Следующий стиль обводит границы вокруг каждого **H1**-элемента и центрирует его на странице. Кроме этого, устанавливается стиль параграфа и гиперссылки.

```
<!--пример 20 : стиль-->
<HEAD>
<STYLE type="text/css">
H1 {border-width: 1; border: solid; text-align: center}
P { color: blue}
a: hover {color: red; text-decoration: overline}
</STYLE></HEAD>
```

Для гиперссылок устанавливаются следующие значения стилей:

a:hover – стиль меняется при наведении курсора;

a:active – стиль меняется при щелчке левой кнопкой мыши;

a:visited – стиль меняется после посещения;

a:link – непосещенная ссылка.

Данные о стиле размещаются в фигурных скобках.

В следующем примере стиль относится к определенным **H1**-элементам. Чтобы скрыть таблицы стилей от старых программ просмотра, их помещают внутрь HTML-комментария:

```
<!--пример 21: стиль в комментарии-->
<HEAD><STYLE type="text/css">
<!-- маскируем
```

```
H1.myclass {border-width: 1; border: solid; text-align:
center}
-->
</STYLE></HEAD>
<BODY>
<H1 class="myclass">H1 is affected by our style</H1>
<H1> This one is not affected by our style </H1>
</BODY>
```

Два тега – **DIV** и **SPAN**, играющие роль скобок, используются, чтобы применить стили к ограниченному фрагменту документа. В следующем примере элемент **SPAN** используется, чтобы вывести несколько слов параграфа прописными буквами и установить стили для других параграфов.

```
<!--пример 22 : применение тегов DEV и SPAN-->
<HEAD><STYLE type="text/css">
SPAN.sc-ex { font-variant: small-caps }
</STYLE></HEAD><BODY>
<P><SPAN class="sc-ex">The first</SPAN> few words are in
small-caps.</P>
<P>Это<SPAN style="font-style:italic"> курсив</SPAN></p>
<P>Это<SPAN style="text-transform-style:uppercase">верхний
регистр </SPAN> </p>
</BODY>
```

В следующем примере используется **DIV** и атрибут **class**, чтобы установить правила для двух параграфов.

```
<!--пример 23 : стиль для фрагментов-->
<HEAD><STYLE type="text/css">
DIV.Abstract { text-align: justify }
</STYLE></HEAD><BODY>
<DIV class="Abstract">
<P>The Chieftain product range is our market winner for the
coming year. This report sets out how to position Chieftain
against competing products.
<P>Chieftain replaces the Commander range, which will
remain on the price list until further notice.
</DIV>
<P style="position:absolute"; top:125px; left:200px > Про-
стой текст для позиционирования, на который накладывается
изображение </p>
<DIV style="position:absolute"; top:125px; left:200px >
<img src="fract.gif"> </DIV>
</BODY>
```

Для третьего параграфа в примере, в котором на текст накладывается изображение, установлено позиционирование: **position:absolute** – точка отсчета: левый угол окна; **top** – вертикальное, **left** – горизонтальное смещение от точки отсчета.

Внешние таблицы стилей позволяют установить стили для нескольких документов, сохранить в файле .**css** и затем изменять их без модификации документа. При этом используются следующие атрибуты элемента **LINK**:

Значение **ref** устанавливается на URL файла стилей. Значение атрибута **type** определяет тип таблицы стилей. Атрибут **rel** устанавливается в таблицу стилей **stylesheet**. Например:

```
<LINK href="mystyle.css" rel="stylesheet" type="text/css">
```

В следующем примере таблица, помещенная в файл **special.css**, устанавливает цвет текста в параграфе зеленым, а границу – красным:

```
P.special
{
 color: green;
 border: solid red;
}
```

Эту таблицу стилей можно связать с HTML-документом с помощью элемента **LINK**:

```
<!--пример 24 : установка стиля для HTML-документа-->
<HTML><HEAD>
<LINK href="special.css" rel="stylesheet" type="text/css">
</HEAD><BODY>
<P class="special">paragraph should have green text.
</BODY></HTML>
```

В контексте использования Java-технологий можно отметить три возможности использования HTML:

1. Использование тегов **<applet> </applet>** для включения java-апплетов в HTML-документ.
2. Использование форм HTML и методов **GET** и **POST** для пересылки запросов и информации из форм серверу для обработки сервлетами.
3. Ответы клиенту, пересылаемые серверу на основании выполнения сервлетов и JSP, также конвертируются в HTML-документ и отображаются на стороне клиента.

ПРИЛОЖЕНИЕ 2

XML и JAVA

Язык разметки XML (Extensible Markup Language) был разработан W3C. Главным преимуществом XML является совместимость данных, представленных в этом формате, с различными приложениями. Для данных XML – это то же самое, что и язык Java для информационных систем.

Язык XML был разработан на базе универсального языка разметки SGML. Собственно язык HTML, как язык разметки гипертекстовых документов, также произошел от SGML.

Основная идея XML – это текстовое представление с помощью тегов, структурированных в виде дерева данных. Древовидная структура хорошо описывает бизнес-объекты, конфигурацию, структуры данных и т.п. Данные в таком формате легко могут быть как построены, так и разобраны на любой системе с использованием любой технологии – для этого нужно лишь уметь работать с текстовыми документами. Почти все современные технологии стандартно поддерживают работу с XML. Кроме того, такое представление данных удобочитаемо (human-readable). Если нужен тег для представления имени, его можно создать: **<name>Igor</name>**.

DTD

Для описания структуры XML-документа используется DTD (Document Type Definition). DTD определяет, какие теги (элементы) могут использоваться в XML-документе, как эти элементы связаны (например, указывать на то, что элемент **<book>** включает дочерние элементы **<price>** и **<author>**), какие атрибуты имеет тот или иной элемент.

Зачем это нужно? В принципе, никто не требует создания DTD для XML-документа, программы-анализаторы будут обрабатывать XML-файл и без DTD. Но в этом случае остается только надеяться, что автор XML-файла правильно его сформировал.

Для того чтобы сформировать DTD, можно создать либо отдельный файл и описать в нем структуру документа, либо включить DTD-описание непосредственно в документ XML.

В первом случае в документ XML помещается ссылка на файл DTD:

```
<?xml version="1.0" standalone="yes" ?>
<! DOCTYPE journal SYSTEM "book.dtd">
```

Во втором случае описание элемента помещается в XML-документ:

```
<?xml version="1.0" ?>
...
<! DOCTYPE book [
<!ELEMENT book (price, author)>
...
]>
```

Описание элемента

Элемент в DTD описывается с помощью дескриптора **!ELEMENT**, в котором указывается название элемента и его содержимое. Так, если нужно определить

элемент **<book>**, у которого есть дочерние элементы **<price>** и **<author>**, то можно сделать это следующим образом:

```
<!ELEMENT price PCDATA>
<!ELEMENT author PCDATA>
<!ELEMENT book (price, author)>
```

В данном случае были определены два элемента **price** и **author** и описано их содержимое с помощью маркера **PCDATA**. Это говорит о том, что элементы могут содержать любую информацию, с которой может работать программа-анализатор (**PCDATA** – parseable character data). Есть также маркеры **EMPTY** – элемент пуст и **ANY** – содержимое документа специально не описывается.

При описании элемента **<book>**, было указано, что он состоит из дочерних элементов **<price>** и **<author>**. Можно расширить это описание с помощью символов '+', '*', '?', используемых для указания количества вхождений элементов. Так, например,

```
<!ELEMENT book (price, author+, caption?)>
```

означает, что элемент **book** содержит один и только один элемент **price**, несколько (минимум один) элементов **author** и необязательный элемент **caption**. Если существует несколько вариантов содержимого элементов, то используется символ '|'. Например:

```
<!ELEMENT book (PCDATA | body)>
```

В данном случае элемент **book** может содержать либо дочерний элемент **body**, либо **PCDATA**.

Описание атрибутов

Атрибуты элементов описываются с помощью дескриптора **!ATTLIST**, внутри которого задаются имя атрибута, тип значения, дополнительные параметры:

```
<!ATTLIST article
      id ID #REQUIRED
      about CDATA #IMPLIED
      type (actual | review | teach )  'actual' ''>
```

В данном случае у элемента **<article>** определяются три атрибута: **id**, **about**, **type**. Существует несколько возможных значений атрибута, это:

CDATA – значением атрибута является любая последовательность символов;

ID – определяет уникальный идентификатор элемента в документе;

IDREF (**IDREFS**) – значением атрибута будет идентификатор (список идентификаторов), определенный в документе;

ENTITY (**ENTITES**) – содержит имя внешней сущности (несколько имен, разделенных запятыми);

NMTOKEN (**NMTOKENS**) – слово (несколько слов, разделенных пробелами).

Опционально можно задать значение по умолчанию для каждого атрибута. Значения по умолчанию могут быть следующими:

#REQUIRED – означает, что значение должно присутствовать в документе;

#IMPLIED – означает, что если значение атрибута не задано, то приложение должно использовать свое собственное значение по умолчанию;

#FIXED – означает, что атрибут может принимать лишь одно значение, то, которое указано в DTD.

Если в документе атрибуту не будет присвоено никакого значения, то его значение будет равно заданному в DTD.

Определение сущности

Сущность (entity) представляет собой некоторое определение, чье содержимое может быть повторно использовано в документе. Описывается сущность с помощью дескриптора **!ENTITY**:

```
<!ENTITY company 'Sun Microsystems'>
...
<sender>&company;</sender>
...
```

Программа-анализатор, которая будет обрабатывать файл, автоматически подставит значение `Sun Microsystems` вместо **&company**. В XML включено несколько внутренних определений:

&amplt – символ <;

&ampgt – символ >;

&ampamp – символ **&**;

&ampapos – символ апострофа ‘;

&ampquot – символ двойной кавычки “.

Кроме этого, есть внешние определения, которые позволяют включать содержимое внешнего файла:

```
<!ENTITY logotype SYSTEM "/image.gif" NDATA GIF87A>
```

Пусть существует XML-документ, содержащий данные адресной книги:

```
<?xml version="1.0"?>
<!DOCTYPE notepad SYSTEM "notepad.dtd">
<notepad>
      <note login="rom">
            <name>Valera Romanchik</name>
            <tel>217819</tel>
            <url>http://www.bsu.by</url>
            <address>
                  <street>Main Str., 35</street>
                  <city>Minsk</city>
                  <country>BLR</country>
            </address>
      </note>
      <note login="goch">
            <name>Igor Blinov</name>
            <tel>430797</tel>
            <url>http://bsu.iba.by</url>
            <address>
                  <street>Deep Forest, 7</street>
                  <city>Polock</city>
                  <country>VCL</country>
```

```
            </address>
      </note>
</notepad>
```

Тогда файл DTD для этого документа будет иметь вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT notepad (note+)>
<!ELEMENT note (name,tel,url,address)>
<!ELEMENT address (street,city,country)>
<!ATTLIST note login ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT url (#PCDATA)>
```

XML-анализаторы

Анализатор (parser) – самый важный инструмент при работе с XML. Каждое приложение, работающее с XML, использует анализатор, который представляет собой некоторый компонент, находящийся между приложением и файлами XML.

Валидирующие и невалидирующие анализаторы

Документы XML могут быть либо well-formed, либо valid. Документы well-formed составлены в соответствии с синтаксическими правилами построения XML-документов. Документы не только сформированы синтаксически правильно, но и следуют некоторой структуре, которая описана в DTD.

Соответственно есть валидирующие и невалидирующие анализаторы. И те, и другие проверяют XML-документ на соответствие синтаксическим правилам, но только валидирующие анализаторы знают, как проверить XML-документ на соответствие структуре, описанной в DTD.

Никакой связи между видом анализатора и видом XML-документа нет. Валидирующий анализатор может разобрать XML-документ, для которого нет DTD, и, наоборот, невалидирующий анализатор может разобрать XML-документ, для которого есть DTD. При этом он просто не будет учитывать описание структуры документа.

Древовидная и событийная модели

Существует два вида взаимодействия приложения и анализатора: использовать модель, основанную на представлении содержимого файла XML в виде дерева объектов, либо событийную модель.

Анализаторы, которые строят древовидную модель, – это DOM-анализаторы (Dynamic Object Model). Анализаторы, которые генерируют события, – это SAX-анализаторы (Simple API for XML).

В первом случае анализатор строит в памяти дерево объектов, соответствующее XML-документу. Далее вся работа ведется именно с этим деревом.

Во втором случае анализатор работает следующим образом: когда происходит анализ документа, анализатор генерирует события, связанные с различными участками XML-файла, а программа, использующая анализатор, решает, как реагировать на эти события. Так, анализатор будет генерировать событие о том, что

он встретил начало документа либо его конец, начало элемента либо его конец, символьную информацию внутри элемента и т.д.

Когда следует использовать DOM-, а когда – SAX-анализаторы?

DOM-анализаторы следует использовать тогда, когда нужно знать структуру документа и может понадобиться изменять эту структуру либо использовать информацию из XML-файла несколько раз.

SAX-анализаторы используются тогда, когда нужно извлечь информацию о нескольких элементах из XML-файла либо когда информация из документа нужна только один раз.

Событийная модель

Как уже отмечалось, SAX-анализатор не строит дерево элементов по содержимому XML-файла. Вместо этого анализатор читает файл и генерирует события, когда находит элементы, атрибуты или текст. На первый взгляд, такой подход менее естествен для приложения, использующего анализатор, так как он не строит дерево, а приложение само должно догадаться, какое дерево элементов описывается в XML-файле.

Однако нужно учитывать, для каких целей используются данные из XML-файла. Очевидно, что нет смысла строить дерево объектов, содержащее десятки тысячи элементов в памяти, если все, что необходимо, – это просто посчитать точное количество элементов в файле.

SAX-анализаторы и Java

SAX API определяет ряд событий, которые будут сгенерированы при разборе документов:

`startDocument` – событие, сигнализирующее о начале документа;

`endDocument` – событие, сигнализирующее о завершении документа;

`startElement` – данное событие будет сгенерировано, когда анализатор полностью обработает содержимое открывающего тега, включая его имя и все содержащиеся атрибуты;

`endElement` – событие, сигнализирующее о завершении элемента;

`characters` – событие, сигнализирующее о том, что анализатор встретил символьную информацию внутри элемента;

`warning`, **`error`**, **`fatalError`** – эти события сигнализируют об ошибках при разборе XML-документа.

В пакете **`org.xml.sax.helpers`** содержится класс **`DefaultHanlder`**, который содержит методы для обработки всех вышеуказанных событий. Для того чтобы создать простейшее приложение, обрабатывающее XML-файл, достаточно сделать следующее:

1. Создать класс, суперклассом которого будет **`DefaultHandler`**, и переопределить методы, отвечающие за обработку интересующих событий.
2. Создать объект-парсер класса **`org.xml.parsers.SAXParser`**.
3. Вызвать метод **`parse()`**, которому в качестве параметров передать имя разбираемого файла и экземпляр созданного на первом шаге класса.

Следующий пример выведет на консоль содержимое XML-документа. Вывод производится в ответ на события, генерируемые анализатором.

```
/* пример # 1 : чтение и вывод XML-документа :
DemoSAXParser.java */
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
class MyHandler extends DefaultHandler {
      public void startElement(String uri,
                  String localName, String qName,
                  Attributes attrs) {
                 String s = "";
      for (int i = 0; i < attrs.getLength(); i++) {
            s = attrs.getQName(i) + "="
            + attrs.getValue(i) + " ";
            }
            System.out.print(qName + " " + s.trim());
      }
      public void endElement(String uri,
            String localName, String qName) {
            System.out.print(qName);            }
      public void characters(char[] ch,
                              int start, int length) {
      System.out.print(new String(ch, start, length));
      }
}
public class DemoSAXParser {
      public static void main(String[] args) {
            try {
      SAXParser parser =
      SAXParserFactory.newInstance().newSAXParser();
      parser.parse("notepad.xml", new MyHandler());
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

В следующем примере производится разбор документа **notepad.xml**, и инициализация на его основе набора объектов.

```
/* пример # 2 : формирование коллекции объектов на основе
XML-документа : MyParserDemo.java */
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.net.URL;
import java.net.MalformedURLException;
```

```
import java.util.Vector;
interface ConstNote {
      int   NAME = 1,   TEL = 2,  URL = 3,
            STREET = 4, CITY = 5, COUNTRY = 6;
}
class DocHandler extends DefaultHandler
            implements ConstNote {
      Vector notes = new Vector();
      Note curr = new Note();
      int current = -1;
      public Vector getNotes() {
            return notes;
      }
      public void startDocument() {
            System.out.println("parsing started");
      }
      public void endDocument() {
            System.out.print("");
      }
      public void startElement(String uri,
                  String localName, String qName,
                  Attributes attrs) {
            if (qName.equals("note")) {
                  curr = new Note();
      curr.setLogin(attrs.getValue(0));
            }
            if (qName.equals("name"))
                  current = NAME;
            else if (qName.equals("tel"))
                  current = TEL;
            else if (qName.equals("url"))
                  current = URL;
            else if (qName.equals("street"))
                  current = STREET;
            else if (qName.equals("city"))
                  current = CITY;
            else if (qName.equals("country"))
                  current = COUNTRY;
      }
      public void endElement(String uri,
             String localName, String qName) {
            if (qName.equals("note"))
                  notes.add(curr);
      }
      public void characters(char[] ch,
                        int start, int length) {
            String s = new String(ch, start, length);
            try {
```

```
                    switch (current) {
                    case NAME:
                          curr.setName(s);
                          break;
                    case TEL:
                          curr.setTel(Integer.parseInt(s));
                          break;
                    case URL:
                          try {
                          curr.setUrl(new URL(s));
                          } catch (MalformedURLException e) {
                          }
                          break;
                    case STREET:
                          curr.address.setStreet(s);
                          break;
                    case CITY:
                          curr.address.setCity(s);
                          break;
                    case COUNTRY:
                          curr.address.setCountry(s);
                          break;
                    }
              } catch (Exception e) {
                    System.out.println(e);
              }
      }
}
public class MyParserDemo {
      public static void main(String[] args) {
            try {
            SAXParser parser =
SAXParserFactory.newInstance().newSAXParser();
                    DocHandler dh = new DocHandler();
                    Vector v;
                    if (dh != null)
                    parser.parse("notepad.xml", dh);
                    v = dh.getNotes();
                    for (int i = 0; i < v.size(); i++)
System.out.println(((Note) v.elementAt(i)).toString());
              } catch (Exception e) {
                    e.printStackTrace();
              }
      }
}
```

В результате на консоль будет выведена следующая информация:

```
parsing started
```

```
rom
 Valera Romanchik 217819 http://www.bsu.by
      address:Main Str., 35 Minsk BLR
goch
 Igor Blinov 430797 http://bsu.iba.by
      address:Deep Forest, 7 Polock VCL
```

Класс, объект которого формируется на основе информации из XML-документа, имеет следующий вид:

```
/* пример # 3 : класс сущности : Note.java */
import java.net.URL;
class Note {
      private String name, login;
      private int tel;
      private URL url;
      public Address address = new Address();
      public void setAddress(Address address) {
            this.address = address;
      }
      public void setLogin(String login) {
            this.login = login;
      }
      public void setName(String name) {
            this.name = name;
      }
      public void setTel(int tel) {
            this.tel = tel;
      }
      public String toString() {
            return login + " " + name + " " + tel + " "
      + url + "\n\t address:" + address.street + " "
      + address.city + " " + address.country;
      }
      class Address {
            String street, city, country;
            public void setCity(String city) {
                  this.city = city;
            }
            public void setCountry(String state) {
                  this.country = state;
            }
            public void setStreet(String street) {
                  this.street = street;
            }
      }
      public void setUrl(URL url) {
            this.url = url;
      }
```

```
}
```

Древовидная модель

DOM (Dynamic object model) представляет собой некоторый общий интерфейс для работы со структурой документа. Одна из целей разработки заключалась в том, чтобы код, написанный для работы с каким-либо DOM-анализатором, мог работать и с любым другим DOM-анализатором.

DOM-анализатор строит дерево, которое представляет содержимое XML-документа, и определяет набор классов, которые представляют каждый элемент в XML-документе (элементы, атрибуты, сущности, текст и т.д.).

В Java включена поддержка DOM. В пакете **`org.w3c.dom`** можно найти интерфейсы, которые представляют вышеуказанные объекты. Реализацией этих интерфейсов занимаются разработчики анализаторов. Разработчики приложений, которые хотят использовать DOM-анализатор, имеют готовый набор методов для манипуляции деревом объектов и не зависят от конкретной реализации используемого анализатора.

Node

Основным объектом DOM является **`Node`** – некоторый общий элемент дерева. Большинство DOM-объектов унаследовано именно от **`Node`**. Для представления элементов, атрибутов, сущностей разработаны свои специализации **`Node`**.

`Node` определяет ряд методов, которые используются для работы с деревом:

`getNodeType()` – возвращает тип объекта (элемент, атрибут, текст, **`CDATA`** и т.д.);

`getParentNode()` – возвращает объект, являющийся родителем текущего узла **`Node`**;

`getChildNodes()` – возвращает список объектов, являющихся дочерними элементами;

`getFirstChild()`, **`getLastChild()`** – возвращает первый и последний дочерние элементы;

`getAttributes()` – возвращает список атрибутов данного элемента.

Attr, Element, Text

Данные интерфейсы унаследованы от интерфейса **`Node`** и используются для работы с конкретными объектами дерева.

Document

Используется для получения информации о документе и изменения его структуры. Это интерфейс представляет собой корневой элемент XML-документа и содержит методы доступа ко всему содержимому документа.

В следующем примере производится разбор документа **`notepad.xml`** с использованием DOM-анализатора и инициализация на его основе набора объектов. При этом используется анализатор XML4J от IBM.

```
// пример # 4 : создание объектов на основе XML: MyDOMDemo.java
import org.w3c.dom.Element;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

```
import org.w3c.dom.Text;
import org.apache.xerces.parsers.DOMParser;
import java.net.URL;
import java.util.Vector;
public class MyDOMDemo {
    public static String getValue(Element e, String name) {
            NodeList nList = e.getElementsByTagName(name);
            Element elem = (Element) nList.item(0);
            Text t = (Text) elem.getFirstChild();
            return t.getNodeValue();
      }
      public static void main(String[] args) {
            Document doc = null;
            DOMParser parser = new DOMParser();
            Vector entries = new Vector();
            try {
                  parser.parse("notepad.xml");
                  doc = parser.getDocument();
                  Element root = doc.getDocumentElement();
                  NodeList noteList =
                        root.getElementsByTagName("note");
                  Element noteElem;
      for (int i = 0; i <noteList.getLength(); i++) {
            noteElem = (Element) noteList.item(i);
            Note e = new Note();
            NodeList list = noteElem.getChildNodes();
            Node log =
                        noteElem.getAttributes().item(0);
            e.setLogin(log.getNodeValue());
            e.setName(getValue(noteElem, "name"));
e.setTel(Integer.parseInt(getValue(noteElem, "tel")));
            e.setUrl(new URL(getValue(noteElem, "url")));
                  Element n =
(Element)noteElem.getElementsByTagName("address").item(0);
            e.address.setStreet(getValue(n, "street"));
            e.address.setCountry(getValue(n, "country"));
            e.address.setCity(getValue(n, "city"));
                  entries.add(e);
                  }
            } catch (Exception e) {
                  System.out.println(e);
            }
            for (int i = 0; i < entries.size(); i++)
            System.out.println(
            ((Note) entries.elementAt(i)).toString());
      }
}
```

XML-документы можно не только читать, но и корректировать.

```
/* пример # 5 : замена информации в файле XML :
JDOMChanger.java */
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;
import java.util.*;
import java.io.FileOutputStream;
public class JDOMChanger {
      static void lookForElement(String name,
String element, String content, String login) {
            SAXBuilder builder = new SAXBuilder();
            try {
                  Document document = builder.build(name);
                  Element root = document.getRootElement();
                  List c = root.getChildren();
                  Iterator i = c.iterator();
                  while (i.hasNext()) {
                  Element e = (Element) i.next();
      if (e.getAttributeValue("login").equals(login)) {
                  e.getChild(element).setText(content);
                        }
                  }
            XMLOutputter serializer = new XMLOutputter();
serializer.output(document, new FileOutputStream(name));
                  System.out.flush();
            } catch (Exception e) {
                  System.out.println(e);
            }
      }
      public static void main(String[] args) {
            String name = "notepad.xml";
FieldChanger.lookForElement(name, "tel", "09", "rom");
      }
}
```

В этом примере использован DOM-анализатор JDOM основанный на идее "if something doesn't work, fix it"

XSL

XML используется для представления информации в виде некоторой структуры, но никоим образом не указывает, как отображать XML-документ. Для того чтобы просмотреть XML-документ, нужно его каким-то образом отформатировать.

Инструкции форматирования XML-документов формируются в так называемые таблицы стилей, и для просмотра XML-документа нужно обработать XML-файл согласно этим инструкциям.

Существует два стандарта стилевых таблиц, опубликованных W3C. Это CSS (Cascading Stylesheet) и XLS (XML Stylesheet Language).

CSS изначально разрабатывался для HTML и представляет из себя набор инструкций, которые указывают браузеру, какой шрифт, размер, цвет использовать для отображения элементов HTML-документа.

XSL более современен, чем CSS, потому что используется для преобразования XML-документа перед отображением. Так, используя XSL, можно построить оглавление для XML-документа, представляющего книгу.

Вообще XSL можно разделить на две части: XSLT (XSL Transformation) и XSLFO (XSL Formatting Objects).

Для того чтобы XML-документ преобразовать согласно инструкциям, находящимся в файле таблицы стилей, необходим XSL Processor.

XSLT

Язык для описания преобразований XML-документа. XSLT используется не только для приведения XML-документов к некоторому "читаемому" виду, но и для изменения структуры XML-документа.

К примеру, XSLT можно использовать для:

- добавления новых элементов в XML-документ;
- создания нового XML-документа на основании заданного (список имен адресной книги);
- предоставления информации из XML-документа с разной степенью детализации;
- преобразования XML-документа в документ HTML.

Пусть требуется построить новый HTML-файл на основе файла **`notepad.xml`**, который в виде таблицы будет выводить **`login`**, **`name`** и **`street`** для каждой записи, присутствующей в адресной книге. Следует воспользоваться XSLT для решения данной задачи. В следующем коде приведено содержимое файла таблицы стилей, который решает поставленную проблему.

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="/">
<html>
<head><title>Notepad Contents</title></head>
<body>
<table border="1">
    <tr>
        <th>Login</th>
        <th>Name</th>
        <th>Street</th>
    </tr>
<xsl:for-each select="notepad/note">
<tr>
<td><xsl:value-of select="@login"/></td>
<td><xsl:value-of select="name"/></td>
<td><xsl:value-of select="address/street"/></td>
</tr>
</xsl:for-each>
```

```
</table>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

Для трансформации одного документа в другой можно использовать, например, следующий код.

```
/* пример # 6 : трансформация XML в HTML :
SimpleTransform.java */
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
public class SimpleTransform {
      public static void main(String[] args) {
            try {
            TransformerFactory tFact =
                  TransformerFactory.newInstance();
            Transformer transformer =
tFact.newTransformer(new StreamSource("notepad.xsl"));
                  transformer.transform(
                  new StreamSource("notepad.xml"),
                  new StreamResult("notepad.html"));
            } catch (TransformerException e) {
                  e.printStackTrace();
            }
      }
}
```

В результате получится HTML-документ следующего вида:

```
<html><head>
<META http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Notepad Contents</title>
</head>
      <body>
      <table border="1">
<tr>
      <th>Login</th><th>Name</th><th>Street</th>
</tr>
<tr>
<td>rom</td><td>Valera Romanchik</td><td>Main Str., 35</td>
</tr>
<tr>
<td>goch</td><td>Igor Blinov</td><td>Deep Forest, 7</td>
</tr>
```

```
</table></body></html>
```

Элементы таблицы стилей

Таблица стилей представляет собой well-formed XML-документ. Эта таблица описывает изначальный документ, конечный документ и то, как трансформировать начальный документ в конечный.

Какие же элементы используются в данном листинге?

```
<xsl:output method="xml" indent="yes"/>
```

Данная инструкция говорит о том, что конечный документ, который получится после преобразования, будет являться XML-документом.

```
<xsl:template match="notepad">
      <names>
            <xsl:apply-templates/>
      </names>
</xsl:template>
```

Инструкция **<xsl:template...>** задает шаблон преобразования. Набор шаблонов преобразования составляет основную часть таблицы стилей. В предыдущем примере приводится шаблон, который преобразует элемент **notepad** в элемент **names**.

Шаблон состоит из двух частей:

1) Параметр **match**, который задает элемент или множество элементов в исходном дереве, к которым будет применяться данный шаблон;
2) Содержимое шаблона, которое будет вставлено в конечный документ.

Нужно отметить, что содержимое параметра **math** может быть довольно сложным. В предыдущем примере просто ограничились именем элемента. Но, к примеру, следующее содержимое параметра **math** указывает на то, что шаблон должен применяться к элементу **url**, содержащему атрибут **protocol** со значением **mailto**:

```
<xsl:template match="url[@protocol='mailto']">
```

Кроме этого, существует набор функций, которые также могут использоваться при объявлении шаблона:

```
<xsl:template match="chapter[position()=2]">
```

Данный шаблон будет применен ко второму по счету элементу **chapter** исходного документа.

Инструкция **<xsl:apply-templates/>** сообщает XSL-процессору о том, что нужно перейти к просмотру дочерних элементов.

XSL-процессор работает по следующему алгоритму. После загрузки исходного XML-документа и таблицы стилей процессор просматривает весь документ от корня до листьев. На каждом шагу процессор пытается применить к данному элементу некоторый шаблон преобразования; если в таблице стилей для текущего просматриваемого элемента есть шаблон, процессор вставляет в результирующий документ содержимое этого шаблона. Когда процессор встречает инструкцию **<xsl:apply-templates/>**, он переходит к дочерним элементам текущего

узла и повторяет процесс, т.е. пытается для каждого дочернего элемента найти соответствие в таблице стилей.

Проверка документа

С помощью DTD и схемы XSD можно проверить документ на корректность. Схема XSD представляет собой более строгое описание XML-документа, чем DTD. Для адресной книги XML-схема **notepad.xsd** выглядит следующим образом.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
    <xs:element name="notepad">
    <xs:complexType>
       <xs:sequence>
         <xs:element ref="note" minOccurs='1'
                  maxOccurs='unbounded'/>
       </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="note">
            <xs:complexType>
                <xs:sequence>
                        <xs:element ref="name"/>
                        <xs:element ref="tel" />
                        <xs:element ref="url" />
                        <xs:element ref="address"/>
                </xs:sequence>
                <xs:attribute name="login" type="xs:ID"
                        use='required'/>
            </xs:complexType>
        </xs:element>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="tel" type="xs:int" />
    <xs:element name="url" type="xs:anyURI"/>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
    <xs:element name="address">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="street" />
                <xs:element ref="city" />
                <xs:element ref="country" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
   </xs:schema>
```

Тогда для проверки документа объекту-парсеру следует дать указание использовать DTD и схему XSD и в XML-документ вместо ссылки на DTD добавить к корневому элементу атрибуты вида:

```
<notepad xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:noNamespaceSchemaLocation='notepad.xsd'>
```

Следующий пример выполняет проверку документа.

```
/* пример # 7 : проверка корректности документа XML :
Validation.java */
import org.w3c.dom.Document;
import org.apache.xerces.parsers.DOMParser;
public class Validation {
      public static void main(String[] args) {
            String filename = "notepad.xml";
            DOMParser parser = new DOMParser();
      //установка обработчика ошибок
            parser.setErrorHandler(new MyErrorHandler());
            try {
      //установка способа проверки с использованием DTD
      parser.setFeature(
      "http://xml.org/sax/features/validation", true);
      //установка способа проверки с использованием XSD
      parser.setFeature(
      "http://apache.org/xml/features/validation/schema",
      true);
                  parser.parse(filename);
                  Document doc = parser.getDocument();
            } catch (Exception e) {
                  System.out.println(e);
            }
            System.out.print("проверка " + filename
                                    + " завершена");
      }
   }
```

Класс обработчика ошибок может выглядеть следующим образом.

```
/* пример # 8 : обработчик ошибок : MyErrorHandler.java */
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;
public class MyErrorHandler implements ErrorHandler {
      public void warning(SAXParseException e) {
            System.out.println(getLineAddress(e) + " - "
                        + e.getMessage());
      }
      public void error(SAXParseException e) {
            System.out.println(getLineAddress(e) + " - "
                        + e.getMessage());
      }
      public void fatalError(SAXParseException e) {
```

```
            System.out.println(getLineAddress(e) + " - "
                        + e.getMessage());
      }
      private String getLineAddress(SAXParseException e) {
            //определение строки и столбца ошибки
return e.getLineNumber() + " : " + e.getColumnNumber();
      }
   }
```

Для того чтобы убедиться в работоспособности кода следует внести в исходный XML-документ ошибку. Например, сделать идентичными значения атрибута **login**. Тогда в результате запуска на консоль будет выведено следующее сообщение обработчика об ошибке вида:

```
14 : 22 - Datatype error: ID 'goch'  has to be unique.
   проверка notepad.xml завершена
```

ПРИЛОЖЕНИЕ 3

Введение в технологию Struts

Проект Struts был запущен в мае 2000 г. К. Макклэнаханом (Craig R. McClanahan) для обеспечения возможности разработки приложений с архитектурой, базирующейся на парадигме Model/View/Controller. В июле 2001 г. был выпущен первый релиз Struts 1.0. Struts является частью проекта Jakarta, поддерживаемого Apache Software Foundation. Цель проекта Jakarta Struts – разработка среды с открытым кодом для создания Web-приложений с помощью технологий Java Servlet and JSP.

Шаблон проектирования MVC (Model-View-Controller)

При использовании шаблона MVC поток выполнения приложения всегда обязан проходить через контроллер приложения. Контроллер направляет запросы – в данном случае HTTP(S)-запросы – к соответствующему обработчику. Обработчики запроса связаны с бизнес-моделью, и в итоге каждый разработчик приложения должен только обеспечить взаимодействие между запросом и бизнес-моделью. В результате реакции системы на запрос вызывается соответствующая JSP-страница, выполняющая в данной схеме роль представления.

В результате модель отделена от представления, и все связывающие их команды проходят через контроллер приложения. Приложение, соответствующее этим принципам, становится более понятным с точки зрения разработки, поддержки и совершенствования, а отдельные его части довольно легко могут быть использованы повторно.

Состав Struts

Согласно шаблону Model/View/Controller, Struts имеет три основных компонента: сервлет-контроллер, который входит в Struts, JSP-страницы и бизнес-логику приложения.

Сервлет-контроллер Struts определяет и направляет HTTP-запрос к соответствующему запрашиваемой JSP-странице наследнику класса
`org.apache.struts.action.Action`,
который создается разработчиком приложения. Во время инициализации контроллер считывает (parse) конфигурационный файл **`struts-config.xml`** который однозначно определяет все соответствия и альтернативы
`org.apache.struts.action.ActionMappings`
для всех запросов данного приложения.

JSP-страница представления состоит, как правило, из статического HTML-кода, а также из динамического содержания, основанного на инициализации в запросе специальных тегов действий (**`action tags`**). Среда Struts включает большой и разнообразный набор стандартных тегов действий, назначение которых описано в спецификации Struts. Кроме того, существует стандартное средство для определения своих собственных тегов.

Модель инкапсулирует бизнес-логику приложения. Управление обычно передается обратно через контроллер соответствующему представлению, этот вызов соответствует запросу JSP-страницы. Перенаправление осуществляется путем

обращения к набору соответствий (mappings) между бизнес-моделью и представлением. Предложенная схема обеспечивает слабое связывание (Low Coupling) между представлением и бизнес-моделью, что делает разработку и поддержку приложения значительно проще.

Приведенный в главе 19 пример распределенного приложения входа в систему с проверкой логина и пароля можно легко переписать с использованием технологии Struts.

```
/* пример # 1 : Action класс : LoginAction.java */
package test.struts;
import java.io.IOException;
import javax.servlet.*;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
public class LoginAction extends Action {
  public ActionForward perform(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response)
                  throws IOException, ServletException {
        //обработка запросов
    ActionErrors errors = new ActionErrors();
    try {
      LoginForm actionForm = (LoginForm)form;
      String cmd = actionForm.getCmd();
      if ("login".equals(cmd)) {
        //процесс логина в систему
        String login = actionForm.getLogin();
        String password = actionForm.getPassword();
        LoginLogic loginLogic = new LoginLogic();
        if (loginLogic.checkLogin(login, password)) {
          //обработка успешного входа в систему
               request.setAttribute("user", login);
          return mapping.findForward("success");
        } else {
   //обработка ситуации неверного логина или пароля
          errors.add(ActionErrors.GLOBAL_ERROR,
new ActionError("error.login.incorrectLoginOrPassword"));
          return mapping.findForward("back");
        }
      } else {
          //загрузка формы для логина
        return mapping.findForward("back");
      }
    } finally {
```

```
      if (errors != null && !errors.empty()) {
        //сохранение ошибок в объекте запроса
           saveErrors(request, errors);
      }
    }
  }
}
```

Класс **LoginForm**, объект которого представляет форму, соответствующую странице ввода логина и пароля, выглядит следующим образом:

```
/* пример # 2 : класс хранения информации, передаваемой
Login.jsp : LoginForm.java */
package test.struts;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
public class LoginForm extends ActionForm {
  private String login;
  private String password;
  private String cmd;
 //очищает поля формы
 public void reset(ActionMapping mapping,
                   HttpServletRequest request) {
    super.reset(mapping, request);
    this.password = null;
    this.login = null;
    this.cmd = null;
  }
  public void setCmd(String cmd) {
    this.cmd = cmd;
  }
  public void setLogin(String login) {
    this.login = login;
  }
  public void setPassword(String password) {
    this.password = password;
  }
  public String getCmd() {
    return cmd;
  }
  public String getLogin() {
    return login;
  }
  public String getPassword() {
    return password;
  }
}
```

Класс **LoginLogic**, представляющий бизнес-логику, не был изменен. Его необходимо перенести в соответствующую папку проекта Struts. JSP-страницы были изменены следующим образом:

```
<%--пример # 3 : стартовая страница в оформлении Struts :
Login.jsp --%>
<%@ page errorPage="Error.jsp" %>
<%-- Подключение библиотек пользовательских тегов --%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:html locale="true">
<head>
<%-- вывод сообщения из .properties проекта --%>
      <title><bean:message key="jsp.Login.title"/></title>
<%-- генерируется тег <base> с атрибутом href, в котором
будет URL текущего application. При этом все относительные
ссылки на странице ответа отсчитываются от этого значения
--%>
      <html:base />
</head>
<body>
<%-- пользовательский тег для отображения ошибок списком,
открывающие/закрывающие значения которого берутся из файла
ресурсов --%>
<html:errors/>
<h3><bean:message key="jsp.Login.header"/></h3>
<hr><%-- форма для логина --%>
<html:form  action="/login" method="POST">
<%-- скрытый параметр, указывающий, что происходит вход в
систему --%>
      <html:hidden property="cmd" value="login"/>
      <bean:message key="jsp.Login.field.login"/>:<br>
      <%-- текстовое поле для имени пользователя --%>
      <html:text property="login"/><br>
      <bean:message key="jsp.Login.field.password"/>:<br>
      <%-- текстовое поле для пароля пользователя --%>
      <html:password property="password"
                        redisplay="false"/><br>
      <html:submit><bean:message
       key="jsp.Login.button.submit"/> </html:submit>
</html:form>
<hr>
</body>
</html:html>
```

Следует обратить внимание на пользовательские теги вида `<bean:message key="jsp.Login.header"/>`, которые выводят текст, предварительно размещенный в специальном файле **`ApplicationResources.properties`**. В этом файле хранятся все статические текстовые данные, которые извлекаются по ключу, указанному в свойстве пользовательского тега **`key`**. Например, в **`LoginAction`** при создании объектов ошибок в их конструкторе был указан ключ, по которому в этом файле берется текст ошибки:

```
errors.add(ActionErrors.GLOBAL_ERROR, new ActionError
("error.login.incorrectLoginOrPassword"));
```

Для данного приложения файл ресурсов может быть создан в виде:

```
# пример # 4 : файл ресурсов :
# ApplicationResources.properties
# header и footer, которые будут использоваться для
# обрамления ошибок, выдаваемых тегом <errors/>.
errors.header=<ul>
errors.footer=</ul>
# разметка для элемента списка ошибок из <errors/>,
# который указывает, что логин или пароль неверны.
error.login.incorrectLoginOrPassword=<li>incorrect login or
password.</li>

# текстовая информация на Login.jsp
jsp.Login.title=Login
jsp.Login.header=Login
jsp.Login.field.login=Login
jsp.Login.field.password=Password
jsp.Login.button.submit=Enter

# текстовая информация на Main.jsp
jsp.Main.title=Welcome
jsp.Main.header=Welcome

# текстовая информация на Error.jsp
jsp.Error.title=Error
jsp.Error.header=Error
jsp.Error.returnToMain=Return to login page.
```

Имя и месторасположение этого файла настраиваются в **`struts-config.xml`**, который является основным конфигурационным файлом для приложения, построенного на основе Struts.

```
пример # 5 : конфигурация action, forward, resource и т.д.
: struts-config.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software
Foundation//DTD Struts Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>

      <!—источники данных к БД (если есть)-->
      <data-sources>
      </data-sources>
      <!-- Action формы-->
            <form-beans>
      <form-bean name="loginForm"
                  type="test.struts.LoginForm">
            </form-bean>
      </form-beans>
      <!-- глобальные форварды (если есть) -->
```

```
      <global-forwards>
      </global-forwards>
      <!-- настройки action -->
      <action-mappings>
            <!-- Action для процесса логина -->
            <action name="loginForm"
                  path="/login"
                  scope="request"
                  type="test.struts.LoginAction"
                  validate="false">
<!-- форварды на JSP, доступные из данного action -->
      <forward name="success" path="/Main.jsp">
                  </forward>
      <forward name="back" path="/Login.jsp">
                  </forward>
            </action>
      </action-mappings>
<!-- Message Resources -->
<!-- имя файла ресурсов .properties, в котором будет хранить-
ся вся статическая текстовая информация для приложения -->
      <message-resources parameter=
"resources.ApplicationResources"/>
</struts-config>
```

В теге **<action>** при помощи параметра **path** со значением **/login** связываются страница JSP и класс **ActionForm**, ей соответствующий. Это значение указывается в **Login.jsp** в теге **FORM ACTION**. Далее прописывается путь к сервлету, в который и будет передана вся информация, извлеченная из запроса. В теге **<forward>** размещаются имена, в частности `success` и `back`, ассоциированные с путями вызова страниц **/Login.jsp** и **/Main.jsp** соответственно. Указанные имена передаются в метод **findForward()** класса **ActionMapping** в качестве параметра при различных вариантах завершения работы сервлета.

Файл **web.xml** в проекте также обязательно присутствует и должен указывать на месторасположение главного сервлета **ActionServlet** и файла **struts-config.xml**, а также указывать **servlet-mapping** для контроллера.

```
пример # 6 : конфигурационный файл приложения : web.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Web Application 2.3//EN" "http://java.sun.com/dtd/web-
app_2_3.dtd">
<web-app id="WebApp">
      <display-name>Study</display-name>
      <!-- настройка стандартного контроллера Struts –
сервлета ActionServlet  -->
      <servlet>
            <servlet-name>action</servlet-name>
```

```
            <servlet-class>
org.apache.struts.action.ActionServlet</servlet-class>
            <!-- месторасположение основного настроечного
файла Struts – struts-config.xml  -->
            <init-param>
            <param-name>config</param-name>
<param-value>WEB-INF/struts-config.xml</param-value>
            </init-param>
            <init-param>
                  <param-name>debug</param-name>
                  <param-value>2</param-value>
            </init-param>
            <init-param>
                  <param-name>detail</param-name>
                  <param-value>2</param-value>
            </init-param>
            <init-param>
                  <param-name>validate</param-name>
                  <param-value>true</param-value>
            </init-param>
            <load-on-startup>1</load-on-startup>
      </servlet>
      <!-- соответствие (mapping) для стратсовского кон-
троллера, указывающее, что контроллер будет вызываться, ес-
ли адрес заканчивается на .do -->
      <servlet-mapping>
          <servlet-name>action</servlet-name>
          <url-pattern>*.do</url-pattern>
      </servlet-mapping>
<!-- подключение стандартных struts-ских библиотек пользо-
вательских тегов (библиотек может быть больше для поздних
версий) -->
      <taglib>
<taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
<taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
      </taglib>
      <taglib>
<taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
<taglib-location>/WEB-INF/struts-html.tld</taglib-location>
      </taglib>
      <taglib>
    <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
<taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
      </taglib>
</web-app>
```

Оставшиеся JSP-страницы были изменены несущественно:

```
<%-- пример # 7 : страница, вызываемая после прохождения
идентификации : Main.jsp --%>
<%@ page errorPage="Error.jsp" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:html locale="true">
<head>
      <title><bean:message key="jsp.Main.title"/></title>
      <html:base/>
</head>
<body>
<html:errors/>
<h3><bean:message key="jsp.Main.header"/>:
<%=(String) request.getAttribute("user")%></h3>
</body>
</html:html>
```

Страница ошибок, к которой осуществляется переход в случае возникновения исключений, модернизирована следующим образом:

```
<%--пример # 8 : страница, вызываемая при ошибке :
Error.jsp --%>
<%@ page isErrorPage="true" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:html locale="true">
<head>
      <title><bean:message key="jsp.Error.title"/></title>
      <html:base/>
</head>
<body>
<h3><bean:message key="jsp.Error.header"/></h3>
<hr>
<%=(exception != null)? exception.toString() :
"unknown error"%>
<hr>
<a href="login.do"><bean:message
                   key="jsp.Error.returnToMain"/></a>
</body>
</html:html>
```

Кроме того, чтобы использовать технологию Struts, необходимо подключить **`struts.jar`** с Java-кодом движка и библиотеки пользовательских тегов:

```
struts-bean.tld
struts-html.tld
struts-logic.tld
```

Чтобы запустить проект, следует вызвать из браузера:

```
http://localhost:8080/Study/login.do
```

Содержание

ЧАСТЬ 3. ТЕХНОЛОГИЯ РАЗРАБОТКИ WEB-ПРИЛОЖЕНИЙ

Краткое содержание

Список источников

1. К. Арнольд, Дж. Гослинг, Д. Холмс. Язык программирования Java. 3-е изд. – М: “Вильямс”, 2001. – 624 с.
2. Б. Эккель. Философия Java. – СПб: Питер, 2001. – 880 с.
3. Я. Ф. Дарвин. Java. Сборник рецептов для профессионалов. – СПб.: Питер, 2002. – 768 с.
4. К. С. Хорстманн, Г. Корнелл. Библиотека профессионала. Java 2. Том 1. Основы. – М.: “Вильямс”, 2004. – 848 с.
5. К. С. Хорстманн, Г. Корнелл. Библиотека профессионала. Java 2. Том 2. Тонкости программирования. – М.: “Вильямс”, 2002. – 1120 с.
6. М. Холл. Сервлеты и JavaServer Pages. Библиотека программиста. – СПб.: Питер, 2001. – 496 с.
7. Sun Developer Network Site. http://java.sun.com/
8. The World Wide Web Consortium. http://w3c.org

Производственно-практическое издание

БЛИНОВ Игорь Николаевич
РОМАНЧИК Валерий Станиславович

JAVA 2

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Ответственный за выпуск *В.М. Стрельченя*
Технический редактор *Н.Е. Куркова*
Компьютерный набор *А.И. Блинов, А.И. Блинова*
Компьютерная верстка *Н.Е. Куркова*
Дизайн обложки *М.В. Голенкова*

Подписано в печать с готовых диапозитивов 19.04.2005 г.
Формат 70 x 100 1/16. Бумага офсетная. Гарнитура Times.
Печать офсетная. Усл. печ. л. 32,5. Уч. изд. л. 13,48.
Тираж 1010 экз. Заказ

Издательское УП «УниверсалПресс»
ЛИ № 02330/056977 от 30.04.2004 г.
Республика Беларусь, 220039, г. Минск, ул. Брилевская, д. 3, оф. 1.
Контактный телефон (017) 224-89-15.

Отпечатано с диапозитивов заказчика
в типографии ООО «Поликрафт»,
Лицензия № 02330/0148704 от 30.04.2004 г.
Республика Беларусь, г. Минск, ул. Я. Колоса, 73.