И.Н.БЛИНОВ
В.С.РОМАНЧИК

Java

ПРОМЫШЛЕННОЕ ПРОГРАММИРОВАНИЕ

Java SE 6
Объектно-ориентированный подход
Шаблоны проектирования GoF и GRASP
Конкурирующие операции
Сервлеты и JSP
JSTL
Struts
Hibernate
Портлеты
ANT
JDBC и SQL
UML
HTML и JavaScript
JUnit

УНИВЕРСАЛПРЕСС

И.Н. Блинов, В.С. Романчик

Java.

ПРОМЫШЛЕННОЕ ПРОГРАММИРОВАНИЕ

Практическое пособие

Минск
«УниверсалПресс»
2007

УДК 004.434
ББК 32.973.26-018.2
Б69

А в т о р ы :

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

Р е ц е н з е н т ы :

доцент, кандидат физико-математических наук И.М. Галкин,
доцент, кандидат технических наук А.Е. Люлькин

Блинов, И.Н.

Б69 Java. Промышленное программирование : практ. пособие / И.Н. Блинов, В.С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с.

ISBN 978-985-6699-63-7

Пособие предназначено для программистов, начинающих и продолжающих изучение технологий Java SE 6 и J2EE. В его первой части рассматриваются основы языка Java и концепции объектно-ориентированного программирования. Во второй части изложены важнейшие аспекты применения библиотек классов языка Java, включая файлы, коллекции, сетевые и многопоточные приложения, а также взаимодействие с XML. В третьей части приведены основы программирования распределенных информационных систем с применением сервлетов, JSP, баз данных и собственных тегов разработчика.

В конце каждой главы даются тестовые вопросы по материалу главы и задания для выполнения. В приложениях приведены дополнительные материалы, относящиеся к использованию HTML, XML, JavaScript, а также краткое описание популярных технологий Struts и Hibernate для разработки распределенных систем, объединяющих возможности J2EE и XML.

УДК 004.434
ББК 32.973.26-018.2

ISBN 978-985-6699-63-7

КРАТКОЕ СОДЕРЖАНИЕ

СОДЕРЖАНИЕ

Часть 2. ИСПОЛЬЗОВАНИЕ КЛАССОВ И БИБЛИОТЕК

Часть 3. ТЕХНОЛОГИИ РАЗРАБОТКИ WEB-ПРИЛОЖЕНИЙ

ПРЕДИСЛОВИЕ

Пособие расширяет и включает переработанную и обновленную версию предыдущей книги авторов «Java 2. Практическое руководство», изданную в 2005 г. Рассмотренный материал относится к программированию на Java SE 6 и J2EE.

Книга написана на основе учебных материалов, использующихся в процессе обучения студентов механико-математического факультета и факультета прикладной математики и информатики Белгосуниверситета, а также слушателей курсов повышения квалификации и преподавательских тренингов EPAM Systems, Sun Microsystems и других учебных центров по ряду направлений технологий Java. При изучении Java знание других языков необязательно, книгу можно использовать для обучения программированию на языке Java «с нуля».

Интересы авторов, направленные на обучение, определили структуру этой книги. Она предназначена как для начинающих изучение Java-технологий, так 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 этому курсу успешно сдают различные экзамены, получают международные сертификаты и в состоянии участвовать в командной разработке промышленных программных проектов.

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

В конце каждой главы даются тестовые вопросы по материалам 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 в информационных системах, основанных на применении Java-технологий, краткое описание портальных приложений и популярных технологий Struts и Hibernate для разработки информационных систем, а также Apache Ant для сборки этих приложений.

В создании некоторых приложений участвовали сотрудники EPAM Systems: приложение «UML» написано совместно с Валерием Масловым; приложение «Базы данных и язык SQL» написано Тимофеем Савичем; корректировка главы «XML&Java» и приложений «Struts» и «Hibernate» выполнена Сергеем Волчком; корректировка приложения «Apache Ant» и раздел «Основные понятия ООП» выполнены Евгением Пешкуром; приложение «JavaScript» создано при участии Александра Чеушева.

В разработке примеров принимали участие студенты механико-математического факультета и факультета прикладной математики и информатики БГУ. Авторы благодарны всем, принимавшим участие в подготовке этой книги.

Часть 1.
ОСНОВЫ ЯЗЫКА JAVA

В первой части книги излагаются вопросы, относящиеся к основам языка Java и технологии объектно-ориентированного программирования.

Глава 1
ВВЕДЕНИЕ В КЛАССЫ И ОБЪЕКТЫ

Основные понятия ООП

Возможности программирования всегда были ограничены либо возможностями компьютера, либо возможностями человека. В прошлом веке главным ограничением были низкие производительные способности компьютера. В настоящее время физические ограничения отошли на второй план. Со всё более глубоким проникновением компьютеров во все сферы человеческой деятельности, программные системы становятся всё более простыми для пользователя и сложными по внутренней архитектуре. Программирование стало делом команды и на смену алгоритмическим идеологиям программирования пришли эвристические, позволяющие достичь положительного результата различными путями.

Базовым способом борьбы со сложностью программных продуктов стало объектно-ориентированное программирование (ООП), являющееся в настоящее время наиболее популярной парадигмой. ООП предлагает способы мышления и структурирования кода.

ООП − методология программирования, основанная на представлении программного продукта в виде совокупности объектов, каждый из которых является экземпляром конкретного класса. ООП использует в качестве базовых элементов эвристическое взаимодействие объектов.

Объект − реальная именованная сущность, обладающая свойствами и проявляющая свое поведение.

В применении к объектно-ориентированным языкам программирования понятие объекта и класса конкретизируется, а именно:

Объект − обладающий именем набор данных (полей объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним. Имя используется для доступа к полям и методам, составляющим объект. В предельных случаях объект может не содержать полей или методов, а также не иметь имени.

Любой объект относится к определенному классу.

Класс содержит описание данных и операций над ними.

В классе дается обобщенное описание некоторого набора родственных, реально существующих объектов. Объект − конкретный экземпляр класса.

В качестве примера можно привести чертеж танка или его описание (класс) и реальный танк (экземпляр класса, или объект).

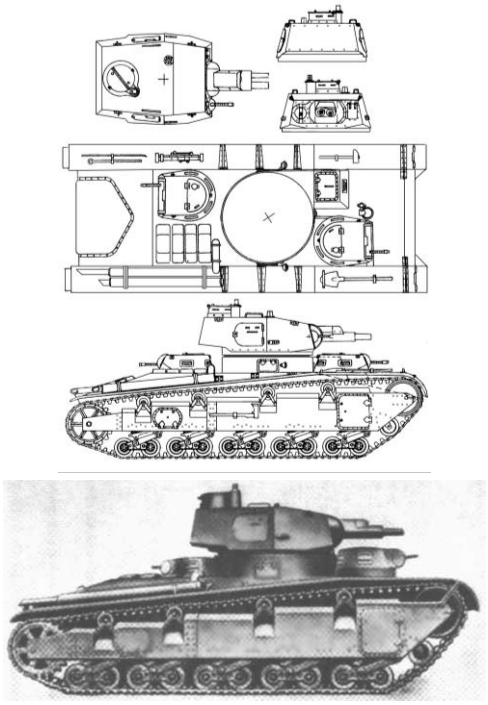


Рис. 1.1. Описание класса (чертеж) и реальный объект

Класс принято обозначать в виде прямоугольника, разделенного на три части:

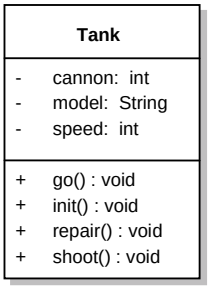


Рис. 1.2. Графическое изображение класса

Объектно-ориентированное программирование основано на принципах:

- абстрагирования данных;
- инкапсуляции;
- наследования;
- полиморфизма;
- «позднего связывания».

Инкапсуляция (encapsulation) − принцип, объединяющий данные и код, манипулирующий этими данными, а также защищающий в первую очередь данные от прямого внешнего доступа и неправильного использования. Другими словами, доступ к данным класса возможен только посредством методов этого же класса.

Наследование (inheritance) − это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним свойства и методы, характерные только для него.

Наследование бывает двух видов:

одиночное − класс (он же подкласс) имеет один и только один суперкласс (предок);

множественное − класс может иметь любое количество предков (в Java запрещено).

Графически наследование часто изображается в виде диаграмм UML:

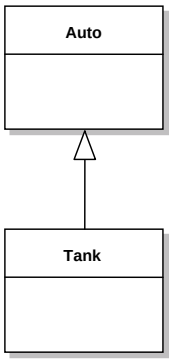


Рис. 1.3. Графическое изображение наследования

Класс «Auto» называется суперклассом, а «Tank» − подклассом.

Полиморфизм (polymorphism) − механизм, использующий одно и то же имя метода для решения двух или более похожих, но несколько отличающихся задач.

Целью полиморфизма применительно к ООП является использование одного имени для задания общих для класса действий. В более общем смысле концепцией полиморфизма является идея «один интерфейс, множество методов».

Механизм «позднего связывания» в процессе выполнения программы определяет принадлежность объекта конкретному классу и производит вызов метода, относящегося к классу, объект которого был использован.

Механизм «позднего связывания» позволяет определять версию полиморфного метода во время выполнения программы. Другими словами, иногда невозможно на этапе компиляции определить, какая версия переопределенного метода будет вызвана на том или ином шаге программы.

Краеугольным камнем наследования и полиморфизма предстает следующая парадигма: «***объект подкласса может использоваться всюду, где используется объект суперкласса***».

При вызове метода класса он ищется в самом классе. Если метод существует, то он вызывается. Если же метод в текущем классе отсутствует, то обращение происходит к родительскому классу и вызываемый метод ищется у него. Если поиск неудачен, то он продолжается вверх по иерархическому дереву вплоть до корня (верхнего класса) иерархии.

Язык Java

Объектно-ориентированный язык Java, разработанный в Sun Microsystems, предназначен для создания переносимых на различные платформы и операционные системы программ. Язык Java нашел широкое применение в Интернет-приложениях, добавив на статические и клиентские Web-страницы динамическую графику, улучшив интерфейсы и реализовав вычислительные возможности. Но объектно-ориентированная парадигма и кроссплатформенность привели к тому, что уже буквально через несколько лет после своего создания язык практически покинул клиентские страницы и перебрался на сервера. На стороне клиента его место занял язык JavaScript.

При создании язык Java предполагался более простым, чем его синтаксический предок C++. На сегодняшний день с появлением версий J2SE 1.5.0 (Тигр) и Java SE 6 (Мустанг) возможности языка Java существенно расширились и во многом перекрывают функциональность C/C++/C#. Отсутствие указателей (наиболее опасного средства языка C++) нельзя считать сужением возможностей, а тем более недостатком, это просто требование безопасности. Возможность работы с произвольными адресами памяти через бестиповые указатели позволяет игнорировать защиту памяти. Отсутствие в Java множественного наследования легко заменяется на более понятные конструкции с применением, например, интерфейсов.

Системная библиотека классов языка Java содержит классы и пакеты, реализующие и расширяющие базовые возможности языка, а также сетевые возможности, взаимодействие с базами данных, графические интерфейсы и многое другое. Методы классов, включенных в эти библиотеки, вызываются из JVM (Java Virtual Machine) во время интерпретации программы.

В Java все объекты программы расположены в динамической памяти – куче (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++. В свою очередь, в Java предложен усовершенствованный механизм работы с коллекциями, реализующими основные динамические структуры данных и списки. 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 изменена концепция организации динамического распределения памяти: отсутствуют способы программного 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 существуют конструкторы, но отсутствуют деструкторы (применяется автоматическая сборка мусора), не используется оператор **goto** и слово **const**, хотя они являются зарезервированными словами языка.

Ключевые и зарезервированные слова языка Java:

abstract	**continue**	**for**	**new**	**switch**
assert	**default**	**goto**	**package**	**synchronized**
boolean	**do**	**if**	**private**	**this**
break	**double**	**implements**	**protected**	**throw**
byte	**else**	**import**	**public**	**throws**
case	**enum**	**instanceof**	**return**	**transient**
catch	**extends**	**int**	**short**	**try**
char	**final**	**interface**	**static**	**void**
class	**finally**	**long**	**strictfp**	**volatile**
const	**float**	**native**	**super**	**while**

Кроме ключевых слов, в Java существуют три литерала: **null**, **true**, **false**, не относящиеся к ключевым и зарезервированным словам. Зарезервированные слова: **const**, **goto**.

Нововведения версий 5.0 и 6.0

В версии языка J2SE 5.0 внесены некоторые изменения и усовершенствования:

- введена возможность параметризации класса;
- поддерживается перечисляемый тип;
- 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;
- используется поддержка стандарта Unicode 4.0;
- добавлены аннотации, новые возможности в ядре и др.

Для версии Java SE 6 характерны высокая скорость, стабильность и оптимальное потребление памяти.

Изменения и усовершенствования:

- новый механизм исполнения сценариев Scripting API;

- поддержка Java-XML Web Service (JAX-WS) для создания приложений поколения Web 2.0;
- улучшены возможности интернационализации ПО, в том числе использования различных региональных форматов и методов преобразования данных;
- новый набор java.awt.Desktop API;
- поддержка области состояния: два новых класса, SystemTray и TrayIcon;
- модернизация в Java Foundation Classes (JFC) и Swing;
- Java-XML Binding (JAXB 2.0);
- JDBC 4.0.

Простое приложение

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

```
// пример # 1 : простое линейное  приложение: First.java
package chapt01;

public class First {
      public static void main(String[] args) {
       // вывод строк
           System.out.print("Мустанг ");
           System.out.println("уже здесь!");
      }
   }
```

В следующем примере то же самое будет сделано с использованием метода класса, реализованного на основе простейшего применения объектно-ориентированного программирования:

```
/* пример # 2 : простое объектно-ориентированное приложение :
FirstProgram.java */
package chapt01;

public class FirstProgram {
      public static void main(String[] args) {
      //объявление и создание объекта firstObject
            MustangLogic firstObject = new MustangLogic();
      //вызов метода, содержащего вывод строки
            firstObject.jumpMustang();
      }
}
// пример # 3 : простой класс: MustangLogic
class MustangLogic {
      public void jumpMustang() {// определение метода
      // вывод строки
            System.out.println("Мустанг уже здесь!");
      }
}
```

Здесь класс **FirstProgram** используется для того, чтобы определить метод **main()**, который запускается автоматически интерпретатором Java и может называться контроллером этого простейшего приложения. Метод **main()** содержит аргументы-параметры командной строки **String[] args**, представляющие массив 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 при работе с классом в целом, а не только с объектом класса. Символы верхнего и нижнего регистров здесь различаются, как и в C++. Тело метода **main()** содержит объявление объекта

```
MustangLogic firstObject = new MustangLogic();
```

и вызов его метода

```
firstObject.jumpMustang();
```

Вывод строки "Мустанг уже здесь!" в примере осуществляет метод **println()** (**ln** – переход к новой строке после вывода) свойства **out** класса **System**, который подключается к приложению автоматически вместе с пакетом **java.lang**. Приведенную программу необходимо поместить в файл **FirstProgram.java** (расширение **.java** обязательно), имя которого совпадает с именем класса.

Объявление классов предваряет строка

```
package chapt01;
```

указывающая на принадлежность классов пакету с именем **chapt01**, который является на самом деле каталогом на диске. Для приложения, состоящего из двух классов, наличие пакетов не является необходимостью. При отсутствии слова **package** классы будут отнесены к пакету по умолчанию, размещенному в корне проекта. Если же приложение состоит из нескольких сотен классов (вполне обычная ситуация), то размещение классов по пакетам является жизненной необходимостью.

Классы из примеров 2 и 3 сохраняются в файлах **FirstProgram.java**. На практике рекомендуется хранить классы в отдельных файлах.

Простейший способ компиляции – вызов строчного компилятора из корневого каталога (в нем находится каталог **chapt01**):

```
javac chapt01/FirstProgram.java
```

При успешной компиляции создаются файлы **FirstProgram.class** и **Mustang.class**. Запустить этот виртуальный код можно с помощью интерпретатора Java:

```
java chapt01.FirstProgram
```

Здесь к имени приложения **FirstProgram.class** добавляется имя пакета **chapt01**, в котором он расположен.

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 JDK установлена в директории (для Windows) **c:\jdk1.6.0**, то каталог, который компилятор Java будет рассматривать как корневой для иерархии пакетов, можно вручную задавать с помощью переменной среды окружения **CLASSPATH** в виде:

```
CLASSPATH=.;c:\jdk1.6.0\
```

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:\jdk1.6.0\bin
```

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

Однако при одновременном использовании нескольких различных версий компилятора и различных библиотек применение переменных среды окружения начинает мешать эффективной работе, так как при выполнении приложения поиск класса осуществляется независимо от версии. Когда виртуальная машина обнаруживает класс с подходящим именем, она его и подгружает. Такая ситуация предрасполагает к ошибкам, порой трудноопределимым. Поэтому переменные окружения лучше не определять вовсе.

Следующая программа отображает в окне консоли аргументы командной строки метода **main()**. Аргументы представляют последовательность 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 # 4 : вывод аргументов командной строки: OutArgs.java */
package chapt01;

public class OutArgs {
      public static void main(String[] args) {
            for (String str : args)
                  System.out.printf("Apr-> %s%n", str);
      }
}
```

В данном примере используется новый вид цикла версии Java 5.0 **for** языка Java и метод форматированного вывода **printf()**. Тот же результат был бы получен при использовании традиционного цикла

```
for (int i = 0; i < args.length; i++)
   System.out.println("Apr-> " + args[i]);
```

Запуск этого приложения осуществляется с помощью следующей командной строки вида:

```
java chapt01.OutArgs 2007 Mustang "Java SE 6"
```

что приведет к выводу на консоль следующей информации:

```
Apr-> 2007
Apr-> Mustang
Apr-> Java SE 6
```

Приложение, запускаемое с аргументами командной строки, может быть использовано как один из способов ввода строковых данных.

Классы и объекты

Классы в языке Java объединяют поля класса, методы, конструкторы и логические блоки. 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 те объявления полей, методов и классов, перед которыми 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 ИмяКласса [extends СуперКласс]
    [implements список_интерфейсов]{/*определение класса*/}
```

Спецификатор доступа к классу может быть **public** (класс доступен в данном 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 дружественный (friendly). Такой класс доступен только в текущем пакете. Спецификатор friendly при объявлении вообще не используется и не является ключевым словом языка. Это слово используется, чтобы как-то определить значение по умолчанию.

Любой класс может наследовать свойства и методы суперкласса, указанного после ключевого слова **extends**, и включать множество интерфейсов, перечисленных через запятую после ключевого слова **implements**. Интерфейсы представляют собой абстрактные классы, содержащие только нереализованные методы.

// пример # 5 : простой пример Java Beans класса: User.java

```
package chapt01;

public class User {
      public int numericCode;//нарушение инкапсуляции
      private String password;

      public void setNumericCode(int value) {
            if(value > 0) numericCode = value;
            else numericCode = 1;
      }
      public int getNumericCode() {
            return numericCode;
      }
      public void setPassword(String pass) {
            if (pass != null) password = pass;
            else password = "11111";
      }
      public String getPassword() {
      //public String getPass() {//некорректно – неполное имя метода
            return password;
      }
}
```

Класс **User** содержит два поля **numericCode** и **password**, помеченные как **public** и **private**. Значение поля **password** можно изменять только при помощи методов, например **setPassword()**. Поле **numericCode** доступно непосредственно через объект класса **User**. 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: UserView.java : Runner.java */
package chapt01;

class UserView {
      public static void welcome(User obj) {
  System.out.printf("Привет! Введен код: %d, пароль: %s",
        obj.getNumericCode(), obj.getPassword());
      }
}
public class Runner {
       public static void main(String[] args) {
             User user = new User();
             user.numericCode = 71;//некорректно - прямой доступ
             // user.password = null; // поле недоступно
             user.setPassword("pass"); //корректно
             UserView.welcome(user);
     }
}
```

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

```
Привет! Введен код: 71, пароль: pass
```

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

```
User user; //объявление ссылки
user = new User(); //создание объекта
```

Однако эти два действия обычно объединяют в одно:

```
User user = new User();/*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.

Сравнение объектов

Операции сравнения ссылок на объекты не имеют особого смысла, так как при этом сравниваются адреса. 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 всех классов и переопределяется в произвольном классе для определения эквивалентности содержимого двух объектов этого класса.

```
/* пример # 7 : сравнение строк и объектов : ComparingStrings.java */
package chapt01;

public class ComparingStrings {
     public static void main(String[] args) {
          String s1, s2;
          s1 = "Java";
          s2 = s1;  // переменная ссылается на ту же строку
          System.out.println("сравнение ссылок "
             + (s1 == s2));  // результат true

          // создание нового объекта добавлением символа
          s1 += '2';
          // s1-="a"; //ошибка, вычитать строки нельзя
          // 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
     }
}
```

Консоль

Взаимодействие с консолью с помощью потока **System.in** представляет собой один из простейших способов передачи информации в приложение. В следующем примере рассматривается ввод информации в виде символа из потока ввода, связанного с консолью, и последующего вывода на консоль символа и его числового кода.

```
// пример # 8 : чтение символа из потока System.in : DemoSystemIn.java
package chapt01;

public class ReadCharRunner {

  public static void main(String[] args) {
     int x;
     try {
        x = System.in.read();
        char c = (char)x;
        System.out.println("Код символа: " + c + " =" + x);
     } catch (java.io.IOException e) {
        e.printStackTrace();
     }
   }
}
```

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

Ввод блока информации осуществляется с помощью чтения строки из консоли. Далее строка может быть использована в исходном виде или преобразована к требуемому виду.

```
// пример # 9 : чтение строки из консоли : ReadCharRunner.java
package chapt01;
import java.io.*;//подключение пакета классов

public class ReadCharRunner {

	public static void main(String[] args) {
	/* байтовый поток ввода System.in передается конструктору потока
		чтения при создании объекта класса InputStreamReader */
		InputStreamReader is =
			new InputStreamReader(System.in);
	/* производится буферизация данных, исключающая необходимость
	 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 из консоли в блоке try */
			String name = bis.readLine();
			System.out.println("Привет, " + name);
		} catch (IOException e) {
			System.err.print("ошибка ввода " + e);
		}
	}
}
```

В результате запуска приложения будет выведено, например, следующее:

```
Введите Ваше имя и нажмите <Enter>:
Остап
Привет, Остап
```

Позже будут рассмотрены более удобные способы извлечения информации из потока ввода, в качестве которого может фигурировать не только консоль, но и дисковый файл, сокетное соединение и пр.

Кроме того, в шестой версии языка существует возможность поддержать национальный шрифт с помощью метода **printf()** определенного для класса **Console**.

```
/* пример # 10 : использование метода printf() класса Console: PrintDeutsch.java */
public class PrintDeutsch {
	public static void main(String[] args) {
		String str = "über";
```

```
            System.out.println(str);
            Console con = System.console();
            con.printf("%s", str);
      }
}
```

В результате будет выведено:

□ber
über

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 # 11 : простой апплет: FirstApplet.java
import java.awt.Graphics;
import java.util.Calendar;

public class FirstApplet extends javax.swing.JApplet {
      private Calendar calendar;

      public void init() {
            calendar = Calendar.getInstance();
            setSize(250,80);
      }
      public void paint(Graphics g) {
            g.drawString("Апплет запущен:", 20, 15);
            g.drawString(
                  calendar.getTime().toString(), 20, 35);
      }
}
```

Для вывода текущего времени и даты в этом примере был использован объект **Calendar** 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.1.4.

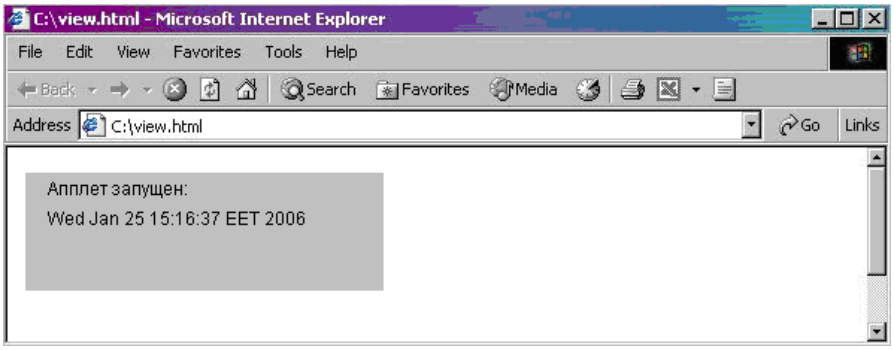


Рис. 1.4. Запуск и выполнение апплета

Для запуска апплетов можно использовать также входящую в JDK программу **`appletviewer.exe`**.

Задания к главе 1

Вариант А

1. Создать класс **`Hello`**, который будет приветствовать любого пользователя, используя командную строку.
2. Создать приложение, которое отображает в окне консоли аргументы командной строки метода **`main()`** в обратном порядке.
3. Создать приложение, выводящее **`n`** строк с переходом и без перехода на новую строку.
4. Создать приложение для ввода пароля из командной строки и сравнения его со строкой-образцом.
5. Создать программу ввода целых чисел как аргументов командной строки, подсчета их суммы (произведения) и вывода результата на консоль.
6. Создать приложение, выводящее фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания. Для получения последней даты и времени использовать класс **`Calendar`** из пакета **`java.util`**.
7. Создать апплет на основе предыдущего задания и запустить его с помощью HTML-документа.

Вариант В

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

1. Четные и нечетные числа.
2. Наибольшее и наименьшее число.
3. Числа, которые делятся на 3 или на 9.
4. Числа, которые делятся на 5 и на 7.
5. Элементы, расположенные методом пузырька по убыванию модулей.

6. Все трехзначные числа, в десятичной записи которых нет одинаковых цифр.
7. Наибольший общий делитель и наименьшее общее кратное этих чисел.
8. Простые числа.
9. Отсортированные числа в порядке возрастания и убывания.
10. Числа в порядке убывания частоты встречаемости чисел.
11. “Счастливые” числа.
12. Числа Фибоначчи: f0 = f1 = 1, f (n) = f (n–1) + f (n–2).
13. Числа-палиндромы, значения которых в прямом и обратном порядке совпадают.
14. Элементы, которые равны полусумме соседних элементов.
15. Период десятичной дроби **p = m/n** для первых двух целых положительных чисел **n** и **m**, расположенных подряд.
16. 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 модификатор доступа к методу **main()** на **public**;
4) убрать параметр из объявления метода **main()**.

Вопрос 1.2.

Выберите истинные утверждения о возможностях языка Java: (выберите 2)

1) возможно использование оператора **goto**;
2) возможно создание метода, не принадлежащего ни одному классу;
3) поддерживается множественное наследование классов;
4) запрещена перегрузка операторов;
5) поддерживается 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 И ОПЕРАТОРЫ

Любая программа манипулирует информацией (простыми данными и объектами) с помощью операторов. Каждый оператор производит результат из значений своих операндов или изменяет непосредственно значение операнда.

Базовые типы данных и литералы

В языке Java используются базовые типы данных, значения которых размещаются в стековой памяти (stack). Эти типы обеспечивают более высокую производительность вычислений по сравнению с объектами. Кроме этого, для каждого базового типа имеются классы-оболочки, которые инкапсулируют данные базовых типов в объекты, располагаемые в динамической памяти (heap).

Определено восемь базовых типов данных, размер каждого из которых остается неизменным независимо от платформы. Беззнаковых типов в Java не существует. Каждый тип данных определяет множество значений и их представление в памяти. Для каждого типа определен набор операций над его значениями.

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	`9223720368547758O7L`
`float`	32	0.0	`3.40282347E+38`
`double`	64	0.0	`1.797693134486231570E+308`

В Java используются целочисленные литералы, например: **`35`** – целое десятичное число, **`071`** – восьмеричное значение, **`0x51`** – шестнадцатеричное значение. Целочисленные литералы по умолчанию относятся к типу **`int`**. Если необходимо определить длинный литерал типа **`long`**, в конце указывается символ **`L`** (например: **`0xffffL`**). Если значение числа больше значения, помещающегося в **`int`** (**`2147483647`**), то Java автоматически предполагает, что оно типа **`long`**. Литералы с плавающей точкой записываются в виде **`1.618`** или в экспоненциальной форме **`0.112E-05`** и относятся к типу **`double`**, таким образом, действительные числа относятся к типу **`double`**. Если необходимо определить литерал типа **`float`**, то в конце литерала следует добавить символ **`F`**. Символьные лите-

ралы определяются в апострофах (**'a'**, **'\n'**, **'\141'**, **'\u005a'**). Для размещения символов используется формат Unicode, в соответствии с которым для каждого символа отводится два байта. В формате Unicode первый байт содержит код управляющего символа или национального алфавита, а второй байт соответствует стандартному ASCII коду, как в C++. Любой символ можно представить в виде **'\u***code***'**, где *code* представляет двухбайтовый шестнадцатеричный код символа. Java поддерживает управляющие символы, не имеющие графического изображения;

'\n'– новая строка, **'\r'** – переход к началу, **'\f'** – новая страница, **'\t'**– табуляция, **'\b'** – возврат на один символ, **'\uxxxx'** – шестнадцатеричный символ Unicode, **'\ddd'**– восьмеричный символ и др. Начиная с J2SE 5.0 используется формат Unicode 4.0. Поддержку четырехбайтным символам обеспечивает наличие специальных методов в классе **Character.**

К литералам относятся булевские значения **true** и **false**, а также **null** – значение по умолчанию для ссылки на объект. При инициализации строки всегда создается объект класса **String** – это не массив символов и не строка. Строки, заключенные в двойные апострофы, считаются литералами и размещаются в пуле литералов, но в то же время такие строки представляют собой объекты.

В арифметических выражениях автоматически выполняются расширяющие преобразования типа **byte → short → int → long → float → double.** Java автоматически расширяет тип каждого **byte** или **short** операнда до **int** в выражениях. Для сужающих преобразований необходимо производить явное преобразование вида **(тип)значение**. Например:

byte b = (byte)128; *//преобразование int в byte*

Указанное в данном примере преобразование необязательно, так как в операциях присваивания литералов при инициализации преобразования выполняются автоматически. При инициализации полей класса и локальных переменных с использованием арифметических операторов автоматически выполняется приведение литералов к объявленному типу без необходимости его явного указания, если только их значения находятся в допустимых пределах, кроме инициализации объектов классов-оболочек. Java не позволяет присваивать переменной значение более длинного типа, в этом случае необходимо явное преобразование типа. Исключение составляют операторы инкремента (++), декремента (--) и сокращенные операторы (+=, /= и т.д.). При явном преобразовании **(тип)значение** 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, литералы и операции над ними :TypeByte.java */
package chapt02;

public class TypeByte {
      public static void main(String[] args) {
            byte b = 1, b1 = 1 + 2;
            final byte B = 1 + 2;
            //b = b1 + 1; //ошибка приведения типов
```

```
            /* b1 – переменная, и на момент выполнения кода b = b1 + 1;
            может измениться, и выражение b1 + 1 может превысить до-
            пустимый размер byte- типа */
            b = (byte)(b1 + 1);
            b = B + 1; // работает
            /* B - константа, ее значение определено, компилятор вычисля-
            ет значение выражения B + 1, и если его размер не превышает
            допустимого для byte типа, то ошибка не возникает */
            //b = -b; //ошибка приведения типов
            b = (byte) -b;
            //b = +b; //ошибка приведения типов
            b = (byte) +b;
            int i = 3;
            //b = i; //ошибка приведения типов, int больше чем byte
            b = (byte) i;
            final int I = 3;
            b = I;        // работает
            /*I –константа. Компилятор проверяет, не превышает ли ее
            значение допустимый размер для типа byte, если не превышает,
            то ошибка не возникает */
            final int I2 = 129;
            //b=I2; //ошибка приведения типов, т.к. 129 больше, чем 127
            b = (byte) I2;

            b += i++;    // работает
            b += 1000;   // работает
            b1 *= 2;     // работает
            float f = 1.1f;
            b /= f;      // работает
            /* все сокращенные операторы автоматически преобразуют ре-
            зультат выражения к типу переменной, которой присваивается
            это значение. Например, b /= f; равносильно b = (byte)(b / f); */
        }
}
```

Переменная базового типа, объявленная как член класса, хранит нулевое значение, соответствующее своему типу. Если переменная объявлена как локальная переменная в методе, то перед использованием она обязательно должна быть проинициализирована, так как она не инициализируется по умолчанию нулем. Область действия и время жизни такой переменной ограничена блоком **{}**, в котором она объявлена.

Документирование кода

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

@author – задает сведения об авторе;

@version – задает номер версии класса;

@exception – задает имя класса исключения;

@param – описывает параметры, передаваемые методу;

@return – описывает тип, возвращаемый методом;

@deprecated – указывает, что метод устаревший и у него есть более совершенный аналог;

@since – с какой версии метод (член класса) присутствует;

@throws – описывает исключение, генерируемое методом;

@see – что следует посмотреть дополнительно.

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

В качестве примера можно рассмотреть снабженный комментариями слегка измененный класс **User** из предыдущей главы.

```
package chapt02;

public class User {
	/**
	 * personal user's code
	 */
	private int numericCode;
	/**
	 * user's password
	 */
	private String password;
	/**
	 * see also chapter #3 "Classes"
	 */
	public User() {
		password = "default";
	}
	/**
	 * @return the numericCode
	 * return the numericCode
	 */
	public int getNumericCode() {
		return numericCode;
	}
	/**
	 * @param numericCode the numericCode to set
	 * parameter numericCode to set
	 */
	public void setNumericCode(int numericCode) {
		this.numericCode = numericCode;
	}
	/**
	 * @return the password
	 * return the password
	 */
```

```
    public String getPassword() {
          return password;
    }
    /**
     * @param password the password to set
     * parameter password to set
     */
    public void setPassword(String password) {
          this.password = password;
    }
}
```

Сгенерированный для этого класса HTML-документ будет иметь вид:

chapt02

Class User

```
java.lang.Object
  └ chapt02.User
```

```
public class User
extends java.lang.Object
```

Field Summary	
private int	**numericCode** personal user's code
private java.lang.String	**password** user's password

Constructor Summary
User() see also chapter #3 "Classes"

Method Summary	
int	**getNumericCode**()
java.lang.String	**getPassword**()

Рис. 2.1. Фрагмент документации для класса User

Операторы

Операторы Java практически совпадают с операторами C++ и имеют такой же приоритет. Поскольку указатели в Java отсутствуют, то отсутствуют операторы `*`, `&`, `->`, `delete` для работы с ними. Операторы работают с базовыми типами и объектами классов-оболочек над базовыми типами. Операторы **+** и **+=** производят также действия с по конкатенации с операндами типа **`String`**. Логические операторы **==**, **!=** и оператор присваивания **=** применимы к операндам любого объектного и базового типов, а также литералам. Применение оператора присваивания к объектным типам часто приводит к ошибке несовместимости типов, поэтому такие операции необходимо тщательно контролировать.

Операции над числами: **+**, **–**, *****, **%**, **/**, **++**, **––** а также битовые операции **&**, **|**, **^**, **~** и операции сдвига 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 других алгоритмических языках, 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 (или деление нацело для целочисленных значений)
`+=`	Сложение (с присваиванием)	`/=`	Деление (с присваиванием)
`-`	Бинарное вычитание и унарное изменение знака	`%`	Остаток от деления (деление по модулю)
`-=`	Вычитание (с присваиванием)	`%=`	Остаток от деления (с присваиванием)
`*`	Умножение	`++`	Инкремент (увеличение значения на единицу)
`*=`	Умножение (с присваиванием)	`--`	Декремент (уменьшение значения на единицу)

Битовые операторы над целочисленными типами

`|`	*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

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

Логические операторы

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

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

Логические операции выполняются только над значениями типов **boolean** и **Boolean** (**true** или **false**).

```
// пример # 2 : битовые операторы и %: DemoOperators.java
package chapt02;

public class DemoOperators {
    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 вида:

booleanзначение ? первое : второе

Если **booleanзначение** равно **true**, вычисляется значение выражения **первое** и оно становится результатом всего оператора, иначе результатом является значение выражения **второе**.

Оператор **instanceof** возвращает значение **true**, если объект является экземпляром данного типа. Например, для иерархии наследования:

```
class Course extends Object {}
class BaseCourse extends Course {}
```

```
class FreeCourse extends BaseCourse {}
class OptionalCourse extends Course {}
```

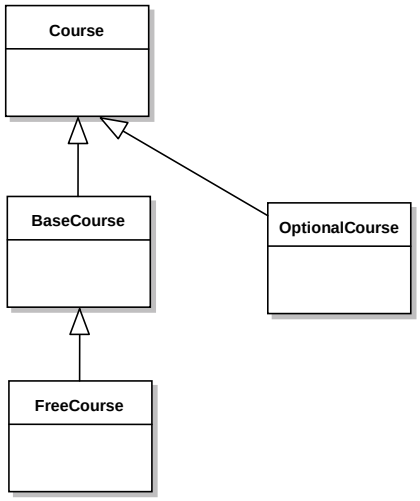


Рис. 2.2. Иерархия наследования

применение оператора **instanceof** может выглядеть следующим образом при вызове метода **doLogic(Course c)**:

```
void doLogic(Course c) {
      if (c instanceof BaseCourse) {/*реализация для BaseCourse и
                                                    FreeCourse*/
      } else if (c instanceof OptionalCourse) {/*реализация для
                                                    OptionalCourse*/
      } else {/*реализация для Course*/}
}
```

Результатом действия оператора **instanceof** будет истина, если объект является объектом данного типа или одного из его подклассов, но не наоборот. Проверка на принадлежность объекта к классу **Object** всегда даст истину, поскольку любой класс является наследником класса **Object**. Результат применения этого оператора по отношению к ссылке на значение **null** всегда ложь, потому что **null** нельзя причислить к какому-либо типу. В то же время литерал **null** можно передавать в методы по ссылке на любой объектный тип и использовать в качестве возвращаемого значения. Базовому типу значение **null** присвоить нельзя, так же как использовать ссылку на базовый тип в операторе **instanceof**.

Классы-оболочки

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

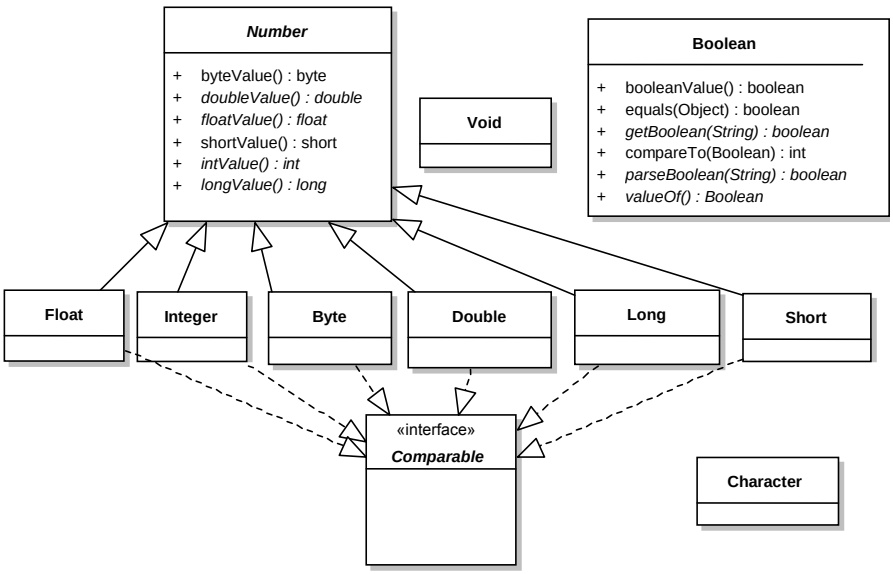


Рис. 2.3. Классы-оболочки

Объекты этих классов представляют ссылки на участки динамической памяти, в которой хранятся их значения, и являются классами-оболочками для значений базовых типов. Классы, соответствующие числовым базовым типам, находятся в библиотеке **java.lang**, являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable**, представляющий собой интерфейс для определения возможности сравнения объектов одного типа между собой. Объекты классов-оболочек по умолчанию получают значение **null**.

Переменную базового типа можно преобразовать к соответствующему объекту, передав ее значение конструктору при объявлении объекта. Объекты класса могут быть преобразованы к любому базовому типу методами тип**Value()** или обычным присваиванием.

Класс **Character** не является подклассом **Number**, 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 # 3 : преобразование типов данных: CastTypes.java */
package chapt02;

public class CastTypes {
     public static void main(String[] args) {
          Float f1 = new Float(10.71); // 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
            byte b0=(byte)(float)f1;// Float в byte
            //b2 = (byte)f1; // невозможно!!!
            /*f1 – не базовый тип, а класс */
            short s = (short) d; // double в short
                  /* Character в int */
            Character ch = new Character('3');
            int i = Character.digit(ch.charValue(), 10);
System.out.printf("f1=%1.2e s1=%s s2=%s%n", f1, s1, s2);
System.out.printf("b=%o d=%.1f b0=%d s=%d i=%d",
                        b, d , b0, s , i);
      }
}
```

Результатом выполнения данного кода будет:

```
f1=1.07e+01 s1=0.0 s2=10.71
b=170 d=120,0 b0=10 s=120 i=3
```

Метод **valueOf(String str)** определен для всех классов-оболочек, соответствующих примитивным типам, и выполняет действия по преобразованию значения, заданного в виде строки, к значению соответствующего объектного типа данных.

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

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

В версии 5.0 введен процесс автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно (автоупаковка). При этом нет необходимости в создании соответствующего объекта с использованием оператора **new**. Например:

```
Integer iob = 71;
```

Автораспаковка – процесс извлечения из объекта-оболочки значения базового типа. Вызовы таких методов, как **intValue()**, **doubleValue()** становятся излишними.

Допускается участие объектов в арифметических операциях, однако не следует этим злоупотреблять, поскольку упаковка/распаковка является ресурсоемким процессом:

```
// пример # 4 : autoboxing & unboxing: NewProperties.java
package chapt02;

public class NewProperties {
     public static void main(String[] args) {
            Integer j = 71; //создание объекта+упаковка
            Integer k = ++j; //распаковка+операция+упаковка
            int i = 2;
            k = i + j + k;
     }
```

}

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

```
Integer j = null; //объект не создан!
int i = j;  //генерация исключения во время выполнения
```

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

```
int i = 128;//заменить на 127  !!!
Integer oa = i; //создание объекта+упаковка
Integer ob = i;
System.out.println("oa==i " + (oa == i));// true
System.out.println("ob==i " + (ob == i));// true
System.out.println("oa==ob " + (oa == ob));// false(ссылки разные)
System.out.println("equals ->" + oa.equals(i)
                  + ob.equals(i)
                  + oa.equals(ob));// true
```

Метод **equals()** сравнивает не значения объектных ссылок, а значения объектов, на которые установлены эти ссылки. Поэтому вызов **oa.equals(ob)** возвращает значение **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 или 7F вместо 7
```

вызывает ошибку компиляции.

Кроме того, стало возможным создавать объекты и массивы, сохраняющие различные базовые типы без взаимных преобразований, с помощью ссылки на класс **Number**, а именно:

```
Number n1 = 1;
Number n2 = 7.1;
Number array[] = {71, 7.1, 7L};
```

При автоупаковке значения базового типа возможны ситуации с появлением некорректных значений и непроверяемых ошибок.

Переменная базового типа всегда передается в метод по значению, а переменная класса-оболочки – по ссылке.

Операторы управления

Оператор выбора **if** имеет следующий синтаксис:

```
if (boolexp) { /*операторы*/} //1
else { /*операторы*/ } //2
```

Если выражение **boolexp** принимает значение **true**, то выполняется группа операторов **1**, иначе – группа операторов **2**. При отсутствии оператора **else** опе-

раторы, расположенные после окончания оператора **if** (строка 2), выполняются вне зависимости от значения булевского выражения оператора **if**. Допустимо также использование конструкции-лесенки **if {} else if {}**.

Аналогично C++ используется оператор **switch:**

```
switch(exp) {
     case exp1:{/*операторы*/}
          break;
     case expN:{/*операторы*/}
          break;
     default:  {/*операторы*/}
}
```

При совпадении условий вида **exp==exp1** выполняются подряд все блоки операторов до тех пор, пока не встретится оператор **break**, Значения **exp1,…, expN** должны быть константами и могут иметь значения типа **int**, **byte**, **short**, **char** или **enum**.

Операторы условного перехода следует применять так, чтобы нормальный ход выполнения программы был очевиден. После **if** следует располагать код, удовлетворяющий нормальной работе алгоритма, после **else** побочные и исключительные варианты. Аналогично для оператора **switch** нормальное исполнение алгоритма следует располагать в инструкциях **case** (наиболее вероятные варианты размещаются раньше остальных), альтернативное или для значений по умолчанию – в инструкции **default**.

В Java существует четыре вида циклов, первые три из них аналогичны соответствующим циклам в языке C++:

```
while (boolexp) { /*операторы*/ } // цикл с предусловием

do { /*операторы*/ } while (boolexp); // цикл с постусловием

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

Здесь по традиции **exp1** – начальное выражение, **boolexp** – условие выполнения цикла, **exp3** – выражение, выполняемое в конце итерации цикла (как правило, это изменение начального значения). Циклы выполняются, пока булевское выражение **boolexp** равно **true**.

Некоторые рекомендации при проектировании циклов:

- проверка условия для всех циклов выполняется только один раз за одну итерацию, для циклов **for** и **while** – перед итерацией, для цикла **do/while** – по окончании итерации.
- цикл **for** следует использовать при необходимости выполнения алгоритма строго определенное количество раз. Цикл **while** используется в случае, когда неизвестно число итераций для достижения необходимого результата, например, поиск необходимого значения в массиве или коллекции. Этот цикл применяется для организации бесконечных циклов в виде **while(true)**.
- для цикла **for** не рекомендуется в цикле изменять индекс цикла.

- условие завершения цикла должно быть очевидным, чтобы цикл не «сорвался» в бесконечный цикл.
- для индексов следует применять осмысленные имена.
- циклы не должны быть слишком длинными. Такой цикл претендует на выделение в отдельный метод.
- вложенность циклов не должна превышать трех.

В версии 5.0 введен еще один цикл, упрощающий доступ к массивам и коллекциям:

```
for(ТипДанных имя : имяОбъекта){ /*операторы*/ }
```

При работе с массивами и коллекциями с помощью данного цикла можно получить доступ ко всем их элементам без использования индексов.

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

Изменять значения элементов массива с помощью такого цикла нельзя. Данный цикл может обрабатывать и единичный объект, если его класс реализует интерфейсы **Iterable** и **Iterator**.

В языке Java расширились возможности оператора прерывания цикла **break** и оператора прерывания итерации цикла **continue**, которые можно использовать с меткой, например:

```
int j = -3;
OUT: while(true) {
     for(;;)
          while (j < 10) {
               if (j == 0)
                    break OUT;
               else {
                    j++;
                    System.out.printf("%d ", j);
               }
          }
}
System.out.print("end");
```

Здесь оператор **break** разрывает цикл, помеченный меткой **OUT**. Тем самым решается вопрос об отсутствии необходимости в операторе `goto` для выхода из вложенных циклов.

Массивы

Массив представляет собой объект, где имя массива является объектной ссылкой. Элементами массива могут быть значения базового типа или объекты. Индексирование элементов начинается с нуля. Все массивы в языке Java являются динамическими, поэтому для создания массива требуется выделение памяти с помощью оператора **new** или прямой инициализации. Значения элементов неинициализированного массива, для которого выделена память, устанавливаются в значения по умолчанию для массива базового типа или **null** для массива объектных ссылок. 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 : массивы и ссылки: ArrRef.java */
package chapt02;

public class ArrRef {
     public static void main(String[] args) {
          int myRef[], my;// объявление ссылки на массив  и переменной
          float[] myRefFloat, myFloat; // два массива!
          // объявление с инициализацией нулевыми значениями по умолчанию
          int myDyn[] = new int[10];
          /*объявление с инициализацией */
          int myInt[] = {5, 7, 9, -5, 6, -2};//6 элементов
          byte myByte[] = {1, 3, 5};//3 элемента
          /*объявление с помощью ссылки на Object */
          Object myObj = new float[5];
          // допустимые присваивания ссылок
          myRef = myDyn;
          myDyn = myInt;
          myRefFloat = (float[])myObj;
          myObj = myByte;// источник ошибки для следующей строки
          myRefFloat = (float[])myObj;// ошибка выполнения
          // недопустимые присваивания ссылок (нековариантность)
          // myInt = myByte;
          //myInt = (int[])myByte;
     }
}
```

Ссылка на **Object** может быть проинициализирована массивом любого типа и любой размерности. Обратное действие требует обязательного приведения типов и корректно только в случае, если тип значений массива и тип ссылки совпадают. Если же ссылка на массив объявлена с указанием типа, то она может содержать данные только указанного типа и присваиваться другой ссылке такого же типа. Приведение типов в этом случае невозможно.

Присваивание **myDyn=myInt** приведет к тому, что значения элементов массива **myDyn** будут утрачены и две ссылки будут установлены на один массив **myInt**, то есть будут ссылаться на один и тот же участок памяти.

Массив представляет собой безопасный объект, поскольку все элементы инициализируются и доступ к элементам невозможен за пределами границ. Размерность массива хранится в его свойстве **length**.

Многомерных массивов в Java не существует, но можно объявлять массив 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.

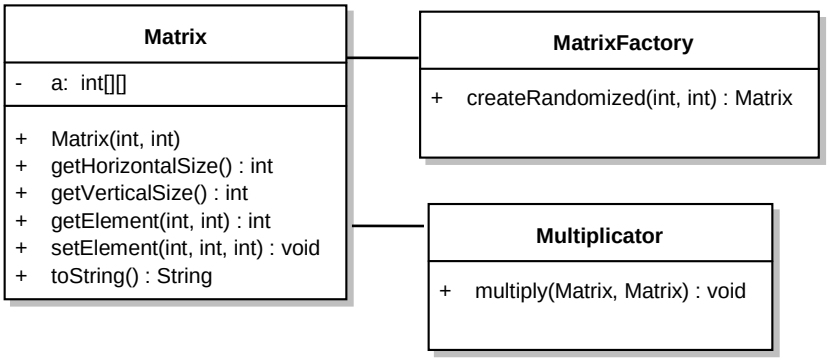


Рис. 2.4. Диаграмма классов для создания и умножения матриц

```
/* пример # 6 : класс хранения матрицы : Matrix.java */
package chapt02;

public class Matrix {
      private int[][] a;

      public Matrix(int n, int m) {
            // создание и заполнение нулевыми значениями
            a = new int[n][m];
      }
      public int getVerticalSize() {
            return a.length;
      }
      public int getHorizontalSize() {
            return a[0].length;
      }
      public int getElement(int i, int j) {
            return a[i][j];
      }
      public void setElement(int i, int j, int value) {
            a[i][j] = value;
      }
      public String toString() {
            String s = "\nMatrix : " + a.length +
                        "x" + a[0].length + "\n";
            for (int[] vector : a) {
                  for (int value : vector)
                        s+= value+" ";
                  s += "\n";
            }
```

```
            return s;
      }
}
```

/ пример # 7 : класс-создатель матрицы : MatrixFactory.java */*

```
package chapt02;

public class MatrixFactory {

      public static Matrix createRandomized(int n, int m) {
            Matrix matrix = new Matrix(n, m);
            for (int i = 0; i < n; i++) {
                  for (int j = 0; j < m; j++) {
      matrix.setElement(i, j, (int)(Math.random()*m*m));
                  }
            }
            return matrix;
      }
}
```

/ пример # 8 : произведение двух матриц : Multiplicator.java */*

```
package chapt02;

public class Multiplicator {

      public static Matrix multiply(Matrix p, Matrix q)
                  throws MultipleException {
            int v = p.getVerticalSize();
            int h = q.getHorizontalSize();
            int temp = p.getHorizontalSize();
            // проверка возможности умножения
            if (temp != q.getVerticalSize())
                  throw new MultipleException();
            // создание матрицы результата
            Matrix result = new Matrix(v, h);
            // умножение
            for (int i = 0; i < v; i++)
                  for (int j = 0; j < h; j++) {
                        int value = 0;
                        for (int k = 0; k < temp; k++) {
      value += p.getElement(i, k) * q.getElement(k, j);
                        }
                        result.setElement(i, j, value);
                  }
            return result;
      }
}
```

/ пример # 9 : исключительная ситуация матрицы : MultipleException.java */*

```
package chapt02;
public class MultipleException extends Exception {}
```

```
/* пример # 10 : класс, запускающий приложение : Runner.java */
package chapt02;
public class Runner {
      public static void main(String[] args) {
          int n = 2, m = 3, l = 4;
          Matrix p = MatrixFactory.createRandomized(n, m);
          Matrix q = MatrixFactory.createRandomized(m, l);
            System.out.println("Matrix first is: " + p);
            System.out.println("Matrix second is: " + q);

            try {
            Matrix result = Multiplicator.multiply(p, q);
            System.out.println("Matrix product is: "
                                          + result);
            } catch (MultipleException e){
                  System.err.println("Matrices could"
                              + " not be multiplied: ");
            }
      }
}
```

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

```
Matrix first is:
Matrix : 2x3
6 4 2
0 8 4

Matrix second is:
Matrix : 3x4
8 0 2 7
6 1 0 0
1 2 4 5

Matrix product is:
Matrix : 2x4
74 8 20 52
52 16 16 20
```

Массивы объектов фактически не отличаются от массивов базовых типов. Они в действительности представляют собой массивы ссылок, проинициализированных по умолчанию значением **null**. Выделение памяти для хранения объектов массива должно производиться для каждой объектной ссылки в массиве.

Класс Math

Класс **java.lang.Math** содержит только статические методы для физических и технических расчетов, а также константы **E** и **PI**.

Все методы класса вызываются без создания экземпляра класса (создать экземпляр класса **Math** невозможно). В классе определено большое количество методов для математических вычислений, а также ряд других полезных методов, таких как **floor()**, **ceil()**, **rint()**, **round()**, **max()**, **min()**, которые выполняют задачи по округлению, поиску экстремальных значений, нахождению ближайшего целого и т.д. Рассмотрим пример обработки значения случайного числа, полученного с помощью метода **random()** класса **Math**.

```
/* пример # 11 : использование методов класса Math: MathMethods.java */
package chapt02;

public class MathMethods {
     public static void main(String[] args) {
          final int MAX_VALUE = 10;
          double d;
          d = Math.random() * MAX_VALUE;
          System.out.println("d = " + d);
          System.out.println("Округленное до целого ="
                    + Math.round(d));
          System.out.println("Ближайшее целое, "
                    + " <= исходного числа ="
                    + Math.floor(d));
          System.out.println("Ближайшее целое, "
                    + " >= исходного числа ="
                    + Math.ceil(d));
          System.out.println("Ближайшее целое значение"
                    + "к числу =" + Math.rint(d));
     }
}
```

Один из вариантов выполнения кода представлен ниже:

```
d = 0.08439575016076173
Округленное до целого =0
Ближайшее целое, <= исходного числа =0.0
Ближайшее целое, >= исходного числа =1.0
Ближайшее целое значение к числу =0.0
```

Управление приложением

Все приложения автоматически импортируют пакет **java.lang**. Этот пакет содержит класс **java.lang.System**, предназначенный для выполнения ряда системных действий по обслуживанию работающего приложения. Объект этого класса создать нельзя.

Данный класс, кроме полей **System.in**, **System.out** и **System.err**, предназначенных для ввода/вывода на консоль, содержит целый ряд полезных методов:

void gc() – запуск механизма «сборки мусора»;

void exit(int status) – прекращение работы виртуальной java-машины (JVM);

void setIn(InputStream in), **void setOut(PrintStream out)**, **void setErr(PrintStream err)** – переназначение стандартных потоков ввода/вывода;

Properties getProperties() – получение всех свойств;

String getProperty(String key) – получение значения конкретного свойства;

void setSecurityManager(SecurityManager s), **SecurityManager getSecurityManager()** – получение и установка системы безопасности;

void load(String filename) – запуск программы из ОС;

void loadLibrary(String libname) – загрузка библиотеки;

void arrayCopy(параметры**)** – копирование части одного массива в другой.

Управлять потоком выполнения приложения можно с помощью класса **java.lang.Runtime**. Объект класса **Runtime** создается при помощи вызова статического метода **getRuntime()**, возвращающего объект **Runtime**, который ассоциирован с данным приложением. Остановить виртуальную машину можно с помощью методов **exit(int status)** и **halt(int status)**. Существует несколько возможностей по очистке памяти: **gc()**, **runFinalization()** и др. Определить общий объем памяти и объем свободной памяти можно с помощью методов **totalMemory()** и **freeMemory()**.

*/*пример # 12 : информация о состоянии оперативной памяти: RuntimeDemo.java*/*

```
package chapt02;

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());
             //инициализация процесса
          ProcessBuilder pb =
          new ProcessBuilder("mspaint","c:\\temp\\cow.gif");

          try {
                   pb.start(); // запуск mspaint.exe
          } catch (java.io.IOException e) {
                   System.err.println(e.getMessage());
          }
          System.out.println("Свободная память после "
          + "запуска mspaint.exe: " + rt.freeMemory());
```

```
            System.out.println("Список команд: "
                                        + pb.command());
    }
}
```

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
Список команд: [mspaint, c:\temp\cow.gif]
```

В примере использованы возможности класса **java.lang.ProcessBuilder**, обеспечивающего запуск внешних приложений с помощью метода **start()**, в качестве параметров которого применяются строки с именем запускаемого приложения и загружаемого в него файла. Внешнее приложение использует для своей загрузки и выполнения память операционной системы.

Метод **arraycopy()** класса **System**, позволяет копировать часть одного массива в другой, начиная с указанной позиции.

```
/* пример # 13 : копирование массива: ArrayCopyDemo.java */
package chapt02;

public class ArrayCopyDemo {
    public static void main(String[] args) {
        int mas1[] = { 1, 2, 3 },
        mas2[] = { 4, 5, 6, 7, 8, 9 };
        show("mas1[]: ", mas1);
        show("mas2[]: ", 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.printf("%n после arraycopy(): ");
        show("mas1[]: ", mas1);
        show("mas2[]: ", mas2);
    }
    private static void show(String s, int[] mas) {
        System.out.printf("%n%s", s);
        for (int i : mas) System.out.printf("%d ", i);
    }
}
```

В результате будет выведено:

```
mas1[]:  1 2 3
```

```
mas2[]:  4 5 6 7 8 9
после arraycopy():
mas1[]:  1 2 3
mas2[]:  4 5 1 2 3 9
```

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 и самую длинную строки. Вывести найденные строки и их длину.
2. Ввести **n** строк с консоли. Упорядочить и вывести строки в порядке возрастания (убывания) значений их длины.
3. Ввести **n** строк с консоли. Вывести на консоль те строки, длина которых меньше (больше) средней, а также длину.
4. Ввести **n** слов с консоли. Найти слово, в котором число различных символов минимально. Если таких слов несколько, найти первое из них.
5. Ввести **n** слов с консоли. Найти количество слов, содержащих только символы латинского алфавита, а среди них – количество слов с равным числом гласных и согласных букв.
6. Ввести **n** слов с консоли. Найти слово, символы в котором идут в строгом порядке возрастания их кодов. Если таких слов несколько, найти первое из них.
7. Ввести **n** слов с консоли. Найти слово, состоящее только из различных символов. Если таких слов несколько, найти первое из них.
8. Ввести **n** слов с консоли. Среди слов, состоящих только из цифр, найти слово-палиндром. Если таких слов больше одного, найти второе из них.
9. Написать программы решения задач 1–8, осуществляя ввод строк как аргументов командной строки.
10. Используя оператор **switch**, написать программу, которая выводит на экран сообщения о принадлежности некоторого значения **k** интервалам (-10k, 0], (0, 5], (5, 10], (10, 10k].
11. Используя оператор **switch**, написать программу, которая выводит на экран сообщения о принадлежности некоторого значения **k** интервалам (-10k, 5], [0, 10], [5, 15], [10, 10k].
12. Написать программу, которая выводит числа от 1 до 25 в виде матрицы 5x5 слева направо и сверху вниз.

13. Написать программу, позволяющую корректно находить корни квадратного уравнения. Параметры уравнения должны задаваться с командной строки.
14. Ввести число от 1 до 12. Вывести на консоль название месяца, соответствующего данному числу. (Осуществить проверку корректности ввода чисел).

Вариант В

Ввести с консоли **n** – размерность матрицы **a**[**n**][**n**]. Задать значения элементов матрицы в интервале значений от **-n** до **n** 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 (180, 270) градусов 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 все строки и столбцы, его содержащие.
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.
14. Округлить все элементы матрицы до целого числа.
15. Найти количество всех седловых точек матрицы. (Матрица **A** имеет седловую точку $\mathbf{A}_{\mathbf{i},\mathbf{j}}$, если $\mathbf{A}_{\mathbf{i},\mathbf{j}}$ является минимальным элементом в **i**-й строке и максимальным в **j**-м столбце).
16. Перестроить матрицу, переставляя в ней строки так, чтобы сумма элементов в строках полученной матрицы возрастала.
17. Найти число локальных минимумов. (Соседями элемента матрицы назовем элементы, имеющие с ним общую сторону или угол. Элемент матрицы называется локальным минимумом, если он строго меньше всех своих соседей.)
18. Найти наибольший среди локальных максимумов. (Элемент матрицы называется локальным максимумом, если он строго больше всех своих соседей.)

19. Перестроить заданную матрицу, переставляя в ней столбцы так, чтобы значения их характеристик убывали. (Характеристикой столбца прямоугольной матрицы называется сумма модулей его элементов).
20. Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине – в позиции (2,2), следующий по величине – в позиции (3,3) и т. д., заполнив таким образом всю главную диагональ.

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 идентификаторов являются корректными?

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

Вопрос 2.4.

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.

Глава 3
КЛАССЫ

Класс представляет описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Классы – основной элемент абстракции языка Java, основное назначение которого, кроме реализации назначенного ему контракта, это сокрытие реализации. Классы всегда взаимодействуют друг с другом и объединяются в пакеты. Из пакетов создаются модули, которые взаимодействуют друг с другом только через ограниченное количество методов и классов, не имея никакого представления о процессах, происходящих внутри других модулей.

Имя класса в пакете должно быть уникальным. Физически пакет представляет собой каталог, в который помещаются программные файлы, содержащие реализацию классов.

Классы позволяют провести декомпозицию поведения сложной системы до множества элементарных взаимодействий связанных объектов. Класс определяет структуру и/или поведение некоторого элемента предметной области, для которой разрабатывается программная модель.

Определение простейшего класса без наследования имеет вид:

```
class ИмяКласса {
    {}//логические блоки
    // дружественные данные и методы
    private// закрытые данные и методы
    protected// защищенные данные и методы
    public// открытые данные и методы
}
```

Переменные класса и константы

Классы инкапсулируют переменные и методы – члены класса. Переменные класса объявляются в нем следующим образом:

```
спецификатор тип имя;
```

В языке Java могут использоваться статические переменные класса, объявленные один раз для всего класса со спецификатором **static** и одинаковые для всех экземпляров (объектов) класса, или переменные экземпляра класса, создаваемые для каждого объекта класса. Поля класса объявляются со спецификаторами доступа **public**, **private**, **protected** или по умолчанию без спецификатора. Кроме данных – членов класса, в методах класса используются локальные переменные и параметры методов. В отличие от переменных класса, инкапсулируемых нулевыми элементами, переменные методов не инициализируются по умолчанию.

Переменные со спецификатором **final** являются константами. Спецификатор **final** можно использовать для переменной, объявленной в методе, а также для параметра метода.

В следующем примере рассматриваются объявление и инициализация значений полей класса и локальных переменных метода, а также использование параметров метода:

```
/* пример # 1 : типы атрибутов и переменных: Second.java */
package chapt03;
import java.util.*;

class Second {
     private int x; // переменная экземпляра класса
     private int y = 71; // переменная экземпляра класса
     public final int CURRENT_YEAR = 2007; // константа
     protected static int bonus; // переменная класса
     static String version = "Java SE 6"; // переменная класса
     protected Calendar now;

     public int method(int z) { // параметр метода
          z++;
          int a; // локальная переменная метода
          //a++; // ошибка компиляции, значение не задано
          a = 4; //инициализация
          a++;
          now = Calendar.getInstance();//инициализация
          return a + x + y + z;
     }
}
```

В рассмотренном примере в качестве переменных экземпляра класса, переменных класса и локальных переменных метода использованы данные базовых типов, не являющиеся ссылками на объекты (кроме **String**). Данные могут быть ссылками, назначить которым реальные объекты можно с помощью оператора **new**.

Ограничение доступа

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

- по умолчанию – дружественные члены класса доступны классам, находящимся в том же пакете;
- **private** – члены класса доступны только членам данного класса;
- **protected** – члены класса доступны классам, находящимся в том же пакете, и подклассам – в других пакетах;
- **public** – члены класса доступны для всех классов в этом и других пакетах.

Член класса (поле или метод), объявленный **public**, доступен из любого места вне класса. Все, что объявлено **private**, доступно только методам внутри класса и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден и доступен 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**. Действие спецификаторов доступа распространяется только на тот элемент класса, перед которым они стоят.

Спецификатор доступа **public** может также стоять перед определением внешнего (enclosing) класса. Если данный спецификатор отсутствует, то класс недоступен из других пакетов.

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 (потери ссылки). Аналогом деструктора является метод **finalize().** Исполняющая среда языка Java будет вызывать его каждый раз, когда сборщик мусора будет уничтожать объект класса, которому не соответствует ни одна ссылка.

```
/* пример # 2 : перегрузка конструктора: Quest.java */
package chapt03;

public class Quest {
      private int id;
      private String text;
      // конструктор без параметров (по умолчанию)
      public Quest() {
          super();/* если класс будет объявлен без конструктора, то
                     компилятор предоставит его именно в таком виде*/
      }
      // конструктор с параметрами
      public Quest(int idc, String txt) {
            super();/*вызов конструктора суперкласса явным образом
            необязателен, компилятор вставит его автоматически*/
            id = idc;
            text = txt;
      }
}
```

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

```
Quest a = new Quest();//инициализация полей значениями по умолчанию
Quest b = new Quest(71, "Сколько бит занимает boolean?");
```

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

Если конструктор в классе не определен, Java предоставляет конструктор по умолчанию без параметров, который инициализирует поля класса значениями по умолчанию, например: **0**, **false**, **null**. Если же конструктор с параметрами определен, то конструктор по умолчанию становится недоступным и для его вызова необходимо явное объявление такого конструктора. Конструктор подкласса всегда вызывает конструктор суперкласса. Этот вызов может быть явным или неявным и всегда располагается в первой строке кода конструктора. Если конструктору суперкласса нужно передать параметры, то необходим явный вызов:

```
super(параметры);
```

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

```
/* пример # 3 : вычисление расстояния между точками: Point.java:
LocateLogic.java: Runner.java */
package chapt03;

public class Point {
/* объект инициализируется при создании и не изменяется */
      private final double x;
      private final double y;

      public Point(final double xx, final double yy) {
            super();
            x = xx;
            y = yy;
      }
      public double getX() {
            return x;
      }
      public double getY() {
            return y;
      }
}
package chapt03;

public class LocateLogic {
      public double calculateDistance(
                                    Point t1, Point t2) {
            /* вычисление расстояния */
            double dx = t1.getX() - t2.getX();
            double dy = t1.getY() - t2.getY();
            return Math.hypot(dx, dy);
      }
}
package chapt03;
public class Runner {
```

```
    public static void main(String[] args) {
    // локальные переменные не являются членами класса
          Point t1 = new Point(5, 10);
          Point t2 = new Point(2, 6);
          System.out.print("расстояние равно : "
          + new LocateLogic().calculateDistance(t1, t2));
    }
}
```

В результате будет выведено:

```
расстояние равно : 5.0
```

Конструктор объявляется со спецификатором **public**, чтобы была возможность вызывать его при создании объекта в любом пакете приложения. Спецификатор **private** не позволяет создавать объекты вне класса, а спецификатор «по умолчанию» – вне пакета. Спецификатор **protected** позволяет создавать объекты в текущем пакете и для подклассов в других пакетах.

Методы

Изобретение методов является вторым по важности открытием после создания компьютера. Метод – основной элемент структурирования хода.

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

```
returnType methodName(список_параметров) {
    // тело метода
    return value;  // если нужен возврат значения (returnType не void)
}
```

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

```
objectName.methodName();
```

Методы-конструкторы по имени вызываются автоматически только при создании объекта класса с помощью оператора **new.**

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] returnType methodName(список_параметров)
                                [throws список_исключений]
```

Как и для полей класса, спецификатор доступа к методам может быть **public**, **private**, **protected** и по умолчанию. При этом методы суперкласса можно перегружать или переопределять в порожденном подклассе.

Объявленные в методе переменные являются локальными переменными метода, а не членами классов, и не инициализируются значениями по умолчанию при создании объекта класса или вызове метода.

Статические методы и поля

Поля данных, объявленные в классе как **static**, являются общими для всех объектов класса и называются переменными класса. Если один объект изменит значение такого поля, то это изменение увидят все объекты. Для работы со статическими атрибутами используются статические методы, объявленные со спецификатором **static**. Такие методы являются методами класса, не привязаны ни к какому объекту и не содержат указателя **this** на конкретный объект, вызвавший метод. Статические методы реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции. По причине недоступности указателя **this** статические поля и методы не могут обращаться к нестатическим полям и методам напрямую, так как для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

```
// пример # 4 : статические метод и поле: Mark.java
package chapt03;

public class Mark {
      private int mark = 3;
      public static int coeff = 5;

      public double getResult() {
            return (double)coeff*mark/100;
      }
      public static void setCoeffFloat(float c) {
            coeff = (int)coeff*c;;
      }
      public void setMark(int mark) {
            this.mark = mark;
      }
//из статического метода нельзя обратиться к нестатическим полям и методам
      /*public static int getResult() {
            setMark(5);//ошибка
            return coeff*mark/100;//ошибка
      }*/
}
```

При создании двух объектов

```
Mark ob1 = new Mark();
Mark ob2 = new Mark();
```

Значение **ob1.coeff** и **ob2.coeff** и равно **5**, поскольку располагается в одной и той же области памяти. Изменить значение статического члена можно прямо через имя класса:

```
Mark.coeff = 7;
```

Вызов статического метода также следует осуществлять с помощью указания: **ClassName.*methodName*()**, а именно:

```
Mark.setCoeffFloat();
```

```
float z = Math.max(x, y); // определение максимума из двух значений
System.exit(1);  // экстренное завершение работы приложения
```

Статический метод можно вызывать также с использованием имени объекта, но такой вызов снижает качество кода и 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 осуществляется по большей части посредством задания имени класса, которому они принадлежат.

Модификатор final

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода. Методы, объявленные как **final**, нельзя замещать в подклассах, для классов – создавать подклассы. Например:

```
/* пример # 5 : final-поля и методы: Rector.java: ProRector.java */
package chapt03;

public class Rector {

	// инициализированная константа
	final int ID = (int)(Math.random()*10);
	// неинициализированная константа
	final String NAME_RECTOR;

	public Rector() {
	// инициализация в конструкторе
		NAME_RECTOR = "Старый";// только один раз!!!
	}
	// {NAME_RECTOR = "Новый";} // только один раз!!!

	public final void jobRector() {
		// реализация
		// ID = 100; //ошибка!
	}
	public boolean checkRights(final int num) {
		// id = 1; //ошибка!
		final int CODE = 72173394;
		if (CODE == num) return true;
		else return false;
	}
	public static void main(String[] args) {
		System.out.println(new Rector().ID);
	}
}
package chapt03;

public class ProRector extends Rector {
	// public void jobRector(){} //запрещено!
}
```

Константа может быть объявлена как поле класса, но не проинициализирована. В этом случае она должна быть проинициализирована в логическом блоке класса, заключенном в **{}**, или конструкторе, но только в одном из указанных мест. Значение по умолчанию константа получить не может в отличие от переменных класса. Константы могут быть объявлены в методах как локальные или как параметры метода. В обоих случаях значения таких констант изменять нельзя.

Абстрактные методы

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

```
/* пример # 6 : абстрактный класс и метод: AbstractCourse.java */
public abstract class AbstractCourse {
     private String name;
     public AbstractCourse()  { }
     public abstract void changeTeacher(int id);/*определение
                                        метода отсутствует*/
     public setName(String n){
            name = n;
     }
}
```

В отличие от интерфейсов, абстрактный класс может содержать и абстрактные, и неабстрактные методы, а может и не содержать ни одного абстрактного метода.

Подробнее абстрактные класса и интерфейсы изучаются в главе «Абстрактные классы. Интерфейсы. Пакеты».

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 loadCripto(int num);
```

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

Модификатор synchronized

При использовании нескольких потоков управления в одном приложении 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 уведомления о возвращении блокировки объекта **notifyAll()**, **notify()** и метода остановки потока **wait()** класса **Object** (суперкласса для всех классов языка Java) предполагает использование модификатора **synchronized**, так как эти методы предназначены для работы с потоками.

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, например:

```
{ /* код */ }
static { /* код */ }
```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов и обращения к полям текущего класса. 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**, представляющую собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static**. В этом случае он вызывается только один раз в жизненном цикле приложения при создании объекта или при обращении к статическому методу (полю) данного класса.

```
/*пример # 7 : использование логических блоков при объявлении класса:
 Department.java: DemoLogic.java */
package chapt03;

public class Department {
	{
		System.out.println("logic (1) id=" + this.id);
	}
	static {
		System.out.println("static logic");
	}
	private int id = 7;

	public Department(int d) {
		id = d;
		System.out.println("конструктор id=" + id);
	}
	int getId() {
		return id;
	}
	{
		id = 10;
		System.out.println("logic (2) id=" + id);
	}
}
package chapt03;

public class DemoLogic {
	public static void main(String[] args) {
```

```
            Department obj = new Department(71);
            System.out.println("значение id=" + obj.getId());
      }
}
```

В результате выполнения этой программы будет выведено:

```
static logic
logic (1) id=0
logic (2) id=10
конструктор id=71
значение id=71
```

В первой строке вывода поле **id** получит значение по умолчанию, так как память для него выделена при создании объекта, а значение еще не проинициализировано. Во второй строке выводится значение **id**, равное **10**, так как после инициализации атрибута класса был вызван логический блок, изменивший его значение.

Перегрузка методов

Метод называется перегруженным, если существует несколько его версий с одним и тем же именем, но с различным списком параметров. Перегрузка реализует «раннее связывние». Перегрузка может ограничиваться одним классом. Методы с одинаковыми именами, но с различающимися списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными. Если в последнем случае списки параметров совпадают, то имеет место другой механизм – переопределение метода.

Статические методы могут перегружаться нестатическими и наоборот – без ограничений.

При вызове перегруженных методов следует избегать ситуаций, когда компилятор будет не в состоянии выбрать тот или иной метод.

/ пример # 8 : вызов перегруженных методов: NumberInfo.java */*

```
package chapt04;

public class NumberInfo {
      public static void viewNum(Integer i) {//1
            System.out.printf("Integer=%d%n", i);
      }
      public static void viewNum(int i) {//2
            System.out.printf("int=%d%n", i);
      }
      public static void viewNum(Float f) {//3
            System.out.printf("Float=%.4f%n", f);
      }
      public static void viewNum(Number n) {//4
            System.out.println("Number=" + n);
      }
      public static void main(String[] args) {
            Number[] num =
```

```
                {new Integer(7), 71, 3.14f, 7.2 };
            for (Number n : num)
                viewNum(n);

            viewNum(new Integer(8));
            viewNum(81);
            viewNum(4.14f);
            viewNum(8.2);
        }
}
```

Может показаться, что в результате компиляции и выполнения данного кода будут последовательно вызваны все четыре метода, однако в консоль будет выведено:

```
Number=7
Number=71
Number=3.14
Number=7.2
Integer=8
int=81
Float=4,1400
Number=8.2
```

То есть во всех случаях при передаче в метод элементов массива был вызван четвертый метод. Это произошло вследствие того, что выбор варианта перегруженного метода происходит на этапе компиляции и зависит от типа массива **num**. То, что на этапе выполнения в метод передается другой тип (для первых трех элементов массива), не имеет никакого значения, так как выбор уже был осуществлен заранее.

При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

С одной стороны, этот механизм снижает гибкость, с другой – все возможные ошибки при обращении к перегруженным методам отслеживаются на этапе компиляции, в отличие от переопределенных методов, когда их некорректный вызов приводит к возникновению исключений на этапе выполнения.

При перегрузке всегда следует придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

Параметризованные классы

К наиболее важным новшествам версии языка J2SE 5 можно отнести появление параметризованных (generic) классов и методов, позволяющих использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Применение generic-классов для создания типизированных коллекций будет рассмотрено в главе «Коллекции». Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Приведем пример generic-класса с двумя параметрами:

```
/*пример # 9 : объявление класса с двумя параметрами : Subject.java */
package chapt03;

public class Subject <T1, T2> {
	private T1 name;
	private T2 id;

	public Subject() {
	}
	public Subject(T2 ids, T1 names) {
		id = ids;
		name = names;
	}
}
```

Здесь **T1**, **T2** – фиктивные объектные типы, которые используются при объявлении членов класса и обрабатываемых данных. При создании объекта компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект. В качестве параметров классов запрещено применять базовые типы.

Объект класса **Subject** можно создать, например, следующим образом:

```
Subject<String,Integer> sub =
     new Subject<String,Integer>();
         char ch[] = {'J','a','v','a'};
Subject<char[],Double> sub2 =
                new Subject<char[],Double>(ch, 71D );
```

В объявлении **sub2** имеет место автоупаковка значения **71D** в **Double**.

Параметризированные типы обеспечивают типовую безопасность.

Ниже приведен пример параметризованного класса **Optional** с конструкторами и методами, также инициализация и исследование поведения объектов при задании различных параметров.

```
/*пример # 10 : создание и использование объектов параметризованного
 класса: Optional.java: Runner.java */
package chapt03;

public class Optional <T> {
	private T value;

	public Optional() {
	}
	public Optional(T value) {
		this.value = value;
	}
	public T getValue() {
		return value;
	}
	public void setValue(T val) {
		value = val;
```

```
	}
	public String toString() {
		if (value == null) return null;
	    return value.getClass().getName() + " " + value;
	}
}
package chapt03;

public class Runner {
	public static void main(String[] args) {
	//параметризация типом Integer
		Optional<Integer> ob1 =
				new Optional<Integer>();
		ob1.setValue(1);
		//ob1.setValue("2");// ошибка компиляции: недопустимый тип
		int v1 = ob1.getValue();
		System.out.println(v1);
	//параметризация типом String
		Optional<String> ob2 =
				new Optional<String>("Java");
		String v2 = ob2.getValue();
		System.out.println(v2);
	//ob1 = ob2; //ошибка компиляции – параметризация не ковариантна

	//параметризация по умолчанию – Object
		Optional ob3 = new Optional();
		System.out.println(ob3.getValue());
		ob3.setValue("Java SE 6");
		System.out.println(ob3.toString());/* выводится
				тип объекта, а не тип параметризации */

		ob3.setValue(71);
		System.out.println(ob3.toString());

		ob3.setValue(null);
	}
}
```

В результате выполнения этой программы будет выведено:

```
1
Java
null
java.lang.String Java SE 6
java.lang.Integer 71
```

В рассмотренном примере были созданы объекты типа **Optional**: **ob1** на основе типа **Integer** и **ob2** на основе типа **String** при помощи различных конструкторов. При компиляции вся информация о generic-типах стирается и заменя-

ется для членов класса и методов заданными типами или типом **Object**, если параметр не задан, как для объекта **ob3**. Такая реализация необходима для обеспечения совместимости с кодом, созданным в предыдущих версиях языка.

Объявление generic-типа в виде **<T>**, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса. Переменные такого типа могут вызывать только методы класса **Object**. Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class OptionalExt <T extends Тип> {
     private T value;
}
```

Такая запись говорит о том, что в качестве типа **T** разрешено применять только классы, являющиеся наследниками (суперклассами) класса **Тип**, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

Часто возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом. В этом случае при определении метода следует применить метасимвол **?**. Метасимвол также может использоваться с ограничением **extends** для передаваемого типа.

```
/*пример # 11 : использование метасимвола в параметризованном классе:
Mark.java, Runner.java */
package chapt03;

public class Mark<T extends Number> {
      public T mark;

      public Mark(T value) {
            mark = value;
      }
      public T getMark() {
            return mark;
      }
      public int roundMark() {
            return Math.round(mark.floatValue());
      }
      /* вместо */ // public boolean sameAny(Mark<T> ob) {
      public boolean sameAny(Mark<?> ob) {
            return roundMark() == ob.roundMark();
      }
      public boolean same(Mark<T> ob) {
            return getMark() == ob.getMark();
      }
}
package chapt03;
```

```
public class Runner {
      public static void main(String[] args) {
      // Mark<String> ms = new Mark<String>("7"); //ошибка компиляции
            Mark<Double> md = new Mark<Double>(71.4D);//71.5d
            System.out.println(md.sameAny(md));
            Mark<Integer> mi = new Mark<Integer>(71);
            System.out.println(md.sameAny(mi));
            // md.same(mi); //ошибка компиляции
            System.out.println(md.roundMark());
      }
}
```

В результате будет выведено:

```
true
true
71
```

Метод **sameAny(Mark<?> ob)** может принимать объекты типа **Mark**, инициализированные любым из допустимых для этого класса типов, в то время как метод с параметром **Mark<T>** мог бы принимать объекты с инициализацией того же типа, что и вызывающий метод объект.

Для generic-типов существует целый ряд ограничений. Например, невозможно выполнить явный вызов конструктора generic-типа:

```
class Optional <T> {
      T value = new T();
}
```

так как компилятор не знает, какой конструктор может быть вызван и какой объем памяти должен быть выделен при создании объекта.

По аналогичным причинам generic-поля не могут быть статическими, статические методы не могут иметь generic-параметры или обращаться к generic-полям, например:

```
/*пример # 12 : неправильное объявление полей параметризованного класса:
Failed.java */
package chapt03;

class Failed <T1, T2> {
      static T1 value;
      T2 id;

      static T1 getValue() {
          return value;
        }
      static void use() {
          System.out.print(id);
       }
   }
```

Параметризованные методы

Параметризованный (generic) 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 Тип> returnType methodName(T arg) {}
<T> returnType methodName(T arg) {}
```

Описание типа должно находиться перед возвращаемым типом. 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.

Generic-методы могут находиться как в параметризованных классах, так и в обычных. Параметр метода может не иметь никакого отношения к параметру своего класса. Метасимволы применимы и к generic-методам.

/ пример # 13 : параметризованный метод: GenericMethod.java */*

```
public class GenericMethod {
     public static <T extends Number> byte asByte(T num) {
             long n = num.longValue();
             if (n >= -128 && n <= 127) return (byte)n;
             else return 0;
     }
     public static void main(String [] args) {
          System.out.println(asByte(7));
          System.out.println(asByte(new Float("7.f")));
          // System.out.println(asByte(new Character('7'))); // ошибка компиляции
     }
}
```

Объекты типа **Integer** (**int** будет в него упакован) 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

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

/ пример # 14: определение количества параметров метода: DemoVarargs.java */*

```
package chapt03;

public class DemoVarargs {
      public static int getArgCount(Integer... args) {
             if (args.length == 0)
                   System.out.print("No arg=");
             for (int i : args)
                   System.out.print("arg:" + i + "  ");
             return args.length;
      }
```

```
      public static void main(String args[]) {
        System.out.println("N=" + getArgCount(7, 71, 555));
        Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
        System.out.println("N=" + getArgCount(i));
        System.out.println(getArgCount());
      }
}
```

В результате выполнения этой программы будет выведено:

```
arg:7  arg:71  arg:555  N=3
arg:1  arg:2  arg:3  arg:4  arg:5  arg:6  arg:7  N=7
No arg=0
```

В примере приведен простейший метод с переменным числом параметров. Метод **getArgCount()** выводит все переданные ему 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. Метод может быть вызван и без аргументов.

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

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

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {}
void methodName(String...args) {}
```

В следующем примере приведены три перегруженных метода и несколько вариантов их вызова. Отличительной чертой является возможность метода с аргументом **Object... args** принимать не только объекты, но и массивы:

```
/* пример # 15 : передача массивов: DemoOverload.java */
package chapt03;

public class DemoOverload {
      public static void printArgCount(Object... args) {//1
      System.out.println("Object args: " + args.length);
      }
      public static void printArgCount(Integer[]...args){//2
      System.out.println("Integer[] args: " + args.length);
      }
      public static void printArgCount(int... args) {//3
            System.out.print("int args: " + +args.length);
      }
      public static void main(String[] args) {
            Integer[] i = { 1, 2, 3, 4, 5 };

            printArgCount(7, "No", true, null);
            printArgCount(i, i, i);
            printArgCount(i, 4, 71);
            printArgCount(i);//будет вызван метод 1
```

```
            printArgCount(5, 7);
            // printArgCount();//неопределенность!
      }
}
```

В результате будет выведено:

```
Object args: 4
Integer[] args: 3
Object args: 3
Object args: 5
int args: 2
```

При передаче в метод **printArgCount()** единичного массива **i** компилятор отдает предпочтение методу с параметром **Object... args**, так как имя массива является объектной ссылкой и потому указанный параметр будет ближайшим. Метод с параметром **Integer[]...args** не вызывается, так как ближайшей объектной ссылкой для него будет **Object[]...args**. Метод с параметром **Integer[]...args** будет вызван для единичного массива только в случае отсутствия метода с параметром **Object...args**.

При вызове метода без параметров возникает неопределенность из-за невозможности однозначного выбора.

Не существует также ограничений и на переопределение подобных методов.

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

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

Перечисления

Типобезопасные перечисления (typesafe enums) в Java представляют собой классы и являются подклассами абстрактного класса **java.lang.Enum**. При этом объекты перечисления инициализируются прямым объявлением без помощи оператора **new**. При инициализации хотя бы одного перечисления происходит инициализация всех без исключения оставшихся элементов данного перечисления.

В качестве простейшего применения перечисления можно рассмотреть следующий код:

```
/* пример # 16 : применение перечисления: SimpleUseEnum.java */
package chapt02;

enum Faculty {
      MMF, FPMI, GEO
}
public class SimpleUseEnum {
      public static void main(String args[]) {
            Faculty current;
            current = Faculty.GEO;
            switch (current) {
              case GEO:
                  System.out.print(current);
                  break;
              case MMF:
```

```
                System.out.print(current);
                break;
            // case LAW : System.out.print(current);//ошибка компиляции!
              default:
             System.out.print("вне case: " + current);
            }
      }
}
```

В операторах **case** используются константы без уточнения типа перечисления, так как его тип определен в **switch**.

Перечисление как подкласс класса **Enum** может содержать поля, конструкторы и методы, реализовывать интерфейсы. Каждый тип **enum** может использовать методы:

static enumType[] values() – возвращает массив, содержащий все элементы перечисления в порядке их объявления;

static T valueOf(Class<T> enumType, String arg) – возвращает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки;

static enumType valueOf(String arg) – возвращает элемент перечисления, соответствующий значению передаваемой строки;

int ordinal() – возвращает позицию элемента перечисления.

/ пример # 17 : объявление перечисления с методом : Shape.java */*

```
package chapt02;

enum Shape {
      RECTANGLE, TRIANGLE, CIRCLE;
      public double square(double x, double y) {
            switch (this) {
            case RECTANGLE:
                  return x * y;
            case TRIANGLE:
                  return x * y / 2;
            case CIRCLE:
                  return Math.pow(x, 2) * Math.PI;
            }
            throw new EnumConstantNotPresentException(
                  this.getDeclaringClass(),this.name());
      }
}
```

/ пример # 18 : применение перечисления: Runner.java */*

```
package chapt02;

public class Runner {
      public static void main(String args[]) {
            double x = 2, y = 3;
            Shape[] arr = Shape.values();
```

```
            for (Shape sh : arr)
                  System.out.printf("%10s = %5.2f%n",
                              sh, sh.square(x, y));
      }
}
```

В результате будет выведено:

```
RECTANGLE =  6,00
 TRIANGLE =  3,00
   CIRCLE = 12,57
```

Каждый из элементов перечисления в данном случае представляет собой в том числе и арифметическую операцию, ассоциированную с методом **square()**. Без **throw** данный код не будет компилироваться, так как компилятор не исключает появления неизвестного элемента. Данная инструкция позволяет указать на возможную ошибку при появлении необъявленной фигуры. Поэтому и при добавлении нового элемента необходимо добавлять соответствующий ему **case**.

/ пример # 19 : конструкторы и члены перечисления: DeanDemo.java */*

```
package chapt02;

enum Dean {
   MMF("Бендер"), FPMI("Балаганов"), GEO("Козлевич");
      String name;

      Dean(String arg) {
            name = arg;
      }
      String getName() {
            return name;
      }
}
package chapt02;

public class DeanDemo {
  public static void main(String[] args) {
   Dean dn = Dean.valueOf("FPMI");
   System.out.print(dn.ordinal());
   System.out.println(" : " + dn + " : " + dn.getName());
      }
}
```

В результате будет выведено:

```
1 : FPMI : Балаганов
```

Однако на перечисления накладывается целый ряд ограничений.

Им запрещено:

- быть суперклассами;
- быть подклассами;
- быть абстрактными;
- создавать экземпляры, используя ключевое слово **new**.

Аннотации

Аннотации – мета-теги, которые добавляются к коду и применяются к объявлению пакетов, типов, конструкторов, методов, полей, параметров и локальным переменным. В результате можно задать зависимость, например, метода от другого метода. Аннотации позволяют избежать создания шаблонного кода во многих ситуациях, активируя утилиты для его генерации из аннотаций в исходном коде.

В следующем коде приведено объявление аннотации.

```
/* пример # 20 : многочленная аннотация : RequestForCustomer.java */
package chapt03;

public @interface RequestForCustomer {
      int level();
      String description();
      String date();
}
```

Ключевому слову **interface** предшествует символ **@**. Такая запись сообщает компилятору об объявлении аннотации. В объявлении также есть три метода-члена: **int level()**, **String description()**, **String date()**.

Все аннотации содержат только объявления методов, добавлять тела этим методам не нужно, так как их реализует сам язык. Кроме того, эти методы не могут содержать параметров, секции **throws** и действуют скорее как поля. Допустимые типы возвращаемого значения: базовые типы, **String**, **Enum**, **Class** и массив любого из вышеперечисленных типов.

Все типы аннотаций автоматически расширяют интерфейс **java.lang.annotation.Annotation**. В этом интерфейсе даны методы: **hashCode()**, **equals()** и **toString()**, определенные в типе **Object**. В нем также приведен метод **annotationType()**, который возвращает объект типа **Class**, представляющий вызывающую аннотацию.

После объявления аннотации ее можно использовать для аннотирования объявлений. Объявление любого типа может иметь аннотацию, связанную с ним. Например, можно снабжать примечаниями классы, методы, поля, параметры и константы типа **enum**. Даже к аннотации можно добавить аннотацию. Во всех случаях аннотация предшествует объявлению.

Применяя аннотацию, нужно задавать значения для ее методов-членов. Далее приведен фрагмент, в котором аннотация **RequestForCustomer** сопровождает объявление метода:

```
@RequestForCustomer (
   level = 2,
   description = "Enable time",
   date = "10/10/2007"
)
public void customerThroughTime() {
   //...
}
```

Данная аннотация связана с методом **customerThroughTime()**. За именем аннотации, начинающимся с символа **@**, следует заключенный в круглые скобки список инициализирующих значений для методов-членов. Для того чтобы передать значение методу-члену, имени этого метода присваивается значение. Таким образом, в приведенном фрагменте строка **"Enable time"** присваивается методу **description()**, члену аннотации типа **RequestForCustomer**. При этом в присваивании после имени **description** нет круглых скобок. Когда методу-члену передается инициализирующее значение, используется только имя метода. Следовательно, в данном контексте методы-члены выглядят как поля.

```
/* пример # 21 : применение аннотации: Request.java */
package chapt03;
import java.lang.reflect.Method;

public class Request {
      @RequestForCustomer(level = 2,
                          description = "Enable time",
                          date = "10/10/2007")
      public void customerThroughTime() {
            try {
            Class c = this.getClass();
            Method m = c.getMethod("customerThroughTime");
            RequestForCustomer ann =
            m.getAnnotation(RequestForCustomer.class);
                  //запрос аннотаций
                  System.out.println(ann.level() + " "
                              + ann.description() + " "
                              + ann.date());
            } catch (NoSuchMethodException e) {
                  System.out.println("метод не найден");
            }
      }
      public static void main(String[] args) {
            Request ob = new Request();
            ob.customerThroughTime();
      }

   }
```

В результате будет выведено:

```
2 Enable time 10/10/2007
```

Если аннотация объявляется в отдельном файле, то ей нужно задать правило сохранения **RUNTIME** в виде кода

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME) //правило сохранения
```

помещаемого перед объявлением аннотации, которое предоставляет максимальную продолжительность существования аннотации.

С правилом **SOURCE** аннотация существует только в исходном тексте программы и отбрасывается во время компиляции.

Аннотация с правилом сохранения **CLASS** помещается в процессе компиляции в файл **.class**, но не доступна в JVM во время выполнения.

Аннотация, заданная с правилом сохранения **RUNTIME**, помещается в файл **.class** в процессе компиляции и доступна в JVM во время выполнения. Следовательно, правило **RUNTIME** предлагает максимальную продолжительность существования аннотации.

Основные типы аннотаций: аннотация-маркер, одночленная и многочленная.

Аннотация-маркер не содержит методов-членов. Цель – пометить объявление. В этом случае достаточно присутствия аннотации. Поскольку у интерфейса аннотации-маркера нет методов-членов, достаточно определить наличие аннотации.

```
public @interface TigerAnnotation {}
```

Для проверки наличия аннотации-маркера используется метод **isAnnotationPresent()**.

Одночленная аннотация содержит единственный метод-член. Для этого типа аннотации допускается краткая условная форма задания значения для метода-члена. Если есть только один метод-член, то просто указывается его значение при создании аннотации. Имя метода-члена указывать не нужно. Но для того чтобы воспользоваться краткой формой, следует для метода-члена использовать имя **value()**.

Многочленные аннотации содержат несколько методов-членов. Поэтому используется полный синтаксис (**имя_параметра = значение**) для каждого параметра.

В языке Java определено семь типов встроенных аннотаций, четыре типа – **@Retention**, **@Documented**, **@Target** и **@Inherited** – импортируются из пакета **java.lang.annotation**. Оставшиеся три – **@Override**, **@Deprecated** и **@Suppresswarnings** – из пакета **java.lang**.

Аннотации получают все более широкое распространение и активно используются в различных технологиях.

Задания к главе 3

Вариант А

1. Определить класс **Вектор** размерности n. Реализовать методы сложения, вычитания, умножения, инкремента, декремента, индексирования. 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 **Вектор** 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 **Вектор** 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 **Матрица** 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 **Матрица** 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 **Комплекс**. 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 масссив 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 **Полином** 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 **Рациональный полином** с полем типа **Полином**. 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 биекции.
15. Определить класс **Определённый интеграл** с аналитически подынтегральной функцией. Написать несколько конструкторов. Создать методы для вычисления значения по формуле левых прямоугольников, по формуле правых прямоугольников, по формуле средних прямоугольников, по формуле трапеций, по формуле Симпсона (параболических трапеций).
16. Определить класс **Массив** с аналитически подынтегральной функцией. Написать несколько конструкторов. Создать методы сортировки: обменная сортировка (метод пузырька); обменная сортировка «Шейкер-сортировка», сортировка посредством выбора (метод простого выбора), сортировка вставками: метод хеширования (сортировка с вычислением адреса), сортировка вставками (метод простых вставок), сортировка бинарного слияния, сортировка Шелла (сортировка с убывающим шагом).
17. Построить класс **Дерево**. Определить несколько конструкторов. Реализовать методы для отображение статистики об общем числе вершин в дереве, имеющих нечетные индексы, имеющих четные индексы, имеющих индексы, превышающие задаваемый пользователем порог. Определить метод вычисление расстояния между вершинами.

Вариант В

Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы **`set`**`Тип`**`()`**, **`get`**`Тип`**`()`**, **`toString()`**. 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. **Airlines**: 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.

Какие из ключевых слов могут быть использованы при объявлении конструктора?

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

Вопрос 3.2.

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 3.3.

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 3.4.

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.

Глава 4

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Наследование

Отношение между классами, при котором характеристики одного класса (суперкласса) передаются другому классу (подклассу) без их повторного описания, называется ***наследованием***.

Подкласс наследует переменные и методы суперкласса, используя ключевое слово **`extends`**. Класс может также реализовывать любое число интерфейсов, используя ключевое слово – **`implements`**. Подкласс имеет прямой доступ ко всем открытым переменным и методам родительского класса, как будто они находятся в подклассе. Исключение составляют члены класса, помеченные **`private`** (во всех случаях) и «по умолчанию» для подкласса в другом пакете. В любом случае (даже если ключевое слово **`extends`** отсутствует) класс автоматически наследует свойства суперкласса всех классов – класса **`Object`**.

Множественное наследование классов запрещено, хотя его аналог предоставляет реализация интерфейсов, которые не являются классами и содержат описание набора методов, позволяющих задать поведение объекта класса, реализующего эти интерфейсы. Наличие общих методов, которые должны быть реализованы в разных классах, обеспечивают им сходную функциональность.

Подкласс дополняет члены базового класса своими переменными и методами, имена которых могут частично совпадать с именами членов суперкласса. Если имена методов совпадают, а параметры различаются, то такое явление называется перегрузкой методов.

В подклассе можно объявить (переопределить) метод с тем же именем, списком параметров и возвращаемым значением, что и у метода суперкласса.

Способность ссылки динамически определять версию переопределенного метода в зависимости от переданного ссылке в сообщении типа объекта называется ***полиморфизмом***.

Полиморфизм является основой для реализации механизма динамического или «позднего связывания».

В следующем примере переопределяемый метод **`typeEmployee()`** находится в двух классах **`Employee`** и **`Manager`**. В соответствии с принципом полиморфизма вызывается метод, наиболее близкий к текущему объекту.

```
/* пример # 1 : наследование класса и переопределение метода:
Employee.java: Manager.java:  Runner.java */
package chapt04;

public class Employee {// рядовой сотрудник
      private int id;
```

```
	public Employee(int idc) {
		super();/* по умолчанию, необязательный явный вызов
				конструктора суперкласса */
		id = idc;
	}
	public int getId() {
		return id;
	}
	public void typeEmployee() {
		//…
		System.out.println("Работник");
	}
}

package chapt04;
// сотрудник с проектом, за который он отвечает

public class Manager extends Employee {
	private int idProject;

	public Manager(int idc, int idp) {
		super(idc); /* вызов конструктора суперкласса
				с параметром */
		idProject = idp;
	}
	public int getIdProject() {
		return idProject;
	}
	public void typeEmployee() {
		//…
		System.out.println("Менеджер");
	}
}
package chapt04;

public class Runner {
	public static void main(String[] args) {
		Employee b1 = new Employee(7110);
		Employee b2 = new Manager(9251, 31);
		b1.typeEmployee();// вызов версии из класса Employee
		b2.typeEmployee();// вызов версии из класса Manager
		// b2.getIdProject();// ошибка компиляции!!!
		((Manager) b2).getIdProject();
		Manager b3 = new Manager(9711, 35);
		System.out.println(b3.getIdProject());// 35
		System.out.println(b3.getId());// 9711
	}
}
```

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

При объявлении совпадающих по сигнатуре (имя, тип, область видимости) 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. Для доступа к полям текущего объекта можно использовать указатель **this**, для доступа к полям суперкласса – указатель **super**. Другие возможности рассмотрены в следующем примере:

/ пример # 2 : создание объекта подкласса и доступ к полям с одинаковыми именами: Course.java: BaseCourse.java: Logic.java */*

```
package chapt04;

public class Course {
      public int id = 71;

      public Course() {
       System.out.println("конструктор класса Course");
       id = getId();//!!!
       System.out.println(" id=" + id);
      }
      public int getId() {
            System.out.println("getId() класса Course");
            return id;
      }
}
package chapt04;

public class BaseCourse extends Course {
      public int id = 90;// так делать не следует!

      public BaseCourse() {
       System.out.println("конструктор класса BaseCourse");
       System.out.println(" id=" + getId());
      }
      public int getId() {
          System.out.println("getId() класса BaseCourse");
          return id;
      }
}
package chapt04;

public class Logic {
```

```
      public static void main(String[] args) {
            Course objA = new BaseCourse();
            BaseCourse objB = new BaseCourse();
            System.out.println("objA: id=" + objA.id);
            System.out.println("objB: id=" + objB.id);
            Course objC = new Course();
      }
}
```

В результате выполнения данного кода последовательно будет выведено:

```
конструктор класса Course
getId() класса BaseCourse
 id=0
конструктор класса BaseCourse
getId() класса BaseCourse
 id=90
конструктор класса Course
getId() класса BaseCourse
 id=0
конструктор класса BaseCourse
getId() класса BaseCourse
 id=90
objA: id=0
objB: id=90
конструктор класса Course
getId() класса Course
 id=71
```

Метод **getId()** содержится как в классе **Course**, так и в классе **BaseCourse** и является переопределенным. При создании объекта класса **BaseCourse** одним из способов:

```
Course objA = new BaseCourse();
BaseCourse objB = new BaseCourse();
```

в любом случае перед вызовом конструктора **BaseCourse()** вызывается конструктор класса **Course**. Но так как в обоих случаях создается объект класса **BaseCourse**, то вызывается метод **getId()**, объявленный в классе **BaseCourse**, который в свою очередь оперирует полем **id**, еще не проинициализированным для класса **BaseCourse**. В результате **id** получит значение по умолчанию, т.е. нуль.

Воспользовавшись преобразованием типов вида **((BaseCourse)objA).id** или **((Course)objB).id**, легко можно получить доступ к полю **id** из соответствующего класса.

Использование final

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

```
// класс ConstCourse не может быть суперклассом
final class ConstCourse {/*код*/}
```

```
// следующий класс невозможен
class BaseCourse extends ConstCourse {/*код*/}
```

Использование super и this

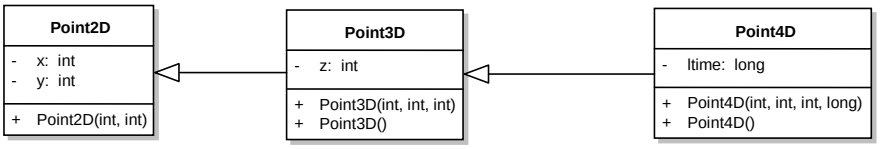
Ключевое слово **super** используется для вызова конструктора суперкласса и для доступа к члену суперкласса. Например:

```
super(список_параметров);/* вызов конструктора суперкласса
                            с передачей параметров или без нее*/
super.id = 71; /* обращение к атрибуту суперкласса */
super.getId(); // вызов метода суперкласса
```

Вторая форма **super** используется для доступа из подкласса к переменной **id** суперкласса. Третья форма специфична для Java и обеспечивает вызов из подкласса переопределенного метода суперкласса, 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 метод «знает», какой объект его вызвал. Вместо обращения к атрибуту **id** в методах можно писать **this.id,** хотя и не обязательно, так как записи **id** и **this.id** равносильны.

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




```
// пример # 3 : this в конструкторе: Point2D.java, Point3D.java, Point4D.java
package chapt04;

public class Point2D {
      private int x, y;

      public Point2D(int x, int y) {
            this.x = x;//this используется для присваивания полям класса
            this.y = y;//x, y, значений параметров конструктора x, y, z
      }
}
package chapt04;

public class Point3D extends Point2D {
      private int z;

      public Point3D(int x, int y, int z) {
            super(x, y);
            this.z = z;
      }
```

```
        public Point3D() {
            this(-1,-1,-1);// вызов конструктора Point3D с параметрами
        }
}
package chapt04;

public class Point4D extends Point3D{
        private long time;

        public Point4D(int x, int y, int z, long time) {
                super(x, y, z);
                this.time = time;
        }
        public Point4D() {
                // по умолчанию super();
        }
}
```

В классе **Point3D** второй конструктор для завершения инициализации объекта обращается к первому конструктору. Такая конструкция применяется в случае, когда в класс требуется добавить конструктор по умолчанию с обязательным использованием уже существующего конструктора.

Ссылка **this** используется в методе для уточнения того, о каких именно переменных **x**, **y** и **z** идет речь в методе, а конкретно для доступа к переменным класса из метода, если в методе есть локальные переменные с тем же именем, что и у класса. 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 объекта во время выполнения, называется ***поздним связыванием***. При вызове метода его поиск происходит сначала в данном классе, затем в суперклассе, пока метод не будет найден или не достигнут **Object** – суперкласс для всех классов.

Если два метода с одинаковыми именами и возвращаемыми значениями находятся в одном классе, то списки их параметров должны отличаться. То же относится к методам, наследуемым из суперкласса. Такие методы являются перегружаемыми (overloading). При обращении вызывается тот метод, список параметров которого совпадает со списком параметров вызова. Если объявление метода подкласса полностью, включая параметры, совпадает с объявлением метода суперкласса (порождающего класса), то метод подкласса переопределяет (overriding) метод суперкласса. 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 раз-

личных подклассов, в зависимости от того, на объект какого подкласса у него имеется ссылка.

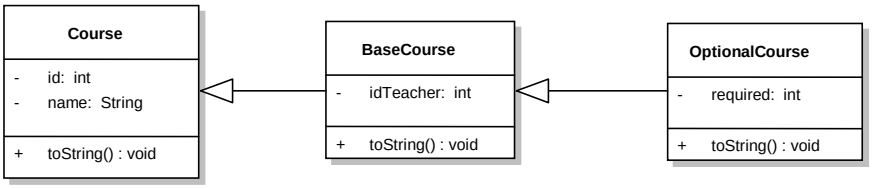


Рис. 4.1. Пример реализации полиморфизма

```
/* пример # 4 : динамическое связывание методов:  Course.java: BaseCourse.java:
OptionalCourse.java: DynDispatcher.java */
package chapt04;

public class Course {
      private int id;
      private String name;

      public Course(int i, String n) {
            id = i;
            name = n;
      }
      public String toString() {
            return "Название: " + name + "(" + id + ")";
      }
}
package chapt04;

public class BaseCourse extends Course {
      private int idTeacher;

      public BaseCourse(int i, String n, int it) {
            super(i, n);
            idTeacher = it;
      }
      public String toString() {
            /* просто toString() нельзя!!!
               метод будет вызывать сам себя, что
               приведет к ошибке во время выполнения  */
   return
      super.toString() + " препод.(" + idTeacher + ")";
      }
}
package chapt04;

public class OptionalCourse extends BaseCourse {
      private boolean required;
```

```
      public OptionalCourse(int i, String n, int it,
                                              boolean r) {
            super(i, n, it);
            required = r;
      }
      public String toString() {
        return super.toString() + " required->" + required;
      }
}
package chapt04;

public class DynDispatcher{
      public void infoCourse(Course c) {
            System.out.println(c.toString());
            //System.out.println(c);//идентично
      }
}
package chapt04;

public class Runner {
      public static void main(String[] args) {
            DynDispatcher d = new DynDispatcher();
            Course cc = new Course(7, "МА");
            d.infoCourse(cc);
            BaseCourse bc = new BaseCourse(71, "МП", 2531);
            d.infoCourse(bc);
            OptionalCourse oc =
                  new OptionalCourse(35, "ФА", 4128, true);
            d.infoCourse(oc);
      }
}
```

Результат:

```
Название: МА(7)
Название: МП(71) препод.(2531)
Название: ФА(35) препод.(4128) required->true
```

Следует помнить, что при вызове **toString()** обращение **super** всегда происходит к ближайшему суперклассу. Аналогично при вызове **super()** в конструкторе обращение происходит к соответствующему конструктору непосредственного суперкласса.

Основной вывод: выбор версии переопределенного метода производится на этапе выполнения кода.

Все методы Java являются виртуальными (ключевое слово virtual, как в C++, не используется).

Статические методы могут быть переопределены в подклассе, но не могут быть полиморфными, так как их вызов не затрагивает объекты. Их следует вызывать только с использованием имени класса.

Методы подставки

С пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип.

```
/* пример # 5 : методы-подставки:  CourseHelper.java:
BaseCourseHelper.java: RunnerCourse.java*/
package chapt04;

public class CourseHelper {
      public Course getCourse(){
             System.out.println("Course");
            return new Course();
      }
}
package chapt04;

public class BaseCourseHelper extends CourseHelper {
       public BaseCourse getCourse(){
              System.out.println("BaseCourse");
            return new BaseCourse();
      }
}
package chapt04;

public class RunnerCourse {
      public static void main(String[] args) {
            CourseHelper bch = new BaseCourseHelper();
            Course course = bch.getCourse();
            //BaseCourse course = bch.getCourse();//ошибка компиляции
            System.out.println(bch.getCourse().id);
      }
}
```

В данной ситуации при компиляции в подклассе **BaseCourseHelper** создаются два метода. При обращении к методу **getCourse()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении вызывается метод-подставка. Обращение к полю производится по типу ссылки, возвращаемой методом **getCourse()**, то есть к полю класса **Course**.

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:

```
Transport s1 = new Bus();
Transport s2 = new Tram();
```

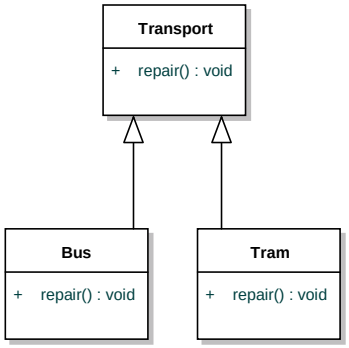


Рис. 4.2. Пример реализации полиморфизма

Базовый класс **Transport** предоставляет общий интерфейс для своих подклассов. Порожденные классы **Bus** и **Tram** перекрывают эти определения для обеспечения уникального поведения.

```
/* пример # 5 : полиморфизм: Transport.java: Bus.java: Tram.java:
RepairingCenter.java: Runner.java*/
package chapt04;
import java.util.Random;

class Transport {
      public void repair() {/* пустая реализация */
      }
}
class Bus extends Transport {
      public void repair() {
            System.out.println("отремонтирован АВТОБУС");
      }
}
class Tram extends Transport {
      public void repair() {
            System.out.println("отремонтирован ТРАМВАЙ");
      }
}
class RepairingFactory {//шаблон Factory
      public Transport getClassFromFactory(int numMode) {
            switch (new Random().nextInt(numMode)) {
             case 0:
                  return new Bus();
             case 1:
                  return new Tram();
             default:
```

```
                    throw new IllegalArgumentException();
                    // assert false;
                    // return null;
                    /*
                * if((int)(Math.random() * numMode)==0) return new Bus(); else
                * return new Tram(); как альтернативный и не очень удачный
                * вариант. Почему?
                     */
                }
        }
}
public class Runner {
        public static void main(String[] args) {
                RepairingFactory rc = new RepairingFactory();
                Transport[] box = new Transport[15];

        for (int i = 0; i < box.length; i++)
                /* заполнение массива единицами проверямого транспорта */
          box[i] = rc.getClassFromFactory(2);// 2 вида транспорта
                for (Transport s : box)
                        s.repair();// вызов полиморфного метода
        }
}
```

В процессе выполнения приложения будет случайным образом сформирован массив из автобусов и трамваев и информация об их ремонте будет выведена на консоль.

Класс **RepairingFactory** содержит метод **getClassFromFactory(int numMode)**, который возвращает ссылку на случайно выбранный объект подкласса класса **Transport** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **Bus** или **Tram**. Метод **main()** содержит массив из ссылок **Transport**, заполненный с помощью вызова **getClassFromFactory()**. 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 **repair()** вызывается для каждого случайным образом выбранного объекта.

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

Статические методы и полиморфизм

Переопределение статических методов класса не имеет практического смысла, так как обращение к статическому атрибуту или методу осуществляется посредством задания имени класса, которому они принадлежат. К статическим методам принципы «позднего связывания» 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 # 6 : поведение статического метода при «переопределении»:
Runner.java */
package chapt04;

class Base {
      public static void assign() {
            System.out.println(
                  "метод assign() из Base");
      }
}
class Sub extends Base {
      public static void assign() {
            System.out.println(
                  "метод assign() из Sub");
      }
}
public class Runner {
      public static void main(String[] args) {
            Base ob1 = new Base();
            Base ob2 = new Sub();
            Sub ob3 = new Sub();
            ob1.assign();//некорректный вызов статичесого метода
            ob2.assign();//следует вызывать Base.assign();
            ob3.assign();
      }
}
```

В результате выполнения данного кода будет выведено:

```
метод assign() из Base
метод assign() из Base
метод assign() из Sub
```

При таком способе инициализации объектов **ob1** и **ob2**, метод **assign()** будет вызван из класса **Base**. Для объекта **ob3** будет вызван собственный метод **assign()**, что следует из способа объявления объекта. Если же спецификатор **static** убрать из объявления методов, то вызовы методов будут осуществляться в соответствии с принципами полиморфизма.

Статические методы всегда следует вызывать через имя класса, в котором они объявлены, а именно:

```
Base.assign();
Sub.assign();
```

Вызов статических методов через объект считается нетипичным и нарушающим смысл статического определения.

Абстракция и абстрактные классы

Множество предметов реального мира обладает некоторым набором общих характеристик и правил поведения. Абстрактное понятие «Геометрическая фигура» может содержать описание геометрических параметров и расположения центра тяжести в системе координат, а также возможности определения площади и

периметра фигуры. Однако в общем случае дать конкретную реализацию приведенных характеристик и функциональности невозможно ввиду слишком общего их определения. Для конкретного понятия, например «Квадрат», дать описание линейных размеров и определения площади и периметра не составляет труда. Абстрагирование понятия должно предоставлять абстрактные характеристики предмета реального мира, а не его ожидаемую реализацию. Грамотное выделение абстракций позволяет структурировать код программной системы в целом и повторно использовать абстрактные понятия для конкретных реализаций при определении новых возможностей абстрактной сущности.

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. При этом допустимо объявлять ссылку на абстрактный класс, но инициализировать ее можно только объектом производного от него класса. Абстрактные классы могут содержать и полностью реализованные методы, а также конструкторы и поля данных.

С помощью абстрактного класса объявляется контракт (требования к функциональности) для его подклассов. Примером может служить уже рассмотренный выше абстрактный класс **Number** и его подклассы **Byte**, **Float** и другие. Класс **Number** объявляет контракт на реализацию ряда методов по преобразованию данных к значению конкретного базового типа, например **floatValue()**. Можно предположить, что реализация метода будет различной для каждого из классов-оболочек. Хотя объект класса **Number** нельзя создать, он может получить численное значение любого базового типа. Однако у самого класса нет возможности преобразовать это значение к конкретному базовому типу.

/ пример # 7 : абстрактный класс и метод : AbstractManager.java */*

```
package chapt04;

public abstract class AbstractManager {
      private int id;
      public AbstractManager(int id) {// конструктор
            this.id = id;
      }
      // абстрактный метод
      public abstract void assignGroupToCourse(
                  int groupId, String nameCourse);
}
```

/ пример # 8 : подкласс абстрактного класса : CourseManager.java */*

```
package chapt04;


// assignGroupToCourse() должен быть реализован в подклассе
public class CourseManager extends AbstractManager {
      public void assignGroupToCourse(
                  int groupId, String nameCourse) {
            //...
```

```
            System.out.println("группа " + groupId
                  + " назначена на курс " + nameCourse);
      }
}
```

/ пример # 9 : объявление объектов и вызов методов : Runner.java */*

```
package chapt04;

public class Runner {
      public static void main(String[] args) {
            AbstractManager mng; // можно объявить ссылку
          // mng = new AbstractManager();нельзя создать объект!
            mng = new CourseManager();
            mng.assignGroupToCourse(10, "Алгебра");
      }
}
```

В результате будет получено:

```
группа 10 назначена на курс Алгебра
```

Ссылка на абстрактный суперкласс **mng** инициализируется объектом подкласса, в котором реализованы все абстрактные методы суперкласса. С помощью этой ссылки могут вызываться реализованные методы абстрактного класса, если они не переопределены в подклассе.

Класс Object

На вершине иерархии классов находится класс **Object**, который является суперклассом для всех классов. Ссылочная переменная типа **Object** может указывать на объект любого другого класса, на любой массив, так как массивы реализуются как классы. В классе **Object** определен набор методов, который наследуется всеми классами:

protected Object clone() – создает и возвращает копию вызывающего объекта;

boolean equals(Object ob) – предназначен для переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов;

Class<? extends Object> getClass() – возвращает объект типа **Class**;

protected void finalize() – вызывается перед уничтожением объекта автоматическим сборщиком мусора (garbage collection);

int hashCode() – возвращает хэш-код объекта;

String toString() – возвращает представление объекта в виде строки.

Методы **notify()**, **notifyAll()** и **wait()** будут рассмотрены в главе «Потоки выполнения».

Если при создании класса предполагается проверка логической эквивалентности объектов, которая не выполнена в суперклассе, следует переопределить два метода: **equals(Object ob)** и **hashCode()**. Кроме того, переопределе-

ние этих методов необходимо, если логика приложения предусматривает использование элементов в коллекциях. Метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае. При переопределении метода **equals()** должны выполняться соглашения, предусмотренные спецификацией языка Java, а именно:

- рефлексивность – объект равен самому себе;
- симметричность – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- транзитивность – если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- непротиворечивость – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- ненулевая ссылка при сравнении с литералом **null** всегда возвращает значение **false**.

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

Метод **hashCode()** переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта. Его следует переопределять всегда, когда переопределен метод **equals()**. Метод **hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа **должны** иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа **могут** иметь различные хэш-коды.

Один из способов создания правильного метода **hashCode()**, гарантирующий выполнение соглашений, приведен ниже, в примере # 10.

Метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**. Метод **toString()** класса **Object** возвращает строку с описанием объекта в виде:

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

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

```
/* пример # 10 : переопределение методов equals(), hashCode, toString():
Student.java */
package chapt04;
```

```
public class Student {
	private int id;
	private String name;
	private int age;

	public Student(int id, String name, int age){
		this.id = id;
		this.name = name;
		this.age = age;
	}
	public int getId() {
		return id;
	}
	public String getName() {
		return name;
	}
	public int getAge() {
		return age;
	}
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (obj instanceof Student){ //warning
			Student temp = (Student) obj;
			return this.id == temp.id &&
				name.equals(temp.name) &&
				this.age == temp.age;
		} else
			return false;
	}
	public int hashCode() {
		return (int)(31 * id + age
		+ ((name == null) ? 0 : name.hashCode()));
	}
	public String toString() {
		return getClass().getName() + "@name" + name
				+ " id:" + id + " age:" + age;
	}
}
```

Выражение **`31 * id + age`** гарантирует различные результаты вычислений при перемене местами значений полей, а именно если **`id=1`** и **`age=2`**, то в результате будет получено **`33`**, если значения поменять местами, то **`63`**. Такой подход применяется при наличии у классов полей базовых типов.

Метод **`equals()`** переопределяется для класса **`Student`** таким образом, чтобы убедиться в том, что полученный объект является объектом типа **`Student`** или одним из его наследников, а также сравнить содержимое полей **`id`**, **`name`** и

age соответственно у вызывающего метод объекта и объекта, передаваемого в качестве параметра.

*/*пример # 11 : класс студента факультета: SubStudent.java */*

```
package chapt04;

public class SubStudent extends Student {
     private int idFaculty;

     public SubStudent (int id, String n, int a, int idf){
             super(id, n, a);
             this.idFaculty = idf;
       }
}
```

*/*пример # 12 : демонстрация работы метода equals() при наследовании: StudentEq.java */*

```
package chapt04;

public class StudentEq {
     public static void main(String[] args) {
             Student p1 = new Student(71, "Петров", 19);
             Student p2 = new Student(71, "Петров", 19);
             SubStudent p3 =
                  new SubStudent(71, "Петров", 19, 5);
             System.out.println(p1.equals(p2));
             System.out.println(p1.equals(p3));
             System.out.println(p3.equals(p1));
     }
}
```

В результате выполнения данного кода будет выведено следующее:

```
true
true
true
```

Переопределенный таким образом метод **equals()** позволяет сравнивать объекты суперкласса с объектами подклассов, но только по тем полям, которые являются общими. При наследовании с добавлением новых полей в подкласс использование метода сравнения из суперкласса приводит к некорректным результатам.

Эту проблему можно легко разрешить, если вместо строки с пометкой *//warning* в метод **equals()** класса **Student** подставить непосредственную проверку на соответствие типов сравниваемых объектов с использованием объекта класса **Class** в виде:

```
if (getClass() == obj.getClass())
```

то в результате будет выведено:

```
true
false
false
```

В то же время такая реализация метода **equals()** будет возвращать истину при сравнении объектов класса **SubStudent** с одинаковыми значениями полей, унаследованных от класса **Student**.

Клонирование объектов

Объекты в методы передаются по ссылке, в результате чего в метод передается ссылка на объект, находящийся вне метода. Поэтому если в методе изменить значение поля объекта, то это изменение коснется исходного объекта. Во избежание такой ситуации для защиты внешнего объекта следует создать клон (копию) объекта в методе. Класс **Object** содержит **protected**-метод **clone()**, осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод **clone()** как **public** для обеспечения возможности вызова из другого пакета. В переопределенном методе следует вызвать базовую версию метода **super.clone()**, которая и выполняет собственно клонирование. Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс **Cloneable**. Интерфейс **Cloneable** не содержит методов относится к помеченным (tagged) интерфейсам, а его реализация гарантирует, что метод **clone()** класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение **CloneNotSupportedException**. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора. В языке C++ аналогичный механизм реализован с помощью конструктора копирования.

```
/* пример # 13 : класс, поддерживающий клонирование: Student.java */
package chapt04;

public class Student implements Cloneable {/*включение
                                              интерфейса */
      private int id = 71;

      public int getId() {
            return id;
      }
      public void setId(int value) {
            id = value;
      }
      public Object clone() {//переопределение метода
            try {
                  return super.clone();//вызов базового метода
            } catch (CloneNotSupportedException e) {
                  throw new AssertionError("невозможно!");
            }
      }
}
```

```
/* пример # 14 : безопасная передача по ссылке: DemoSimpleClone.java */
package chapt04;
```

```
public class DemoSimpleClone {
      private static void changeId(Student p) {
            p = (Student) p.clone();//клонирование
            p.setId(1000);
            System.out.println("->id = " + p.getId());
            }
      public static void main(String[] args) {
            Student ob = new Student();
            System.out.println("id = " + ob.getId());
            changeId(ob);
            System.out.println("id = " + ob.getId());
      }
}
```

В результате будет выведено:

```
id = 71
->id = 1000
id = 71
```

Если закомментировать вызов метода **clone()**, то выведено будет следующее:

```
id = 71
->id = 1000
id = 1000
```

Такое решение эффективно только в случае, если поля клонируемого объекта представляют собой значения базовых типов и их оболочек или неизменяемых (immutable) объектных типов. Если же поле клонируемого типа является изменяемым объектным типом, то для корректного клонирования требуется другой подход. Причина заключается в том, что при создании копии поля оригинал и копия представляют собой ссылку на один и тот же объект. В этой ситуации следует также клонировать и объект поля класса.

```
/* пример # 15 : глубокое клонирование: Student.java */
package chapt04;
import java.util.ArrayList;

public class Student implements Cloneable {
      private int id = 71;
      private ArrayList<Mark> lm = new ArrayList<Mark>();

      public int getId() {
            return id;
      }
      public void setId(int id) {
            this.id = id;
      }
      public ArrayList<Mark> getMark() {
            return lm;
      }
```

```
    public void setMark(ArrayList<Mark> lm) {
         this.lm = lm;
    }
    public Object clone() {
          try {
                Student copy =(Student)super.clone();
                copy.lm = (ArrayList<Mark>)lm.clone();
                return copy;
          } catch (CloneNotSupportedException e) {
                throw new AssertionError(
                            "отсутствует Cloneable!");
          }
    }
}
```

Такое клонирование возможно только в случае, если тип атрибута класса также реализует интерфейс **Cloneable** и переопределяет метод **clone()**. В противном случае вызов метода невозможен, так как он просто недоступен. Следовательно, если класс имеет суперкласс, то для реализации механизма клонирования текущего класса необходимо наличие корректной реализации такого механизма в суперклассе. При этом следует отказаться от использования объявлений **final** для полей объектных типов по причине невозможности изменения их значений при реализации клонирования.

"Сборка мусора" и освобождение ресурсов

Так как объекты создаются динамически с помощью операции **new**, а уничтожаются автоматически, то желательно знать механизм ликвидации объектов и способ освобождения памяти. Автоматическое освобождение памяти, занимаемой объектом, выполняется 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 **System.gc()** или **Runtime.getRuntime().gc()**, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным. Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов утративших все ссылки.

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций могут применяться два способа: конструкция **try-finally** и механизм finalization. Конструкция **try-finally** является предпочтительной, абсолютно надежной и будет рассмотрена в девятой главе. Запуск механизма finalization определяется алгоритмом сборки мусора и до его непосредственного исполнения может пройти сколь угодно много времени. Из-за всего этого поведение метода **finalize()** может повлиять на корректную работу программы, особенно при смене JVM. Если существует возможность освободить ресурсы или выполнить другие подобные действия без привлечения этого механизма, то лучше без него

обойтись. Виртуальная машина вызывает этот метод всегда, когда она собирается уничтожить объект данного класса. Внутри метода **finalize()**, вызываемого непосредственно перед освобождением памяти, следует определить действия, которые должны быть выполнены до уничтожения объекта.

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

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

Ключевое слово **protected** запрещает доступ к **finalize()** коду, определенному вне этого класса. Метод **finalize()** вызывается только перед самой "сборкой мусора", а не тогда, когда объект выходит из области видимости, то есть заранее невозможно определить, когда **finalize()** будет выполнен, и недоступный объект может занимать память довольно долго. В принципе этот метод может быть вообще не выполнен! Недопустимо в приложении доверять такому методу критические по времени действия по освобождению ресурсов.

```
/* пример # 16 : класс Manager с поддержкой finalization : Manager.java */
package chapt04;

class Manager {
      private int id;

      public Manager(int value) {
            id = value;
      }
      protected void finalize() throws Throwable {
          try {
          //освобождение ресурсов
        System.out.println("объект будет удален, id=" + id);
          } finally {
              super.finalize();
          }
    }
}
package chapt04;

public class FinalizeDemo {
     public static void main(String[] args) {
          Manager d1 = new Manager(1);
          d1 = null;
          Manager d2 = new Manager(2);
          Object d3 = d2;                    //1
          //Object d3 = new Manager (3);     //2
          d2 = d1;
          System.gc();// просьба выполнить "сборку мусора"
     }
}
```

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

```
объект будет удален, id=1
```

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

```
объект будет удален, id=1
объект будет удален, id=2
```

Если не вызвать явно метод **`finalize()`** суперкласса, то он не будет вызван автоматически. Еще одна опасность заключается в том, что если при выполнении данного метода возникнет исключительная ситуация, то она будет проигнорирована и приложение будет продолжать выполняться, что также представляет опасность для его корректной работы.

Задания к главе 4

Вариант А

Создать приложение, удовлетворяющее требованиям, приведенным в задании. Аргументировать принадлежность классу каждого создаваемого метода и корректно переопределить для каждого класса методы **`equals()`**, **`hashCode()`**, **`toString()`**.

1. Создать объект класса **Текст**, используя класс **Абзац**. Методы: дополнить текст, вывести на консоль текст, заголовок текста.
2. Создать объект класса **Автомобиль**, используя класс **Колесо**. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса **Самолет**, используя класс **Крыло**. Методы: летать, задавать маршрут, вывести на консоль маршрут.
4. Создать объект класса **Беларусь**, используя класс **Область**. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса **Планета**, используя класс **Материк**. Методы: вывести на консоль название материка, планеты, количество материков.
6. Создать объект класса **Звездная система**, используя классы **Планета**, **Звезда**, **Луна**. Методы: вывести на консоль количество планет в звездной системе, название звезды, добавление планеты в систему.
7. Создать объект класса **Компьютер**, используя классы **Винчестер**, **Дисковод**, **ОЗУ**. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера.
8. Создать объект класса **Квадрат**, используя классы **Точка, Отрезок.** Методы: задание размеров, растяжение, сжатие, поворот, изменение цвета.
9. Создать объект класса **Круг**, используя классы **Точка, Окружность.** Методы: задание размеров, изменение радиуса, определение принадлежности точки данному кругу.
10. Создать объект класса **Котёнок**, используя классы **Животное, Кошка.** Методы: вывести на консоль имя, подать голос, рожать потомство (создавать себе подобных).

11. Создать объект класса **Наседка**, используя классы **Птица, Кукушка.** Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса **Текстовый файл**, используя класс **Файл.** Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса **Одномерный массив**, используя класс **Массив.** Методы: создать, вывести на консоль, выполнить операции (сложить, вычесть, перемножить).
14. Создать объект класса **Простая дробь**, используя класс **Число.** Методы: вывод на экран, сложение, вычитание, умножение, деление.
15. Создать объект класса **Дом**, используя классы **Окно, Дверь.** Методы: закрыть на ключ, вывести на консоль количество окон, дверей.
16. Создать объект класса **Роза**, используя классы **Лепесток, Бутон.** Методы: расцвести, завять, вывести на консоль цвет бутона.
17. Создать объект класса **Дерево**, используя классы **Лист.** Методы: зацвести, опасть листьям, покрыться инеем, пожелтеть листьям.
18. Создать объект класса **Пианино**, используя класс **Клавиша.** Методы: настроить, играть на пианино, нажимать клавишу.
19. Создать объект класса **Фотоальбом**, используя класс **Фотография.** Методы: задать название фотографии, дополнить фотоальбом фотографией, вывести на консоль количество фотографий.
20. Создать объект класса **Год**, используя классы **Месяц, День.** Методы: задать дату, вывести на консоль день недели по заданной дате, рассчитать количество дней, месяцев в заданном временном промежутке.
21. Создать объект класса **Сутки**, используя классы **Час, Минута.** Методы: вывести на консоль текущее время, рассчитать время суток (утро, день, вечер, ночь).
22. Создать объект класса **Птица**, используя класс **Крылья.** Методы: летать, питаться.
23. Создать объект класса **Тигр**, используя класс **Когти.** Методы: рычать, бежать, добывать пищу.
24. Создать объект класса **Гитара**, используя класс **Струна.** Методы: играть, натягивать струну.

Вариант В

Построить модель программной системы.

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 при 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 **Дома. 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.
13. **Система Периодические издания. Читатель** может сделать **Заявку**, предварительно выбрав периодические **Издания** из списка. Система подсчитывает сумму для оплаты. **Читатель** оплачивает **заявку. Администратор** добавляет **Заявку** в «черный список», если **Клиент** не оплачивает её в определённый срок.

14. **Система Заказ гостиницы**. **Клиент** оставляет **Заявку** на **Номер**, указав количество мест в номере, класс апартаментов и время пребывания. **Администратор** рассматривает **Заявку**, подтверждает или отклоняет её. Результат просматривает **Клиент**. В случае подтверждения **Заявки Клиент** оплачивает услуги.
15. **Система Жилищно-коммунальные услуги**. **Квартиросъемщик** отправляет **Заявку**, в которой указывает род работ, масштаб и желаемое время выполнения. **Диспетчер** формирует соответствующую **Бригаду** и регистрирует её в **Плане работ**. **Диспетчер** может отклонить **Заявку** в случае занятости всех **Бригад**.
16. **Система Прокат автомобилей**. **Клиент** выбирает **Автомобиль** из списка доступных, заполняет форму **Заказа**, указывая паспортные данные, срок аренды. **Администратор** может отклонить **Заявку**, указав причины отказа. При подтверждении **Заявки Клиент** оплачивает **Заказ**. **Система** выписывает сумму. В случае повреждения **Автомобиля Клиентом Администратор** вносит соответствующие пометки.

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 два)

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
ПРОЕКТИРОВАНИЕ КЛАССОВ

Шаблоны проектирования GRASP

При создании классов, распределении обязанностей и способов взаимодействия объектов перед программистом возникает серьезная проблема моделирования взаимодействия классов и объектов. Представляются широкие возможности выбора. Неоптимальный выбор может сделать системы и их отдельные компоненты непригодными для поддержки, восприятия, повторного использования и расширения. Систематизация приемов программирования и принципов организации классов получила название шаблона.

Основные принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах GRASP (General Responsibility Assignment Software Patterns).

При этом были сформулированы основные принципы и общие стандартные решения, придерживаясь которых можно создавать хорошо структурированный и понятный код.

Шаблон Expert

При проектировании классов на первом этапе необходимо определить общий принцип распределения обязанностей между классами проекта, а именно: в первую очередь определить кандидатов в информационные эксперты – классы, обладающие информацией, требуемой для выполнения своей функциональности.

Простые эксперты определять достаточно просто, но часто возникает необходимость дополнять класс атрибутами, и в этом случае необходимо сделать правильный выбор. Например, в подсистеме прохождения назначенного теста некоторому классу необходимо знать число вопросов, на которые получен ответ на текущий момент времени в процессе тестирования. Какой класс должен отвечать за знание количества вопросов, на которые дан ответ на текущий момент времени при прохождении теста, если определены следующие классы?

```
/*пример # 1 : шаблон Expert : LineRequestQuest.java :Test.java : Quest.java */
      public class Test {//информационный эксперт
            private int idTest;
            private int numberQuest;
            private String testName;
            // реализация конструкторов и методов
      }
      public class LineRequestQuest {
            private int questID;
            // реализация конструкторов и методов
      }
      public class Quest {//информационный эксперт
            private int idQuest;
            private int testID;
            // реализация конструкторов и методов
```

```
}
```

Необходимо узнать количество вопросов из теста, на которые дан ответ, то есть число созданных объектов класса **`LineRequestQuest`**. Такой информацией обладает лишь экземпляр объекта **`Test`**, так как этот класс ответствен за знание общего количества вопросов в тесте. Следовательно, с точки зрения шаблона Expert объект **`Test`** подходит для выполнения этой обязанности, т.е. является информационным экспертом.

```
/*пример # 2 : шаблон Expert : Test.java */
        class Test {
                private int idTest;
                private int numberQuest;
                private String testName;
                private int currentNumberQuest;

                public int getCurrentNumberQuest() {
                        // реализация
                }
                // реализация конструкторов и методов
}
```

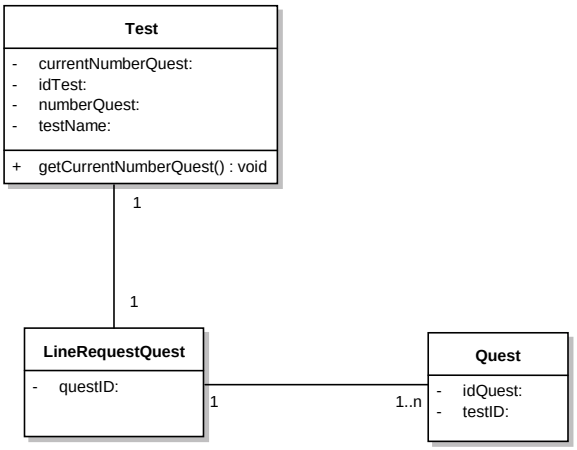


Рис. 5.1. Применение шаблона Expert

Преимущества следования шаблону Expert:

- сохранение инкапсуляции информации при назначении ответственности классам, которые уже обладают необходимой информацией для обеспечения своей функциональности;
- уклонение от новых зависимостей способствует обеспечению низкой степени связанности между классами (Low Coupling);
- добавление соответствующего метода способствует высокому зацеплению (Highly Cohesive) классов, если класс уже обладает информацией для обеспечения необходимой функциональности.

Однако назначение чрезмерно большого числа ответственностей классу при использовании шаблона Expert может привезти к получению слишком сложных классов, которые перестанут удовлетворять шаблонам Low Coupling и High Cohesion.

Шаблон Creator

Существует большая вероятность того, что класс проще, если он будет большую часть своего жизненного цикла ссылаться на создаваемые объекты.

После определения информационных экспертов следует определить классы, ответственные за создание нового экземпляра некоторого класса. Следует назначить классу **B** обязанность создавать экземпляры класса **A**, если выполняется одно из следующих условий:

- класс **B** агрегирует (aggregate) объекты **A**;
- класс **B** содержит (contains) объекты **A**;
- класс **B** записывает или активно использует (records or closely uses) экземпляры объектов **A**;
- классы **B** и **A** относятся к одному и тому же типу, и их экземпляры составляют, агрегируют, содержат или напрямую используют другие экземпляры того же класса;
- класс **B** содержит или получает данные инициализации (has the initializing data), которые будут передаваться объектам **A** при его создании.

Если выполняется одно из указанных условий, то класс **B** – создатель (creator) объектов **A.**

Инициализация объектов – стандартный процесс. Грамотное распределение обязанностей при проектировании позволяет создать слабо связанные независимые простые классы и компоненты.

В соответствии с шаблоном необходимо найти класс, который должен отвечать за создание нового экземпляра объекта **Quest** (агрегирующий экземпляры объектов **Quest**).

Поскольку объект **LineRequestQuest** использует объект **Quest**, согласно шаблону Creator он является кандидатом для выполнения обязанности, связанной с созданием экземпляров объектов **Quest**. В этом случае обязанности могут быть распределены следующим образом:

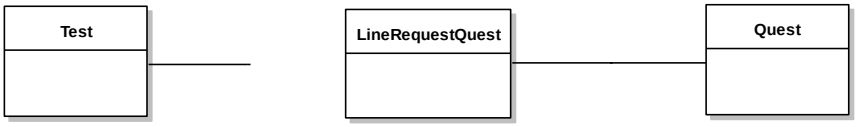


Рис. 5.2. Пример реализации шаблона Creator

```
/* пример # 3 : шаблон Creator: Qest.java: LineRequestQuest.java :Test.java */
          public class Test {
             private int idTest;
             private int numberQuest;
             private String testName;
             private int currentNumberQuest;
             // реализация конструкторов и методов
             }
          public class LineRequestQuest {
             private int questID;

             public void answerQuest() {
             // реализация
```

```
    Vector q = new Vector();
    q.add(makeRequest(параметры));
    //
    }
    public Quest makeRequest(параметры) {
          // реализация
          return new Quest(параметры);
    }
}
public class Quest{
    private int idQuest;
    private int testID;

    public Quest() {}
    // реализация конструкторов и методов
}
```

Шаблон Creator способствует низкой зависимости между классами (Low Coupling), так как экземпляры класса, которым необходимо содержать ссылку на некоторые объекты, должны создавать эти объекты. При создании некоторого объекта самостоятельно класс тем самым перестает быть зависимым от класса, отвечающего за создание объектов для него. Распределение обязанностей выполняется в процессе создания диаграммы взаимодействия классов.

Шаблон Low Coupling

Степень связанности классов определяет, насколько класс связан с другими классами и какой информацией о других классах он обладает. При проектировании отношений между классами следует распределить обязанности таким образом, чтобы степень связанности оставалась низкой.

Наличие классов с высокой степенью связанности нежелательно, так как:

- изменения в связанных классах приводят к локальным изменениям в данном классе;
- затрудняется понимание каждого класса в отдельности;
- усложняется повторное использование, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.

Пусть требуется создать экземпляр класса **Quest** и связать его с объектом **Test**. В предметной области регистрация объекта **Test** выполняется объектом **Course.**

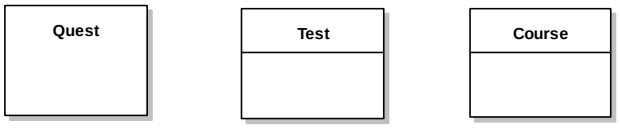


Рис. 5.3. Классы, которые необходимо связать

Далее экземпляр объекта **Course** должен передать сообщение **makeQuest()** объекту **Test**. Это значит, что в текущем тесте были получены идентификаторы всех вопросов, составляющих тест и становится возможным создание объектов типа **Quest** и наполнение собственно теста.

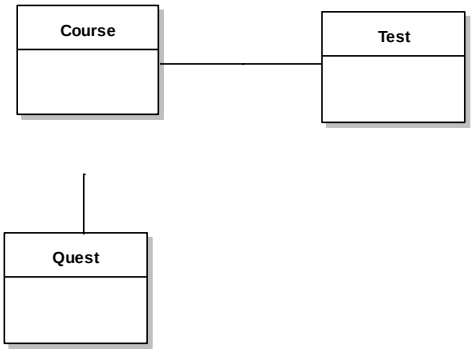


Рис. 5.4. Пример плохой реализации шаблона Low Coupling

```
/* пример # 4 : шаблон Low Coupling : Qest.java : Test.java : Course.java */
public class Course {
   private int id;
   private String name;

   public void makeTest(){
         Test test = new Test(параметры);
         // реализация
   while(условие){
         Quest quest = new Quest(параметры);
         // реализация
         test.addTest(quest);
         }
         // реализация
   }
}
public class Test {
   // поля
   public void addTest(Quest quest){
         // реализация
   }
}
public class Quest {
   // поля и методы
}
```

При таком распределении обязанностей предполагается, что класс **Course** связывается с классом **Quest**.

Второй вариант распределения обязанностей с уклонением класса **Course** от создания объектов вопросов представлен на рисунке 5.5.

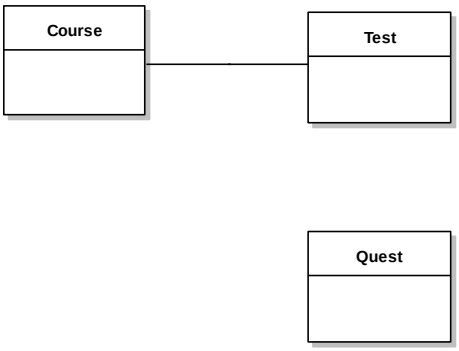


Рис. 5.5. Пример правильной реализации шаблона Low Coupling

```
/* пример # 5 : шаблон Low Coupling: Qest.java: Test.java: Course.java */
public class Course {
   private int id;
   private String name;

   public void makeTest() {
         Test test = new Test(параметры);
         // реализация
         test.addTest();
         // реализация
   }
}
public class Test {
   // поля
   public void addTest() {
         // реализация
     while(условие) {
       Quest quest = new Quest(параметры);
         // реализация
         }
   }
}
public class Quest {
   // поля и методы
}
```

Какой из методов проектирования, основанный на распределении обязанностей, обеспечивает низкую степень связанности?

В обоих случаях предполагается, что объекту **Test** должно быть известно о существовании объекта **Quest**.

При использовании первого способа, когда объект **Quest** создается объектом **Course**, между этими двумя объектами добавляется новая связь, тогда как второй способ степень связывания объектов не увеличивает. Более предпочтителен второй способ, так как он обеспечивает низкую связываемость.

В ООП имеются некоторые стандартные способы связывания объектов **A** и **B**:

- объект **A** содержит атрибут, который ссылается на экземпляр объекта **B**;

- объект **A** содержит метод, который ссылается на экземпляр объекта **B**, что подразумевает использование **B** в качестве типа параметра, локальной переменной или возвращаемого значения;
- класс объекта **A** является подклассом объекта **B**;
- **B** является интерфейсом, а класс объекта **A** реализует этот интерфейс.

Шаблон Low Coupling нельзя рассматривать изолированно от других шаблонов (Expert, Creator, High Cohesion). Не существует *абсолютной меры* для определения слишком высокой степени связывания.

Преимущества следования шаблону Low Coupling:

- изменение компонентов класса мало сказывается на других классах;
- принципы работы и функции компонентов можно понять, не изучая другие классы.

Шаблон High Cohesion

С помощью этого шаблона можно обеспечить возможность управления сложностью, распределив обязанности, поддерживая высокую степень зацепления.

Зацепление – мера сфокусированности класса. При высоком зацеплении обязанности класса тесно связаны между собой, и класс не выполняет работ непомерных объёмов. Класс с низкой степенью зацепления выполняет много разнородных действий или не связанных между собой обязанностей.

Возникают проблемы, связанные с тем, что класс:

- труден в понимании, так как необходимо уделять внимание несвязным (неродственным) идеям;
- сложен в поддержке и повторном использовании из-за того, что он должен быть использован вместе с зависимыми классами;
- ненадежен, постоянно подвержен изменениям.

Классы со слабым зацеплением выполняют обязанности, которые можно легко распределить между другими классами.

Пусть необходимо создать экземпляр класса **Quest** и связать его с заданным тестом. Какой класс должен выполнять эту обязанность? В предметной области сведения о вопросах на текущий момент времени при прохождении теста записываются в объекте **Course**, согласно шаблону для создания экземпляра объекта **Quest** можно использовать объект **Course**. Тогда экземпляр объекта **Course** сможет отправить сообщение **makeTest()** объекту **Test**. За прохождение теста отвечает объект **Course**, т.е. объект **Course** частично несет ответственность за выполнение операции **makeTest()**. Однако если и далее возлагать на класс **Course** обязанности по выполнению все новых функций, связанных с другими системными операциями, то этот класс будет слишком перегружен и будет обладать низкой степенью зацепления.

Этот шаблон необходимо применять при оценке эффективности каждого проектного решения.

Виды зацепления:

1. *Очень слабое зацепление*. Единоличное выполнение множества разнородных операций.

```
/*пример # 6 : очень слабое зацепление : Initializer.java */
          public class Initializer {
```

```
        public void createTCPServer(String port) {
                // реализация
        }
        public int connectDataBase(URL url) {
                // реализация
        }
        public void createXMLDocument(String name) {
                // реализация
        }
    }
```

2. *Слабое зацепление*. Единоличное выполнение сложной задачи из одной функциональной области.

```
/*пример # 7 : слабое зацепление : NetLogicCreator.java */
    public class NetLogicCreator {
        public void createTCPServer() {
                // реализация
        }
        public void createTCPClient() {
                // реализация
        }
        public void createUDPServer() {
                // реализация
        }
        public void createUDPClient() {
                // реализация
        }
    }
```

3. *Среднее зацепление*. Несложные обязанности в нескольких различных областях, логически связанных с концепцией этого класса, но не связанных между собой.

```
/*пример # 8 : среднее зацепление : TCPServer.java */
  public class TCPServer {
        public void createTCPServer() {
                // реализация
        }
        public void receiveData() {
                // реализация
        }
        public void sendData() {
                // реализация
        }
        public void compression() {
                // реализация
        }
        public void decompression() {
                // реализация
        }
    }
```

4. *Высокое зацепление.* Среднее количество обязанностей из одной функциональной области при взаимодействии с другими классами.

*/*пример # 9 : высокое зацепление : TCPServerCreator.java : DataTransmission.java : CodingData.java */*

```
public class TCPServerCreator {
   public void createTCPServer() {
         // реализация
   }
}
public class DataTransmission {
   public void receiveData() {
         // реализация
   }
   public void sendData() {
         // реализация
   }
}
public class CodingData {
   public void compression() {
         // реализация
   }
   public void decompression() {
         // реализация
   }
}
```

Если обнаруживается, что используется слишком негибкий дизайн, который сложен в поддержке, следует обратить внимание на классы, которые не обладают свойством зацепления или зависят от других классов. Эти классы легки в узнавании, поскольку они сильно взаимосвязаны с другими классами или содержат множество неродственных методов. Как правило, классы, которые не обладают сильной зависимостью с другими классами, обладают свойством зацепления и наоборот. При наличии таких классов необходимо реорганизовать их структуру таким образом, чтобы они по возможности не являлись зависимыми и обладали свойством зацепления.

Шаблон Controller

Одной из базовых задач при проектировании информационных систем является определение класса, отвечающего за обработку системных событий. При необходимости посылки внешнего события прямо объекту приложения, которое обрабатывает это событие, как минимум один из объектов должен содержать ссылку на другой объект, что может послужить причиной очень негибкого дизайна, если обработчик событий зависит от типа источника событий или источник событий зависит от типа обработчика событий.

В простейшем случае зависимость между внешним источником событий и внутренним обработчиком событий заключается исключительно в передаче событий. Довольно просто обеспечить необходимую степень независимости между источником событий и обработчиком событий, используя интерфейсы. Интер-

фейсов может оказаться недостаточно для обеспечения поведенческой независимости между источником и обработчиком событий, когда отношения между этими источником и обработчиком достаточно сложны.

Можно избежать зависимости между внешним источником событий и внутренним обработчиком событий путем введения между ними дополнительного объекта, который будет работать в качестве посредника при передаче событий. Этот объект должен быть способен справляться с любыми другими сложными аспектами взаимоотношений между объектами.

Согласно шаблону Controller, производится делегирование обязанностей по обработке системных сообщений классу, если он:

- представляет всю организацию или всю систему в целом (внешний контроллер);
- представляет активный объект из реального мира, который может участвовать в решении задачи (контроллер роли);
- представляет искусственный обработчик всех системных событий прецедента и называется `Прецедент`**`Handler`** (контроллер прецедента) .

Для всех системных событий в рамках одного прецедента используется один и тот же контролер.

Controller – это класс, не относящийся к интерфейсу пользователя и отвечающий за обработку системных событий. Использование объекта-контроллера обеспечивает независимость между внешними источниками событий и внутренними обработчиками событий, их типом и поведением. Выбор определяется зацеплением и связыванием.

Раздутый контроллер (волшебный сервлет) выполняет слишком много обязанностей. Признаки:

- в системе имеется единственный класс контроллера, получающий все системные сообщения, которых поступает слишком много (внешний или ролевой контроллер);
- контроллер имеет много полей (информации) и методов (ассоциаций), которые необходимо распределить между другими классами.

Шаблоны проектирования GoF

Шаблоны проектирования GoF – это многократно используемые решения широко распространенных проблем, возникающих при разработке программного обеспечения. Многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения воплощены в краткой и легко применимой на практике форме.

«Любой шаблон описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново». (Кристофер Александер).

В общем случае шаблон состоит из четырех основных элементов:

1. *Имя.* Точное имя предоставляет возможно сразу понять проблему и определить решение. Уровень абстракции при проектировании повышается.
2. *Задача.* Область применения в рамках решения конкретной проблемы.

3. *Решение.* Абстрактное описание элементов дизайна задачи проектирования и способа ее решения с помощью обобщенного набора классов.
4. *Результаты.*

Шаблоны классифицируются по разным критериям, наиболее распространенным из которых является назначение шаблона. Вследствие этого выделяются порождающие шаблоны, структурные шаблоны и шаблоны поведения.

Порождающие шаблоны

Порождающие шаблоны предназначаются для организации процесса создания объектов.

К порождающим шаблонам относятся:

Abstract Factory (**Абстрактная Фабрика**) – предоставляет интерфейс для создания связанных между собой объектов семейств классов без указания их конкретных реализаций;

Factory (**Фабрика**) – создает объект из иерархического семейства классов на основе передаваемых данных (частный случай Abstract Factory);

Builder (**Строитель**) – создает объект класса различными способами;

Singleton (**Одиночка**) – гарантирует существование только одного экземпляра класса;

Prototype (**Прототип**) – применяется при создании сложных объектов. На основе прототипа объекты сохраняются и воссоздаются, н-р путем копирования;

Factory Method (**Фабричный Метод**) – определяет интерфейс для создания объектов, при этом объект класса создается с помощью методов подклассов.

Шаблон Factory

Необходимо определить механизм создания объектов по заданному признаку для классов, находящихся в иерархической структуре. Основной класс шаблона представляет собой класс, который имеет методы для создания одного объекта из нескольких возможных на основе передаваемых данных.

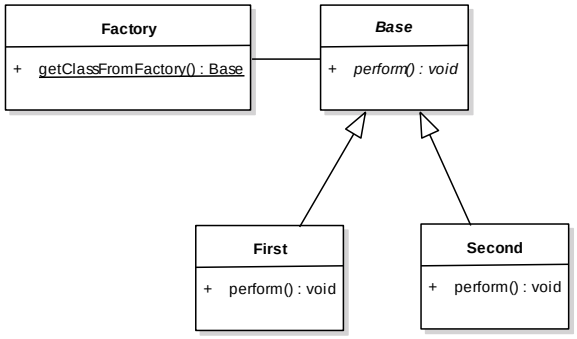


Рис. 5.6. Пример реализации шаблона Factory

Решением проблемы может быть создание класса **`ClassFactory`** с одним методом **`getClassFromFactory(String id)`**, возвращаемым значением которого будет ссылка на класс-вершину **`Base`** иерархии создаваемых классов. В качестве параметра метода передается некоторое значение, в соответствии с

которым будет осуществляться инициализация объекта одного из подклассов класса **Base**.

Программная реализация может быть представлена в общем виде следующим образом.

*/*пример # 10 : создание объектов с помощью шаблона Factory : Base.java : First.java : Second.java : ClassFactory.java : Main.java */*

```
package chapt05.factory;
public abstract class Base {
      public abstract void perform();
}
package chapt05.factory;
public class First extends Base {
      public void perform() {
            System.out.println("First");
    }
}
package chapt05.factory;
public class Second extends Base {
    public void perform() {
            System.out.println("Second");
    }
}
package chapt05.factory;
public class ClassFactory {
  private enum  Signs {FIRST, SECOND}
    public static Base getClassFromFactory(String id) {
      Signs sign = Signs.valueOf(id.toUpperCase());
      switch(sign){
       case FIRST  : return new First();
       case SECOND : return new Second();
       default : throw new EnumConstantNotPresentException(
             Signs.class, sign.name());
      }
   }
}
package chapt05.factory;
public class Main {
    public static void main(String args[]) {
        Base ob1 =
            ClassFactory.getClassFromFactory("first");
        Base ob2 =
            ClassFactory.getClassFromFactory("second");
        ob1.perform();
        ob2.perform();
    }
}
```

Один из примеров применения данного шаблона уже был рассмотрен в примере # 5 предыдущей главы.

Шаблон AbstractFactory

Необходимо создавать объекты классов, не имеющих иерархической связи, но логически связанных между собой. Абстрактный класс-фабрика определяет общий интерфейс таких фабрик. Его подклассы обладают конкретной реализацией методов по созданию разных объектов.

Предложенное решение изолирует конкретные классы. Так как абстрактная фабрика реализует процесс создания классов-фабрик и саму процедуру инициализации объектов, то она изолирует приложение от деталей реализации классов.

Одна из возможных реализаций шаблона предложена в следующем примере. Классы фабрики создаются по тому же принципу, по которому в предыдущем шаблоне создавались объекты.

```
/*пример # 11 : создание классов-фабрик по заданному признаку :
AbstractFactory.java */
package chapt05.abstractfactory;
public class AbstractFactory {
      enum Color { BLACK, WHITE };
      public static BaseFactory getFactory(String data) {
            Color my = Color.valueOf(data.toUpperCase());
            switch (my){
                  case BLACK : return new BlackFactory();
                  case WHITE : return new WhiteFactory();
                  default : throw new
EnumConstantNotPresentException(Signs.class, sign.name());
            }
      }
}
```

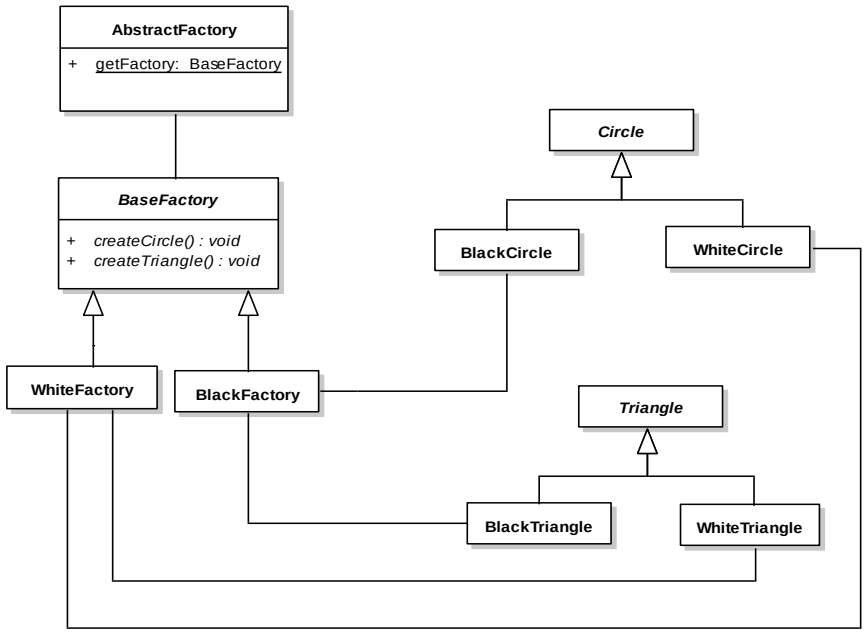


Рис. 5.7. Пример реализации шаблона AbstractFactory

Производители объектов реализуют методы по созданию не связанных иерархическими зависимостями объектов. Класс **BaseFactory** – абстрактная фабрика, а классы **BlackFactory** и **WhiteFactory** конкретные производители объектов, наследуемые от нее. Конкретные фабрики могут создавать черные или белые объекты-продукты.

*/*пример # 12 : классы-фабрики по созданию несвязанных объектов : BaseFactory.java : BlackFactory.java : WhiteFactory.java */*

```
package chapt05.abstractfactory;
public abstract class BaseFactory {
      public abstract Circle createCircle(double radius);
      public abstract Triangle createTriangle(double a,
                                                double b);
}
package chapt05.abstractfactory;
public class BlackFactory extends BaseFactory {
      public Circle createCircle(double radius) {
            return new BlackCircle(radius);
      }
      public Triangle createTriangle(double a, double b){
            return new BlackTriangle(a,b);
      }
}
package chapt05.abstractfactory;
public class WhiteFactory extends BaseFactory {
      public Circle createCircle(double radius) {
            return new WhiteCircle(radius);
      }
      public Triangle createTriangle(double a, double b){
            return new WhiteTriangle(a, b);
      }
}
```

Рассматриваются два вида классов-продуктов: **Circle**, **Triangle**. Каждый из них может быть представлен в одном из двух цветов: белом или черном.

*/*пример # 13 : классы-продукты : Circle.java : Triangle.java */*

```
package chapt05.abstractfactory;
public abstract class Circle {
      protected double radius;
      protected String color;
      public abstract void square();
}
package chapt05.abstractfactory;
public class BlackCircle extends Circle {
      public BlackCircle(double radius){
            this.radius = radius;
            color = "Black";
      }
```

```
	public void square(){
		double s = Math.PI * Math.pow(radius, 2);
		System.out.println(color + " Circle"
				+ "  Square =  " + s);
	}
}
package chapt05.abstractfactory;
public class WhiteCircle extends Circle{
	public WhiteCircle(double radius){
		this.radius = radius;
		color = "White";
	}
	public void square(){
		double s = Math.PI * Math.pow(radius, 2);
		System.out.println(color + " Circle  "
				+ "Square =  " + s);
	}
}
package chapt05.abstractfactory;
public abstract class Triangle {
	protected double a, b;
	protected String color;
	public abstract void square();
}
package chapt05.abstractfactory;
public class BlackTriangle extends Triangle {
	public BlackTriangle (double a, double b){
		this.a = a;
		this.b = b;
		color = "Black";
	}
	public void square(){
		double s = a * b / 2;
		System.out.println(color + " Triangle"
				+ "  Square =  " + s);
	}
}
package chapt05.abstractfactory;
public class WhiteTriangle extends Triangle {
	public WhiteTriangle (double a,double b) {
		this.a = a;
		this.b = b;
		color = "White";
	}
	public void square(){
		double s = 0.5 * a * b;
		System.out.println(color + " Triangle"
				+ "  Square =  " + s);
	}
}
```

Ниже будут созданы объекты всех классов и всех цветов.

```
/*пример # 14 : демонстрация работы шаблона AbstractFactory : Main.java */
package chapt05.abstractfactory;
public class Main {
      public static void main(String[] args) {
            BaseFactory factory1 =
                  AbstractFactory.getFactory("black");
            BaseFactory factory2 =
                  AbstractFactory.getFactory("white");
            Circle ob1 = factory1.createCircle(1.232);
            Circle ob2 = factory2.createCircle(1);
            Triangle ob3 = factory1.createTriangle(12,5);
            Triangle ob4 = factory2.createTriangle(1,7);

            ob1.square();
            ob2.square();
            ob3.square();
            ob4.square();
            }
}
```

Шаблон Builder

Необходимо задать конструирование сложного объекта, определяя для него только тип и содержимое. Детали построения объекта остаются скрытыми.

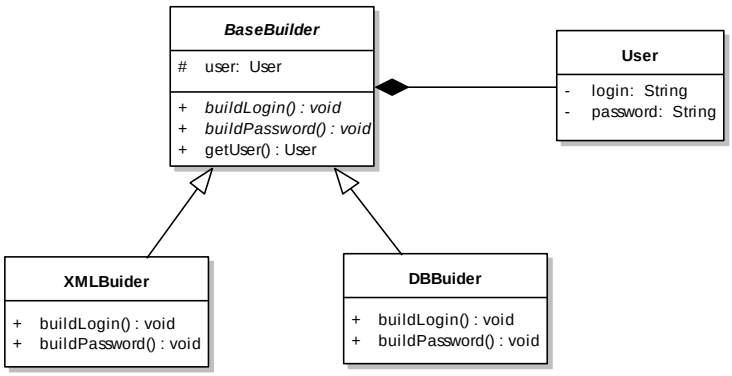


Рис. 5.8. Пример реализации шаблона Builder

Класс **BaseBuilder** определяет абстрактный интерфейс для создания частей объекта сложного класса **User**. Классы **XMLBuilder** и **DBBuilder** конструируют и собирают вместе части объекта класса **User**, а также представляет внешний интерфейс для доступа к нему. В результате объекты-строители могут работать с разными источниками, определяющими содержимое, не требуя при этом никаких изменений. При использовании этого шаблона появляется возможность контролировать пошагово весь процесс создания объекта-продукта.

Простая реализация шаблона Builder приведена ниже.

*/*пример # 15 : «сложный» для построения объект : User.java */*

```
package chapt05.builder;
public class User {
    private String login = "Guest";
    private String password = "Kc";

    public String getLogin() {
        return login;
    }
    public void setLogin(String login) {
        this.login = login;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Класс **BaseBuilder** – абстрактный класс-строитель, объявляющий в качестве поля ссылку на создаваемый объект и абстрактные методы его построения. Классы **XMLBuilder** и **DBBuilder** – наследуемые от него классы, реализующие специальные способы создания объекта. Таким образом, используя один класс **User** можно создать или администратора или модератора.

*/*пример # 16 : разные способы построения объекта : BaseBuilder.java : XML-Builder.java: DBBuilder.java */*

```
package chapt05.builder;
public abstract class BaseBuilder {
    protected User user = new User();

    public User getUser() {
        return user;
    }
    public abstract void buildLogin();
    public abstract void buildPassword();
}
package chapt05.builder;
public class XMLBuilder extends BaseBuilder {

    public void buildLogin() {
      //реализация
        user.setLogin("Admin");
    }
    public void buildPassword() {
      //реализация
        user.setPassword("Qu");
    }
}
```

```
package chapt05.builder;
public class DBBuilder extends BaseBuilder {

    public void buildLogin() {
      //реализация
        user.setLogin("Moderator");
    }
    public void buildPassword() {
      //реализация
        user.setPassword("Ku");
    }
}
```

Процесс создания объектов с использованием одного принципа реализован ниже.

```
/*пример # 17 : тестирование процесса создания объекта : Main.java */
package chapt05.builder;
public class Main {
    private static User buildUser(BaseBuilder builder) {
        builder.buildLogin();
        builder.buildPassword();
        return builder.getUser();
    }
    public static void main(String args[]) {
        User e1 = buildUser(new XMLBuilder());
        User e2 = buildUser(new DBBuilder());

        System.out.println(e1.getLogin());
        System.out.println(e1.getPassword());
        System.out.println(e2.getLogin());
        System.out.println(e2.getPassword());
    }
}
```

Шаблон Singleton

Необходимо создать объект класса таким образом, чтобы гарантировать невозможность инициализации другого объекта того же класса. Обычно сам класс контролирует наличие единственного экземпляра и он же предоставляет при необходимости к нему доступ.

```
/*пример # 18 : реализация шаблона «Одиночка» : Singleton.java */
package chapt05.singleton;
public class Singleton {

    private static Singleton instance = null;
    private SingletonTrust() {
    }
    public static Singleton getInstance() {
        if (instance == null) {
            System.out.println("Creating Singleton");
            instance = new Singleton();
        }
```

```
        return instance;
    }
}
```

Класс объявляет метод **`getInstance()`**, который позволяет клиентам получать контролируемый доступ к единственному экземпляру. Этот шаблон позволяет уточнять методы через подклассы, а также разрешить появление более чем одного экземпляра.

Структурные шаблоны

Структурные шаблоны отвечают за композицию объектов и классов.

К структурным шаблонам относятся:

Proxy (**Заместитель**) – подменяет сложный объект более простым и осуществляет контроль доступа к нему;

Composite (**Компоновщик**) – группирует объекты в иерархические структуры и позволяет работать с единичным объектом так же, как с группой объектов;

Adapter (**Адаптер**) – применяется при необходимости использовать вместе несвязанные классы. Поведение адаптируемого класса при этом изменяется на необходимое;

Bridge (**Мост**) – разделяет представление класса и его реализацию, позволяя независимо изменять то и другое;

Decorator (**Декоратор**) – представляет способ изменения поведения объекта без создания подклассов. Позволяет использовать поведение одного объекта в другом;

Facade (**Фасад**) – создает класс с общим интерфейсом высокого уровня к некоторому числу интерфейсов в подсистеме.

Шаблон Bridge

Необходимо отделить абстракцию (Abstraction) от ее реализации (Implementor) так, чтобы и то и другое можно было изменять независимо. Шаблон Bridge используется в тех случаях, когда существует иерархия абстракций и соответствующая иерархия реализаций. Причем не обязательно точное соответствие между абстракциями и реализациями. Обычно абстракция определяет операции более высокого уровня, чем реализация.

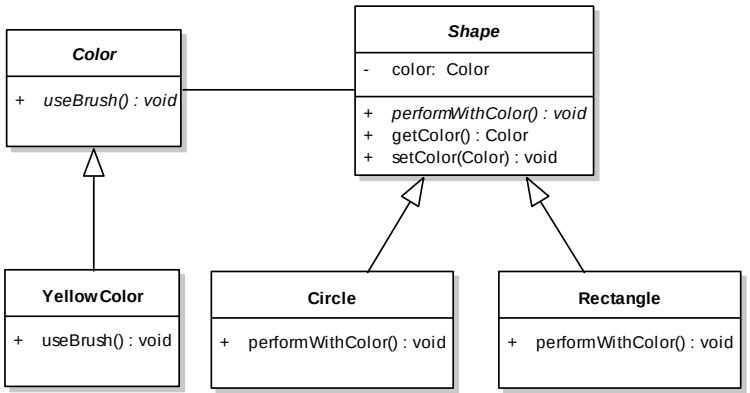


Рис. 5.9. Пример реализации шаблона Bridge

*/*пример # 18 : Implementor и его подкласс : Color.java: YellowColor.java */*

```
package chapt05.implementor;
public abstract class Color { //implementor
    public abstract void useBrush();
}
package chapt05.implementor;
public class YellowColor extends Color {
    public void useBrush() {
        System.out.println("BrushColor - Yellow");
    }
}
```

Класс **Color** – абстрактный, реализующий Implementor. Класс **YellowColor** – уточняющий подкласс класса **Color**.

*/*пример # 19 : абстракция и ее уточнения : Shape.java : Circle.java : Rectangle.java */*

```
package chapt05.abstraction;
import chapt05.implementor.*;
public abstract class Shape { //abstraction
    protected Color color;

    public Shape (){
      color = null;
    }
    public Color getColor() {
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
    }
    public abstract void performWithColor();
}
package chapt05.abstraction;
import chapt05.implementor.*;
public class Circle extends Shape {
    public Circle(Color color) {
        setColor(color);
    }
    public void performWithColor() {
       System.out.println("Performing in Circle class");
       color.useBrush();
    }
}
package chapt05.abstraction;
import chapt05.implementor.*;
public class Rectangle extends Shape {
    public Rectangle(Color color) {
        setColor(color);
    }
```

```
    public void performWithColor() {
       System.out.println("Performing in Rectangle class");
       color.useBrush();
    }
}
```

Класс **Shape** – абстракция, классы **Circle** и **Rectangle** – уточненные абстракции.

```
/*пример # 20 : использование шаблона Bridge : Main.java */
package chapt05.bridge;
import chapt05.abstraction.*;
import chapt05.implementor.*;
public class Main {
    public static void main(String args[]) {
        YellowColor color = new YellowColor();
        new Rectangle(color).performWithColor();
        new Circle(color).performWithColor();
    }
}
```

Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно динамически изменять и конфигурировать во время выполнения. Иерархии классов Abstraction и Implementor независимы и поэтому могут иметь любое число подклассов.

Шаблон Decorator

Необходимо расширить функциональные возможности объекта, используя прозрачный для клиента способ. Расширяемый класс реализует тот же самый интерфейс, что и исходный класс, делегируя исходному классу выполнение базовых операций. Шаблон Decorator даёт возможность динамического изменения поведения объектов в процессе выполнения приложения.

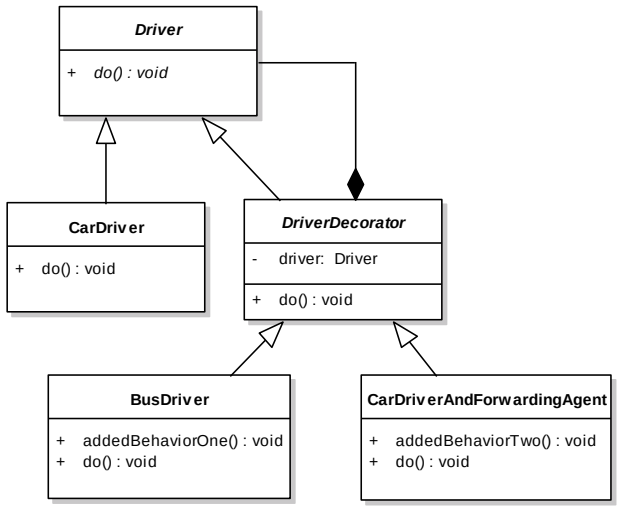


Рис. 5.10. Пример реализации шаблона Decorator

```
/*пример # 21 : определение интерфейса для компонентов : Driver.java */
package chapt05.decorator;
public abstract class Driver {
      public abstract void do();
   }
```

Класс **DriverDecorator** определяет для набора декораторов интерфейс, соответствующий интерфейсу класса **Driver**, и создает необходимые ссылки.

```
/*пример # 22 : интерфейс-декоратор для класса Driver : DriverDecorator.java */
package chapt05.decorator;
public abstract class DriverDecorator extends Driver {
      protected Driver driver;

      public DriverDecorator(Driver driver) {
            this.driver = driver;
      }
      public void do() {
            driver.do();
      }
   }
```

Класс **CarDriver** определяет класс, функциональность которого будет расширена.

```
/*пример # 23 : класс просто водителя : CarDriver.java */
package chapt05.decorator;
public class CarDriver extends Driver {
      public void do() { //базовая операция
            System.out.println("I am a driver");
      }
   }
```

Класс **BusDriver** добавляет дополнительную функциональность **addedBehaviorOne()** необходимую для водителя автобуса, используя функциональность **do()** класса **CarDriver**.

```
/*пример # 24 : класс водителя автобуса: BusDriver.java */
package chapt05.decorator;
public class BusDriver extends DriverDecorator {

      public BusDriver(Driver driver) {
            super(driver);
      }
      private void addedBehaviorOne() {
            System.out.println("I am bus driver");
      }
      public void do() {
            driver.do();
            addedBehaviorOne();
      }
   }
```

Класс **CarDriverAndForwardingAgent** добавляет дополнительную функциональность **addedBehaviorTwo()** необходимую для водителя-экспедитора, используя функциональность **do()** класса **CarDriver**.

```
/*пример # 25 : класс водителя-экспедитора:CarDriverAndForwardingAgent.java*/
package chapt05.decorator;
public class CarDriverAndForwardingAgent
                              extends DriverDecorator {

      public CarDriverAndForwardingAgent(Driver driver){
            super(driver);
      }
      private void addedBehaviorTwo() {
            System.out.println("I am a Forwarding Agent");
      }
      public void do() {
            driver.do();
            addedBehaviorTwo();
      }
   }
```

Создав экземпляр класса **CarDriver** можно делегировать ему выполнение задач, связанных с водителем автобуса или водителя-экспедитора, без написания специализированных полновесных классов.

```
/*пример # 26 : использование шаблона Decorator : Main.java */
package chapt05.decorator;
public class Main {
      public static void main(String args[]){
            Driver carDriver = new CarDriver();
            Main runner = new Main();
            runner.doDrive(carDriver);

            runner.doDrive(new BusDriver(carDriver));
            runner.doDrive(
            new CarDriverAndForwardingAgent(carDriver));
            }
      public void doDrive(Driver driver){
            driver.do();
      }
   }
```

Шаблоны поведения

Шаблоны поведения характеризуют способы взаимодействия классов или объектов между собой.

К шаблонам поведения относятся:

Chain of Responsibility (Цепочка Обязанностей) – организует независимую от объекта-отправителя цепочку не знающих возможностей друг друга объектов-получателей, которые передают запрос друг другу;

Command (**Команда**) – используется для определения по некоторому признаку конкретного класса, которому будет передан запрос для обработки;

Iterator (**Итератор**) – позволяет последовательно обойти все элементы коллекции или другого составного объекта, не зная деталей внутреннего представления данных;

Mediator (**Посредник**) – позволяет снизить число связей между классами при большом их количестве, выделяя один класс, знающий все о методах других классов;

Memento (**Хранитель**) – сохраняет текущее состояние объекта для дальнейшего восстановления;

Observer (**Наблюдатель**) – позволяет при зависимости между объектами типа «один ко многим» отслеживать изменения объекта;

State (**Состояние**) – позволяет объекту изменять свое поведение за счет изменения внутреннего объекта состояния;

Strategy (**Стратегия**) – задает набор алгоритмов с возможностью выбора одного из классов для выполнения конкретной задачи во время создания объекта;

Template Method (**Шаблонный Метод**) – создает родительский класс, использующий несколько методов, реализация которых возложена на производные классы;

Visitor (**Посетитель**) – представляет операцию в одном или нескольких связанных классах некоторой структуры, которую вызывает специфичный для каждого такого класса метод в другом классе.

Шаблон Command

Необходимо создать объект-команду, метод которого может быть вызван, а сам объект может быть сохранен и передан в качестве параметра метода или возвращен как любой другой объект. Инкапсулирует запрос как объект.

Объект-источник запроса отделяется от команды, но от его типа зависит, какая из команд будет выполнена.

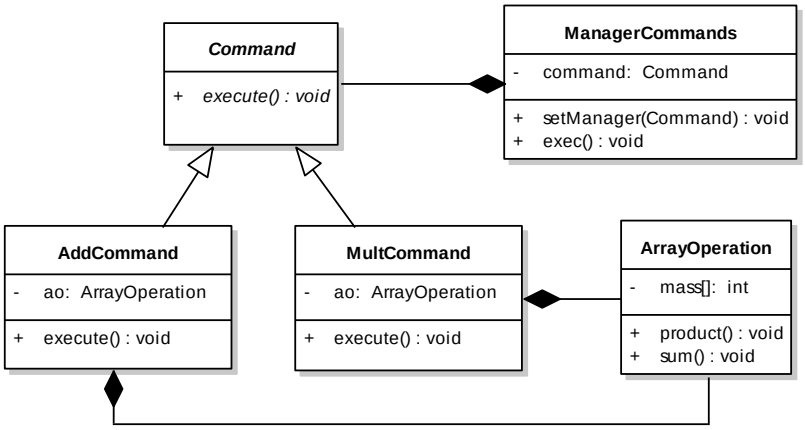


Рис. 5.11. Пример реализации шаблона Command

*/*пример # 27 : описание команды и конкретные реализации : Command.java : AddCommand.java : MultCommand.java */*

```
package chapt05.command;
public abstract class Command {
      public abstract void execute();
}
package chapt05.command;
import chapt05.receiver.*;
public class MultCommand extends Command {
      private ArrayOperation ao;

      public MultCommand (ArrayOperation ao) {
            this.ao  = ao;
      }
      public  void execute () {
            this.ao.product();
      }
}
package chapt05.command;
import chapt05.receiver.*;
public class AddCommand extends Command {
      private ArrayOperation ao;

      public AddCommand (ArrayOperation ao) {
            this.ao  = ao;
      }
      public  void execute() {
            this.ao.sum();
      }
}
```

*/*пример # 28 : класс Receiver (получатель) - располагает информацией о способах выполнения операций : ArrayOperation.java */*

```
package chapt05.receiver;
public class ArrayOperation {
      private int[] mass;

      public ArrayOperation(int[] mass) {
            this.mass = mass;
      }
      public void sum() {
            int sum = 0;
            for (int i : mass)
                  sum += i;
            System.out.println(sum);
      }
      public void product() {
            int mult = 1;
            for (int i : mass)
```

```
                mult *= i;
            System.out.println(mult);
        }
    }
```

*/*пример # 29 : класс Invoker (инициатор)-обращается к команде для выполнения запроса : ManagerCommands.java */*

```
package chapt05.invoker;
import chapt05.command.*;
public class ManagerCommands {
      private Command command;

      public ManagerCommands(Command command) {
            this.command = command;
      }
      public void setManager(Command command) {
            this.command = command;
      }
      public void exec() {
            command.execute();
      }
}
```

*/*пример # 30 : простое использование шаблона Command : Main.java */*

```
package chapt05;
import chapt05.invoker.*;
import chapt05.receiver.*;
import chapt05.command.*;
public class Main {
      public static void main(String[] args) {
            int mass[] = {-1, 71, 45, -20, 48, 60, 19};
      /*класс получатель(Receiver)-располагают информацией о способах
                                                выполнения операций*/
      ArrayOperation receiver = new ArrayOperation (mass);
      //инициализация команды
      Command operation1 = new MultCommand(receiver);
      Command operation2 = new AddCommand(receiver);
//класс инициатор (Invoker)-обращается к команде для выполнения запроса
ManagerCommands manager = new ManagerCommands(operation1);
            manager.exec();
            manager.setManager(operation2);
            manager.exec();
      }
   }
```

Объект-команда получен прямой инициализацией на основе переданного параметра. На практике данный объект создается или извлекается из коллекции на основе признака вызываемой команды. Объект **ManagerCommands** инициализируется командой и обладает простым интерфейсом для выполнения специализируемой задачи. В этом случае появляется возможность изменять реакцию приложения на запрос команды простой заменой объекта-управления.

Шаблон Strategy

Необходимо определить семейство алгоритмов, инкапсулировать каждый из них и сделать их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

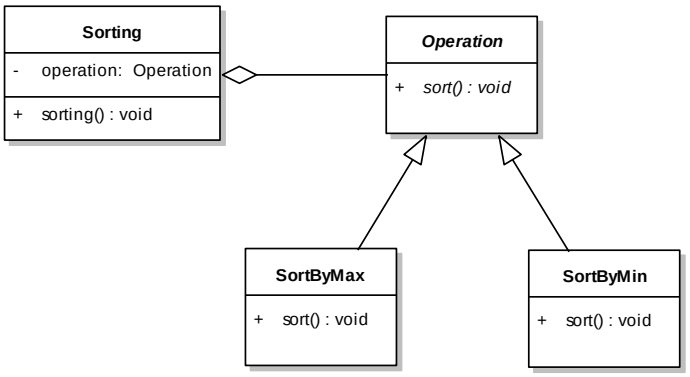


Рис. 5.12. Пример реализации шаблона Strategy

Класс **Operation** объявляет общий для всех поддерживаемых алгоритмов интерфейс, которым пользуется класс **Sorting** для вызова конкретного алгоритма, определенного в классе **SortByMax** или **SortByMin**. Класс **Sorting** конфигурируется объектом класса **SortByXxx**, объявляет ссылку на объект класса **Operation** и может определять интерфейс, позволяющий объекту **Operation** получить доступ к информации, в данном случае для сортировки массива.

Использование шаблона позволяет отказаться от условных операторов при выборе нужного поведения. Стратегии могут предлагать различные реализации одного и того же поведения. Класс-клиент вправе выбирать подходящую стратегию в зависимости от своих требований.

```
/*пример # 31 : общий интерфейс и классы конкретных стратегий :
Operation.java : SortByMax.java : SortByMin.java */
package chapt05.strategy;
public abstract class Operation {
      public abstract void sort(int mass[]);
}
package chapt05.strategy;
public class SortByMax extends Operation {
      public void sort(int mass[]) {
            for (int i = 0; i < mass.length; ++i) {
                  for (int j = i; j < mass.length; ++j) {
                        if (mass[j] > mass[i]){
                              int m = mass[i];
                              mass[i] = mass[j];
                              mass[j] = m;
                        }
                  }
            }
```

```
            System.out.print("SortByMax : ");
            for (int i : mass)
                  System.out.print(i + " ");
            System.out.println('\n');           }
}
package chapt05.strategy;
public class SortByMin extends Operation {
      public void sort(int mass[]) {
            for (int i = 0; i < mass.length; ++i) {
                  for (int j = i; j < mass.length; ++j) {
                        if (mass[j] < mass[i]){
                              int m = mass[i];
                              mass[i] = mass[j];
                              mass[j] = m;
                        }
                  }
            }
            System.out.print("SortByMin : ");
            for (int i : mass)
                  System.out.print(i + " ");
            System.out.println('\n');}
}
/*пример # 32 : класс выбора стратегии : Sorting.java */
package chapt05.strategy;
public class Sorting {
    private Operation operation = null;
    public Sorting(int operation){
        switch(operation) {
            case 1: this.operation = new SortByMax();
                        break;
            case 2: this.operation = new SortByMin();
                        break;
            default: System.out.println(
                        "Такая операция отсутствует");
                  }
     }
    public void sorting(int[] mass) {
      if (operation != null) operation.sort(mass);
      else return;
   }
}
/*пример # 33 : использование шаблона Strategy: Main.java */
package chapt05.strategy;
public class Main {
    public static void main(String args[]) {
      int mass[] = {28, 9, 71, 8, 35, 5, 51};
      Sorting cont1 = new Sorting(1);
      Sorting cont2 = new Sorting(2);
```

```
            cont1.sorting(mass);
            cont2.sorting(mass);
        }
}
```

Шаблон Observer

Требуется определить связь «один ко многим» между объектами таким образом, чтобы при изменении состояния одного объекта все связанные с ним объекты оповещались об этом и автоматически изменяли свое состояние. В языке Java этот шаблон используется под названием Listener.

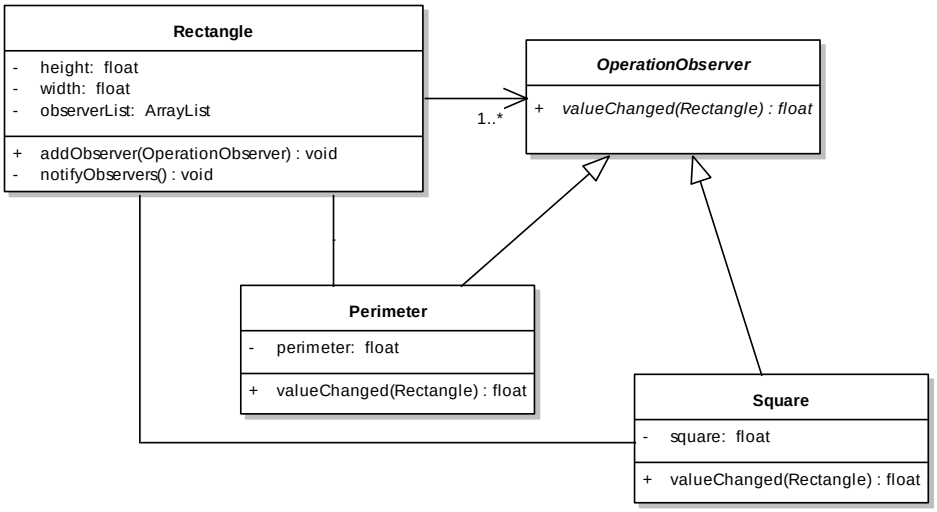


Рис. 5.13. Пример реализации шаблона Observer

Класс **`Rectangle`** (субъект) располагает информацией о своих наблюдателях и предоставляет интерфейс для регистрации и уведомления наблюдателей. Класс **`OperationObserver`** (наблюдатель) определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта. Класс **`Perimeter`** (конкретный наблюдатель) хранит или получает ссылку на объект класса **`Rectangle`**, сохраняет данные и реализует интерфейс обновления, определенный в классе **`OperationObserver`** для поддержки согласованности с субъектом.

Шаблон обеспечивает автоматическое уведомление всех подписавшихся на него объектов. Кроме этого, применение шаблона Observer абстрагирует связь субъекта и наблюдателя. Субъект имеет информацию только о том, что у него есть некоторое число наблюдателей, каждый из которых подчиняется интерфейсу абстрактного класса-наблюдателя.

*/*пример # 34 : класс-субъект, за которым следят все классы-наблюдатели : Rectangle.java */*

```
package chapt05.observer;
import java.util.*;
```

```
public class Rectangle {
      private float width;
      private float height;
      private ArrayList<OperationObserver> observerList =
                        new ArrayList<OperationObserver>();
      public Rectangle(float width, float height) {
            this.width = width;
            this.height = height;
      }
      public void addObserver(OperationObserver observer) {
            observerList.add(observer);
      }
      public float getWidth() {
            return width;
      }
      public float getHeight() {
            return height;
      }
      public void setWidth(float width) {
            this.width = width;
            notifyObservers();
      }
      public void setHeight(float height) {
            this.height = height;
            notifyObservers();
      }
      private void notifyObservers() {
            Iterator it = observerList.iterator();
            while (it.hasNext()) {
      ((OperationObserver) it.next()).valueChanged(this);
            }
      }
      public String toString() {
            String s = "";
            Iterator it = observerList.iterator();
            while (it.hasNext()) {
                  s = s +
      ((OperationObserver) it.next()).toString() + '\n';
            }
            return s;
      }
}
```

Классы **Perimeter** и **Square** наследуются от абстрактного класса **OperationObserver** и являются наблюдателями. Как только субъект **Rectangle** изменяется, состояние этих объектов также подвергается изменению в соответствии с реализованным интерфейсом.

```
/*пример # 35 : классы-наблюдатели : OperationObserver.java : Square.java :
Perimeter.java */
package chapt05.observer;
public abstract class OperationObserver {
    public abstract float valueChanged(Rectangle observed);
}
package chapt05.observer;
public class Perimeter extends OperationObserver {
      private float perimeter;
      public float valueChanged(Rectangle observed) {
            return perimeter =
      2 * (observed.getWidth() + observed.getHeight());
      }
      public String toString() {
            return "P = " + perimeter;
      }
}
package chapt05.observer;
public class Square extends OperationObserver  {
      private float square;
      public float valueChanged(Rectangle observed) {
            return square =
            observed.getWidth() * observed.getHeight();
      }
      public String toString() {
            return "S = " + square;
      }
}
/*пример # 36 : использование шаблона Observer : Main.java */
package chapt05.observer;
public class Main {
    public static void main(String args[]) {
        Rectangle observed = new Rectangle(5, 3);
        System.out.println(observed.toString());
        observed.addObserver(new Square());
        observed.addObserver(new Perimeter());
        observed.setWidth(10);
        System.out.println(observed.toString());
        observed.setHeight(8);
        System.out.println(observed.toString());
    }
}
```

Антишаблоны проектирования

Big ball of mud. «Большой Ком Грязи» – термин для системы или просто программы, которая не имеет хоть немного различимой архитектуры. Как правило, включает в себя более одного антишаблона. Этим страдают системы, разработанные людьми без подготовки в области архитектуры ПО.

Software Bloat. «Распухание ПО» – пренебрежительный термин, используемый для описания тенденций развития новейших программ в направлении использования бóльших объемов системных ресурсов (место на диске, ОЗУ), чем предшествующие версии. В более общем контексте применяется для описания программ, которые используют больше ресурсов, чем необходимо.

Yo-Yo problem. «Проблема Йо-Йо» возникает, когда необходимо разобраться в программе, иерархия наследования и вложенность вызовов методов которой очень длинна и сложна. Программисту вследствие этого необходимо лавировать между множеством различных классов и методов, чтобы контролировать поведение программы. Термин происходит от названия игрушки йо-йо.

Magic Button. Возникает, когда код обработки формы сконцентрирован в одном месте и, естественно, никак не структурирован.

Magic Number. Наличие в коде многократно повторяющихся одинаковых чисел или чисел, объяснение происхождения которых отсутствует.

Gas Factory. «Газовый Завод» – необязательный сложный дизайн или для простой задачи.

Analiys paralisys. В разработке ПО «Паралич анализа» проявляет себя через чрезвычайно длинные фазы планирования проекта, сбора необходимых для этого артефактов, программного моделирования и дизайна, которые не имеют особого смысла для достижения итоговой цели.

Interface Bloat. «Распухший Интерфейс» – термин, используемый для описания интерфейсов, которые пытаются вместить в себя все возможные операции над данными.

Smoke And Mirrors. Термин «Дым и Зеркала» используется, чтобы описать программу либо функциональность, которая еще не существует, но выставляется за таковую. Часто используется для демонстрации финального проекта и его функционала.

Improbability Factor. «Фактор Неправдоподобия» – ситуация, при которой в системе наблюдается некоторая проблема. Часто программисты знают о проблеме, но им не разрешено ее исправить отчасти из-за того, что шанс всплытия наружу у этой проблемы очень мал. Как правило (следуя закону Мерфи), она всплывает и наносит ущерб.

Creeping featurism. Используется для описания ПО, которое выставляет напоказ вновь разработанные элементы, доводя до высокой степени ущербности по сравнению с ними другие аспекты дизайна, такие как простота, компактность и отсутствие ошибок. Как правило, существует вера в то, что каждая новая маленькая черта информационной системы увеличит ее стоимость.

Accidental complexity. «Случайная сложность» – проблема в программировании, которой легко можно было избежать. Возникает вследствие неправильного понимания проблемы или неэффективного планирования.

Ambiguous viewpoint. Объектно-ориентированные модели анализа и дизайна представляются без внесения ясности в особенности модели. Изначально эти модели обозначаются с точки зрения визуализации структуры программы. Двусмысленные точки зрения не поддерживают фундаментального разделения интерфейсов и деталей представления.

Boat anchor. «Корабельный Якорь» – часть бесполезного компьютерного «железа», единственное применение для которого – отправить на утилизацию. Этот термин появился в то время, когда компьютеры были больших размеров. В

настоящее время термин «Корабельный Якорь» стал означать классы и методы,, которые по различным причинам не имеют какого-либо применения в приложении и в принципе бесполезны. Они только отвлекают внимание от действительно важного кода.

Busy spin. Техника, при которой процесс непрерывно проверяет изменение некоторого состояния, например ожидает ввода с клавиатуры или разблокировки объекта. В результате повышается загрузка процессора, ресурсы которого можно было бы перенаправить на исполнения другого процесса. Альтернативным путем является использование сигналов. Большинство ОС поддерживают погружение потока в состояние «сон» до тех пор, пока ему отправит сигнал другой поток в результате изменения своего состояния.

Caching Failure. «Кэширование Ошибки» – тип программного бага (bug), при котором приложение сохраняет (кэширует) результаты, указывающие на ошибку даже после того, как она исправлена. Программист исправляет ошибку, но флаг ошибки не меняет своего состояния, поэтому приложение все еще не работает.

Задания к главе 5

Вариант А

Выполнить описание логики системы и использовать шаблоны проектирования для определения организации классов разрабатываемой системы. Использовать объекты классов и подклассов для моделирования реальных ситуаций и взаимодействий 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 **Пассажироперевозчик** и подклассы **Самолет**, **Поезд**, **Автомобиль**. Задать правила выбора транспорта в зависимости от расстояния и наличия путей сообщения.
3. Создать суперкласс **Учащийся** и подклассы **Школьник** и **Студент**. Определить способы обучения и возможности его продолжения.
4. Создать суперкласс **Музыкальный инструмент** и классы **Ударный, Струнный**, **Духовой**. Определить правила организации и управления оркестром.
5. Создать суперкласс **Животное** и подклассы **Собака**, **Кошка**, **Тигр**, **Мустанг**, **Дельфин**. С помощью шаблонов задать способы обитания.
6. Создать базовый класс **Садовое дерево** и производные классы **Яблоня**, **Вишня**, **Груша**, **Слива**. Принять решение о пересадке каждого дерева в зависимости от возраста и плодоношения.

Тестовые задания к главе 5

Вопрос 5.1.

Какой шаблон создает объекты путем их копирования?

1) Factory;
2) Prototype;
3) Builder;
4) Singleton.

Вопрос 5.2.

Какие из шаблонов относятся к порождающим? (выберите два)

1) Factory;
2) Command;
3) Strategy;
4) Singleton.

Вопрос 5.3.

Какой шаблон позволяет обращаться к группе объектов таким же образом как и к одному?

1) Visitor;
2) Composite;
3) Prototype;
4) Observer.

Вопрос 5.4.

Какой шаблон реализует постоянную часть алгоритма, а реализацию изменяемой оставляет потомкам?

1) Strategy;
2) Decorator;
3) Template Method;
4) Visitor.

Вопрос 5.5.

Какой шаблон подменяет собой сложный объект и контролирует доступ к нему.

1) Adapter;
2) Decorator;
3) Proxy;
4) Bridge.

Глава 6
ИНТЕРФЕЙСЫ И ВНУТРЕННИЕ КЛАССЫ

Интерфейсы

Интерфейсы подобны полностью абстрактным классам, но не являются классами. Ни один из объявленных методов не может быть реализован внутри интерфейса. В языке Java существуют два вида интерфейсов: интерфейсы, определяющие контракт для классов посредством методов, и интерфейсы, реализация которых автоматически (без реализации методов) придает классу определенные свойства. К последним относятся, например, интерфейсы **`Cloneable`** и **`Serializable`**, отвечающие за клонирование и сохранение объекта в информационном потоке соответственно.

Все объявленные в интерфейсе методы автоматически трактуются как **`public`** и **`abstract`**, а все поля – как **`public`**, **`static`** и **`final`**, даже если они так не объявлены. Класс может реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **`implements`**, дополняющего определение класса. После этого класс обязан реализовать все методы, полученные им от интерфейсов, или объявить себя абстрактным классом.

На множестве интерфейсов также определена иерархия наследования, но она не имеет отношения к иерархии классов.

Определение интерфейса имеет вид:

```
[public] interface Имя [extends Имя1, Имя2,…, ИмяN] {
                                        /*реализация интерфейса*/}
```

Например:

```
/* пример # 1 : объявление интерфейсов: LineGroup.java, Shape.java */
package chapt06;

public interface LineGroup {
       // по умолчанию public abstract
       double getPerimeter();// объявление метода
}

package chapt06;

public interface Shape extends LineGroup {
      //int id; // ошибка, если нет инициализации
      //void method(){} /* ошибка, так как абстрактный метод не может
                          иметь тела!  */
       double getSquare();  // объявление метода
}
```

Для более простой идентификации интерфейсов в большом проекте в сообществе разработчиков действует негласное соглашение о добавлении к имени интерфейса символа **'I'**, в соответствии с которым вместо имени **Shape** можно записать **IShape.**

Класс, который будет реализовывать интерфейс **Shape**, должен будет определить все методы из цепочки наследования интерфейсов. В данном случае это методы **getPerimeter()** и **getSquare().**

Интерфейсы обычно объявляются как **public**, потому что описание функциональности, предоставляемое ими, может быть использовано в нескольких пакетах проекта. Интерфейсы с областью видимости в рамках пакета могут использоваться только в этом пакете и нигде более.

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.

Реализация интерфейсов классом может иметь вид:

```
[доступ] class ИмяКласса implements Имя1, Имя2,…, ИмяN {
                                                /*код класса*/}
```

Здесь **Имя1, Имя2,…, ИмяN** – 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 : реализация интерфейса: Rectangle.java */
package chapt06;

public class Rectangle implements Shape {
      private double a, b;

      public Rectangle(double a, double b) {
            this.a = a;
            this.b = b;
      }
      //реализация метода из интерфейса
      public double getSquare() {//площадь прямоугольника
            return a * b;
      }
      //реализация метода из интерфейса
      public double getPerimeter() {
            return 2 * (a + b);
      }
}
```

```
/* пример # 3 : реализация интерфейса: Circle.java */
package chapt06;

public class Circle implements Shape {
```

```
	private double r;

	public Circle(double r) {
		this.r = r;
	}
	public double getSquare() {//площадь круга
		return Math.PI * Math.pow(r, 2);
	}
	public double getPerimeter() {
		return 2 * Math.PI * r;
	}
}
```

```
/* пример # 4 : неполная реализация интерфейса: Triangle.java */
package chapt06;
/* метод getSquare() в данном абстрактном классе не реализован */
public abstract class Triangle implements Shape {
	private double a, b, c;

	public Triangle(double a, double b, double c) {
		this.a = a;
		this.b = b;
		this.c = c;
	}
	public double getPerimeter() {
		return a + b + c;
	}
}
```

```
/* пример # 5 : свойства ссылки на интерфейс : Runner.java */
package chapt06;

public class Runner {
	public static void printFeatures(Shape f) {
	   System.out.printf("площадь:%.2f периметр: %.2f%n",
	             f.getSquare(), f.getPerimeter());
	}
	public static void main(String[] args) {
		Rectangle r = new Rectangle(5, 9.95);
		Circle c = new Circle(7.01);
		printFeatures(r);
		printFeatures(c);
	}
}
```

В результате будет выведено:

```
площадь:49,75 периметр: 29,90
площадь:154,38 периметр: 44,05
```

Класс **Runner** содержит метод **printFeatures(Shape f)**, 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 в классе **ShapeCreator** используются классы и интерфейсы, определенные выше, и объявляется ссылка на интерфейсный тип. Такая ссылка может указывать на экземпляр любого класса, который реализует объявленный интерфейс. При вызове метода через такую ссылку будет вызываться его реализованная версия, основанная на текущем экземпляре класса. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

```
/* пример # 6  : динамический связывание методов : ShapeCreator.java */
package chapt06;

public class ShapeCreator {
      public static void main(String[] args) {
            Shape sh; /*  ссылка на интерфейсный тип */
            Rectangle re = new Rectangle(5, 9.95);
            sh = re;
            sh.getPerimeter();//вызов метода класса Rectangle
            Circle cr = new Circle(7.01);
            sh = cr;  // присваивается ссылка на другой объект
            sh.getPerimeter();//вызов метода класса Circle

            // cr=re; // ошибка! разные ветви наследования
      }
}
```

Невозможно приравнивать ссылки на классы, находящиеся в разных ветвях наследования, так как не существует никакого способа привести один такой тип к другому. По этой же причине ошибку вызовет попытка объявления объекта в виде:

```
Circle c = new Rectangle(1, 5);
```

Пакеты

Любой класс Java относится к определенному пакету, который может быть неименованным (unnamed или default package), если оператор **package** отсутствует. Оператор **package** имя, помещаемый в начале исходного программного файла, определяет именованный пакет, т.е. область в пространстве имен классов, где определяются имена классов, содержащихся в этом файле. Действие оператора **package** указывает на месторасположение файла относительно корневого каталога проекта. Например:

```
package chapt06;
```

При этом программный файл будет помещен в подкаталог с названием **chapt06**. Имя пакета при обращении к классу из другого пакета присоединяется к имени класса: **chapt06.Student**. Внутри указанной области можно выделить подобласти:

```
package chapt06.bsu;
```

Общая форма файла, содержащего исходный код Java, может быть следующая:

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

В реальных проектах пакеты часто именуются следующим образом:

- обратный интернет-адрес производителя программного обеспечения, а именно для `www.bsu.by` получится `by.bsu`;
- далее следует имя проекта, например: `eun`;
- затем располагаются пакеты, определяющие собственно приложение.

При использовании классов перед именем класса через точку надо добавлять полное имя пакета, к которому относится данный класс. На рисунке приведен далеко не полный список пакетов реального приложения. Из названий пакетов можно определить, какие примерно классы в нем расположены, не заглядывая внутрь. При создании пакета всегда следует руководствоваться простым правилом: называть его именем простым, но отражающим смысл, логику поведения и функциональность объединенных в нем классов.

by.bsu.eun
by.bsu.eun.administration.constants
by.bsu.eun.administration.dbhelpers
by.bsu.eun.common.constants
by.bsu.eun.common.dbhelpers.annboard
by.bsu.eun.common.dbhelpers.courses
by.bsu.eun.common.dbhelpers.guestbook
by.bsu.eun.common.dbhelpers.learnres
by.bsu.eun.common.dbhelpers.messages
by.bsu.eun.common.dbhelpers.news
by.bsu.eun.common.dbhelpers.prepinfo
by.bsu.eun.common.dbhelpers.statistics
by.bsu.eun.common.dbhelpers.subjectmark
by.bsu.eun.common.dbhelpers.subjects
by.bsu.eun.common.dbhelpers.test
by.bsu.eun.common.dbhelpers.users
by.bsu.eun.common.menus
by.bsu.eun.common.objects
by.bsu.eun.common.servlets
by.bsu.eun.common.tools
by.bsu.eun.consultation.constants
by.bsu.eun.consultation.dbhelpers
by.bsu.eun.consultation.objects
by.bsu.eun.core.constants
by.bsu.eun.core.dbhelpers
by.bsu.eun.core.exceptions
by.bsu.eun.core.filters
by.bsu.eun.core.managers
by.bsu.eun.core.taglibs

Рис. 6.1. Организация пакетов приложения

Каждый класс добавляется в указанный пакет при компиляции. Например:

```
// пример # 7 : простейший класс в пакете: CommonObject.java
package by.bsu.eun.objects;

public class CommonObject implements Cloneable {
      public CommonObject() {
            super();
      }
      public Object clone()
            throws CloneNotSupportedException {
            return super.clone();
      }
}
```

Класс начинается с указания того, что он принадлежит пакету **by.bsu.eun.objects**. Другими словами, это означает, что файл **CommonObject.java** находится в каталоге **objects**, который, в свою очередь, находится в каталоге **bsu**, и так далее. 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: **by.bsu.eun.objects.CommonObject**. Чтобы избежать таких длинных имен, используется ключевое слово **import.** Например:

```
import by.bsu.eun.objects.CommonObject;
```

или

```
import by.bsu.eun.objects.*;
```

Во втором варианте импортируется весь пакет, что означает возможность доступа к любому классу пакета, но только не к подпакету и его классам. В практическом программировании следует использовать индивидуальный **import** класса, чтобы при анализе кода была возможность быстро определить месторасположение используемого класса.

Доступ к классу из другого пакета можно осуществить следующим образом:

```
// пример # 8 : доступ к пакету: UserStatistic.java
package by.bsu.eun.usermng;

public class UserStatistic
                  extends by.bsu.eun.objects.CommonObject {
      private long id;
      private int mark ;

      public UserStatistic() {
            super();
      }
      public long getId() {
            return id;
      }
      public void setId(long id) {
```

```
            this.id = id;
      }
      public int getMark() {
            return mark;
      }
      public void setMark(int mark) {
            this.mark = mark;
      }
}
```

При импорте класса из другого пакета рекомендуется всегда указывать полный путь с указанием имени импортируемого класса. Это позволяет в большом проекте легко найти определение класса, если возникает необходимость посмотреть исходный код класса.

// пример # 9 : доступ к пакету: CreatorStatistic.java

```
package by.bsu.eun.actions;
import by.bsu.eun.objects.CommonObject;
import by.bsu.eun.usermng.UserStatistic;

public class CreatorStatistic {
   public static UserStatistic createUserStatistic(long id)
{
            UserStatistic temp = new UserStatistic();
            temp.setId(id);
            // чтение информации из базы данных по id пользователя
            int mark = полученное значение;
            temp.setMark(mark);
            return temp;
      }
      public static void main(String[] args) {
            UserStatistic us = createUserStatistic(71);
            System.out.println(us.getMark());
      }
}
```

Если пакет не существует, то его необходимо создать до первой компиляции, если пакет не указан, класс добавляется в пакет без имени (unnamed). При этом unnamed-каталог не создается. Однако в реальных проектах классы вне пакетов не создаются, и не существует причин отступать от этого правила.

Статический импорт

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

// пример # 10 : статический импорт: ImportDemo.java

```
package chapt06;
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/3)));
    }
}
```

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

```
import static java.lang.Math.E;
import static java.lang.Math.cos;//для одного метода
```

Внутренние классы

Классы могут взаимодействовать друг с другом не только посредством наследования и использования ссылок, но и посредством организации логической структуры с определением одного класса в теле другого.

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 «События»). Одной из важнейших причин использования внутренних классов является возможность независимого наследования внутренними классами. Фактически при этом реализуется множественное наследование со своими преимуществами и проблемами.

В качестве примеров можно рассмотреть взаимосвязи классов «Корабль», «Двигатель» и «Шлюпка». Объект класса «Двигатель» расположен внутри (невидим извне) объекта «Корабль» и его деятельность приводит «Корабль» в движение. Оба этих объекта неразрывно связаны, то есть запустить «Двигатель» можно только посредством использования объекта «Корабль», например, из машинного отделения. Таким образом, перед инициализацией объекта внутреннего класса «Двигатель» должен быть создан объект внешнего класса «Корабль».

Класс «Шлюпка» также является логической частью класса «Корабль», однако ситуация с его объектами проще по причине того, что данные объекты могут быть использованы независимо от наличия объекта «Корабль». Объект класса «Шлюпка» только использует имя (на борту) своего внешнего класса. Такой внутренний класс следует определять как **static**. Если объект «Шлюпка» используется без привязки к какому-либо судну, то класс следует определять как обычный независимый класс.

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 внешнего 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 метода 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:

```
public class Ship {
      // поля и конструкторы
      // abstract, final, private, protected - допустимы
      public class Engine {  // определение внутреннего класса
            // поля и методы
            public void launch() {
               System.out.println("Запуск двигателя");
            }
      }// конец объявления внутреннего класса
      public void init() {// метод внешнего класса
        // объявление объекта внутреннего класса
            Engine eng = new Engine();
            eng.launch();
      }
}
```

При таком объявлении объекта внутреннего класса **Engine** в методе внешнего класса **Ship** нет реального отличия от использования какого-либо другого внешнего класса, кроме объявления внутри класса **Ship**. Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

```
Ship.Engine obj = new Ship().new Engine();
```

Основное отличие от внешнего класса состоит в больших возможностях ограничения видимости внутреннего класса по сравнению с обычным внешним классом. Внутренний класс может быть объявлен как **private**, что обеспечивает его полную невидимость вне класса-владельца и надежное сокрытие реализации. В этом случае ссылку **obj**, приведенную выше, объявить было бы нельзя. Создать объект такого класса можно только в методах и логических блоках внешнего класса. Использование **protected** позволяет получить доступ к внутреннему классу для класса в другом пакете, являющегося суперклассом внешнего класса.

После компиляции объектный модуль, соответствующий внутреннему классу, получит имя **Ship$Engine.class**.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к 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:

```
public class WarShip extends Ship {
     protected class SpecialEngine extends Engine {}
}
```

Если внутренний класс наследуется обычным образом другим классом (после **extends** указывается **ИмяВнешнегоКласса.ИмяВнутреннегоКласса**), то он теряет доступ к полям своего внешнего класса, в котором он был объявлен.

```
public class Motor extends Ship.Engine {
     public Motor(Ship obj) {
          obj.super();
     }
}
```

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

Внутренние классы позволяют окончательно решить проблему множественного наследования, когда требуется наследовать свойства нескольких классов.

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

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

```
/* пример # 11 : взаимодействие внешнего и внутреннего классов : Student.java :
AnySession.java */
package chapt06;

public class Student {
     private int id;
     private ExamResult[] exams;

     public Student(int id) {
          this.id = id;
     }

     private class ExamResult {// внутренний класс
          private String name;
          private int mark;
          private boolean passed;

          public ExamResult(String name) {
               this.name = name;
               passed = false;
          }
          public void passExam() {
               passed = true;
          }
          public void setMark(int mark) {
               this.mark = mark;
          }
```

```
        public int getMark() {
             return mark;
        }
        public int getPassedMark() {
             final int PASSED_MARK = 4;// «волшебное» число
             return PASSED_MARK;
        }
        public String getName() {
             return name;
        }
        public boolean isPassed() {
             return passed;
        }
    } // окончание внутреннего класса

public void setExams(String[] name, int[] marks) {
            if (name.length != marks.length)
                  throw new IllegalArgumentException();
            exams = new ExamResult[name.length];
            for (int i = 0; i < name.length; i++) {
                  exams[i] = new ExamResult(name[i]);
                  exams[i].setMark(marks[i]);
      if (exams[i].getMark() >= exams[i].getPassedMark())
                        exams[i].passExam();
            }
public String toString() {
            String res = "Студент: " + id + "\n";
      for (int i = 0; i < exams.length; i++)
            if (exams[i].isPassed())
            res += exams[i].getName() + " сдал \n";
      else
            res += exams[i].getName() + " не сдал \n";

      return res;
      }
}
package chapt06;

public class AnySession {
     public static void main(String[] args) {
          Student stud = new Student(822201);
          String ex[] = {"Механика","Программирование"};
          int marks[] = { 2, 9 };
          stud.setExams(ex, marks);
          System.out.println(stud);
     }
}
```

В результате будет выведено:

```
Студент: 822201
Механика не сдал
Программирование сдал
```

Внутренний класс определяет сущность предметной области “результат экзамена” (класс **ExamResult**), которая обычно непосредственно связана в информационной системе с объектом класса **Student**. Класс **ExamResult** в данном случае определяет только методы доступа к своим атрибутам и совершенно невидим вне класса **Student**, который 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. Видимость такого класса регулируется областью видимости 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. Класс, объявленный внутри метода, не может быть объявлен как **static**, а также не может содержать статические поля и методы.

```
/*пример # 12 : внутренний класс, объявленный внутри метода:
TeacherLogic.java*/
package chapt06;

public abstract class AbstractTeacher {
      private int id;
      public AbstractTeacher(int id) {
            this.id = id;
      }
      public abstract boolean excludeStudent(String name);
}
package chapt06;

public class Teacher extends AbstractTeacher {

      public Teacher(int id) {
            super(id);
      }
      public boolean excludeStudent(String name) {
            return false;
      }
}
package chapt06;

public class TeacherCreator {
      public TeacherCreator(){}

      public AbstractTeacher createTeacher(int id) {
            // объявление класса внутри метода
         class Dean extends AbstractTeacher {
```

```
                  Dean(int id){
                        super(id);
                  }
            public boolean excludeStudent(String name) {
                        if (name != null) {
                        // изменение статуса студента в базе данных
                              return true;
                        }
                        else return false;
                  }
            }// конец внутреннего класса

            if (isDeanId(id))
                  return new Dean(id);
            else return new Teacher(id);
      }
      private static boolean isDeanId(int id) {
            // проверка декана из БД или
            return (id  == 777);
      }
}
package chapt06;

public class TeacherLogic {
      public static void excludeProcess(int deanId,
                  String name) {
            AbstractTeacher teacher =
            new TeacherCreator().createTeacher(deanId);

            System.out.println("Студент: " + name
            + " отчислен:" + teacher.excludeStudent(name));
      }
      public static void main(String[] args) {
            excludeProcess(700, "Балаганов");
            excludeProcess(777, "Балаганов");
      }
}
```

В результате будет выведено:

```
Студент: Балаганов отчислен:false
Студент: Балаганов отчислен:true
```

Класс **Dean** объявлен в методе **createTeacher(int id)**, и соответственно объекты этого класса можно создавать только внутри этого метода, из любого другого места внешнего класса внутренний класс недоступен. Однако существует возможность получить ссылку на класс, объявленный внутри метода, и использовать его специфические свойства. При компиляции данного кода с внутренним классом ассоциируется объектный модуль со сложным именем **TeacherCreator$1Dean**, тем не менее однозначно определяющим связь между внешним и внутренним классами. Цифра **1** в имени говорит о том, что в других методах класса могут быть объявлены внутренние классы с таким же именем.

Вложенные (nested) классы

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

Вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него.

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. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класса. Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс.

```
/* пример # 13 : вложенный класс: Ship.java : RunnerShip.java */
package chapt06;

public class Ship {
	private int id;
	// abstract, final, private, protected - допустимы
	public static class LifeBoat {
		public static void down() {
			System.out.println("шлюпки на воду!");
		}
		public void swim() {
			System.out.println("отплытие шлюпки");
		}
	}
}
package chapt06;

public class RunnerShip {
	public static void main(String[] args) {
		// вызов статического метода
		Ship.LifeBoat.down();
		// создание объекта статического класса
		Ship.LifeBoat lf = new Ship.LifeBoat();
		// вызов обычного метода
		lf.swim();
	}
}
```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему. Объект **lf** вложенного класса создается с использованием имени внешнего класса без вызова его конструктора.

Класс, вложенный в интерфейс, по умолчанию статический. На него не накладывается никаких особых ограничений, и он может содержать поля и методы как статические, так и нестатические.

/ пример # 14 : класс вложенный в интерфейс: Faculty.java : University.java */*

```
package chapt06;

public interface University {
      int NUMBER_FACULTY = 20;

      class LearningDepartment {// static по умолчанию
            public int idChief;

            public static void assignPlan(int idFaculty) {
                  // реализация
            }
            public void acceptProgram() {
                  // реализация
            }
      }
}
```

Такой внутренний класс использует пространство имен интерфейса.

Анонимные (anonymous) классы

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому объекту для обработки событий, реализации блоков прослушивания и т.д. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного, единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора **new**.

Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Этот прием эффективен в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области (или одноразового) применения метода.

Конструкторы анонимных классов нельзя определять и переопределять. Анонимные классы допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными.

/ пример # 15 : анонимные классы: TypeQuest.java: RunnerAnonym.java */*

```
package chapt06;

public class TypeQuest {
      private int id = 1;

      public TypeQuest() {
      }
```

```
      public TypeQuest(int id) {
            this.id = id;
      }
      public void addNewType() {
            // реализация
            System.out.println(
            "добавлен вопрос на соответствие");
      }
}
package chapt06;

public class RunnerAnonym {
      public static void main(String[] args) {
       TypeQuest unique = new TypeQuest() {// анонимный класс #1
                  public void addNewType() {
                        // новая реализация метода
                        System.out.println(
                  "добавлен вопрос со свободным ответом");
                  }
            };// конец объявления анонимного класса
            unique.addNewType();

            new TypeQuest(71) {// анонимный класс #2
                  private String name = "Drag&Drop";

                  public void addNewType() {
                        // новая реализация метода #2
            System.out.println("добавлен " + getName());
                  }
                  String getName() {
                        return name;
                  }
            }.addNewType();

            TypeQuest standard = new TypeQuest(35);
            standard.addNewType();
      }
}
```

В результате будет выведено:

```
добавлен вопрос со свободным ответом
добавлен Drag&Drop
добавлен вопрос на соответствие
```

При запуске приложения происходит объявление объекта **unique** с применением анонимного класса, в котором переопределяется метод **addNewType()**. Вызов данного метода на объекте **unique** приводит к вызову версии метода из анонимного класса, который компилируется в объектный модуль с именем **RunnerAnonym$1**. Процесс создания второго объекта с анонимным типом применяется в программировании значительно чаще, особенно при реализации клас-

сов-адаптеров и реализации интерфейсов в блоках прослушивания. В этом же объявлении продемонстрирована возможность объявления в анонимном классе полей и методов, которые доступны объекту вне этого класса.

Для перечисления объявление анонимного внутреннего класса выглядит несколько иначе, так как инициализация всех элементов происходит при первом обращении к типу. Поэтому и анонимный класс реализуется только внутри объявления типа **enum**, как это сделано в следующем примере.

```
/* пример # 16 : анонимный класс в перечислении : EnumRunner.java */
package chapt06;

enum Shape {
      RECTANGLE, SQUARE,
      TRIANGLE {// анонимный класс
            public double getSquare() {// версия для TRIANGLE
                  return a*b/2;
            }
      };
      public double a, b;

      public void setShape(double a, double b){
            if ((a<=0 || b<=0) || a!=b && this==SQUARE)
                  throw new IllegalArgumentException();
            else
                  this.a = a;
                  this.b = b;
      }
      public double getSquare(){// версия для RECTANGLE и SQUARE
            return a * b;
      }
      public String getParameters() {
            return "a=" + a + ", b=" + b;
      }
}
public class EnumRunner {
      public static void main(String[] args) {
            int i = 4;
            for (Shape f : Shape.values()) {
                  f.setShape(3, i--);
      System.out.println(f.name()+"-> " + f.getParameters()
                        + " площадь= " + f.getSquare());
                  }
      }
}
```

В результате будет выведено:

```
RECTANGLE-> a=3.0, b=4.0 площадь= 12.0
SQUARE-> a=3.0, b=3.0 площадь= 9.0
TRIANGLE-> a=3.0, b=2.0 площадь= 3.0
```

Объектный модуль для такого анонимного класса будет скомпилирован с именем **Shape$1**.

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 **СССР** 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.
11. Создать класс **Художественная Выставка** с внутренним классом, с помощью объектов которого можно хранить информацию о картинах, авторах и времени проведения выставок.
12. Создать класс **Календарь** с внутренним классом, с помощью объектов которого можно хранить информацию о выходных и праздничных днях.
13. Создать класс **Shop** (магазин) с внутренним классом, с помощью объектов которого можно хранить информацию об отделах, товарах и услуг.
14. Создать класс **Справочная Служба Общественного Транспорта** с внутренним классом, с помощью объектов которого можно хранить информацию о времени, линиях маршрутов и стоимости проезда.
15. Создать класс **Computer** (компьютер) с внутренним классом, с помощью объектов которого можно хранить информацию об операционной системе, процессоре и оперативной памяти.

16. Создать класс **Park** (парк) с внутренним классом, с помощью объектов которого можно хранить информацию об аттракционах, времени их работы и стоимости.
17. Создать класс **Cinema** (кино) с внутренним классом, с помощью объектов которого можно хранить информацию об адресах кинотеатров, фильмах и времени сеансов.
18. Создать класс **Программа Передач** с внутренним классом, с помощью объектов которого можно хранить информацию о названии телеканалов и программ.
19. Создать класс **Фильм** с внутренним классом, с помощью объектов которого можно хранить информацию о продолжительности, жанре и режиссерах фильма.

Вариант B

В заданиях варианта B 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.

Вариант C

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` Абитуриент ← `abstract class` Студент ← `class` Студент-Заочник.**
3. **`interface` Сотрудник ← `class` Инженер ← `class` Руководитель.**
4. **`interface` Здание ← `abstract class` Общественное Здание ← `class` Театр.**
5. **`interface` Mobile ← `abstract class` Siemens Mobile ← `class` Model.**
6. **`interface` Корабль ← `abstract class` Военный Корабль ← `class` Авианосец.**
7. **`interface` Врач ← `class` Хирург ← `class` Нейрохирург.**
8. **`interface` Корабль ← `class` Грузовой Корабль ← `class` Танкер.**
9. **`interface` Мебель ← `abstract class` Шкаф ← `class` Книжный Шкаф.**
10. **`interface` Фильм ← `class` Отечественный Фильм ← `class` Комедия.**
11. **`interface` Ткань ← `abstract class` Одежда ← `class` Костюм.**
12. **`interface` Техника ← `abstract class` Плеер ← `class` Видеоплеер.**
13. **`interface` Транспортное Средство ← `abstract class` Общественный Транспорт ← `class` Трамвай.**

14. **interface Устройство Печати ← class Принтер ← class Лазерный Принтер**.
15. **interface Бумага ← abstract class Тетрадь ← class Тетрадь Для Рисования**.
16. **interface Источник Света ← class Лампа ← class Настольная Лампа**.

Тестовые задания к главе 6

Вопрос 6.1.

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 6.2.

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{
    class MyClass {}}
```
5)
```
public interface MyInterface{
    public final static int i;
    public abstract int result(int i);}
```

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 Quest5 {
  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.

Часть 2.
ИСПОЛЬЗОВАНИЕ КЛАССОВ И БИБЛИОТЕК

Во второй части книги рассмотрены вопросы использования классов Java при работе со строками и файлами, для хранения объектов, при создании пользовательских интерфейсов, многопоточное и сетевое программирование с использованием классов из пакетов ***java.util***, ***java.text***, ***java.net***, ***java.io***, ***java.awt***, ***javax.swing*** *и др.*

Из-за ограниченности объема книги детальное рассмотрение библиотек классов невозможно. Подробное описание классов и методов можно найти в документации по языку Java, которой необходимо пользоваться каждому Java-программисту.

Глава 7
ОБРАБОТКА СТРОК

Строка в языке Java – это основной носитель текстовой информации. Это не массив символов типа **char**, а объект соответствующего класса. Системная библиотека Java содержит классы **String**, **StringBuilder** и **StringBuffer**, поддерживающие работу со строками и определенные в пакете **java.lang**, подключаемом автоматически. Эти классы объявлены как **final**, что означает невозможность создания собственных порожденных классов со свойствами строки. Кроме того, для форматирования и обработки строк применяются классы **Formatter**, **Pattern**, **Matcher** и другие.

Класс String

Каждая строка, создаваемая с помощью оператора **new** или с помощью литерала (заключённая в двойные апострофы), является объектом класса **String**. Особенностью объекта класса **String** является то, что его значение не может быть изменено после создания объекта при помощи какого-либо метода класса, так как любое изменение строки приводит к созданию нового объекта. При этом ссылку на объект класса **String** можно изменить так, чтобы она указывала на другой объект и тем самым на другое значение.

Класс **String** поддерживает несколько конструкторов, например: **String()**, **String(String str)**, **String(byte asciichar[])**, **String(char[] unicodechar)**, **String(StringBuffer sbuf)**, **String(StringBuilder sbuild)** и др. Эти конструкторы используются

для создания объектов класса **String** на основе инициализации значениями из массива типа **char**, **byte** и др. Например, при вызове конструктора

```
new String(str.getChars(), "UTF-8"),
```

где **str** – строка в формате Unicode, можно установить необходимый алфавит с помощью региональной кодировки в качестве второго параметра конструктора, 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:

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

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

int compareTo(String s) и **compareToIgnoreCase(String s)** – лексикографическое сравнение строк с учетом и без учета регистра. Метод осуществляет вычитание кодов символов вызывающей и передаваемой в метод строк и возвращает целое значение. Метод возвращает значение нуль в случае, когда **equals()** возвращает значение **true**;

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

String substring(int n, int m) – извлечение из строки подстроки длины **m-n**, начиная с позиции **n**. Нумерация символов в строке начинается с нуля;

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

int length() – определение длины строки;

int indexOf(char ch) – определение позиции символа в строке;

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

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

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

String intern() – заносит строку в "пул" литералов и возвращает ее объектную ссылку;

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

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

boolean isEmpty() – возвращает **true**, если длина строки равна **0**;

byte[] getBytes(), **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** – извлечение символов строки в массив байт или символов;

static String format(String format, Object... args), **format(Locale l, String format, Object... args)** – генерирует форматированную строку, полученную с использованием формата, интернационализации и др.;

String[] split(String regex), **split(String regex, int limit)** – поиск вхождения в строку заданного регулярного выражения (разделителя) и деление исходной строки в соответствии с этим на массив строк.

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 */
package chapt07;

public class DemoString {
   static int i;

   public static void main(String[] args) {
      char s[] = { 'J', 'a', 'v', 'a' }; // массив
      // комментарий содержит результат выполнения кода
      String str = new String(s); // str="Java"
      if (!str.isEmpty()) {
         i = str.length(); // i=4
         str = str.toUpperCase(); // str="JAVA"
         String num = String.valueOf(6); // num="6"
         num = str.concat("-" + num); // num="JAVA-6"
         char ch = str.charAt(2); // ch='V'
         i = str.lastIndexOf('A'); // i=3 (-1 если нет)
         num = num.replace("6", "SE"); // num="JAVA-SE"
         str.substring(0, 4).toLowerCase(); // java
         str = num + "-6";// str="JAVA-SE-6"
         String[] arr = str.split("-");
         for (String ss : arr)
            System.out.println(ss);
      } else { System.out.println("String is empty!");

      }
   }
}
```

В результате будет выведен массив строк:

```
JAVA
SE
6
```

При использовании методов класса **String**, изменяющих строку, создается новый измененный объект класса **String**. Сохранить изменения в объекте класса **String** можно только с применением оператора присваивания, т.е.

установкой ссылки на этот новый объект. В следующем примере будет выведено последнее после присваивания значение **str**.

```
/* пример # 2 : передача строки по ссылке: RefString.java */
package chapt07;

public class RefString {
     public static void changeStr(String s) {
          s.concat(" Microsystems");// создается новая строка
     }
     public static void main(String[] args) {
          String str = new String("Sun");
          changeStr(str);
          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(String s)** приводит к созданию нового объекта.

В следующем примере рассмотрены особенности 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 */
package chapt07;

public class EqualStrings {
     public static void main(String[] args) {
          String s1 = "Java";
          String s2 = "Java";
          String s3 = new String("Java");
          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 строковые объекты расположены в различных участках памяти, значения их хэш-кодов совпадают.

Т.к. в Java все ссылки хранятся в стеке, а объекты – в куче, то при создании объекта **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
package chapt07;

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

Ниже рассмотрена сортировка массива строк методом выбора.

```
// пример # 5 : сортировка: SortArray.java
package chapt07;

public class SortArray {
     public static void main(String[] args) {
String a[] = {"  Alena", "Alice  ", " alina",
"  albina", "   Anastasya", "   ALLA ", "AnnA  "};
          for(int j = 0; j < a.length; j++)
               a[j] = a[j].trim().toLowerCase();
          for(int j = 0; j < a.length - 1; j++)
            for(int i = j + 1; i < a.length; i++)
```

```
                if(a[i].compareTo(a[j]) < 0)    {
                    String temp = a[j];
                    a[j] = a[i];
                    a[i] = temp;
                }
            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.

Классы StringBuilder и StringBuffer

Классы **StringBuilder** и **StringBuffer** являются "близнецами" и по своему предназначению близки к классу **String**, но, в отличие от последнего, содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно изменять.

Основным и единственным отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего. В версии 1.5.0 был добавлен непотокобезопасный (следовательно, более быстрый в обработке) класс **StringBuilder**, который следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

С помощью соответствующих методов и конструкторов объекты классов **StringBuffer**, **StringBuilder** и **String** можно преобразовывать друг в друга. Конструктор класса **StringBuffer** (также как и **StringBuilder**) может принимать в качестве параметра объект **String** или 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:

void setLength(int n) – установка размера буфера;

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

int capacity() – возвращение текущего размера буфера;

StringBuffer append(параметры**)** – добавление к содержимому объекта строкового представления аргумента, который может быть символом, значением базового типа, массивом и строкой;

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

StringBuffer deleteCharAt(int index) – удаление символа;

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

StringBuffer 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 */
package chapt07;

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** конструктор по умолчанию автоматически резервирует некоторый объем памяти (16 символов), 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 ёмкость 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 */
package chapt07;

public class RefStringBuffer {
     public 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 */
package chapt07;

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**.

Форматирование строк

Для создания форматированного текстового вывода предназначен класс **java.util.Formatter**. Этот класс обеспечивает преобразование формата, позволяющее выводить числа, строки, время и даты в любом необходимом разработчику виде.

В классе **Formatter** объявлен метод **format()**, который преобразует переданные в него параметры в строку заданного формата и сохраняет в объекте типа **Formatter**. Аналогичный метод объявлен у классов **PrintStream** и **PrintWriter**. Кроме того, у этих классов объявлен метод **printf()** с параметрами идентичными параметрам метода **format()**, который осуществляет форматированный вывод в поток, тогда как метод **format()** сохраняет изменения в объекте типа **Formatter**. Таким образом, метод **printf()** автоматически использует возможности класса **Fomatter** и подобен функции **printf()** языка С.

Класс **Formatter** преобразует двоичную форму представления данных в форматированный текст. Он сохраняет форматированный текст в буфере, содержимое которого можно получить в любой момент. Можно предоставить классу **Formatter** автоматическую поддержку этого буфера либо задать его явно при создании объекта. Существует возможность сохранения буфера класса **Formatter** в файле.

Для создания объекта класса существует более десяти конструкторов. Ниже приведены наиболее употребляемые:

Formatter()

```
   Formatter(Appendable buf)
   Formatter(Appendable buf, Locale loc)
   Formatter(String filename) throws FileNotFoundException
   Formatter(String filename, String charset)
throws FileNotFoundException, UnsupportedEncodingException
   Formatter(File outF) throws FileNotFoundException
   Formatter(OutStream outStrm)
```

В приведенных образцах **buf** задает буфер для форматированного вывода. Если параметр **buf** равен **null**, класс **Formatter** автоматически размещает объект типа **StringBuilder** для хранения форматированного вывода. Параметр **loc** определяет региональные и языковые настройки. Если никаких настроек не задано, используются настройки по умолчанию. Параметр **filename** задает имя файла, который получит форматированный вывод. Параметр **charset** определяет кодировку. Если она не задана, используется кодировка, установленная по умолчанию. Параметр **outF** передает ссылку на открытый файл, в котором будет храниться форматированный вывод. В параметре **outStrm** передается ссылка на поток вывода, который будет получать отформатированные данные. Если используется файл, выходные данные записываются в файл.

В классе объявлены следующие методы:

Formatter format(String fmtString, Object...args) – форматирует аргументы, переданные в аргументе переменной длины **args** (количество аргументов в списке вывода не фиксировано), в соответствии со спецификаторами формата, содержащимися в **fmtString**. Возвращает вызывающий объект;

Formatter format(Locale loc, String fmtString, Object...args) – форматирует аргументы, переданные в аргументе переменной длины **args**, в соответствии со спецификаторами формата, содержащимися в **fmtString**. При форматировании используются региональные установки, заданные в **loc**. Возвращает вызывающий объект;

IOException ioException() – если объект, приемник отформатированного вывода, генерирует исключение типа **IOException**, возвращает это исключение. В противном случае возвращает **null**;

Locale locale() – возвращает региональные установки вызывающего объекта;

Appendable out() – возвращает ссылку на базовый объект-приемник для выходных данных;

void flush() – переносит информацию из буфера форматирования и производит запись в указанное место выходных данных, находящихся в буфере. Метод чаще всего используется объектом класса **Formatter**, связанным с файлом;

void close() – закрывает вызывающий объект класса **Formatter**, что приводит к освобождению ресурсов, используемых объектом. После закрытия объекта типа **Formatter** он не может использоваться повторно. Попытка использовать закрытый объект приводит к генерации исключения типа **FormatterClosedEception**;

String toString() – возвращает объект типа **String**, содержащий отформатированный вывод.

При форматировании используются спецификаторы формата:

Спецификатор формата	Выполняемое форматирование
%a	Шестнадцатеричное значение с плавающей точкой
%b	Логическое (булево) значение аргумента
%c	Символьное представление аргумента
%d	Десятичное целое значение аргумента
%h	Хэш-код аргумента
%e	Экспоненциальное представление аргумента
%f	Десятичное значение с плавающей точкой
%g	Выбирает более короткое представление из двух: **%e** или **%f**
%o	Восьмеричное целое значение аргумента
%n	Вставка символа новой строки
%s	Строковое представление аргумента
%t	Время и дата
%x	Шестнадцатеричное целое значение аргумента
%%	Вставка знака **%**

Так же возможны спецификаторы с заглавными буквами: **%A** (эквивалентно **%a**). Форматирование с их помощью обеспечивает перевод символов в верхний регистр.

```
/*пример # 9 : форматирование строки при помощи метода format():
 SimpleFormatString.java */
package chapt07;
import java.util.Formatter;

public class SimpleFormatString {
      public static void main(String[] args){
            Formatter f = new Formatter(); // объявление объекта
// форматирование текста по формату %S, %c
f.format("This %s is about %n%S %c", "book","java",'6');
            System.out.print(f);
      }
}
```

В результате выполнения этого кода будет выведено:

```
This book is about
JAVA 6
```

```
/*пример # 10 : форматирование чисел с использованием спецификаторов %x,
%o, %a, %g: FormatterDemoNumber.java */
package chapt07;
import java.util.Formatter;

public class FormatterDemoNumber {
      public static void main(String[] args) {
            Formatter f = new Formatter();
```

```
            f.format("Hex: %x, Octal:  %o", 100, 100);
            System.out.println(f);
            f = new Formatter();
            f.format("%a", 100.001);
            System.out.println(f);
            f = new Formatter();
            for (double i = 1000; i < 1.0e+10; i *= 100) {
                  f.format("%g ", i);
                  System.out.println(f);
            }
      }
}
```

В результате выполнения этого кода будет выведено:

```
Hex: 64, Octal:  144
0x1.90010624dd2f2p6
1000.00
1000.00 100000
1000.00 100000 1.00000e+07
1000.00 100000 1.00000e+07 1.00000e+09
```

Все спецификаторы для форматирования даты и времени могут употребляться только для типов **long**, **Long**, **Calendar**, **Date**.

В таблице приведены некоторые из спецификаторов формата времени и даты.

Спецификатор формата	Выполняемое преобразование
%tH	Час (00 – 23)
%tI	Час (1 – 12)
%tM	Минуты как десятичное целое (00 – 59)
%tS	Секунды как десятичное целое (00 – 59)
%tL	Миллисекунды (000 – 999)
%tY	Год в четырехзначном формате
%ty	Год в двузначном формате (00 – 99)
%tB	Полное название месяца (“Январь”)
%tb или **%th**	Краткое название месяца (“янв”)
%tm	Месяц в двузначном формате (1 – 12)
%tA	Полное название дня недели (“Пятница”)
%ta	Краткое название дня недели (“Пт”)
%td	День в двузначном формате (1 – 31)
%tR	То же что и "**%tH:%tM**"
%tT	То же что и "**%tH:%tM:%tS**"
%tr	То же что и "**%tI:%tM:%tS %Tp**" где **%Tp** = (AM или PM)
%tD	То же что и "**%tm/%td/%ty**"
%tF	То же что и "**%tY–%tm–%td**"
%tc	То же что и "**%ta %tb %td %tT %tZ %tY**"

```
/*пример # 11 : форматирование даты и времени:
FormatterDemoTimeAndDate.java */
package chapt07;
import java.util.*;

public class FormatterDemoTimeAndDate {
      public static void main(String args[]) {
            Formatter f = new Formatter();
            Calendar cal = Calendar.getInstance();

            // вывод в 12-часовом временном формате
            f.format("%tr", cal);
            System.out.println(f);

            // полноформатный вывод времени и даты
            f = new Formatter();
            f.format("%tc", cal);
            System.out.println(f);

            // вывод текущего часа и минуты
            f = new Formatter();
            f.format("%tl:%tM", cal, cal);
            System.out.println(f);

            // всевозможный вывод месяца
            f = new Formatter();
            f.format("%tB %tb %tm", cal, cal, cal);
            System.out.println(f);
      }
}
```

В результате выполнения этого кода будет выведено:

```
03:28:08 PM
Пт янв 06 15:28:08 EET 2006
3:28
Январь янв 01
```

Спецификатор точности применяется только в спецификаторах формата **%f**, **%e**, **%g** для данных с плавающей точкой и в спецификаторе **%s** – для строк. Он задает количество выводимых десятичных знаков или символов. Например, спецификатор **%10.4f** выводит число с минимальной шириной поля 10 символов и с четырьмя десятичными знаками. Принятая по умолчанию точность равна шести десятичным знакам.

Примененный к строкам спецификатор точности задает максимальную длину поля вывода. Например, спецификатор **%5.7s** выводит строку длиной не менее пяти и не более семи символов. Если строка длиннее, конечные символы отбрасываются.

Ниже приведен пример на использование флагов форматирования.

```
/*пример # 12: применение флагов форматирования: FormatterDemoFlags.java */
package chapt07;
```

```
import java.util.*;

public class FormatterDemoFlags {
      public static void main(String[] args) {
            Formatter f = new Formatter();

            // выравнивание вправо
            f.format("|%10.2f|", 123.123);
            System.out.println(f);

            // выравнивание влево
            // применение флага '-'
            f = new Formatter();
            f.format("|%-10.2f|", 123.123);
            System.out.println(f);

            f = new Formatter();
            f.format("% (d", -100);
            // применение флага ' ' и '('
            System.out.println(f);

            f = new Formatter();
            f.format("%,.2f", 123456789.34);
            // применение флага ','
            System.out.println(f);

            f = new Formatter();
            f.format("%.4f", 1111.1111111);
            // задание точности представления для чисел
            System.out.println(f);

            f = new Formatter();
f.format("%.16s", "Now I know class java.util.Formatter");
            // задание точности представления для строк
            System.out.println(f);

      }
}
```

В результате выполнения этого кода будет выведено:

```
|    123,12|
|123,12    |
(100)
123 456 789,34
1111,1111
Now I know class
```

У класса **Formatter** есть полезное свойство, которое позволяет задавать аргумент, к которому следует применить конкретный спецификатор формата. По умолчанию соответствие между спецификаторами и аргументами, на которые они воздействуют, устанавливается в соответствии с порядком их следования, слева

направо. Это означает, что первый спецификатор формата соответствует первому аргументу, второй спецификатор – второму аргументу и т. д. Однако, используя порядковый номер или индекс аргумента, можно указать явное соответствие спецификатора формата аргументу.

Порядковый номер аргумента указывается за знаком **%** в спецификаторе формата и имеет следующий формат: **N$**. Символ **N** обозначает порядковый номер нужного аргумента, нумерация аргументов начинается с единицы.

```
/*пример #13: применение порядкового номера аргумента:
FormatterDemoArguments.java */
package chapt07;
import java.util.Formatter;

public class FormatterDemoArguments {
      public static void main(String[] args) {
            Formatter f = new Formatter();
            Number n[] = { 4, 2.2, 3, 1.1 };
            f.format("%4$.1f %2$.1f %3$d %1$d", n[0], n[1],
                  n[2], n[3]);
            System.out.println(f);
      }
}
```

В результате выполнения этого кода будет выведено:

```
1,1 2,2 3 4
```

Такой же вывод легко получить, используя метод **printf()** в виде:

```
System.out.printf("%4$.1f %2$.1f %3$d %1$d", n[0], n[1],
                                          n[2], n[3]);
```

Лексический анализ текста

Класс **StringTokenizer** содержит методы, позволяющие разбивать текст на лексемы, отделяемые разделителями. Набор разделителей по умолчанию: пробел, символ табуляции, символ новой строки, перевод каретки. В задаваемой строке разделителей можно указывать другие разделители, например «= , ; : ».

Класс **StringTokenizer** имеет конструкторы:

```
StringTokenizer(String str);
StringTokenizer(String str, String delimiters);
StringTokenizer(String str, String delimiters,
                                    Boolean delimAsToken);
```

Некоторые методы:

String nextToken() – возвращает лексему как **String** объект;

boolean hasMoreTokens() – возвращает **true**, если одна или несколько лексем остались в строке;

int countToken() – возвращает число лексем.

Класс был реализован в самой первой версии языка. Однако в настоящее время существуют более совершенные средства по обработке текстовой информации – регулярные выражения.

Регулярные выражения

Класс **java.util.regex.Pattern** применяется для определения регулярных выражений (шаблонов), для которых ищется соответствие в строке, файле или другом объекте, представляющем последовательность символов. Для определения шаблона применяются специальные синтаксические конструкции. О каждом соответствии можно получить информацию с помощью класса **java.util.regex.Matcher**.

Далее приведены основные логические конструкции для задания шаблона.

Если в строке, проверяемой на соответствие, необходимо, чтобы в какой-либо позиции находился один из символов некоторого символьного набора, то такой набор (класс символов) можно объявить, используя одну из следующих конструкций:

[abc]	**a**, **b** или **c**
[^abc]	символ, исключая **a**, **b** и **c**
[a-z]	символ между **a** и **z**
[a-d[m-p]]	либо между **a** и **d**, либо между **m** и **p**
[e-z&&[dem]]	**e** либо **m** (конъюнкция)

Кроме стандартных классов символов, существуют предопределенные классы символов:

.	любой символ
\d	[**0-9**]
\D	[**^0-9**]
\s	[**\t\n\x0B\f\r**]
\S	[**^\s**]
\w	[**a-zA-Z_0-9**]
\W	[**^\w**]
\p{javaLowerCase}	~ **Character.isLowerCase()**
\p{javaUpperCase}	~ **Character.isUpperCase()**

При создании регулярного выражения могут использоваться логические операции:

ab	после **a** следует **b**
a|b	**a** либо **b**
(a)	**a**

Скобки, кроме их логического назначения, также используются для выделения групп.

Для определения регулярных выражений недостаточно одних классов символов, т. к. в шаблоне часто нужно указать количество повторений. Для этого существуют квантификаторы.

a?	**a** один раз или ни разу
a*	**a** ноль или более раз
a+	**a** один или более раз
a{n}	**a** n раз
a{n,}	**a** n или более раз
a{n,m}	**a** от n до m

Существует еще два типа квантификаторов, которые образованы прибавлением суффикса **?** (слабое, или неполное совпадение) или **+** («жадное», или собственное совпадение) к вышеперечисленным квантификаторам. Неполное совпадение соответствует выбору с наименее возможным количеством символов, а собственное – с максимально возможным.

Класс **Pattern** используется для простой обработки строк. Для более сложной обработки строк используется класс **Matcher**, рассматриваемый ниже.

В классе **Pattern** объявлены следующие методы:

Pattern compile(String regex) – возвращает **Pattern**, который соответствует **regex.**

Matcher matcher(CharSequence input) – возвращает **Matcher**, с помощью которого можно находить соответствия в строке **input.**

boolean matches(String regex, CharSequence input) – проверяет на соответствие строки **input** шаблону **regex.**

String pattern() – возвращает строку, соответствующую шаблону.

String[] split(CharSequence input) – разбивает строку **input**, учитывая, что разделителем является шаблон.

String[] split(CharSequence input, int limit) – разбивает строку **input** на не более чем **limit** частей.

С помощью метода **matches()** класса **Pattern** можно проверять на соответствие шаблону целой строки, но если необходимо найти соответствия внутри строки, например, определять участки, которые соответствуют шаблону, то класс **Pattern** не может быть использован. Для таких операций необходимо использовать класс **Matcher.**

Начальное состояние объекта типа **Matcher** не определено. Попытка вызвать какой-либо метод класса для извлечения информации о найденном соответствии приведет к возникновению ошибки **IllegalStateException**. Для того чтобы начать работу с объектом **Matcher**, нужно вызвать один из его методов:

boolean matches() – проверяет, соответствует ли вся строка шаблону;

boolean lookingAt() – пытается найти последовательность символов, начинающуюся с начала строки и соответствующую шаблону;

boolean find() или **boolean find(int start)** – пытается найти последовательность символов, соответствующих шаблону, в любом месте строки. Параметр **start** указывает на начальную позицию поиска.

Иногда необходимо сбросить состояние **Matcher**'а в исходное, для этого применяется метод **reset()** или **reset(CharSequence input)**, который также устанавливает новую последовательность символов для поиска.

Для замены всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку можно применить метод **replaceAll(String replacement)**.

Для того чтобы ограничить поиск границами входной последовательности, применяется метод **region(int start, int end)**, а для получения значения этих границ – **regionEnd()** и **regionStart()**. С регионами связано несколько методов:

Matcher useAnchoringBounds(boolean b) – если установлен в **true**, то начало и конец региона соответствуют символам **^** и **$** соответственно.

boolean hasAnchoringBounds() – проверяет закрепленность границ.

В регулярном выражении для более удобной обработки входной последовательности применяются группы, которые помогают выделить части найденной подпоследовательности. В шаблоне они обозначаются скобками "**(**" и "**)**". Номера групп начинаются с единицы. Нулевая группа совпадает со всей найденной подпоследовательностью. Далее приведены методы для извлечения информации о группах.

int end() – возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону;

int end(int group) – возвращает индекс последнего символа указанной группы;

String group() – возвращает всю подпоследовательность, удовлетворяющую шаблону;

String group(int group) – возвращает конкретную группу;

int groupCount() – возвращает количество групп;

int start() – возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону;

int start(int group) – возвращает индекс первого символа указанной группы;

boolean hitEnd() – возвращает истину, если был достигнут конец входной последовательности.

Следующий пример показывает как можно использовать возможности классов **Pattern** и **Matcher** для поиска, разбора и разбивки строк.

```
/* пример # 14 : обработка строк с помощью шаблонов :  DemoRegular.java */
package chapt07;
import java.util.regex.*;

public class DemoRegular {
    public static void main(String[] args) {
            //проверка на соответствие строки шаблону
            Pattern p1 = Pattern.compile("a+y");
            Matcher m1 = p1.matcher("aaay");
            boolean b = m1.matches();
            System.out.println(b);
            //поиск и выбор подстроки, заданной шаблоном
            String regex =
                        "(\\w+)@(\\w+\\.)(\\w+)(\\.\\w+)*";
            String s =
             "адреса эл.почты:mymail@tut.by и rom@bsu.by";
            Pattern p2 = Pattern.compile(regex);
            Matcher m2 = p2.matcher(s);
            while (m2.find())
              System.out.println("e-mail: " + m2.group());

//разбивка строки на подстроки с применением шаблона в качестве разделителя
            Pattern p3 = Pattern.compile("\\d+\\s?");
            String[] words =
                  p3.split("java5tiger 77 java6mustang");
```

```
            for (String word : words)
                  System.out.println(word);
    }
}
```

В результате будет выведено:

```
true
e-mail: mymail@tut.by
e-mail: rom@bsu.by
java
tiger
java
mustang
```

Следующий пример показывает, как использовать группы, а также собственные и неполные квантификаторы.

/ пример # 15 : группы и квантификаторы : Groups.java */*

```
package chapt07;

public class Groups {
    public static void main(String[] args) {
        String input = "abdcxyz";
        myMatches("([a-z]*)([a-z]+)", input);
        myMatches("([a-z]?)([a-z]+)", input);
        myMatches("([a-z]+)([a-z]*)", input);
        myMatches("([a-z]?)([a-z]?)", input);
    }
    public static void myMatches(String regex,
            String input) {
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(input);
        if(matcher.matches()) {
            System.out.println("First group: "
                + matcher.group(1));
            System.out.println("Second group: "
                + matcher.group(2));
        } else
            System.out.println("nothing");
        System.out.println();
    }
}
```

Результат работы программы:

```
First group: abdcxy
Second group: z

First group: a
Second group: bdcxyz

First group: abdcxyz
Second group:

nothing
```

В первом случае к первой группе относятся все возможные символы, но при этом остается минимальное количество символов для второй группы.

Во втором случае для первой группы выбирается наименьшее количество символов, т. к. используется слабое совпадение.

В третьем случае первой группе будет соответствовать вся строка, а для второй не остается ни одного символа, так как вторая группа использует слабое совпадение.

В четвертом случае строка не соответствует регулярному выражению, т. к. для двух групп выбирается наименьшее количество символов.

В классе **Matcher** объявлены два полезных метода для замены найденных подпоследовательностей во входной строке.

Matcher appendReplacement(StringBuffer sb, String replacement) – метод читает символы из входной стоки и добавляет их в **sb**. Чтение останавливается на **start()-1** позиции предыдущего совпадения, после чего происходит добавление в **sb** строки **replacement**. При следующем вызове этого метода производится добавление символов, начиная с символа с индексом **end()** предыдущего совпадения.

StringBuffer appendTail(StringBuffer sb) – добавляет оставшуюся часть символов из входной последовательности в **sb**. Как правило, вызывается после одного или нескольких вызовов метода **appendReplacement()**.

Интернационализация текста

Класс **java.util.Locale** позволяет учесть особенности региональных представлений алфавита, символов и проч. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять. Для некоторых стран региональные параметры устанавливаются с помощью констант, например: **Locale.US**, **Locale.FRANCE**. Для других стран объект **Locale** нужно создавать с помощью конструктора:

```
Locale myLocale = new Locale("bel", "BY");
```

Получить доступ к текущему варианту региональных параметров можно следующим образом:

```
Locale current = Locale.getDefault();
```

Если, например, в ОС установлен регион «Россия» или в приложении с помощью **new Locale("ru", "RU")**, то следующий код (при выводе результатов выполнения на консоль)

```
current.getCountry();//код региона
current.getDisplayCountry();//название региона
current.getLanguage();//код языка региона
current.getDisplayLanguage();//название языка региона
```

позволяет получить информацию о регионе в виде:

```
RU
Россия
ru
русский
```

Для создания приложений, поддерживающих несколько языков, существует целый ряд решений. Самое логичное из них – использование взаимодействия классов **java.util.ResourceBundle** и **Locale**. Класс **ResourceBundle** предназначен в первую очередь для работы с текстовыми файлами свойств (расширение **.properties**). Каждый объект **ResourceBundle** представляет собой набор объектов соответствующих подтипов, которые разделяют одно и то же базовое имя, к которому можно получить доступ через поле **parent**. Следующий список показывает возможный набор соответствующих ресурсов с базовым именем **text**. Символы, следующие за базовым именем, показывают код языка, код страны и тип операционной системы. Например, файл **text_de_CH.properties** соответствует объекту **Locale**, заданному кодом языка немецкого (**de**) и кодом страны Швейцарии (**CH**).

```
text.properties
text_ru.properties
text_de_CH.properties
text_en_CA_UNIX.properties
```

Чтобы выбрать определенный объект **ResourceBundle**, следует вызвать метод **ResourceBundle.getBundle(**параметры**)**. Следующий фрагмент выбирает **text** объекта **ResourceBundle** для объекта **Locale**, который соответствует английскому языку, стране Канаде и платформе UNIX.

```
Locale currentLocale = new Locale("en", "CA", "UNIX");
ResourceBundle rb =
   ResourceBundle.getBundle("text", currentLocale);
```

Если объект **ResourceBundle** для заданного объекта **Locale** не существует, то метод **getBundle()** извлечет наиболее общий. В случае если общее определение файла ресурсов не задано, то метод **getBundle()** генерирует исключительную ситуацию **MissingResourceException**. Чтобы этого не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов, а именно: **text.properties** в отличие от частных случаев вида:

```
text_en_US.properties
text_bel_BY.properties
```

В файлах свойств информация должна быть организована по принципу:

```
key1 = value1
key2 = value2
...
```

Например: **str1 = To be or not to be?**

Перечисление всех ключей в виде **Enumeration<String>** можно получить вызовом метода **getKeys()**. Конкретное значение по ключу извлекается методом **String getString(String key)**.

В следующем примере в зависимости от выбора пользователя известная фраза будет выведена на одном из трех языков.

// пример # 16 : поддержка различных языков: HamletInternational.java

```
package chapt8;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Locale;
```

```
import java.util.ResourceBundle;

public class HamletInternational {
      public static void main(String[] args) {
            String country = "", language = "";
            System.out.println("1 - Английский");
            System.out.println("2 - Белорусский");
            System.out.println("Любой символ - Русский");
            char i = 0;
            try {
                  i = (char) System.in.read();
            } catch (IOException e1) {
                  e1.printStackTrace();
            }
            switch (i) {
             case '1':
                  country = "US";
                  language = "EN";
                  break;
             case '2':
                  country = "BY";
                  language = "BEL";
            }
            Locale current = new Locale(language, country);
            ResourceBundle rb =
            ResourceBundle.getBundle("text", current);
            try {
                  String st = rb.getString("str1");
                  String s1 =
            new String(st.getBytes("ISO-8859-1"), "UTF-8");
                  System.out.println(s1);

                  st = rb.getString("str2");
                  String s2 =
            new String(st.getBytes("ISO-8859-1"), "UTF-8");
                  System.out.println(s2);
            } catch (UnsupportedEncodingException e) {
                  e.printStackTrace();
            }
      }
}
```

Файл **text_en_US.properties** содержит следующую информацию:

```
str1 = To be or not to be?
str2 = This is a question.
```

Файл **text_bel_BY.properties**:

```
str1 = Быць або не быць?
str2 = Вось у чым пытанне.
```

Файл **text.properties**:

```
str1 = Быть или не быть?
str2 = Вот в чём вопрос.
```

Интернационализация чисел

Стандарты представления дат и чисел в различных странах могут существенно отличаться. Например, в Германии строка **"1.234,567"** воспринимается как «одна тысяча двести тридцать четыре целых пятьсот шестьдесят семь тысячных», для русских и французов данная строка просто непонятна и не может представлять число.

Чтобы сделать такую информацию конвертируемой в различные региональные стандарты, применяются возможности класса **java.text.NumberFormat**. Первым делом следует задать или получить текущий объект **Locale** с шаблонами регионального стандарта и создать с его помощью объект форматирования **NumberFormat**. Например:

```
NumberFormat nf =
   NumberFormat.getInstance(new Locale("RU"));
```

с конкретными региональными установками или с установленными по умолчанию для приложения:

```
NumberFormat.getInstance();
```

Далее для преобразования строки в число и обратно используются методы **Number parse(String source)** и **String format(double number)** соответственно.

В предлагаемом примере производится преобразование строки, содержащей число, в три различных региональных стандарта, а затем одно из чисел преобразуется из одного стандарта в два других.

// пример # 17 : региональные представления чисел: DemoNumberFormat.java

```
package chapt07;
import java.text.*;
import java.util.Locale;

public class DemoNumberFormat {
      public static void main(String args[]) {
            NumberFormat nfGe =
                  NumberFormat.getInstance(Locale.GERMAN);
            NumberFormat nfUs =
                  NumberFormat.getInstance(Locale.US);
            NumberFormat nfFr =
                  NumberFormat.getInstance(Locale.FRANCE);

            double iGe=0, iUs=0, iFr =0;
      String str = "1.234,567";//строка, представляющая число
            try {
      //преобразование строки в германский стандарт
                  iGe = nfGe.parse(str).doubleValue();
      //преобразование строки в американский стандарт
                  iUs = nfUs.parse(str).doubleValue();
      //преобразование строки во французский стандарт
                  iFr = nfFr.parse(str).doubleValue();
            } catch (ParseException e) {
                  e.printStackTrace();
            }
```

```
        System.out.printf("iGe = %f\niUs = %f\niFr = %f",
                                                iGe, iUs, iFr);

        //преобразование числа из германского в американский стандарт
        String sUs = nfUs.format(iGe);
        //преобразование числа из германского во французский стандарт
        String sFr = nfFr.format(iGe);
        System.out.println("\n" + sUs + "\n" + sFr);
        }
}
```

Результат работы программы:

```
iGe = 1234,567000
iUs = 1,234000
iFr = 1,000000
1,234.567
1 234,567
```

Аналогично выполняются переходы от одного регионального стандарта к другому при отображении денежных сумм.

Интернационализация дат

Учитывая исторически сложившиеся способы отображения даты и времени в различных странах и регионах мира, в языке создан механизм поддержки всех национальных особенностей. Эту задачу решает класс **java.text.DateFormat**. С его помощью учтены: необходимость представления месяцев и дней недели на национальном языке; специфические последовательности в записи даты и часовых поясов; возможности использования различных календарей.

Процесс получения объекта, отвечающего за обработку регионального стандарта даты, похож на создание объекта, отвечающего за национальные представления чисел, а именно:

```
DateFormat df = DateFormat.getDateInstance(
                    DateFormat.MEDIUM, new Locale("BY"));
```

или по умолчанию:

```
DateFormat.getDateInstance();
```

Константа **DateFormat.MEDIUM** указывает на то, что будут представлены только дата и время без указания часового пояса. Для указания часового пояса используются константы класса **DateFormat** со значением **LONG** и **FULL**. Константа **SHORT** применяется для сокращенной записи даты, где месяц представлен в виде своего порядкового номера.

Для получения даты в виде строки для заданного региона используется метод **String format(Date date)** в виде:

```
String dat = df.format(new Date());
```

С помощью метода **Date parse(String source)** можно преобразовать переданную в виде строки дату в объектное представление конкретного регионального формата, например:

```
String str = "April 3, 2006";
Date d = df.parse(str);
```

Класс содержит большое количество методов, позволяющих выполнять разнообразные манипуляции с датой и временем.

В качестве примера рассмотрено преобразование заданной даты в различные региональные форматы.

```
// пример # 18 : региональные представления дат: DemoDateFormat.java
package chapt07;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.*;

public class DemoDateFormat {
      public static void main(String[] args) {
      DateFormat df =
DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.US);

            Date d = null;
            String str = "April 3, 2006";
            try {
                  d = df.parse(str);
                  System.out.println(d);
            } catch (ParseException e) {
                  e.printStackTrace();
            }
            df =
DateFormat.getDateInstance(DateFormat.FULL,
                              new Locale("ru","RU"));
            System.out.println(df.format(d));

            df =
DateFormat.getDateInstance(DateFormat.FULL,Locale.GERMAN);
            System.out.println(df.format(d));

            d = new Date();
      //загрузка в объект df текущего времени
            df = DateFormat.getTimeInstance();
      //представление и вывод времени в текущем формате дат
            System.out.println(df.format(d));
      }
}
```

Результат работы программы:

```
Mon Apr 03 00:00:00 EEST 2006
3 Апрель 2006 г.
Montag, 3. April 2006
05:45:16
```

Чтобы получить представление текущей даты во всех возможных региональных стандартах, можно воспользоваться следующим фрагментом кода:

```
Date d = new Date();
```

```
Locale[] locales =
      DateFormat.getAvailableLocales();
            for (Locale loc : locales) {
                  DateFormat df =
DateFormat.getDateInstance(DateFormat.FULL, loc);
            System.out.println(loc.toString() + "---> "
                              + df.format(d));
            }
```

В результате будут выведены две сотни строк, каждая из которых представляет текущую дату в соответствии с региональным стандартом, выводимым перед датой с помощью инструкции **`loc.toString()`**.

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 `P`, если она не последняя в слове, ошибочно напечатана буква `A` вместо `O`. 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 телеграммы, 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.
22. Преобразовать текст так, чтобы каждое слово начиналось с заглавной буквы.
23. Подсчитать количество содержащихся в данном тексте знаков препинания.
24. В заданном тексте найти сумму всех встречающихся цифр.
25. Из кода Java удалить все комментарии (//, /*, /**).
26. Дан текст на английском языке. Пусть все слова встречаются четное количество раз, за исключением одного. Определить это слово. При сравнении слов регистр не учитывать.
27. Определить сумму всех целых чисел, встречающихся в заданном тексте.
28. Из английского текста удалить все пробелы, если он разделяет два различных знака препинания и если рядом с ним находится еще один пробел.
29. Строка состоит из упорядоченных чисел от 0 до 100000, записанных подряд без пробелов. Определить, что будет подстрокой от позиции n до m.
30. Определить количество вхождений заданного слова в текст, игнорируя регистр символов и считая буквы «е», «ё», и «и», «й» одинаковыми.
31. Преобразовать текст так, чтобы только первые буквы каждого предложения были заглавными.
32. Заменить в тексте все шаблоны типа %user%Бендер%/user% на Бендер.
33. В Java код добавить корректные getter и setter-методы для всех полей данного класса, при их отсутствии.
34. Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.

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 первой 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.
15. В тексте исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.
16. Заменить все одинаковые рядом стоящие символы в тексте одним символом.
17. Вывести в заданном тексте все слова, расположив их в алфавитном порядке.
18. Подсчитать, сколько слов в заданном тексте начинается с прописной буквы.
19. Подсчитать, сколько раз заданное слово входит в текст.
20. Преобразовать каждое слово в тексте, удалив из него все последующие (предыдущие) вхождения первой (последней) буквы этого слова.
21. Вычеркнуть из текста минимальное количество предложений, так чтобы у любых двух оставшихся предложений было хотя бы одно общее слово.

22. Текст из n^2 символов шифруется по следующему правилу:
    – все символы текста записываются в квадратную таблицу размерности n в порядке слева направо, сверху вниз;
    – таблица поворачивается на 90^0 по часовой стрелке;
    – 1-я строка таблицы меняется местами с последней, 2-я – с предпоследней и т.д.
    – 1-й столбец таблицы меняется местами со 2-м, 3-й – с 4-м и т.д.
    – зашифрованный текст получается в результате обхода результирующей таблицы по спирали по часовой стрелке, начиная с левого верхнего угла.
    Зашифровать текст по указанному правилу.
23. На основании правила кодирования, описанного в предыдущем примере, расшифровать заданный набор символов.
24. Исключить из текста подстроку максимальной длины, начинающуюся и заканчивающуюся одним и тем же символом.
25. Осуществить сжатие английского текста, заменив каждую группу из двух или более рядом стоящих символов, на один символ, за которым следует количество его вхождений в группу. К примеру, строка hellowoooorld должна сжиматься в hel2owo4rld.
26. Распаковать текст, сжатый по правилу из предыдущего задания.
27. Определить, удовлетворяет ли имя файла маске. Маска может содержать символы ‘?’ (произвольный символ) и ‘*’ (произвольное количество произвольных символов).
28. Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства – по алфавиту. Словом считать максимальную группу подряд стоящих не пробельных символов.
29. Буквенная запись телефонных номеров основана на том, что каждой цифре соответствует несколько английских букв: 2 – ABC, 3 – DEF, 4 – GHI, 5 – JKL, 6 – MNO, 7 – PQRS, 8 – TUV, 9 – WXYZ. Написать программу, которая находит в заданном телефонном номере подстроку максимальной длины, соответствующую слову из словаря.
30. В заданном тексте найти подстроку максимальной длины, являющуюся палиндромом, т.е. читающуюся слева направо и справа налево одинаково.
31. Осуществить форматирование заданного текста с выравниванием по левому краю. Программа должна разбивать текст на строки с длиной, не превосходящей заданного количества символов. Если очередное слово не помещается в текущей строке, его необходимо переносить на следующую.
32. Изменить программу из предыдущего примера так, чтобы она осуществляла форматирование с выравниванием по обоим краям. Для этого добавить дополнительные пробелы между словами.
33. Добавить к программе из предыдущего примера возможность переноса слов по слогам. Предполагается, что есть доступ к словарю, в котором для каждого слова указано, как оно разбивается на слоги.
34. Пусть массив содержит миллион символов и необходимо сформировать из них строку путем конкатенации. Определить время работы кода. Как можно ускорить процесс, используя класс StringBuffer?

35. Алгоритм Барроуза – Уиллера для сжатия текстов основывается на преобразовании Барроуза – Уиллера. Оно производится следующим образом: для слова рассматриваются все его циклические сдвиги, которые затем сортируются в алфавитном порядке, после чего формируется слово из последних символов отсортированных циклических сдвигов. К примеру, для слова JAVA циклические сдвиги – это JAVA, AVAJ, VAJA, AJAV. После сортировки по алфавиту получим AJAV, AVAJ, JAVA, VAJA. Значит, результат преобразования – слово VJAA. Реализовать программно преобразование Барроуза – Уиллера для данного слова.
36. Восстановить слово по его преобразованию Барроуза – Уиллера. К примеру, получив на вход VJAA, в результате работы программа должна выдать слово JAVA.

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.

Глава 8
ИСКЛЮЧЕНИЯ И ОШИБКИ

Иерархия и способы обработки

Исключительные ситуации (исключения) возникают во время выполнения программы, когда возникшая проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Примерами являются особо «популярные»: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на нуль. При возникновении исключения создается объект, описывающий это исключение. Затем текущий ход выполнения приложения останавливается, и включается механизм обработки исключений. При этом ссылка на объект-исключение передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы. Если в классе используется метод, в котором может возникнуть проверяемая исключительная ситуация, но не предусмотрена ее обработка, то ошибка возникает еще на этапе компиляции. При создании такого метода программист должен включить в код обработку исключений, которые могут генерировать этот метод, или передать обработку исключения на более высокий уровень методу, вызвавшему данный метод.

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

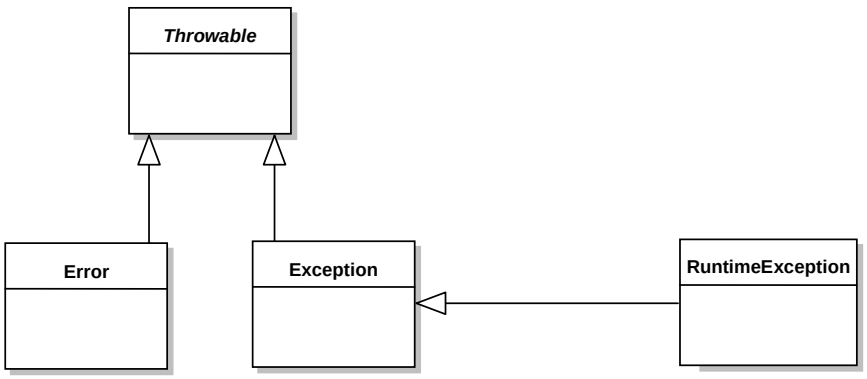


Рис. 8.1. Иерархия основных классов исключений

Исключительные ситуации типа **`Error`** возникают только во время выполнения программы. Такие исключения связаны с серьезными ошибками, к примеру – переполнение стека, и не подлежат исправлению и не могут обрабатываться приложением. Иерархия классов, наследуемых от класса **`Error`**, приведена на рисунке 8.2.

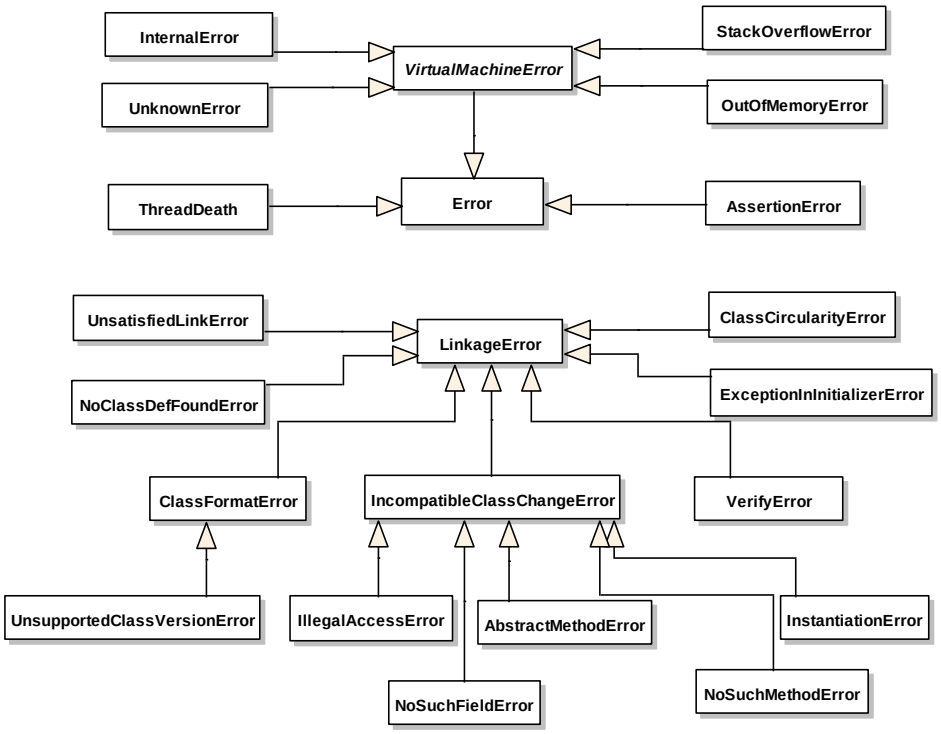


Рис. 8.2. Иерархия классов исключений, наследуемых от класса Error.

На рисунке 8.3 приведена иерархия классов исключений, наследуемых от класса **`Exception`**.

Проверяемые исключения должны быть обработаны в методе, который может их генерировать, или включены в **`throws`**-список метода для дальнейшей обработки в вызывающих методах. Возможность возникновения проверяемого исключения может быть отслежена на этапе компиляции кода.

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы. Иерархия этих исключений приведена на рисунке 8.4. В отличие от проверяемых исключений, класс **`RuntimeException`** и порожденные от него классы относятся к непроверяемым исключениям. Компилятор не проверяет, генерирует ли и обрабатывает ли метод эти исключения. Исключения типа **`RuntimeException`** автоматически генерируются при возникновении ошибок во время выполнения приложения. Таким образом, нет необходимости в проверке генерации исключения вида:

```
if(a==null) throw new NullPionterException();
```

объект класса **`NullPionterException`** при возникновении ошибки будет сгенерирован автоматически. Кроме этого, в любом случае нет необходимости в обработке этого исключения непосредственно в методе или в передаче на обработку вызывающему методу с помощью оператора **`throw`**. В конце концов исключение будет передано в метод **`main()`**, где обрабатывается вызовом метода **`printStackTrace()`**, выдающего данные трассировки.

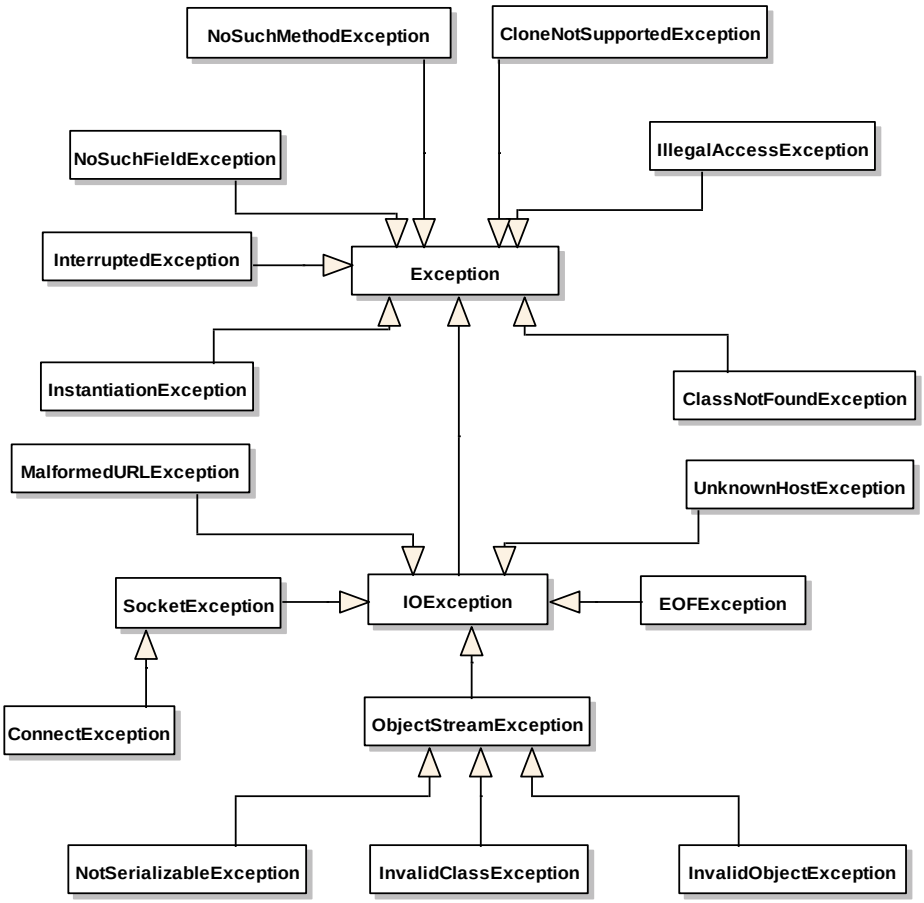


Рис. 8.3. Иерархия классов проверяемых (checked) исключительных ситуаций

Обычно используется один из трех способов обработки исключений:

- перехват и обработка исключения в блоке **`try-catch`** метода;
- объявление исключения в секции **`throws`** метода и передача вызывающему методу (для проверяемых исключений);
- использование собственных исключений.

Первый подход можно рассмотреть на следующем примере. При клонировании объекта в определенных ситуациях может возникать исключительная ситуация типа **`CloneNotSupportedException`**. Например:

```
public void changeObject(Student ob) {
            try {
                  Object temp = ob.clone();
                  //реализация
            } catch (CloneNotSupportedException e) {
                  System.err.print(e);
            }
      }
```

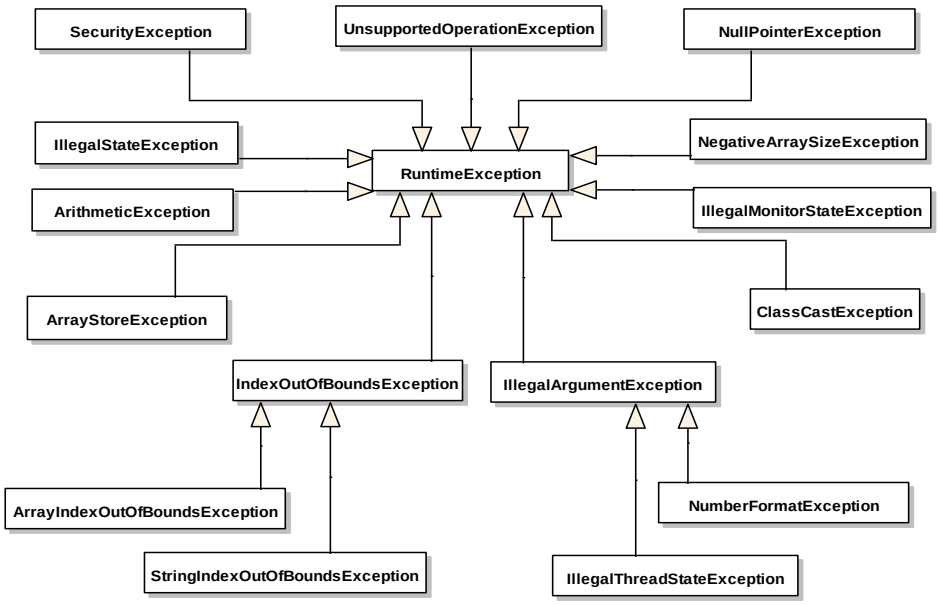


Рис. 8.4. Иерархия непроверяемых (unchecked) исключений

При клонировании может возникнуть исключительная ситуация в случае, если переданный объект не поддерживает клонирование (не включен интерфейс **`Cloneable`**). В этом случае генерируется соответствующий объект, и управление передается блоку **`catch`**, в котором обрабатывается данный тип исключения, иначе блок **`catch`** пропускается. Блок **`try`** похож на обычный логический блок. Блок **`catch(){}`** похож на метод, принимающий в качестве единственного параметра ссылку на объект-исключение и обрабатывающий этот объект.

Второй подход демонстрируется на этом же примере. Метод может генерировать исключения, которые сам не обрабатывает, а передает для обработки другим методам, вызывающим данный метод. В этом случае метод 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 имяМетода(список_аргументов)
        throws список_исключений { }
```

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

```
public void changeObject(Student ob)
            throws CloneNotSupportedException {
            Object temp = ob.clone();
                  //реализация
            }
```

Ключевое слово **`throws`** позволяет разобраться с исключениями методов «чужих» классов, код которых отсутствует. Обрабатывать исключение при этом будет метод, вызывающий **`changeObject()`**:

```
public void load(Student stud) {
     try {
          changeObject(stud);
     } catch (CloneNotSupportedException e) {
          String error = e.toString();
          System.err.println(error);
     }
}
```

Создание собственных исключений будет рассмотрено позже в этой главе.

Если в блоке **try** может быть сгенерировано в разных участках кода несколько типов исключений, то необходимо наличие нескольких блоков **catch**, если только блок **catch** не обрабатывает все типы исключений.

```
/* пример # 1 : обработка двух типов исключений: TwoException.java */
package chapt08;

public 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] = 71;
          } catch(ArithmeticException e) {
               System.err.println("деление на 0" + e);
          } catch(ArrayIndexOutOfBoundsException e) {
               System.err.println(
                       "превышение границ массива: " + e);
          }//последний catch
               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. Однако пример, приведенный выше, носит чисто демонстративный характер и не является образцом хорошего кода, так как в этой ситуации можно было обойтись простой проверкой аргументов на допустиые значения перед выполнением операций. К тому же генерация и обработка исключения – операция значительно более ресурсоемкая, чем вызов оператора **if** для проверки аргумента. Исключения должны применяться только для обработки исключительных ситуаций, и если существует возможность обойтись без них, то следует так и поступить.

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

```
try {/*код, который может вызвать исключение*/
} catch(RuntimeException e) { /* суперкласс RuntimeException
                                 перехватит объекты всех своих подклассов */
```

```
} catch(ArithmeticException e) {} /* не может быть вызван,
                                    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 */
package chapt08;

public 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.err.println(
                         "недопустимый размер буфера: " + e);
               }
          } catch (ArithmeticException e) {
               System.err.println("деление на 0" + e);
          }
     }
}
```

В результате запуска приложения 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.

Третий подход к обработке исключений будет рассмотрен ниже на примере создания пользовательских исключений.

Оператор throw

В программировании часто возникают ситуации, когда программисту необходимо самому инициировать генерацию исключения для указания, например, на заведомо ошибочный результат выполнения операции, на некорректные значения параметра метода и др. Исключительную ситуацию можно создать с помощью оператора **throw**, если объект-исключение уже существует, или инициализировать его прямо после этого оператора. 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**, генерирующей исключение:

```
throw объектThrowable;
```

Объект-исключение может уже существовать или создаваться с помощью оператора **new**:

```
throw new IOException();
```

При достижении оператора **throw** 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**. Инициализация объекта-исключения без оператора **throw** никакой исключительной ситуации не вызовет.

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 */
package chapt08;
import java.io.File;

public class ThrowGeneration {
      public static void connectFile(File file) {
        if (file == null || !file.exists())
            throw new IllegalArgumentException();/*генерация
                                                  исключения */
            //...
       }
     public static void main(String[] args) {
          File f = new File("demo.txt");
          // File f = null;
          try {
               connectFile(f);
          } catch(IllegalArgumentException e) {
               System.err.print("обработка unchecked-"
               + " исключения вне метода: " + e);
     }
}
```

Вызываемый метод **connectFile()** может (при отсутствии файла на диске или при аргументе **null**) генерировать исключение, перехватываемое обработчиком. В результате этого объект непроверяемого исключения **IllegalArgumentException**, как подкласса класса **RuntimeException**, передается обработчику исключений в методе **main()**.

В случае генерации проверяемого исключения компилятор требует 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 исключений, являющихся подклассами класса **RuntimeException** (unchecked) и используемых для отображения программных ошибок, при выполнении приложения **throws** в объявлении должен отсутствовать.

Ключевое слово finally

Возможна ситуация, при которой нужно выполнить некоторые действия по завершению программы (закрыть поток, освободить соединение с базой данных) 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*/}//необязателен
catch(Exception2 e2) {/*обработка исключения e2*/}//необязателен
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 перед выходом из метода даже в том случае, если перед ним были выполнены инструкции вида **return**, **break**, **continue**. Приведем пример:

```
/* пример # 4 : выполнение блоков finally: StudentFinally.java */
package chapt08;

public class StudentFinally {
     private static int age;

     public static void setAge(int age) {
          try {
          //реализация
               if(age <= 0)
               throw new RuntimeException("не бывает");
          } finally {
             System.out.print("освобождение ресурсов");
          //реализация
          }
             System.out.print("конец метода");
     }
     public static int getAgeWoman() {
          try {
               return age - 3;
          } finally {
               return age;
          }
     }
     public static void main(String[] args) {
          try {
          setAge(23);
          setAge(-5);
          } catch (RuntimeException e) {
               e.printStackTrace();
          }
```

```
        System.out.print(getAgeWoman());
    }
}
```

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

Собственные исключения

Разработчик может создать собственное исключение как подкласс класса **Exception** и затем использовать его при обработке ситуаций, не являющихся исключениями с точки зрения языка, но нарушающих логику вещей. Например, появление объектов типа **Human** (Человек) с отрицательным значением поля **age** (возраст).

```
/* пример # 5 : метод, вызывающий исключение, созданное программистом:
 RunnerLogic.java  */
package chapt08;

public class RunnerLogic {
    public static double salary(int coeff)
                                throws SalaryException {
        double d;
        try {
            if((d = 10 - 100/coeff) < 0)
            throw new SalaryException("negative salary");
            else return d;
        } catch (ArithmeticException e) {
            throw new SalaryException("div by zero", e);
        }
    }
    public static void main(String[] args) {
        try {
            double res = salary(3); //или 0, или 71;
        } catch (SalaryException e) {
            System.err.println(e.toString());
            System.err.println(e.getHiddenException());
        }
    }
}
```

При невозможности вычислить значение генерируется объект **ArithmeticException**, обработчик которого, в свою очередь, генерирует исключение **SalaryException**, 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 ин-

формацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину вызова **SalaryException**, он всего лишь должен вызвать метод **getHiddenException()**. Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки **SalaryException**.

```
/* пример # 6 : собственное исключение: SalaryException.java */
package chapt08;

public class SalaryException extends Exception {
     private Exception _hidden;

     public SalaryException(String er) {
          super(er);
     }
     public SalaryException(String er, Exception e) {
          super(er);
          _hidden = e;
     }
     public Exception getHiddenException() {
          return _hidden;
     }
}
```

Разработчики программного обеспечения стремятся к высокому уровню повторного использования кода, поэтому они постарались предусмотреть и закодировать все возможные исключительные ситуации. Поэтому при реальном программировании можно вполне обойтись без создания собственных классов исключений.

Наследование и исключения

Создание сложных распределенных систем редко обходится без наследования и обработки исключений. Следует знать два правила для проверяемых исключений при наследовании:

- переопределяемый метод в подклассе не может содержать в инструкции **throws** исключений, не обрабатываемых в соответствующем методе суперкласса;
- конструктор подкласса должен включить в свой блок **throws** все классы исключений или их суперклассы из блока **throws** конструктора суперкласса, к которому он обращается при создании объекта.

Первое правило имеет непосредственное отношение к расширяемости приложения. Пусть при добавлении в цепочку наследования нового класса его полиморфный метод включил в блок **throws** «новое» проверяемое исключение. Тогда методы логики пріложения, принимающие объект нового класса в качестве параметра и вызывающие данный полиморфный метод, не готовы обрабатывать «новое» исключение, так как ранее в этом не было необходимости. Поэтому при попытке добавления «нового» checked-исключения в полиморфный метод компилятор выдает сообщение об ошибке.

```
/* пример # 7 : полиморфизм и исключения: Stone.java: WhiteStone.java:
BlackStone.java: StoneLogic.java */
package chapt08;

class Stone { //старый класс
     public void build() throws FileNotFoundException {
                    /* реализация*/ }
}
class WhiteStone extends Stone {//старый класс
     public void build() {
          System.out.println("белый каменный шар");
     }
}
public class StoneLogic {//старый класс
     public static void infoStone(Stone stone) {
     try {
       stone.build();//обработка IOException не предусмотрена
     } catch(FileNotFoundException e) {
            System.err.print("файл не найден");
      }
    }
}
class BlackStone extends Stone {//новый класс
     public void build() throws IOException{//ошибка компиляции
          System.out.println("черный каменный шар");
          /* реализация*/
     }
}
```

Если же при объявлении метода суперкласса инструкция **throws** присутствует, то в подклассе эта инструкция может вообще отсутствовать или в ней могут быть объявлены любые исключения, являющееся подклассами исключения из блока **throws** метода суперкласса

Второе правило позволяет защитить программиста от возникновения неизвестных ему исключений при создании объекта.

```
/* пример # 8 : конструкторы и исключения: FileInput.java: SocketInput.java */
package chapt08;
import java.io.FileNotFoundException;
import java.io.IOException;

class FileInput {//старый класс
      public FileInput(String filename)
                  throws FileNotFoundException {
            //реализация
      }
}
class SocketInput extends FileInput {
//старый конструктор
      public SocketInput(String name)
```

```
                    throws FileNotFoundException {
            super(name);
            //реализация
        }
/старый конструктор
        public SocketInput() throws IOException {
            super("file.txt");
            //реализация
        }
/новый конструктор
        public SocketInput(String name, int mode) {/*ошибка
                                                    компиляции*/
            super(name);
            //реализация
        }
    }
```

В приведенном выше случае компилятор не разрешит создать конструктор подкласса, обращающийся к конструктору суперкласса без корректной инструкции **throws**. Если бы это было возможно, то при создании объекта подкласса класса **FileInput** не было бы никаких сообщений о возможности генерации исключения, и при возникновении исключительной ситуации ее источник было бы трудно идентифицировать.

Отладочный механизм assertion

Борьба за качество программ ведется всеми возможными способами. На этапе отладки найти неявные ошибки в функционировании приложения бывает довольно сложно. Например, в методе, устанавливающем возраст пользователя, информация о возрасте извлекается из внешних источников (файл, БД), и в результате получается отрицательное значение. Далее неверные данные влияют на результат вычисления среднего возраста пользователей и т.д. Определять и исправлять такие ситуации позволяет механизм проверочных утверждений (assertion). При помощи этого механизма можно сформулировать требования к входным, выходным и промежуточным данным методов классов в виде некоторых логических условий.

Попытка обработать ситуацию появления отрицательного возраста может выглядеть следующим образом:

```
        int age = ob.getAge();
            if (age >= 0) {
            // реализация
            } else {
            // сообщение о неправильных данных
            }
```

Теперь механизм assertion позволяет создать код, который будет генерировать исключение на этапе отладки проверки постусловия или промежуточных данных в виде:

```
        int age = ob.getAge();
            assert (age >= 0): "NEGATIVE AGE!!!";
            // реализация
```

Правописание инструкции **assert**:

```
assert (boolexp): expression;
assert (boolexp);
```

Выражение **boolexp** может принимать только значение типов **boolean** или **Boolean**, а **expression** – любое значение, которое может быть преобразовано к строке. Если логическое выражение получает значение **false**, то генерируется исключение **AssertionError**, и выполнение программы прекращается с выводом на консоль значения выражения **expression** (если оно задано).

Механизм assertion хорошо подходит для проверки инвариантов, например, перечислений:

```
enum Mono { WHITE, BLACK }
…
String str = "WHITE";//"GRAY"
Mono mono = Mono.valueOf(str);
// …
   switch (mono) {
         case WHITE : // …
               break;
         case BLACK : // …
               break;
         default :
               assert false : "Colored!";
   }
```

Создателями языка не рекомендуется использовать assertion при проверке параметров **public**-методов. В таких ситуациях лучше генерировать исключения одного из типов: **IllegalArgumentException**, **NullPointerException** или собственное исключение. Нет также особого смысла в механизме assertion при проверке пограничных значений переменных, поскольку исключительные ситуации генерируются в этом случае без посторонней помощи.

Assertion можно включать для отдельных классов и пакетов при запуске виртуальной машины в виде:

```
java –enableassertions MyClass
```

или

```
java –ea MyClass
```

Для выключения применяется **–da** или **-disableassertions**.

Задания к главе 8

Вариант А

Выполнить задания на основе варианта А главы 4, 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 8

Вопрос 8.1.

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 два).

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 8.2.

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 8.3.

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 8.4.

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 8.5.

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 9
ФАЙЛЫ. ПОТОКИ ВВОДА/ВЫВОДА

Потоки ввода/вывода используются для передачи данных в файловые потоки, на консоль или на сетевые соединения. Потоки представляют собой объекты соответствующих классов. Библиотека ввода/вывода предоставляет пользователю большое число классов и методов и постоянно обновляется.

Класс File

Для работы с физическим файлами и каталогами (директориями), расположенными на внешних носителях, 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 содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами файла, как права доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д.

Объект класса **File** создается одним из нижеприведенных способов:

```
File myFile = new File("\\com\\myfile.txt");
File myDir  = new File("c:\\jdk1.6.0\\src\\java\\io");
File myFile = new File(myDir, "File.java");
File myFile = new File("c:\\com", "myfile.txt");
File myFile = new File(new URI("Интернет-адрес"));
```

В первом случае создается объект, соответствующий файлу, во втором – подкаталогу. Третий и четвертый случаи идентичны. Для создания объекта указывается каталог и имя файла. В пятом – создается объект, соответствующий адресу в Интернете.

При создании объекта класса **File** любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Существует разница между разделителями, употребляющимися при записи пути к файлу: для системы Unix – "**/**", а для Windows – "****". Для случаев, когда неизвестно, в какой системе будет выполняться код, предусмотрены специальные поля в классе **File**:

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный в любой системе:

```
File myFile  = new File(File.separator + "com"
         + File.separator + "myfile.txt" );
```

Также предусмотрен еще один тип разделителей – для директорий:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

К примеру, для ОС Unix значение **pathSeparator=";"**, а для Windows – **pathSeparator=":"**.

В классе **File** объявлено более тридцати методов, наиболее используемые из них рассмотрены в следующем примере:

```
/* пример # 1 : работа с файловой системой: FileTest.java */
package chapt09;
import java.io.*;
import java.util.*;

public class FileTest {
   public static void main(String[] args) {
   //с объектом типа File ассоциируется файл на диске FileTest2.java
       File fp = new File("chapt09" + File.separator
             + "FileTest2.java");
       if(fp.exists()) {
          System.out.println(fp.getName() + " существует");

         if(fp.isFile()) {//если объект – дисковый файл
          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 :\t"
                        + new Date(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());
             }
       } else
          System.out.println("файл " + fp.getName()
                        + " не существует");
       try{
          if(fp.createNewFile())
             System.out.println("Файл " + fp.getName()
                        + " создан");
       } catch(IOException e) {
          System.err.println(e);
       }
//в объект типа File помещается каталог\директория
// в корне проекта должен быть создан каталог com.learn с несколькими файлами
    File dir = new File("com" + File.separator + "learn");
          if (dir.exists() && dir.isDirectory())/*если объект
                                          является каталогом и если этот
                                          каталог существует */
             System.out.println("каталог "
                   + dir.getName() + " существует");
```

```
            File[] files = dir.listFiles();
            for(int i = 0; i < files.length; i++){
               Date date = new Date(files[i].lastModified());
               System.out.print("\n" + files[i].getPath()
                     + " \t| " + files[i].length() + "\t| "
                     + date.toString());
             //использовать toLocaleString() или toGMTString()
          }
        // метод listRoots() возвращает доступные корневые каталоги
        File root = File.listRoots()[1];
        System.out.printf("\n%s %,d из %,d свободно.",
root.getPath(),root.getUsableSpace(),root.getTotalSpace());
    }
}
```

В результате файл **FileTest2.java** будет очищен, а на консоль выведено:

```
FileTest2.java существует
Путь к файлу:     chapt09\FileTest2.java
Абсолютный путь:  D:\workspace\chapt09\FileTest2.java
Размер файла:     2091
Последняя модификация : Fri Mar 31 12:26:50 EEST 2006
Файл доступен для чтения:     true
Файл доступен для записи:     true
Файл удален:      true
Файл FileTest2.java создан
каталог learn существует
com\learn\bb.txt  | 9   | Fri Mar 24 15:30:33 EET 2006
com\learn\byte.txt| 8   | Thu Jan 26 12:56:46 EET 2006
com\learn\cat.gif | 670 | Tue Feb 03 00:44:44 EET 2004
C:\ 3 665 334 272 из 15 751 376 896 свободно.
```

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

Байтовые и символьные потоки ввода/вывода

При создании приложений всегда возникает необходимость прочитать информацию из какого-либо источника и сохранить результат. Действия по чтению/записи информации представляют собой стандартный и простой вид деятельности. Самые первые классы ввода/вывода связаны с передачей и извлечением последовательности байтов.

Потоки ввода последовательности байтов являются подклассами абстрактного класса **InputStream**, потоки вывода – подклассами абстрактного класса **OutputStream**. Эти классы являются суперклассами для ввода массивов байтов, строк, объектов, а также для выбора из файлов и сетевых соединений. При работе с файлами используются подклассы этих классов соответственно **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом.

Для чтения байта или массива байтов используются абстрактные методы **`read()`** и **`read(byte[] b)`** класса **`InputStream`**. Метод возвращает **`–1`**, если достигнут конец потока данных, поэтому возвращаемое значение имеет тип **`int`**, не **`byte`**. При взаимодействии с информационными потоками возможны различные исключительные ситуации, поэтому обработка исключений вида **`try-catch`** при использовании методов чтения и записи является обязательной. В конкретных классах потоков ввода указанные выше методы реализованы в соответствии с предназначением класса. В классе **`FileInputStream`** данный метод читает один байт из файла, а поток **`System.in`** как встроенный объект подкласса **`InputStream`** позволяет вводить информацию с консоли. Абстрактный метод **`write(int b)`** класса **`OutputStream`** записывает один байт в поток вывода. Оба эти метода блокируют поток до тех пор, пока байт не будет записан или прочитан. После окончания чтения или записи в поток его всегда следует закрывать с помощью метода **`close()`**, для того чтобы освободить ресурсы приложения.

Поток ввода связывается с одним из источников данных, в качестве которых могут быть использованы массив байтов, строка, файл, «pipe»-канал, сетевые соединения и др. Набор классов для взаимодействия с перечисленными источниками приведен на рис. 9.1.

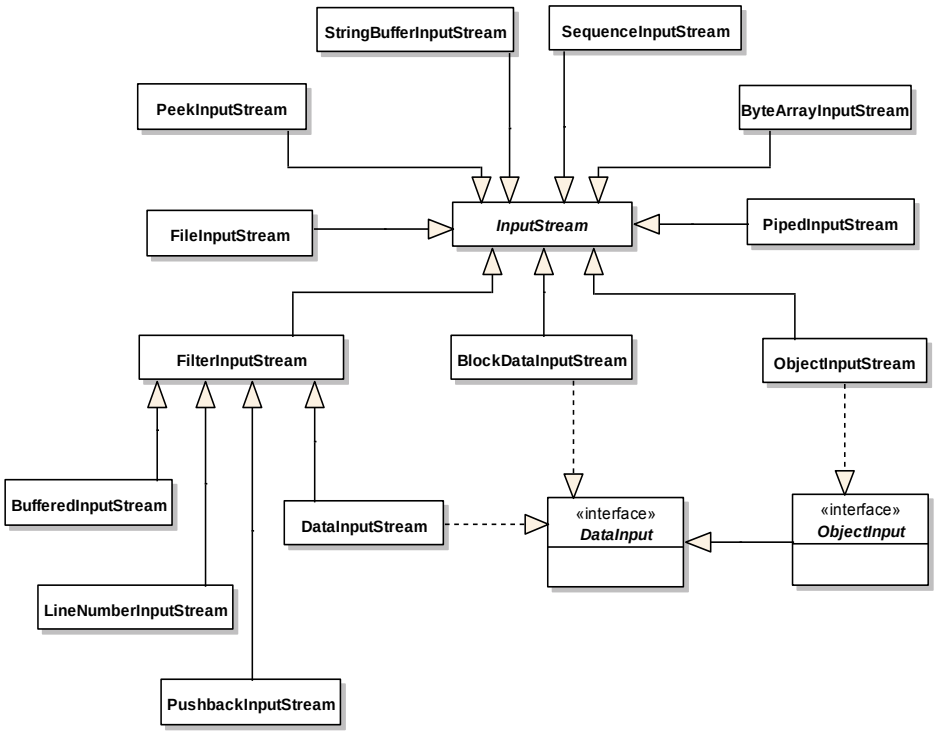


Рис. 9.1. Иерархия классов байтовых потоков ввода

Абстрактный класс **FilterInputStream** используется как шаблон для настройки классов ввода, наследуемых от класса **InputStream**. Класс **DataInputStream** предоставляет методы для чтения из потока данных значений базовых типов, но начиная с версии 1.2 класс был помечен как deprecated и не рекомендуется к использованию. Класс **BufferedInputStream** присоединяет к потоку буфер для ускорения последующего доступа.

Для вывода данных используются потоки следующих классов.

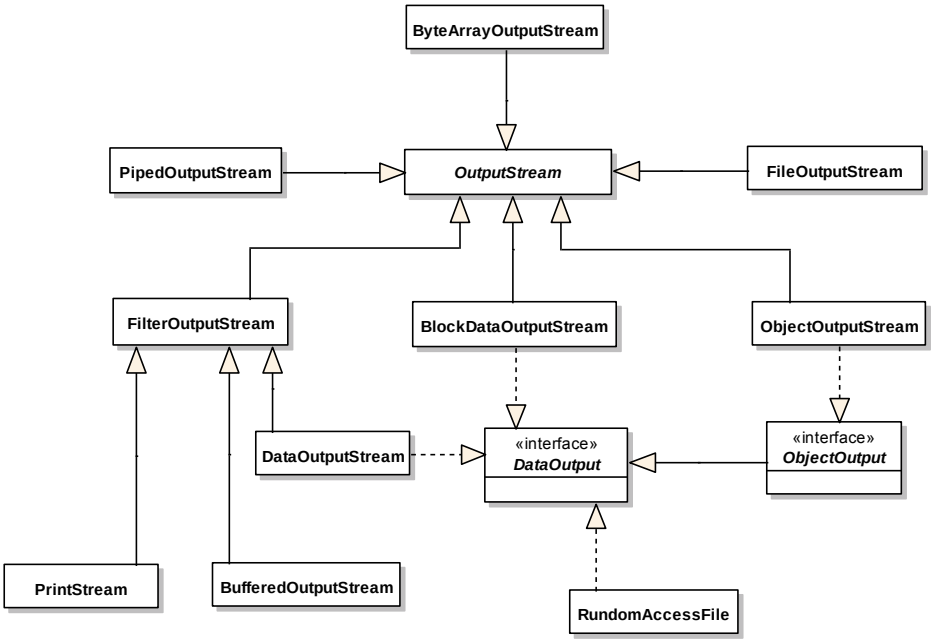


Рис. 9.2. Иерархия классов байтовых потоков вывода

Абстрактный класс **FilterOutputStream** используется как шаблон для настройки производных классов. Класс **BufferedOutputStream** присоединяет буфер к потоку для ускорения вывода и уменьшения доступа к внешним устройствам.

Начиная с версии 1.2 пакет **java.io** подвергся значительным изменениям. Появились новые классы, которые производят скоростную обработку потоков, хотя и не полностью перекрывают возможности классов предыдущей версии.

Для обработки символьных потоков в формате Unicode применяется отдельная иерархия подклассов абстрактных классов **Reader** и **Writer**, которые почти полностью повторяют функциональность байтовых потоков, но являются более актуальными при передаче текстовой информации. Например, аналогом класса **FileInputStream** является класс **FileReader**. Такой широкий выбор потоков позволяет выбрать наилучший способ записи в каждом конкретном случае.

В примерах по возможности используются способы инициализации для различных семейств потоков ввода/вывода.

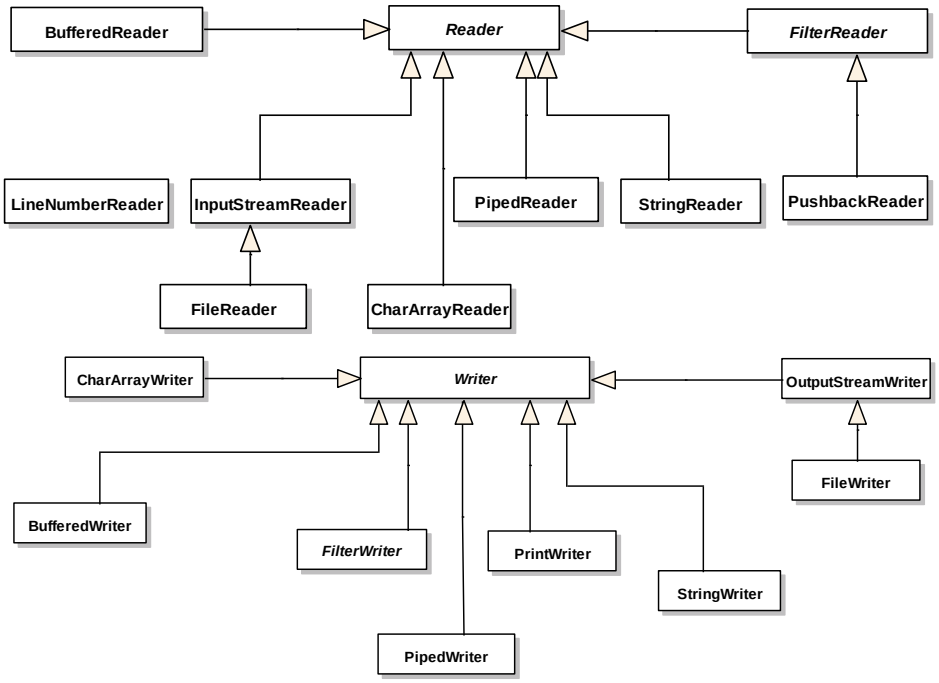


Рис. 9.3. Иерархия символьных потоков ввода/вывода

```
/* пример # 2 : чтение по одному байту (символу) из потока ввода : ReadDemo.java */
package chapt09;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadDemo {
   public static void main(String[] args) {
      File f = new File("file.txt");//должен существовать!

      int b, count = 0;
      try {
         FileReader is = new FileReader(f);
      /* FileInputStream is = new FileInputStream(f);*/ //альтернатива
         while ((b = is.read()) != -1) {/*чтение*/
            System.out.print((char)b);
            count++;
         }
         is.close(); // закрытие потока ввода
       } catch (IOException e) {
            System.err.println("ошибка файла: " + e);
      }
      System.out.print("\n число байт = " + count);
   }
}
```

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

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

// пример # 3 : вывод массива в поток в виде символов и байтов: WriteRunner.java

```
package chapt09;
import java.io.*;

public class WriteRunner {
      public static void main(String[] args) {
            String pArray[] = { "2007 ", "Java SE 6" };
            File fbyte = new File("byte.txt");
            File fsymb = new File("symbol.txt");
            try {
                  FileOutputStream fos =
                        new FileOutputStream(fbyte);
                  FileWriter fw = new FileWriter(fsymb);
                  for (String a : pArray) {
                        fos.write(a.getBytes());
                        fw.write(a);
                  }
                  fos.close();
                  fw.close();
            } catch (IOException e) {
                  System.err.println("ошибка файла: " + e);
            }
      }
}
```

В результате будут получены два файла с идентичным набором данных, но созданные различными способами.

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 байтов, 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 # 4 : запись и чтение из потока: RandomFiles.java */*

```
package chapt09;
import java.io.*;

public class RandomFiles {
```

```
    public static void main(String[] args) {
        double data[] = { 1, 10, 50, 200, 5000 };
        try {
        RandomAccessFile rf =
            new RandomAccessFile("temp.txt", "rw");
            for (double d : data)
                rf.writeDouble(d); // запись в файл
        /* чтение в обратном порядке */
        for (int i = data.length - 1; i >= 0; i--) {
                rf.seek(i * 8);
        // длина каждой переменной типа double равна 8-и байтам
            System.out.println(rf.readDouble());
            }
            rf.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

В результате будет выведено:

```
5000.0
200.0
50.0
10.0
1.0
```

Предопределенные потоки

Система ввода/вывода языка Java содержит стандартные потоки ввода, вывода и вывода ошибок. 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 консолью, но могут быть переназначены на другое устройство.

Для назначения вывода текстовой информации в произвольный поток следует использовать класс **PrintWriter**, являющийся подклассом абстрактного класса **Writer**.

При наиболее удобного вывода информации в файл (или в любой другой поток) следует организовать следующую последовательность инициализации потоков с помощью класса **PrintWriter**:

```
new PrintWriter(new BufferedWriter(
      new FileWriter(new File("file.txt"))));
```

В итоге класс **BufferedWriter** выступает классом-оберткой для класса **FileWriter**, так же как и класс **BufferedReader** для **FileReader**.

Приведенный ниже пример демонстрирует вывод в файл строк и чисел с плавающей точкой.

```
// пример # 5 : вывод в файл: DemoWriter.java
package chapt09;
import java.io.*;

public class DemoWriter {
      public static void main(String[] args) {
            File f = new File("res.txt");
            FileWriter fw = null;
            try {
                  fw = new FileWriter(f, true);
            } catch (IOException e) {
      System.err.println("ошибка открытия потока " + e);
      System.exit(1);
            }
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);

            double[] v = { 1.10, 1.2, 1.401, 5.01 };
            for (double version : v)
                  pw.printf("Java %.2g%n", version);
            pw.close();
      }
}
```

В итоге в файл **res.txt** будет помещена следующая информация:

```
Java 1.1
Java 1.2
Java 1.4
Java 5.0
```

Для вывода данных в файл в текстовом формате использовался фильтрованный поток вывода **PrintWriter** и метод **printf()**. После соединения этого потока с дисковым файлом посредством символьного потока **BufferedWriter** и удобного средства записи в файл **FileWriter** становится возможной запись текстовой информации с помощью обычных методов **println()**, **print()**, **printf()**, **format()**, **write()**, **append()**.

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 соответственно. Этот поток для организации чтения из файла лучше всего инициализировать объектом класса **FileReader** в виде:

```
new BufferedReader(new FileReader(new File("f.txt")));
```

Чтение из созданного в предыдущем примере файла с использованием удобной технологии можно произвести следующим образом:

```
// пример # 6 : чтение из файла: DemoReader.java
package chapt09;
import java.io.*;
```

```
public class DemoReader {
      public static void main(String[] args) {
            try {
      BufferedReader br =
            new BufferedReader(new FileReader("res.txt"));
                  String tmp = "";
                  while ((tmp = br.readLine()) != null) {
                        //пробел использовать как разделитель
                        String[] s = tmp.split("\\s");
                        //вывод полученных строк
                        for (String res : s)
                              System.out.println(res);
                  }
                  br.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

В консоль будет выведено:

```
Java
1.1
Java
1.2
Java
1.4
Java
5.0
```

Сериализация объектов

Кроме данных базовых типов, в поток можно отправлять объекты классов.

Процесс преобразования объектов в потоки байтов для хранения называется сериализацией. Процесс извлечения объекта из потока байтов называется десериализацией. Существует два способа сделать объект сериализуемым.

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**. Спецификаторы **transient** и **static** означают, что поля, помеченные ими, не могут быть предметом сериализации, но существует различие в десериализации. Так, поле со спецификатором **transient** после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением **null**), а поле со спецификатором **static** получает значение по умолчанию в случае отсутствия в области

видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта.

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 его метод **readObject()**, возвращающий ссылку на класс **Object**. После чего следует преобразовать полученный объект к нужному типу.

Необходимо знать, что при использовании **Serializable** десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом не вызывается.

```
/* пример # 7 : запись сериализованного объекта в файл и его десериализация :
Student.java : DemoSerialization.java */
package chapt09;
import java.io.*;

class Student implements Serializable {
   protected static String faculty;
   private String name;
   private int id;
   private transient String password;
   private static final long serialVersionUID = 1L;
   /*значение этого поля для класса будет дано далее*/

   public Student(String nameOfFaculty, String name,
            int id, String password){
      faculty = nameOfFaculty;
      this.name = name;
      this.id = id;
      this.password = password;
   }
   public String toString(){
      return "\nfaculty " + faculty + "\nname " + name
            + "\nID " + id + "\npassword " + password;
   }
}

public class DemoSerialization {
   public static void main(String[] args) {
      // создание и запись объекта
      Student goncharenko =
            new Student("MMF", "Goncharenko", 1, "G017s9");
            System.out.println(goncharenko);
```

```
        File fw = new File("demo.dat");
        try {
           ObjectOutputStream ostream =
              new ObjectOutputStream(
                    new FileOutputStream(fw));

           ostream.writeObject(goncharenko);
           ostream.close();
        } catch (IOException e) {
            System.err.println(e);
        }
        Student.faculty = "GEO";//изменение значения static-поля
    // чтение и вывод объекта
        File fr = new File("demo.dat");
        try {
           ObjectInputStream istream =
                    new ObjectInputStream(
                          new FileInputStream(fr));

           Student unknown =
                    (Student)istream.readObject();
           istream.close();
           System.out.println(unknown);
        } catch (ClassNotFoundException ce) {
              System.err.println(ce);
              System.err.println("Класс не существует");
        } catch (FileNotFoundException fe) {
              System.err.println(fe);
              System.err.println("Файл не найден");
        } catch (IOException ioe) {
              System.err.println(ioe);
              System.err.println("Ошибка доступа");
        }
    }
}
```

В результате выполнения данного кода в консоль будет выведено:

```
faculty MMF
name Goncharenko
ID 1
password G017s9

faculty GEO
name Goncharenko
ID 1
password null
```

В итоге поля **name** и **id** нового объекта **unknown** сохранили значения, которые им были присвоены до записи в файл. Поле **password** со спецификатором **transient** получило значение по умолчанию, соответствующее типу (объект-

ный тип всегда инициализируется по умолчанию значением **null**). Поле **faculty**, помеченное как статическое, получает то значение, которое имеет это поле на текущий момент, то есть при создании объекта **goncharenko** поле получило значение **MMF**, а затем значение статического поля было изменено на **GEO**. Если же объекта данного типа нет в области видимости, то статическое поле также получает значение по умолчанию.

Если поля класса являются объектами другого класса, то необходимо, чтобы тот класс тоже реализовал интерфейс **Serializable.**

При сериализации объекта класса, реализующего интерфейс **Serializable**, учитывается порядок объявления полей в классе. Поэтому при изменении порядка десериализация пройдет некорректно. Это обусловлено тем, что в каждый класс, реализующий интерфейс **Serializable**, на стадии компиляции добавляется поле **private static final long serialVersionUID**. Это поле содержит уникальный идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса – полям, их порядку объявления, методам, их порядку объявления.

Это поле записывается в поток при сериализации класса. Это единственный случай, когда **static**-поле сериализуется.

При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение **java.io.InvalidClassException.** Соответственно, при любом изменении в классе это поле поменяет свое значение.

Если набор полей класса и их порядок жестко определены, методы класса могут меняться. В этом случае сериализации ничего не угрожает, однако стандартный механизм не даст десериализовать данные. В таких случаях можно вручную в классе определить поле **private static final long serialVersionUID.**

Вместо реализации интерфейса **Serializable** можно реализовать **Externalizable**, который содержит два метода:

```
void writeExternal(ObjectOutput out)
void readExternal(ObjectInput in)
```

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра должен сам класс. Для этого в нем должны быть переопределены методы **writeExternal()** и **readExternal()** интерфейса **Externalizable**. Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

При восстановлении **Externalizable**-объекта экземпляр создается путем вызова конструктора без аргументов, после чего вызывается метод **readExternal()**, поэтому необходимо проследить, чтобы в классе был пустой конструктор. Для сохранения состояния вызываются методы **ObjectOutput**, с помощью которых можно записать как примитивные, так и объектные значения. Для корректной работы в соответствующем методе **readExternal()** эти значения должны быть считаны в том же порядке.

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

ObjectInputStream.GetField
ObjectOutputStream.PutField.

Консоль

Одним из классов, предоставляющих дополнительные возможности чтения и последующей типизации информации консоли (или любого другого потока), является **java.util.Scanner**, введенный в пятой версии языка. Также для взаимодействия с консолью применяется класс **java.io.Console**, введенный в шестой версии языка.

```
// пример # 8 : ввод информации : UserHelper.java
package chapt01;
//подключение классов ввода
import java.io.Console;
// обработчик ошибок ввода
import java.util.InputMismatchException;

public class Helper {
  //чтение информации из консоли с помощью класса Console
  public void readFromConsole() {
     Console con = System.console();
     if (con != null) {
        con.printf("Введите числовой код:");
        int code = 0;
        try {
           code = Integer.valueOf(con.readLine());
           System.out.println("Код доступа:" + code);
     } catch (InputMismatchException e) {
           con.printf("неправильный формат кода" + e);
        }
        if (code != 0) {
           con.printf("Введите пароль:");
           String password;
           char passTemp[] =
                  con.readPassword("Введите пароль: ");
           password = new String(passTemp);
           System.out.println("Пароль:" + password);
                     }
     } else {
        System.out.println("Консоль недоступна");
     }
   }
}
// пример # 9 : инициализация объектов и вызов методов: Runner.java
package chapt01;

public class Runner {
      public static void main(String[] args) {
            Helper helper = new Helper();
```

```
            helper.readFromConsole();
        }
}
```

В ответ на запрос можно ввести некоторые данные и получить следующий результат:

```
Введите числовой код:
1001
Введите пароль:
****
Код доступа: 1001
Пароль: pass
```

При вводе значения **code**, не являющегося цифрой, на экран будет выдано сообщение об ошибке при попытке его преобразования в целое число, так как метод **valueOf()** пытается преобразовать строку в целое число, не проверив предварительно, может ли быть выполнено это преобразование.

Класс Scanner

Объект класса **java.util.Scanner** принимает форматированный объект (ввод) и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable** или **ReadableByteChannel**.

Класс определяет следующие конструкторы:

```
Scanner(File source) throws FileNotFoundException
Scanner(File source, String charset)
   throws FileNotFoundException
Scanner(InputStream source)
Scanner(InputStream source, String charset)
Scanner(Readable source)
Scanner(ReadableByteChannel source)
Scanner(ReadableByteChannel source, String charset)
Scanner(String source),
```

где **source** – источник входных данных, а **charset** – кодировка.

Объект класса **Scanner** читает лексемы из источника, указанного в конструкторе, например из строки или файла. Лексема – это набор данных, выделенный набором разделителей (по умолчанию пробелами). В случае ввода из консоли следует определить объект:

```
Scanner con = new Scanner(System.in);
```

После создания объекта его используют для ввода, например целых чисел, следующим образом:

```
write(con.hasNextInt()) {
   int n = con.nextInt();
}
```

В классе **Scanner** определены группы методов, проверяющих данные заданного типа на доступ для ввода. Для проверки наличия произвольной лексемы используется метод **hasNext()**. Проверка конкретного типа производится с помощью одного из методов **boolean hasNext**Тип**()** или **boolean**

hasNextТип**(int radix)**, где **radix** – основание системы счисления. Например, вызов метода **hasNextInt()** возвращает **true**, только если следующая входящая лексема – целое число. Если данные указанного типа доступны, они считываются с помощью одного из методов Тип **next**Тип**()**. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

```
// пример # 10 : разбор файла: ScannerLogic.java : ScannerDemo.java
package chap09;
import java.io.*;
import java.util.Scanner;

class ScannerLogic {
   static String filename = "scan.txt";
   public static void scanFile() {
      try {
         FileReader fr =
            new FileReader(filename);
         Scanner scan = new Scanner(fr);
         while (scan.hasNext()) {//чтение из файла

            if (scan.hasNextInt())
                  System.out.println(
                        scan.nextInt() + ":int");
            else if (scan.hasNextDouble())
                  System.out.println(
                        scan.nextDouble() + ":double");
            else if (scan.hasNextBoolean())
                  System.out.println(
                        scan.nextBoolean() + ":boolean");
            else
                  System.out.println(
                        scan.next() + ":String");
         }
      } catch (FileNotFoundException e) {
         System.err.println(e);
      }
   }
   public static void makeFile() {
      try {
         FileWriter fw =
            new FileWriter(filename);//создание потока для записи
         fw.write("2 Java 1,5 true 1.6 ");//запись данных
         fw.close();
      } catch (IOException e) {
         System.err.println(e);
      }
   }
```

```
}
public class ScannerDemo {
      public static void main(String[] args) {
            ScannerLogic.makeFile();
            ScannerLogic.scanFile();
      }
}
```

В результате выполнения программы будет выведено:

```
2:int
Java:String
1.5:double
true:boolean
1.6:String
```

Процедура проверки типа реализована при с помощью методов **hasNext**Тип**()**. Такой подход предпочтителен из-за отсутствия возможности возникновения исключительной ситуации, так как ее обработка требует на порядок больше ресурсов, чем нормальное течение программы. Для чтения строки из потока ввода применяются методы **next()** или **nextLine()**.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String pattern)**, где **pattern** содержит набор разделителей.

```
/* пример # 11 : применение разделителей: ScannerDelimiterDemo.java*/
package chapt09;
import java.util.Scanner;

public class ScannerDelimiterDemo {
   public static void main(String args[]) {
      double sum = 0.0;

      Scanner scan =
      new Scanner("1,3;2,0;  8,5; 4,8; 9,0; 1; 10");
         scan.useDelimiter(";\\s*");
         while (scan.hasNext()) {
            if (scan.hasNextDouble())
                sum += scan.nextDouble();
            else System.out.println(scan.next());
         }
         System.out.printf("Сумма чисел = " + sum);
      }
   }
}
```

В результате выполнения программы будет выведено:

```
Сумма чисел = 36.6
```

Использование шаблона **"; *"** указывает объекту класса **Scanner**, что ‘;’ и ноль или более пробелов следует рассматривать как разделитель.

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в следующей строке текста. Если шаблон найден, соответствующая ему подстрока извлекается из строки ввода. Если совпадений не найдено, то возвращается **null**.

Методы **String findWithinHorizon(Pattern pattern, int count)** и **String findWithinHorizon(String pattern, int count)** производят поиск заданного шаблона в ближайших **count** символах. Можно пропустить образец с помощью метода **skip(Pattern pattern)**.

Если в строке ввода найдена подстрока, соответствующая образцу **pattern**, метод **skip()** просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод **skip()** генерирует исключение **NoSuchElementException**.

Архивация

Для хранения классов языка Java и связанных с ними ресурсов в языке Java используются сжатые архивные **jar**-файлы.

Для работы с архивами в спецификации Java существуют два пакета – **java.util.zip** и **java.util.jar** соответственно для архивов **zip** и **jar**. Различие форматов **jar** и **zip** заключается только в расширении архива zip. Пакет **java.util.jar** аналогичен пакету **java.util.zip**, за исключением реализации конструкторов и метода **void putNextEntry(ZipEntry e)** класса **JarOutputStream**. Ниже будет рассмотрен только пакет **java.util.jar**. Чтобы переделать все примеры на использование **zip**-архива, достаточно всюду в коде заменить **Jar** на **Zip**.

Пакет **java.util.jar** позволяет считывать, создавать и изменять файлы форматов **jar**, а также вычислять контрольные суммы входящих потоков данных.

Класс **JarEntry** (подкласс **ZipEntry**) используется для предоставления доступа к записям **jar**-файла. Наиболее важными методами класса являются:

void setMethod(int method) – устанавливает метод сжатия записи;

int getMethod() – возвращает метод сжатия записи;

void setComment(String comment) – устанавливает комментарий записи;

String getComment() – возвращает комментарий записи;

void setSize(long size) – устанавливает размер несжатой записи;

long getSize() – возвращает размер несжатой записи;

long getCompressedSize() – возвращает размер сжатой записи;

У класса **JarOutputStream** существует возможность записи данных в поток вывода в **jar**-формате. Он переопределяет метод **write()** таким образом, чтобы любые данные, записываемые в поток, предварительно сжимались. Основными методами данного класса являются:

void setLevel(int level) – устанавливает уровень сжатия. Чем больше уровень сжатия, тем медленней происходит работа с таким файлом;

void putNextEntry(ZipEntry e) – записывает в поток новую **jar**-запись. Этот метод переписывает данные из экземпляра **JarEntry** в поток вывода;

void closeEntry() – завершает запись в поток **jar**-записи и заносит дополнительную информацию о ней в поток вывода;

void write(byte b[], int off, int len) – записывает данные из буфера **b** начиная с позиции **off** длиной **len** в поток вывода;

void finish() – завершает запись данных **jar**-файла в поток вывода без закрытия потока;

void close() – закрывает поток записи.

```
/* пример # 12 : создание jar-архива: PackJar.java */
package chapt09;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.zip.Deflater;

public class PackJar {
   public static void pack(String[] filesToJar,
                  String jarFileName, byte[] buffer) {
      try {
            JarOutputStream jos =
                  new JarOutputStream(
                        new FileOutputStream(jarFileName));
            // метод сжатия
            jos.setLevel(Deflater.DEFAULT_COMPRESSION);
            for (int i = 0; i < filesToJar.length; i++) {
                  System.out.println(i);
            jos.putNextEntry(new JarEntry(filesToJar[i]));

            FileInputStream in =
                        new FileInputStream(filesToJar[i]);
             int len;
              while ((len = in.read(buffer)) > 0)
                  jos.write(buffer, 0, len);
              jos.closeEntry();
              in.close();
            }
            jos.close();
      } catch (IllegalArgumentException e) {
            e.printStackTrace();
            System.err.println("Некорректный аргумент");
      } catch (FileNotFoundException e) {
            e.printStackTrace();
```

```
            System.err.println("Файл не найден");
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println("Ошибка доступа");
        }
    }
    public static void main(String[] args) {
        System.out.println("Создание jar-архива");
        // массив файлов для сжатия
        String[] filesToJar = new String[2];
        filesToJar[0] = "chapt09//UseJar.java";
        filesToJar[1] = "chapt09//UseJar.class";
        byte[] buffer = new byte[1024];
        // имя полученного архива
        String jarFileName = "example.jar";
        pack(filesToJar, jarFileName, buffer);
    }
}
```

Класс **JarFile** обеспечивает гибкий доступ к записям, хранящимся в **jar**-файле. Это очень эффективный способ, поскольку доступ к данным осуществляется гораздо быстрее, чем при считывании каждой отдельной записи. Единственным недостатком является то, что доступ может осуществляться только для чтения. Метод **entries**() извлекает все записи из **jar**-файла. Этот метод возвращает список экземпляров **JarEntry** – по одной для каждой записи в **jar**- файле. Метод **getEntry(String name)** извлекает запись по имени. Метод **getInputStream()** создает поток ввода для записи. Этот метод возвращает поток ввода, который может использоваться приложением для чтения данных записи.

Класс **JarInputStream** читает данные в **jar**-формате из потока ввода. Он переопределяет метод **read()** таким образом, чтобы любые данные, считываемые из потока, предварительно распаковывались.

```
/* пример # 13 : чтение jar-архива: UnPackJar.java */
package chapt09;
import java.io.*;
import java.util.Enumeration;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

public class UnPackJar {
      private File destFile;
      // размер буфера для распаковки
      public final int BUFFER = 2048;

  public void unpack(String destinationDirectory,
                                          String nameJar) {
            File sourceJarFile = new File(nameJar);
            try {
            File unzipDestinationDirectory =
```

```
                        new File(destinationDirectory);
                  // открытие zip-архива для чтения
            JarFile jFile = new JarFile(sourceJarFile);
            Enumeration jarFileEntries = jFile.entries();
            while (jarFileEntries.hasMoreElements()) {
                  // извлечение текущей записи из архива
            JarEntry entry =
                  (JarEntry) jarFileEntries.nextElement();

            String entryname = entry.getName();
                  //entryname = entryname.substring(2);
            System.out.println("Extracting: " + entry);
                  destFile =
            new File(unzipDestinationDirectory, entryname);
                  // определение каталога
                  File destinationParent =
                              destFile.getParentFile();
                  // создание структуры каталогов
                        destinationParent.mkdirs();
                  // распаковывание записи, если она не каталог
                        if (!entry.isDirectory()) {
                              writeFile(jFile, entry);
                        }
                  }
                  jFile.close();
            } catch (IOException ioe) {
                  ioe.printStackTrace();
            }
      }
private void writeFile(JarFile jFile, JarEntry entry)
                              throws IOException {
            BufferedInputStream is =
                  new BufferedInputStream(
                        jFile .getInputStream(entry));
            int currentByte;
            byte data[] = new byte[BUFFER];
            // запись файла на диск
            BufferedOutputStream dest =
                  new BufferedOutputStream(
                  new FileOutputStream(destFile), BUFFER);

      while ((currentByte = is.read(data, 0, BUFFER)) > 0){
                  dest.write(data, 0, currentByte);
            }
            dest.flush();
            dest.close();
            is.close();
      }
```

```
    public static void main(String[] args) {
        System.out.println(
              "Извлечение данных из jar-архива");
        // расположение и имя архива
        String nameJar = "c:\\work\\example.jar";
        // куда файлы будут распакованы
        String destination = "c:\\temp\\";
        new UnPackJar().unpack(destination, nameJar);
    }
}
```

Задания к главе 9

Вариант A

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 Александра 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 Сергея Есенина подсчитать частоту повторяемости каждого слова из заданного списка и вывести эти слова в порядке возрастания частоты повторяемости.
7. В каждом слове сонета Вильяма Шекспира заменить первую букву слова на прописную.
8. Определить частоту повторяемости букв и слов в стихотворении Александра Пушкина.

Вариант B

Выполнить задания из варианта B главы 4, 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 9

Вопрос 9.1.

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 9.2.

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 9.3.

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 9.4.

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

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

Вопрос 9.5.

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

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

Глава 10
КОЛЛЕКЦИИ

Общие определения

Коллекции – это хранилища, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных типов (структур) данных, поддерживающих три основные операции:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: просмотреть элементы, подсчитать их количество и др.

Применение коллекций обусловливается возросшими объемами обрабатываемой информации. Когда счет используемых объектов идет на сотни тысяч, массивы не обеспечивают ни должной скорости, ни экономии ресурсов. Например, процессор UltraSPARC T1 тестировался на обработке информации для электронного магазина, содержащего около 40 тысяч товаров и 125 миллионов клиентов, сделавших 400 миллионов заказов.

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 в языке Java объединены в библиотеке классов **`java.util`** и представляют собой контейнеры для хранения и манипулирования объектами. До появления Java 2 эта библиотека содержала классы только для работы с простейшими 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. Скорость обработки коллекций повысилась по сравнению с предыдущей версией языка за счет отказа от их потокобезопасности. Поэтому если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распределенных приложений, следует использовать коллекции из Java 1.

Так как в коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, следует обезопасить коллекцию от появления ссылок на другие не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выполнения, что повышает трудозатраты на исправление и верификацию кода. Поэтому начиная с версии 5.0 коллекции стали типизированными.

Более удобным стал механизм работы с коллекциями, а именно:

- предварительное сообщение компилятору о типе ссылок, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции;
- отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты (тип **Object**) к требуемому типу.

Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Так как **Object** – суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых. Коллекции – это динамические массивы, связные списки, деревья, множества, хэш-таблицы, стеки, очереди.

Интерфейсы коллекций:

Map<K,V> – карта отображения вида "ключ-значение";

Collection<E> – вершина иерархии остальных коллекций;

List<E> – специализирует коллекции для обработки списков;

Set<E> – специализирует коллекции для обработки множеств, содержащих уникальные элементы.

Все классы коллекций реализуют также интерфейсы **Serializable**, **Cloneable** (кроме **WeakHashMap**). Кроме того, классы, реализующие интерфейсы **List<E>** и **Set<E>**, реализуют также интерфейс **Iterable<E>.**

В интерфейсе **Collection<E>** определены методы, которые работают на всех коллекциях:

boolean add(E obj) – добавляет **obj** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **obj** уже элемент коллекции;

boolean addAll(Collection<? extends E> 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<E> 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;

<T> T[] toArray(T a[]) – копирует элементы коллекции в массив объектов определенного типа.

Для работы с элементами коллекции применяются следующие интерфейсы:

Comparator<T> – для сравнения объектов;

Iterator<E>, **ListIterator<E>**, **Map.Entry<K,V>** – для перечисления и доступа к объектам коллекции.

Интерфейс **Iterator<E>** используется для построения объектов, которые обеспечивают доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом **iterator()**. Такой объект позволяет просматривать

содержимое коллекции последовательно, элемента за элементом. Позиции итератора располагаются в коллекции между элементами. В коллекции, состоящей из N элементов, существует N+1 позиций итератора.

Методы интерфейса **`Iterator<E>:`**

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

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

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

Интерфейс **`ListIterator<E>`** расширяет интерфейс **`Iterator<E>`** и предназначен в основном для работы со списками. Наличие методов **`E previous()`**, **`int previousIndex()`** и **`boolean hasPrevious()`** обеспечивает обратную навигацию по списку. Метод **`int nextIndex()`** возвращает номер следующего итератора. Метод **`void add(E obj)`** позволяет вставлять элемент в список текущей позиции. Вызов метода **`void set(E obj)`** 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 **`K getKey()`** и **`V getValue()`** соответственно. Вызов метода **`V setValue(V value)`** заменяет значение, ассоциированное с текущим ключом.

Списки

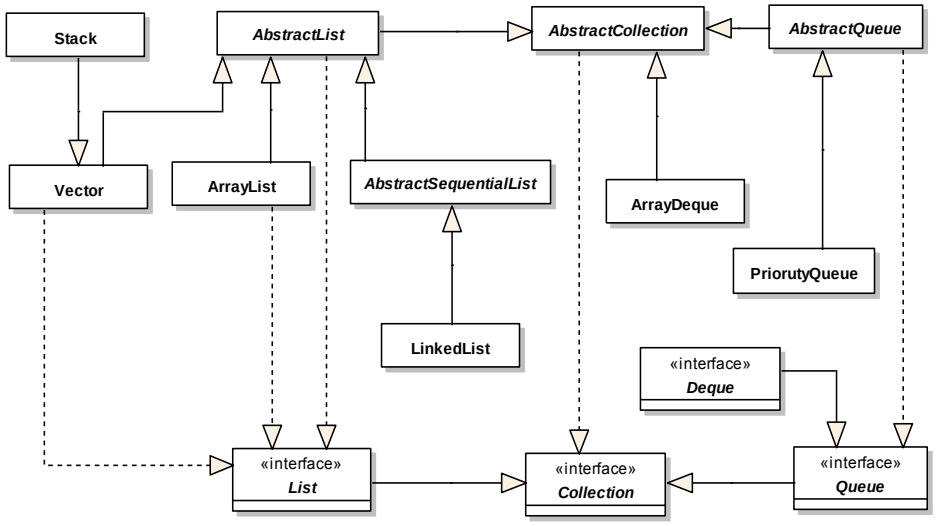


Рис. 10.1. Иерархия наследования списков

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

ArrayList()
ArrayList(Collection <? extends E> c)
ArrayList(int capacity)

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

void add(int index, E element) – вставляет **element** в позицию, указанную в **index**;

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

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

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

E remove(int index) – удаляет объект из позиции **index**;

E set(int index, E element) – заменяет объект в позиции **index**, возвращает при этом удаляемый элемент;

List<E> subList(int fromIndex, int toIndex) – извлекает часть коллекции в указанных границах.

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

```
/* пример # 1 : создание параметризованной коллекции : DemoGeneric.java */
package chapt10;
import java.util.*;

public class DemoGeneric {
     public static void main(String args[]) {
          ArrayList<String> list = new ArrayList<String>();
          // ArrayList<int> b = new ArrayList<int>(); // ошибка компиляции
          list.add("Java");
          list.add("Fortress");
          String res = list.get(0);/* компилятор "знает"
                                      тип значения */
          // list.add(new StringBuilder("C#") ); // ошибка компиляции
          // компилятор не позволит добавить "посторонний" тип
          System.out.print(list);
     }
}
```

В результате будет выведено:

[Java, Fortress]

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 **list**. При этом параметром коллекции может быть только объектный тип.

Следует отметить, что указывать тип следует при создании ссылки, иначе будет позволено добавлять объекты всех типов.

```
/* пример # 2 : некорректная коллекция : UncheckCheck.java */
package chapt10;
import java.util.*;

public class UncheckCheck {
     public static void main(String args[]) {
          ArrayList list = new ArrayList();
          list.add(71);
          list.add(new Boolean("TruE"));
          list.add("Java 1.6.0");

          // требуется приведение типов
            int i = (Integer)list.get(0);
            boolean b = (Boolean)list.get(1);
            String str = (String)list.get(2);
          for (Object ob : list)
            System.out.println("list " + ob);

          ArrayList<Integer> s = new ArrayList<Integer>();
          s.add(71);
          s.add(92);
          // s.add("101");// ошибка компиляции: s параметризован
          for (Integer ob : s)
             System.out.print("int " + ob);
     }
}
```

В результате будет выведено:

```
list 71
list true
list Java 1.6.0
int 71
int 92
```

Чтобы параметризация коллекции была полной, необходимо указывать параметр и при объявлении ссылки, и при создании объекта.

Объект типа **Iterator** может использоваться для последовательного перебора элементов коллекции. Ниже приведен пример заполнения списка псевдослучайными числами, подсчет с помощью итератора количества положительных и удаление из списка неположительных значений.

```
/* пример # 3 : работа со списком : DemoIterator.java */
package chapt10;
import java.util.*;
```

```
public class DemoIterator {
      public static void main(String[] args) {
            ArrayList<Double> c =
                  new ArrayList<Double>(7);
            for(int i = 0 ;i < 10; i++) {
                  double z = new Random().nextGaussian();
                  c.add(z);//заполнение списка
            }
            //вывод списка на консоль
            for(Double d: c) {
            System.out.printf("%.2f ",d);
            }
            int positiveNum = 0;
            int size = c.size();//определение размера коллекции

            //извлечение итератора
            Iterator<Double> it = c.iterator();

      //проверка существования следующего элемента
      while(it.hasNext()) {
      //извлечение текущего элемента и переход к следующему
       if (it.next() > 0) positiveNum++;
       else it.remove();//удаление неположительного элемента
                        }
    System.out.printf("%nКоличество положительных: %d ",
                                          positiveNum);
    System.out.printf("%nКоличество неположительных: %d ",
                              size - positiveNum);
    System.out.println("\nПоложительная коллекция");
            for(Double d : c) {
                  System.out.printf("%.2f ",d);
                  }
      }
}
```

В результате на консоль будет выведено:

```
0,69 0,33 0,51 -1,24 0,07 0,46 0,56 1,26 -0,84 -0,53
Количество положительных: 7
Количество отрицательных: 3
Положительная коллекция
0,69 0,33 0,51 0,07 0,46 0,56 1,26
```

Для доступа к элементам списка может также использоваться интерфейс **ListIterator<E>**, который позволяет получить доступ сразу в необходимую программисту позицию списка. Такой способ доступа возможен только для списков.

```
/* пример # 4 : замена, удаление и поиск элементов : DemoListMethods.java */
package chapt10;
import java.util.*;
```

```
public class DemoListMethods {
      public static void main(String[] args) {
            ArrayList<Character> a =
                  new ArrayList<Character>(5);
            System.out.println("коллекция пуста: "
                                          + a.isEmpty());
            for (char c = 'a'; c < 'h'; ++c) {
                  a.add(c);
            }
            char ch = 'a';
            a.add(6, ch); // заменить 6 на >=8 – ошибка выполнения
            System.out.println(a);
      ListIterator<Character> it;// параметризация обязательна
      it= a.listIterator(2);// извлечение итератора списка в позицию
      System.out.println("добавление элемента в позицию "
                                    + it.nextIndex());
      it.add('X');// добавление элемента без замены в позицию итератора
      System.out.println(a);
            // сравнить методы
      int index = a.lastIndexOf(ch); // a.indexOf(ch);
      a.set(index, 'W'); // замена элемента без итератора
      System.out.println(a + "после замены элемента");
            if (a.contains(ch)) {
                  a.remove(a.indexOf(ch));
                  }
            System.out.println(a + "удален элемент " + ch);
      }
}
```

В результате будет выведено:

```
коллекция пуста: true
[a, b, c, d, e, f, a, g]
добавление элемента в позицию 2
[a, b, X, c, d, e, f, a, g]
[a, b, X, c, d, e, f, W, g]после замены элемента
[b, X, c, d, e, f, W, g]удален элемент a
```

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

Класс **LinkedList<E>** реализует интерфейс **Queue<E>**, что позволяет предположить, что такому списку легко придать свойства очереди. К тому же специализированные методы интерфейса **Queue<E>** по манипуляции первым и

последним элементами такого списка **E element()**, **boolean offer(E o)**, **E peek()**, **E poll()**, **E remove()** работают немного быстрее, чем соответствующие методы класса **LinkedList<E>**.

Методы интерфейса **Queue<E>**:

E element() – возвращает, но не удаляет головной элемент очереди;

boolean offer(E o) – вставляет элемент в очередь, если возможно;

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.

```
/* пример # 5 : добавление и удаление элементов : DemoLinkedList.java */
package chapt10;
import java.util.*;

public class DemoLinkedList {
     public static void main(String[] args){
          LinkedList<Number> a = new LinkedList<Number>();
          for(int i = 10; i <= 15; i++)
             a.add(i);
          for(int i = 16; i <= 20; i++)
             a.add(new Float(i));
          ListIterator<Number> list = a.listIterator(10);
          System.out.println("\n"+ list.nextIndex()
                                        + "-й индекс");
          list.next(); // важно!
          System.out.println(list.nextIndex()
                                        + "-й индекс");
          list.remove(); // удаление элемента с текущим индексом
          while(list.hasPrevious())
             System.out.print(list.previous()+"  "); /*вывод
                                          в обратном порядке*/
          // демонстрация работы методов
          a.removeFirst();
          a.offer(71); // добавление элемента в конец списка
          a.poll(); // удаление нулевого элемента из списка
          a.remove(); // удаление нулевого элемента из списка
          a.remove(1); // удаление первого элемента из списка
          System.out.println("\n" + a);

          Queue<Number> q = a; // список в очередь
          for (Number i : q) // вывод элементов
             System.out.print(i + " ");
          System.out.println(" :size= " + q.size());

          // удаление пяти элементов
```

```
          for (int i = 0; i < 5; i++) {
               Number res = q.poll();
          }
          System.out.print("size= " + q.size());
     }
}
```

В результате будет выведено:

```
10-й индекс
11-й индекс
19.0  18.0  17.0  16.0  15  14  13  12  11  10
[13, 15, 16.0, 17.0, 18.0, 19.0, 71]
13 15 16.0 17.0 18.0 19.0 71  :size= 7
size= 2
```

При реализации интерфейса **Comparator<T>** существует возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа. Для этого необходимо реализовать метод **int compare(T ob1, T ob2)**, принимающий в качестве параметров два объекта для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки. Этот метод автоматически вызывается методом **public static <T> void sort(List<T> list, Comparator<? super T> c)** класса **Collections**, в качестве первого параметра принимающий коллекцию, в качестве второго – объект-comparator, из которого извлекается правило сортировки.

```
/* пример # 6 : авторская сортировка списка: UniqSortMark.java */
package chapt10;
import java.util.Comparator;

public class Student implements Comparator<Student> {
     private int idStudent;
     private float meanMark;

     public Student(float m, int id) {
          meanMark = m;
          idStudent = id;
     }
     public Student() {
     }
     public float getMark() {
          return meanMark;
     }
     public int getIdStudent() {
          return idStudent;
     }
     // правило сортировки
     public int compare(Student one, Student two) {
          return
          (int)(Math.ceil(two.getMark() - one.getMark()));
     }
}
```

```
package chapt10;
import java.util.*;

public class UniqSortMark {
     public static void main(String[] args) {
          ArrayList<Student> p = new ArrayList<Student>();
          p.add(new Student(3.9f, 52201));
          p.add(new Student(3.65f, 52214));
          p.add(new Student(3.71f, 52251));
          p.add(new Student(3.02f, 52277));
          p.add(new Student(3.81f, 52292));
          p.add(new Student(9.55f, 52271));
          // сортировка списка объектов
          try {
       Collections.sort(p, Student.class.newInstance());
          } catch (InstantiationException e1) {
             //невозможно создать объект класса
             e1.printStackTrace();
          } catch (IllegalAccessException e2) {
             e2.printStackTrace();
          }
          for (Student ob : p)
               System.out.printf("%.2f ", ob.getMark());
     }
}
```

В результате будет выведено:

```
9,55 3,90 3,81 3,71 3,65 3,02
```

Метод **boolean equals(Object obj)** интерфейса **Comparator<T>**, который обязан выполнять свой контракт, возвращает **true** только в случае если соответствующий метод **compare()** возвращает **0**.

Для создания возможности сортировки по другому полю **id** класса **Student** следует создать новый класс, реализующий **Comparator** по новым правилам.

```
/* пример # 7 : другое  правило сортировки: StudentId.java */
package chapt10;

public class StudentId implements Comparator<Student> {
     public int compare(Student one, Student two) {
          return two.getIdStudent() - one.getIdStudent();
     }
   }
```

При необходимости сортировки по полю **id** в качестве второго параметра следует объект класса **StudentId**:

```
Collections.sort(p, StudentId.class.newInstance());
```

Параметризация коллекций позволяет разрабатывать безопасные алгоритмы, создание которых потребовало бы несколько больших затрат в предыдущих версиях языка.

Deque

Интерфейс **Deque** определяет «двунаправленную» очередь и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди. Методы обеспечивают удаление, вставку и обработку элементов. Каждый из этих методов существует в двух формах. Одни методы создают исключительную ситуацию в случае неудачного завершения, другие возвращают какое-либо из значений (**null** или **false** в зависимости от типа операции). Вторая форма добавления элементов в очередь сделана специально для реализаций **Deque**, имеющих ограничение по размеру. В большинстве реализаций операции добавления заканчиваются успешно.

В следующим примере реализована работа с интерфейсом **Deque**. Методы **addFirst()**, **addLast()** вставляют элементы в начало и в конец очереди соответственно. Метод **add()** унаследован от интерфейса **Queue** и абсолютно аналогичен методу **addLast()** интерфейса **Deque.**

```
/* пример # 8 : демонстрация Deque : DequeRunner.java */
package chapt10;
import java.util.*;

public class DequeRunner {
      public static void printDeque(Deque<?> d){
                  for (Object de : d)
                        System.out.println(de + "; ");
            }
      public static void main(String[] args) {
            Deque<String> deque = new ArrayDeque<String>();
                  deque.add(new String("5"));
                  deque.addFirst("A");
                  //deque.addLast(new Integer(5));//ошибка компиляции
                  System.out.println(deque.peek());
                  System.out.println("Before:");
                  printDeque(deque);
                  deque.pollFirst();
                  System.out.println(deque.remove(5));
                  System.out.println("After:");
                  printDeque(deque);
      }
}
```

В результате на консоль будет выведено:

```
A
Before:
A;
5;
false
After:
5;
```

Множества

Интерфейс **Set<E>** объявляет поведение коллекции, не допускающей дублирования элементов. Интерфейс **SortedSet<E>** наследует **Set<E>** и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса. Интерфейс **NavigableSet** существенно облегчает поиск элементов.

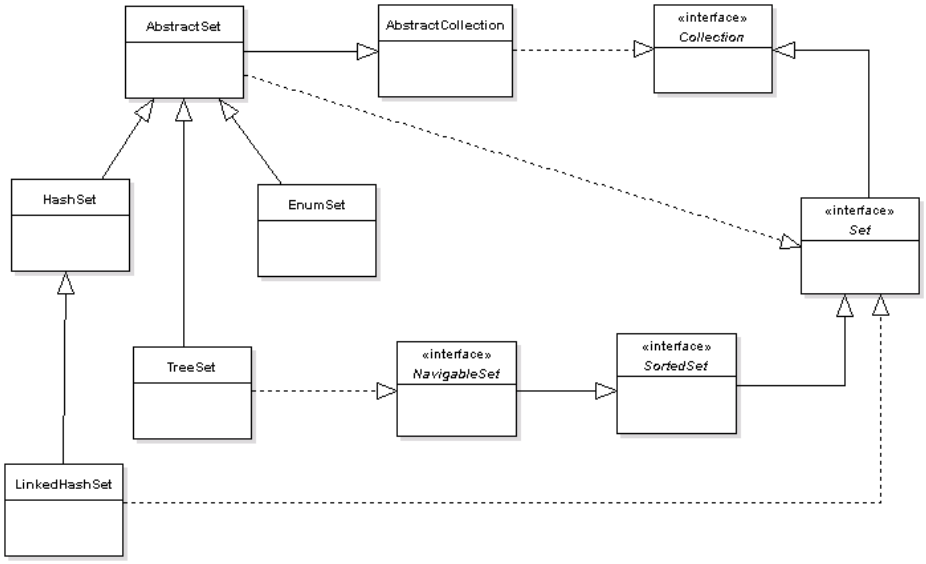


Рис. 10.2. Иерархия наследования множеств

Класс **HashSet<E>** наследуется от абстрактного суперкласса **AbstractSet<E>** и реализует интерфейс **Set<E>**, 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 большим количеством элементов. Все элементы такого множества упорядочены посредством хэш-таблицы, в которой хранятся хэш-коды элементов.

Конструкторы класса:

HashSet()

HashSet(Collection <? extends E> c)

HashSet(int capacity)

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

/ пример # 9 : использование множества для вывода всех уникальных слов из файла : DemoHashSet.java */*

```
package chapt10;
import java.util.*;
import java.io.*;
```

```
public class DemoHashSet {
     public static void main(String[] args) {
     HashSet<String> words = new HashSet<String>(100);
     // использовать коллекции LinkedHashSet или TreeSet
     long callTime = System.nanoTime();
     try {
            BufferedReader in =
                  new BufferedReader(
                        new FileReader("c://pushkin.txt"));
              String line = "";
       while ((line = in.readLine()) != null) {
         StringTokenizer tokenizer =
               new StringTokenizer(line,
                     " (){}[]<>#*!?.,:;-\'\"/");
               while (tokenizer.hasMoreTokens()) {
                    String word = tokenizer.nextToken();
                    words.add(word.toLowerCase());
               }
        }
     } catch (IOException e) {
         System.err.println(e);
     }
     Iterator<String> it = words.iterator();
     while (it.hasNext())
         System.out.println(it.next());
     long totalTime = System.nanoTime()- callTime;
     System.out.println("различных слов: " + words.size()
                  + ", " + totalTime + " наносекунд");
     }
}
```

Класс **TreeSet<E>** для хранения объектов использует бинарное дерево. При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что все добавляемые элементы реализуют интерфейсы **Comparator** и **Comparable**. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Конструкторы класса:

TreeSet()
TreeSet(Collection <? extends E> c)
TreeSet(Comparator <? super E> c)
TreeSet(SortedSet <E> s)

Класс **TreeSet<E>** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **E first()** и **E last()**. Методы **SortedSet<E> subSet(E from, E to)**, **SortedSet<E> tailSet(E from)** и **SortedSet<E> headSet(E to)** предназначены для извлечения определенной части множества. Метод **Comparator <? super E> comparator()** возвращает объект

Comparator, используемый для сортировки объектов множества или **null**, если выполняется обычная сортировка.

/ пример # 10: создание множества из списка и его методы: DemoTreeSet.java */*

```
package chapt10;
import java.util.*;

public class DemoTreeSet {
     public static void main(String[] args) {
          ArrayList<String> c = new ArrayList<String>();
          boolean b;
          for (int i = 0; i < 6; i++)
               c.add((int) (Math.random() * 71) + "Y ");
          System.out.println(c + "список");
          TreeSet<String> set = new TreeSet<String>(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");
          System.out.println(set.comparator()); //null !!!

          // извлечение наибольшего и наименьшего элементов
          System.out.println(set.last() + " "
                    + set.first());
     }
}
```

В результате может быть выведено:

```
[44Y , 56Y , 49Y , 26Y , 49Y , 2Y ]список
[2Y , 26Y , 44Y , 49Y , 56Y ]множество
[2Y , 26Y , 44Y , 49Y , 5 Element, 56Y ]add
null
56Y  2Y
```

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. Для множества, состоящего из обычных строк, используется по умолчанию правило обычной лексикографической сортировки, поэтому метод **comparator()** возвращает **null**.

Если попытаться заменить тип **String** на **StringBuilder** или **StringBuffer**, то создать множество **TreeSet** так просто не удастся. Решением такой задачи будет создание нового класса с полем типа **StringBuilder** и реализацией интерфейса **Comparable<T>** вида:

/ пример # 11 :пользовательский класс, объект которого может быть добавлен в множество TreeSet : Message.java */*

```
package chapt10;
import java.util.*;
```

```
public class Message implements Comparable<Message> {
      private StringBuilder str;
      private int idSender;

      public Message(StringBuilder s, int id) {
            super();
            this.str = s;
            idSender = id;
      }
      public String getStr() {
            return str.toString();
      }
      public int getId() {
            return idSender;
      }
      public int compareTo(Message a0) {
            return (idSender - a0.getId());
      }
}
```

Предлагаемое решение универсально для любых пользовательских типов.

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от абстрактного класса **AbstractSet**. Специально реализован для работы с типами **enum**. Все элементы такой коллекции должны принадлежать единственному типу **enum**, определенному явно или неявно. Внутренне множество представимо в виде вектора битов, обычно единственного **long**. Множества нумераторов поддерживают перебор по диапазону из нумераторов. Скорость выполнения операций над таким множеством очень высока, даже если в ней участвует большое количество элементов.

Создать объект этого класса можно только с помощью статических методов. Метод **EnumSet<E> noneOf(Class<E> elemType)** создает пустое множество нумерованных констант с указанным типом элемента, метод **allOf(Class<E> elementType)** создает множество нумерованных констант, содержащее все элементы указанного типа. Метод **of(E first, E... rest)** создает множество, первоначально содержащее указанные элементы. С помощью метода **complementOf(EnumSet<E> s)** создается множество, содержащее все элементы, которые отсутствуют в указанном множестве. Метод **range(E from, E to)** создает множество из элементов, содержащихся в диапазоне, определенном двумя элементами. При передаче вышеуказанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**.

```
/* пример # 12 : использование множества enum-типов : UseEnumSet.java */
package chapt10;
import java.util.EnumSet;

enum Faculty{ FFSM, MMF, FPMI, FMO, GEO }

public class UseEnumSet {
```

```
    public static void main(String[] args) {
    /*множество set1 содержит элементы типа enum из интервала,
     определенного двумя элементами*/
          EnumSet <Faculty> set1 =
                EnumSet.range(Faculty.MMF, Faculty.FMO);
    /*множество set2 будет содержать все элементы, не содержащиеся
    в множестве set1*/
          EnumSet <Faculty> set2 =
                EnumSet.complementOf(set1);
          System.out.println(set1);
          System.out.println(set2);
    }
}
```

В результате будет выведено:

```
[MMF, FPMI, FMO]
[FFSM, GEO]
```

В следующем примере показано использование интерфейса **NavigableSet**. Метод **first()** возвращает первый элемент из множества. Метод **subSet(E fromElement, E toElement)** возвращает список элементов, находящихся между **fromElement** и **toElement**, причем последний не включается. Методы **headSet(E element)** и **tailSet(E element, boolean inclusive)** возвращают то множество элементов, которое меньше либо больше **element** соответственно. Если **inclusive** равно **true**, то элемент включается в найденное множество и не включается в противном случае.

```
/* пример # 13 : использование множества NavigableSet: NavigableSetTest.java */
package chapt10;
import java.util.*;

public class NavigableSetTest {
  public static void main(String[] args) {
     HashSet<String> city = new HashSet<String>();
     city.add("Minsk");
     city.add("Moscow");
     city.add("Polotsk");
     city.add("Brest");
     NavigableSet<String> ns = new TreeSet<String>(city);
     System.out.println("All: " + ns);
     System.out.println("First: " + ns.first());
     System.out.println("Between Minsk and Polotsk: "
                        + ns.subSet("Minsk","Polotsk"));
     System.out.println("Before Minsk: "
                              + ns.headSet("Minsk"));
     System.out.println("After Minsk: "
                              + ns.tailSet("Minsk",false));
  }
}
```

В результате на консоль будет выведено:

```
All: [Brest, Minsk, Moscow, Polotsk]
First: Brest
Between Minsk and Polotsk: [Minsk, Moscow]
Before Minsk: [Brest]
After Minsk: [Moscow, Polotsk]
```

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. Уникальность объектов-ключей должна обеспечиваться переопределением методов **`hashCode()`** и **`equals()`** пользовательским классом. Если элемент с указанным ключом отсутствует в карте, то возвращается значение **`null`**.

Классы карт отображений:

`AbstractMap<K,V>` – реализует интерфейс **`Map<K,V>`**;

`HashMap<K,V>` – расширяет **`AbstractMap<K,V>`**, используя хэш-таблицу, в которой ключи отсортированы относительно значений их хэш-кодов;

`TreeMap<K,V>` – расширяет **`AbstractMap<K,V>`**, используя дерево, где ключи расположены в виде дерева поиска в строгом порядке.

`WeakHashMap<K,V>` позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения.

`LinkedHashMap<K,V>` запоминает порядок добавления объектов в карту и образует при этом дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

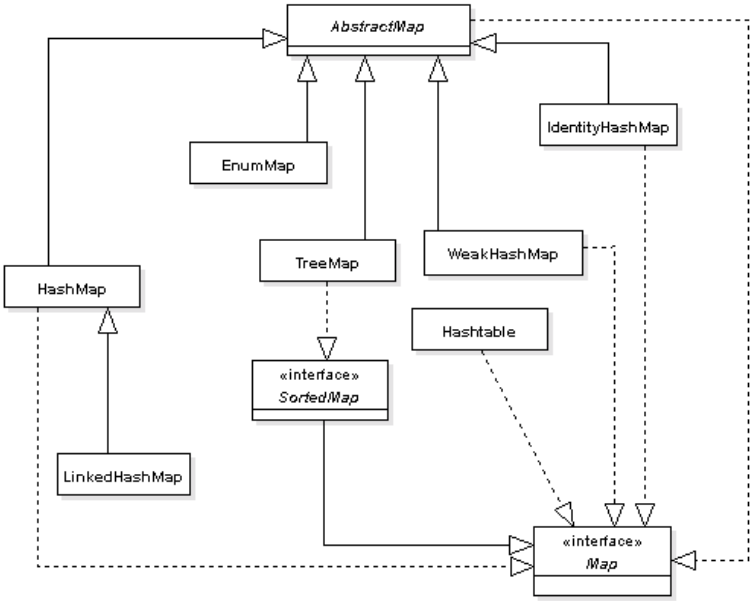


Рис. 10.3. Иерархия наследования карт

Для класса **IdentityHashMap<K,V>** хэш-коды объектов-ключей вычисляются методом **System.identityHashCode()** по адресу объекта в памяти, в отличие от обычного значения **hashCode()**, вычисляемого сугубо по содержимому самого объекта.

Интерфейсы карт:

Map<K,V> – отображает уникальные ключи и значения;

Map.Entry<K,V> – описывает пару "ключ-значение";

SortedMap<K,V> – содержит отсортированные ключи и значения;

NavigableMap<K,V> – добавляет новые возможности поиска по ключу.

Интерфейс **Map<K,V>** содержит следующие методы:

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

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

boolean containsValue(Object value) – возвращает **true**, если вызывающая карта содержит **value** как значение;

Set<Map.Entry<K,V>> entrySet() – возвращает множество, содержащее значения карты;

Set<K> keySet() – возвращает множество ключей;

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

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

void putAll(Map <? extends K, ? extends V> t) – помещает коллекцию **t** в вызывающую карту;

V remove(Object key) – удаляет пару "ключ-значение" по ключу **key**;

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

Интерфейс **Map.Entry<K,V>** содержит следующие методы:

K getKey() – возвращает ключ текущего входа;

V getValue() – возвращает значение текущего входа;

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

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

```
/* пример # 14 : создание хэш-карты и замена элемента по ключу:
DemoHashMap.java */
package chapt10;
import java.util.*;

public class DemoHashMap {
     public static void main(String[] args){
          HashMap<Integer, String> hm =
                    new HashMap<Integer, String>(5);
          for (int i = 11; i < 15; i++)
                    hm.put(i, i + "EL");
```

```
        System.out.println(hm);
        hm.put(12, "14EL");
        System.out.println(hm + "с заменой элемента");
        String a = hm.get(12);
        System.out.println(a + " - найден по ключу '12'");
        /* вывод хэш-таблицы с помощью методов интерфейса
         Map.Entry<K,V> */
        Set<Map.Entry<Integer, String>> setvalue =
                                        hm.entrySet();
        System.out.println(setvalue);
        Iterator<Map.Entry<Integer, String>> i =
                        setvalue.iterator();
        while (i.hasNext()) {
             Map.Entry<Integer, String> me = i.next();
             System.out.print(me.getKey()+" : ");
             System.out.println(me.getValue());
        }
    }
}
```

```
{13=13EL, 14=14EL, 12=12EL, 11=11EL}
{13=13EL, 14=14EL, 12=14EL, 11=11EL}с заменой элемента
14EL - найден по ключу '12'
[13=13EL, 14=14EL, 12=14EL, 11=11EL]
13 : 13EL
14 : 14EL
12 : 14EL
11 : 11EL
```

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

```
/* пример # 15 : применение коллекций при проверке доступа в систему :
DemoSecurity.java */
package chapt10;
import java.util.*;

public class DemoSecurity {
        public static void main(String[] args) {
        CheckRight.startUsing(2041, "Артем");
        CheckRight.startUsing(2420, "Ярослав");
        /*
         *добавление еще одного пользователя и
         * проверка его на возможность доступа
         */
        CheckRight.startUsing(2437, "Анастасия");
        CheckRight.startUsing(2041, "Артем");
    }
}
```

```
/* пример # 16 : класс проверки доступа в систему: CheckRight.java */
package chapt10;
import java.util.*;

public class CheckRight {
     private static HashMap<Integer, String> map =
                         new HashMap<Integer, String> ();

     public static void startUsing(int 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(int 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 **EnumMap<K extends Enum<K>, V>** в качестве ключа может принимать только объекты, принадлежащие одному типу **enum**, который должен быть определен при создании коллекции. Специально организован для обеспечения максимальной скорости доступа к элементам коллекции.

```
/* пример # 17 : пример работы с классом EnumMap: UseEnumMap.java */
package chapt10;
import java.util.EnumMap;

enum User {
      STUDENT, TUTOR, INSTRUCTOR, DEAN
}
class UserPriority {
      private int priority;

      public UserPriority(User k) {
            switch (k) {
```

```
            case STUDENT:
                  priority = 1; break;
            case TUTOR:
                  priority = 3; break;
            case INSTRUCTOR:
                  priority = 7; break;
            case DEAN:
                  priority = 10; break;
            default:
                  priority = 0;
            }
      }
      public int getPriority() {
            return priority;
      }
}
public class UseEnumMap {
      public static void main(String[] args) {
            EnumMap<User, UserPriority> faculty =
new EnumMap<User, UserPriority> (User.class);
            for (User user : User.values()) {
                  faculty.put(user,
                        new UserPriority(user));
            }
            for (User user : User.values()) {
                  System.out.println(user.name()
                        + "-> Priority:" +
((UserPriority) faculty.get(user)).getPriority());
            }
      }
}
```

В результате будет выведено:

```
STUDENT-> Priority:1
TUTOR-> Priority:3
INSTRUCTOR-> Priority:7
DEAN-> Priority:10
```

Унаследованные коллекции

В ряде распределенных приложений, например с использованием сервлетов, до сих пор применяются коллекции, более медленные в обработке, но при этом потокобезопасные, существовавшие в языке Java с момента его создания, а именно карта **`Hashtable<K,V>`**, список **`Vector<E>`** и перечисление **`Enumeration<E>`**. Все они также были параметризованы, но сохранили все свои особенности.

Класс **`Hashtable<K,V>`** реализует интерфейс **`Map`**, но обладает также несколькими интересными методами:

`Enumeration<V> elements()` – возвращает перечисление (аналог итератора) для значений карты;

Enumeration<K> keys() – возвращает перечисление для ключей карты.

```
/* пример # 18 : создание хэш-таблицы и поиск элемента по ключу:
  HashTableDemo.java */
package chapt10;
import java.util.*;
import java.io.*;

public class HashTableDemo {
     public static void main(String[] args) {
          Hashtable<Integer, Double> ht =
                         new Hashtable<Integer, Double>();
          for (int i = 0; i < 5; i++)
               ht.put(i, Math.atan(i));
          Enumeration<Integer> ek = ht.keys();
          int key;
          while (ek.hasMoreElements()) {
               key = ek.nextElement();
               System.out.printf("%4d  ", key);
          }
             System.out.println("");
          Enumeration<Double> ev = ht.elements();
          double value;
          while (ev.hasMoreElements()) {
               value = ev.nextElement();
               System.out.printf("%.2f  ", value);
          }
     }
}
```

В результате в консоль будет выведено:

```
   4     3     2     1     0
1,33  1,25  1,11  0,79  0,00
```

Принципы работы с коллекциями, в отличие от их структуры, со сменой версий языка существенно не изменились.

Класс Collections

Класс **Collections** содержит большое количество статических методов, предназначенных для манипулирования коллекциями.

С применением предыдущих версий языка было разработано множество коллекций, 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 # 19 : проверяемая коллекция: SafeCollection.java  */
package chapt10;
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(7.0); // ошибка времени выполнения
     }
}
```

В этот же класс добавлен целый ряд методов, специализированных для проверки конкретных типов коллекций, а именно: **checkedList()**, **checkedSortedMap()**, **checkedMap()**, **checkedSortedSet()**, **checkedSet()**, а также:

<T> boolean addAll(Collection<? super T> c, T... a) – добавляет в параметризованную коллекцию соответствующие параметризации элементы;

<T> void copy(List<? super T> dest, List<? extends T> src) – копирует все элементы из одного списка в другой;

boolean disjoint(Collection<?> c1, Collection<?> c2) – возвращает **true**, если коллекции не содержат одинаковых элементов;

<T> List <T> emptyList(), **<K,V> Map <K,V> emptyMap()**, **<T> Set <T> emptySet()** – возвращают пустой список, карту отображения и множество соответственно;

<T> void fill(List<? super T> list, T obj) – заполняет список заданным элементом ;

int frequency(Collection<?> c, Object o) – возвращает количество вхождений в коллекцию заданного элемента;

<T extends Object & Comparable <? super T>> T max(Collection<? extends T> coll),

<T extends Object & Comparable <? super T>> T min(Collection<? extends T> coll) – возвращают минимальный и максимальный элемент соответственно;

<T> T max(Collection <? extends T> coll, Comparator<? super T> comp),

<T> T min(Collection<? extends T> coll, Comparator<? super T> comp) – возвращают минимальный и максимальный элемент соответственно, используя **Comparator** для сравнения;

<T> List <T> nCopies(int n, T o) – возвращает список из **n** заданных элементов;

<T> boolean replaceAll(List<T> list, T oldVal, T newVal) – заменяет все заданные элементы новыми;

void reverse(List<?> list) – “переворачивает” список;

void rotate(List<?> list, int distance) – сдвигает список циклически на заданное число элементов;

void shuffle(List<?> list) – перетасовывает элементы списка;

<T> Set <T> singleton(T o), **singletonList(T o)**, **singletonMap(K key, V value)** – создают множество, список и карту отображения, состоящие из одного элемента;

<T extends Comparable<? super T>> void sort(List<T> list),

<T> void sort(List<T> list, Comparator<? super T> c) – сортировка списка, естественным порядком и используя **Comparator** соответственно;

void swap(List<?> list, int i, int j) – меняет местами элементы списка стоящие на заданных позициях.

/ пример # 20 : методы класса Collections: CollectionsDemo.java: MyComparator.java */*

```
package chapt10;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsDemo {
      public static void main(String[] args) {
            MyComparator<Integer> comp =
                  new MyComparator<Integer>();
            ArrayList<Integer> list =
                  new ArrayList<Integer>();

            Collections.addAll(list, 1, 2, 3, 4, 5);
            Collections.shuffle(list);
            print(list);
            Collections.sort(list, comp);
            print(list);
            Collections.reverse(list);
            print(list);
            Collections.rotate(list, 3);
            print(list);
            System.out.println("min: "
            + Collections.min(list, comp));
            System.out.println("max: "
            + Collections.max(list, comp));

            List<Integer> singl =
                  Collections.singletonList(71);
            print(singl);
            //singl.add(21);//ошибка времени выполнения
      }
      static void print(List<Integer> c) {
```

```
            for (int i : c)
                  System.out.print(i + " ");
            System.out.println();
      }
}
package chapt10;
import java.util.Comparator;

public class MyComparator<T> implements Comparator<Integer>
{
      public int compare(Integer n, Integer m) {
            return m.intValue() - n.intValue();
      }
}
```

В результате будет выведено:

```
4 3 5 1 2
5 4 3 2 1
1 2 3 4 5
3 4 5 1 2
min: 5
max: 1
71
```

Класс Arrays

В пакете **java.util** находится класс **Arrays**, который содержит методы манипулирования содержимым массива, а именно для поиска, заполнения, сравнения, преобразования в коллекцию и прочие:

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;

static <T> T[] copyOf(T[] original, int newLength) – заполняет массив определенной длины, отбрасывая элементы или заполняя **null** при необходимости;

static <T> T[] copyOfRange(T[] original, int from, int to) – копирует заданную область массива в новый массив;

<T> List<T> asList(T... a) – метод, копирующий элементы массива в объект типа **List<T>**.

В качестве простого примера применения указанных методов можно привести следующий код.

```
/* пример # 21 : методы класса Arrays : ArraysEqualDemo.java */
package chapt10;
import java.util.*;
```

```
public class ArraysEqualDemo {
    public static void main(String[] args) {
        char m1[] = new char[3];
        char 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'));
           Integer arr[] = {35, 71, 92};
           //вывод массива объектов в строку
           System.out.println(Arrays.deepToString(arr));
           //вычисление хэш-кода исходя и значений элементов
           System.out.println(Arrays.deepHashCode(arr));
           Integer arr2[] = {35, 71, 92};
           //сравнение массивов по седержимому
     System.out.println(Arrays.deepEquals(arr, arr2));
           char m3[] = new char[5];
           // копирование массива
           m3 = Arrays.copyOf(m1, 5);
           System.out.print("массив m3:");
           for (i = 0; i < 5; i++)
                 System.out.print(" " + m3[i]);
    }
}
```

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
[35, 71, 92]
65719
true
массив m3: b c z □ □
```

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. Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.
7. Задать два стека, поменять информацию местами.
8. Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.
9. 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.
10. 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.
11. Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте **`HashMap`**.
12. Умножить два многочлена заданной степени, если коэффициенты многочленов хранятся в различных списках.
13. Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные – в начало этого списка.
14. Ввести строки из файла, записать в список **`ArrayList`**. Выполнить сортировку строк, используя метод **`sort()`** из класса **`Collections`**.
15. Задана строка, состоящая из символов '(', ')', '[', ']', '{', '}'. Проверить правильность расстановки скобок. Использовать стек.
16. Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс **`HashSet`**.
17. Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличающиеся регистром букв, считать различными. Использовать класс **`HashMap`**.

Вариант В

1. В кругу стоят N человек, пронумерованных от 1 до N. При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из

программ должна использовать класс **`ArrayList`**, а вторая – **`LinkedList`**. Какая из двух программ работает быстрее? Почему?

2. Задан список целых чисел и число X. Не используя вспомогательных объектов и не изменяя размера списка, переставить элементы списка так, чтобы сначала шли числа, не превосходящие X, а затем числа, большие X.
3. Написать программу, осуществляющую сжатие английского текста. Построить для каждого слова в тексте оптимальный префиксный код по алгоритму Хаффмена. Использовать класс **`PriorityQueue`**.
4. Реализовать класс Graph, представляющий собой неориентированный граф. В конструкторе класса передается количество вершин в графе. Методы должны поддерживать быстрое добавление и удаление ребер.
5. На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций:
   - добавление/удаление числа;
   - поиск числа, наиболее близкого к заданному (т.е. модуль разницы минимален).
6. Реализовать класс, моделирующий работу N-местной автостоянки. Машина подъезжает к определенному месту и едет вправо, пока не встретится свободное место. Класс должен поддерживать методы, обслуживающие приезд и отъезд машины.
7. 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.
8. 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.
9. 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.
10. 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.
11. Один из способов шифрования данных, называемый «двойным шифрованием», заключается в том, что исходные данные при помощи некоторого преобразования последовательно шифруются на некоторые два ключа K1 и K2. Разработать и реализовать эффективный алгоритм, позволяющий находить ключи K1 и K2 по исходной строке и ее зашифрованному варианту. Проверить, оказался ли разработанный способ дей-

ствительно эффективным, протестировав программу для случая, когда оба ключа K1 и K2 являются 20-битными (время ее работы не должно превосходить одной минуты).

12. На плоскости задано N точек. Вывести в файл описания всех прямых, которые проходят более чем через одну точку из заданных. Для каждой прямой указать, через сколько точек она проходит. Использовать класс **`HashMap`**.
13. На клетчатой бумаге нарисован круг. Вывести в файл описания всех клеток, целиком лежащих внутри круга, в порядке возрастания расстояния от клетки до центра круга. Использовать класс **`PriorityQueue`**.
14. На плоскости задано N отрезков. Найти точку пересечения двх отрезков, имеющую минимальную абсциссу. Использовать класс **`TreeMap`**.
15. На клетчатом листе бумаги закрашена часть клеток. Выделить все различные фигуры, которые образовались при этом. Фигурой считается набор закрашенных клеток, достижимых друг из друга при движении в четырёх направлениях. Две фигуры являются различными, если их нельзя совместить поворотом на угол, кратный 90 градусам, и параллельным переносом. Используйте класс **`HashSet`**.
16. Дана матрица из целых чисел. Найти в ней прямоугольную подматрицу, состоящую из максимального количества одинаковых элементов. Использовать класс **`Stack`**.
17. Реализовать структуру "черный ящик", хранящую множество чисел и имеющую внутренний счетчик K, изначально равный нулю. Структура должна поддерживать операции добавления числа в множество и возвращение K-го по минимальности числа из множества.
18. На прямой гоночной трассе стоит N автомобилей, для каждого из которых известны начальное положение и скорость. Определить, сколько произойдет обгонов.
19. На прямой гоночной трассе стоит N автомобилей, для каждого из которых известны начальное положение и скорость. Вывести первые K обгонов.

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 пользовательских интерфейсов язык Java содержит библиотеки классов, позволяющие создавать и поддерживать окна, использовать элементы управления (кнопки, меню, полосы прокрутки и др.), применять инструменты для создания графических приложений. Графические инструменты и интерфейсы пользователя в языке Java реализованы с помощью двух библиотек:

- Пакет AWT (загружается **`java.awt`**) содержит набор классов, позволяющих выполнять графические операции и создавать элементы управления. Этот пакет поддерживается последующими версиями языка, однако считается весьма ограниченным и недостаточно эффективным.
- Пакет Swing (загружается **`javax.swing`**, имя **`javax`** обозначает, что пакет не является основным, а только расширением языка) содержит улучшенные и обновленные классы, по большей части аналогичные AWT. К именам этих классов добавляется **`J`** (**`JButton`**, **`JLabel`** и т.д.). Пакет является частью библиотеки JFC (Java Foundation Classes), которая содержит большой набор компонентов JavaBeans, предназначенных для создания пользовательских интерфейсов.

Библиотека Swing, в отличие от AWT, более полно реализует парадигму объектно-ориентированного программирования. К преимуществам библиотеки Swing следует отнести повышение надежности, расширение возможностей пользовательского интерфейса, а также независимость от платформы. Кроме того, библиотеку Swing легче использовать, она визуально более привлекательна.

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.

Апплеты используют окна, производные от класса **`Panel`**, графические приложения используют окна, производные от класса **`Frame`**, порожденного от класса **`Window`**.

Иерархия базовых классов AWT и Swing, применяемых для построения визуальных приложений, приведена на рисунке 11.1.

Суперкласс **`java.awt.Component`** является абстрактным классом, инкапсулирующим все атрибуты визуального компонента. Класс содержит большое число методов для создания компонентов управления и событий, с ними связанных.

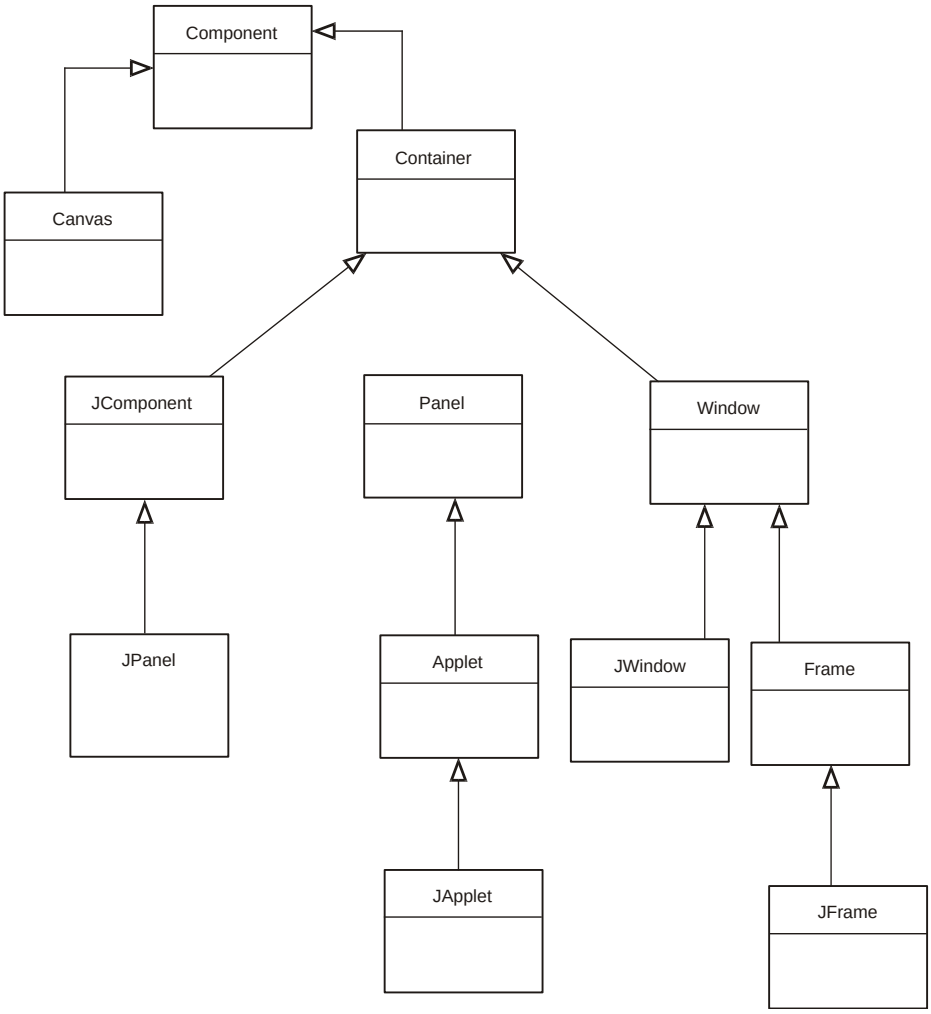


Рис. 11.1. Иерархия классов основных графических компонентов AWT и Swing

Порожденный от него подкласс **`Container`** содержит методы типа **`add()`**, которые позволяют вкладывать в него другие компоненты (объекты) и отвечает за размещение любых компонентов, которые он содержит. Класс **`Container`** порождает классы **`Panel`** и **`Window`** – фундаментальные классы при создании апплетов и фреймов.

Класс **`Panel`** используется апплетом, графический вывод которого рисуется на поверхности объекта **`Panel`** – окна, не содержащего области заголовка, строки меню и обрамления. Основу для его отображения предоставляет браузер.

Графические приложения используют класс **`Window`** (окно верхнего уровня), который является основой для организации фрейма, но сам непосредственно для вывода компонент не используется. Для этого применяется его подкласс **`Frame`**. С помощью объекта типа **`Frame`** создается стандартное окно со строкой заголовка, меню, размерами.

Аналогично классы из пакета Swing используют для вывода графических изображений окна **JPanel** и **JFrame**. Еще один используемый для вывода класс **Canvas**, представляющий пустое окно, в котором можно рисовать, является наследником класса **Component**.

Апплеты

Графические интерфейсы, рисунки и изображения могут быть реализованы в апплетах. Апплеты представляют собой разновидность графических приложений, реализованных в виде классов языка Java, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются браузером как часть документа HTML. Апплеты позволяют вставлять в документы поля, содержание которых меняется во времени, организовывать "бегущие строки", меню, мультипликацию, производить вычисления на клиентской странице. Апплеты выполняются браузером при отображении HTML-документа или просматриваются программой appletviewer. Апплеты не могут записывать файлы и читать информацию из файлов. Эти ограничения связаны с проблемой безопасности клиентского компьютера, поскольку клиенты не желают запускать «троянского коня» в свой компьютер. Существует несколько уровней безопасности, устанавливаемых клиентом для апплетов, загружаемых с сервера (надежные апплеты). Класс **Applet** обеспечивает интерфейс между апплетами и их окружением. Апплеты являются наследниками класса **Applet** из пакета **java.applet** из пакета AWT или его подкласса **JApplet** из пакета Swing.

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 при загрузке апплета для выполнения начальной инициализации. Метод **start()** вызывается каждый раз, когда апплет переносится в поле зрения 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, при выполнении апплета автоматически запускается метод **paint()** класса **Component**. Метод **paint()** не вызывается явно, а только из других методов, например из метода **repaint()**, если необходима перерисовка.

Ниже приведен пример апплета, в котором используются методы **init()**, **paint()**, метод **setColor()** установки цвета символов и метод **drawString()** рисования строк.

```
/* пример # 1 : вывод даты : DateApplet.java */
package chapt11;
import java.awt.Color;
import java.awt.Graphics;
import java.util.Calendar;
import java.util.Formatter;
import javax.swing.JApplet;

public class DateApplet extends JApplet {
```

```
        private Formatter dateFmt = new Formatter();
        private Formatter timeFmt = new Formatter();

        public void init() {
              setSize(180, 100);
              Calendar c = Calendar.getInstance();
              String era = "";
              if (c.get(Calendar.ERA) == 1)
                    era = "н.э.";
              dateFmt.format("%tA %td.%tm.%tY года "
                                + era, c, c, c, c);
              timeFmt.format("%tT", c);
        }
        public void paint(Graphics g) {
              g.setColor(Color.RED);
              g.drawString("Апплет стартовал в " + timeFmt,
                    10, getHeight()/2);
              g.setColor(new Color(0,87,127));
              g.drawString(dateFmt.toString(), 13,
                    getHeight() - 10);
        }
}
```

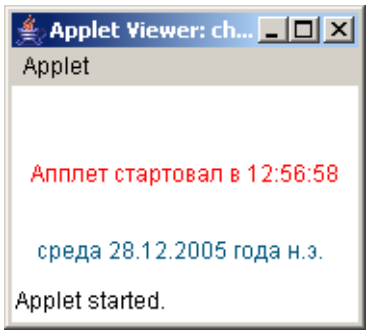


Рис. 11.2. Вывод строки и даты

Доступ к элементам даты осуществлен с помощью класса **java.util.Calendar**. Цвет выводимых символов устанавливается с помощью полей класса **Color**.

После выполнения компиляции имя класса, содержащего байт-код апплета, помещается в тег **<applet** параметры**> </applet>** документа HTML. Например:

```
<html>
<applet code = chapt11.DateApplet.class
        width = 250 height = 250> </applet></html>
```

Исполнителем HTML-документа является браузер, который и запускает соответствующий ссылке апплет.

Большинство используемых в апплетах графических методов, как и использованные в примере методы **setColor()**, **drawString()**, – методы абстракт-

ного базового класса **java.awt.Graphics**. Класс **Graphics** представляет графический контекст для рисования, который затем отображается на физическое устройство. 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.

Методы класса **Graphics** используются для отображения графики как для классов **Applet**, так и для классов **JApplet**.

В примерах 2–4, приведенных ниже, демонстрируется использование методов класса **Graphics** для вывода графических изображений в окно апплета.

```
/* пример # 2 : трехмерные прямоугольники : ThrRect.java */
package chapt11;
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JApplet;

public class ThrRect extends JApplet {
      public void draw3D(Graphics g, int x, int y, int
width, int height, boolean isRaised, boolean isFilled) {
            g.draw3DRect(x, y, width - 1, height - 1,
                  isRaised);
            g.draw3DRect(x + 1, y + 1, width - 3,
                  height - 3, isRaised);
```

```
            g.draw3DRect(x + 2, y + 2, width - 5,
                  height - 5, isRaised);
            if (isFilled)
                  g.fillRect(x + 3, y + 3, width - 6,
                  height - 6);
      }
      public void paint(Graphics g) {
            g.setColor(Color.GRAY);
            draw3D(g, 10, 5, 80, 40, true, false);
            draw3D(g, 130, 5, 80, 40, false, false);
            draw3D(g, 10, 55, 80, 40, true, true);
            draw3D(g, 130, 55, 80, 40, false, true);
        }
}
```

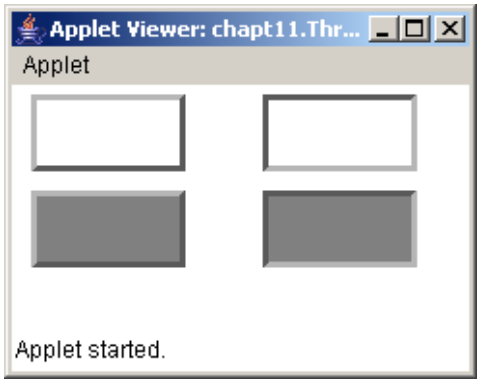


Рис. 11.3. Трехмерные прямоугольники

Пакет **java.awt** содержит большое число классов, используемых для вывода изображения: **Color**, **Font**, **Image**, **Shape**, **Canvas** и т.д. Кроме того, возможности этого пакета расширяют пакеты **java.awt.geom**, **java.awt.color**, **java.awt.image** и другие.

```
/* пример # 3 : построение фигур : BuildShape.java */
package chapt11;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Shape;
import java.awt.geom.*;
import javax.swing.JApplet;

public class BuildShape extends JApplet {
      public void init() {
            setSize(200, 205);
      }
      public void paint(Graphics g) {
            Graphics2D g2 = (Graphics2D) g;
            setBackground(Color.LIGHT_GRAY);
```

```
            g2.rotate(Math.PI / 6);
            drawChessBoard(g);
        //поворот
            g2.rotate(-Math.PI / 6);
            g.setXORMode(new Color(200, 255, 250));
            Shape e = new Ellipse2D.Float(70, 75, 70, 50);
        //рисование эллипса
            g2.fill(e);
        }
        //рисование шахматной доски
        public void drawChessBoard(Graphics g) {
            int size = 16;
            for (int y = 0; y < 8; y++) {
                for (int x = 0; x < 8; x++) {
                    if ((x + y) % 2 == 0)
                        g.setColor(Color.BLUE);
                    else
                        g.setColor(Color.WHITE);
        g.fillRect(75 + x * size, y * size - 25, size, size);
                }
                g.setColor(Color.BLACK);

        g.drawString(new Character(
(char) ('8' - y)).toString(), 66, y * size - 13);
        g.drawString(new Character(
                (char) (y + 'a')).toString(),
                        79 + y * size, 8 * size - 14);
            }
        }

}
```

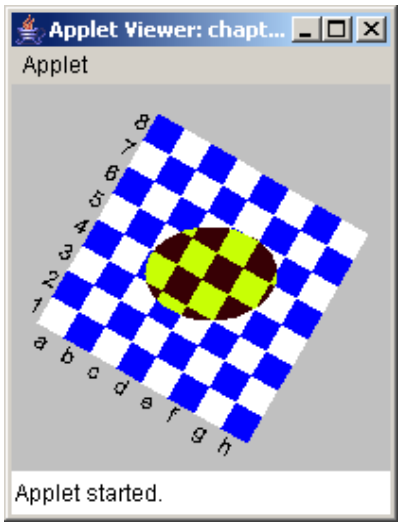


Рис. 11.4. Построение различных фигур

```
// пример #  4 : вывод GIF-изображения : DrawImage.java
package chapt11;
import java.awt.Graphics;
import java.awt.Image;
import javax.swing.JApplet;

public class DrawImage extends JApplet {
   private Image img;

   public void init() {
      //загрузка изображения из корня проекта
       img = getImage(getCodeBase(), "joli.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 chapt11;
import java.awt.Color;
import java.awt.Graphics;
import java.applet.Applet;

public class ReadParam extends Applet {
      private int bNum;
      private boolean state;

      public void start() {// чтение параметров
            String param;
                  param = getParameter("state");
            if(param != null)
             state = new Boolean(param);
            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);
      }
}
```

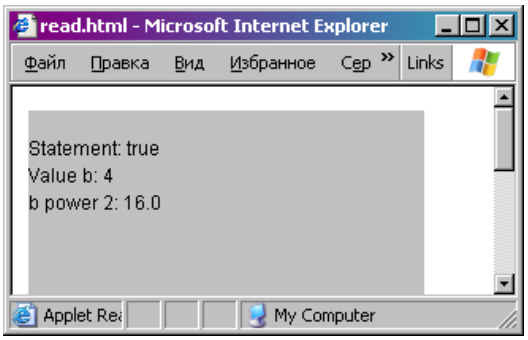


Рис. 11.5. Передача параметров в апплет

Если параметр недоступен, метод **`getParameter()`** возвращает **`null`**.

При применении управляющих компонентов в апплетах принято использовать классы из пакета **`javax.swing`**, в которых компонент на экране создается средствами Java и в минимальной степени зависит от платформы и оборудования. Такими классами-компонентами являются **`JButton`**, **`JCheckBox`**, **`JDialog`**, **`JMenu`**, **`JComboBox`**, **`JMenu`**, **`JTextField`**, **`JTextArea`** и другие. Компоненты, как правило, размещают в контейнер (класс **`Container`**), являющийся неявным объектом любого графического приложения. Явную ссылку на контейнер можно получить с помощью метода **`getContentPane()`**.

// пример # 6 : апплет с компонентом : MyJApplet.java

```
package chapt11;
import java.awt.Container;
import javax.swing.JApplet;
import javax.swing.JLabel;

public class MyJApplet extends JApplet {
      private JLabel lbl = new JLabel("Swing-applet!");

      public void init() {
            Container c = getContentPane();
            c.add(lbl);
      }
}
```

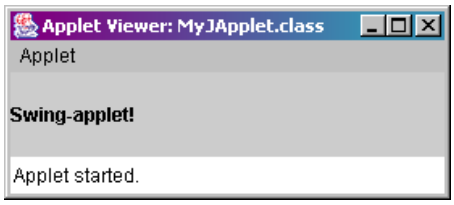


Рис. 11.6. Апплет с меткой

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 можно сделать 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

```
package chapt11;
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JApplet;

public class DemoLC extends JApplet {
      private int starX[] =
{ 112, 87, 6, 71, 47, 112, 176, 151, 215, 136 };

      private int starY[] =
{ 0, 76, 76, 124, 200, 152, 200, 124, 76, 76 };

      private int i;
      private Color c;

      public void init() {
            c = new Color(0, 0, 255);
            setBackground(Color.LIGHT_GRAY);
            i = 1;
      }
      public void start() {
            int j = i * 25;
            if (j < 255)
                  c = new Color(j, j, 255 - j);
            else i = 1;
      }
      public void paint(Graphics g) {
            g.setColor(c);
            g.fillPolygon(starX, starY, starX.length);
```

```
            g.setColor(Color.BLACK);
            g.drawPolygon(starX, starY, starX.length);
      }
      public void stop() {
            i++;
      }
}
```

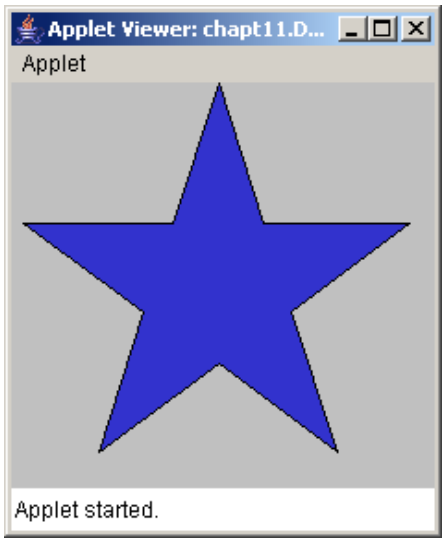


Рис. 11.7. Цвет полигона и жизненный цикл апплета

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 : Fonts.java
package chapt11;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.GraphicsEnvironment;
import javax.swing.JApplet;

public class Fonts extends JApplet {
      private String[] fonts;

      public void init() {
            GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();

            fonts = ge.getAvailableFontFamilyNames();
            setSize(700, 400);
      }
```

```
      public void paint(Graphics g) {
            int xSize = getWidth() / 170;
         for (int i = 0; i < fonts.length; i++) {
            g.setFont(new Font(
fonts[i], Font.PLAIN, 12)); // название, стиль, размер
            g.drawString(fonts[i],
                  170 * (i % xSize), 13 * (i / xSize + 1));
            }
      }
}
```

Рис. 11.8. Вывод списка шрифтов

Фреймы

В Java окно верхнего уровня (не содержащееся в другом окне) называется ***фреймом***. 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 в качестве окна для вывода объект **`Frame`** или **`JFrame`**, в который будут помещаться используемые приложением компоненты GUI (Graphics User Interface). 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 **`Frame`** или **`JFrame`.**

```
/* пример # 9 :простейший фрейм – овалы и дуги : FrameDemo.java */
package chapt11;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JFrame;
```

```
public class FrameDemo extends JFrame {
      private String msg = "My Windows-Application";

      public void paint(Graphics g) {
            int diam = 230;
            drawSphere(g, diam);
            g.drawString(msg, 59, diam + 52);
            g.drawLine(59, diam + 56, 190, diam + 56);
      }
      public void drawSphere(Graphics g, int diam) {
            int r = diam / 2;
            int alpha = 0;
            int k = 20;
            for (int i = 0; i < 4; i++) {
int width = (int) (r * Math.cos(Math.PI / 180 * alpha));
int height = (int) (r * Math.sin(Math.PI / 180 * alpha));
      g.setColor(Color.MAGENTA);
      g.drawArc(10 + r - width, r + height + i * 10,
      2 * width, 80 - i * 20, 0, 180);
      g.drawArc(10 + r - width, r - height + i * 10,
      2 * width, 80 - i * 20, 0, 180);
      g.setColor(Color.BLACK);
      g.drawArc(10 + r - width, r + height + i * 10,
      2 * width, 80 - i * 20, 0, -180);
      g.drawArc(10 + r - width, r - height + i * 10,
      2 * width, 80 - i * 20, 0, -180);
                  alpha += k;
                  k -= 1;
            }
            for (int i = 0; i < 4; i++) {
                  k = (i * i * 17);
            g.drawOval(10 + k / 2, 40, diam - k, diam);
            }
      }
      public static void main(String[] args) {
            FrameDemo fr = new FrameDemo();
            fr.setBackground(Color.LIGHT_GRAY);
            // устанавливается размер окна. Желательно!
            fr.setSize(new Dimension(250, 300));
            // заголовок
            fr.setTitle("Windows-Application");
            // установка видимости. Обязательно!
            fr.setVisible(true);
            // перерисовка - вызов paint()
            fr.repaint();
      }
}
```

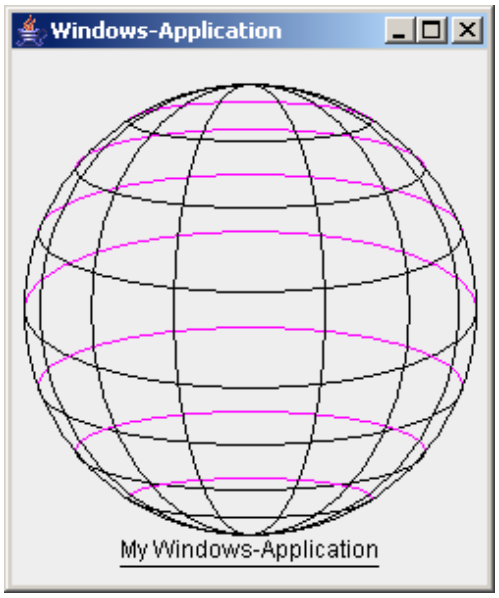


Рис. 11.9. Простейшее графическое приложение

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`** с использованием его метода **`add()`**? (выберите два)

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
СОБЫТИЯ

Основные понятия

Обработка любого события (нажатие кнопки, щелчок мышью и др.) состоит в связывании события с методом, его обрабатывающим. Принцип обработки событий, начиная с Java 2, базируется на модели делегирования событий. В этой модели имеется блок прослушивания события (**EventListener**), который ждет поступления события определенного типа от источника, после чего обрабатывает его и возвращает управление. Источник – это объект, который генерирует событие, если изменяется его внутреннее состояние, например, изменился размер, изменилось значение поля, произведен щелчок мыши по форме или выбор значения из списка. После генерации объект-событие пересылается для обработки зарегистрированному в источнике блоку прослушивания как параметр его методов – обработчиков событий.

Блоки прослушивания **Listener** представляют собой объекты классов, реализующих интерфейсы прослушивания событий, определенных в пакете **java.awt.event**. Соответствующие методы, объявленные в используемых интерфейсах, необходимо явно реализовать при создании собственных классов прослушивания. Эти методы и являются обработчиками события. Передаваемый источником блоку прослушивания объект-событие является аргументом обработчика события. Объект класса – блока прослушивания события необходимо зарегистрировать в источнике методом

```
источник.addСобытиеListener(объект_прослушиватель);
```

После этого объект-прослушиватель (**Listener**) будет реагировать именно на данное событие и вызывать метод «обработчик события». Такая логика обработки событий позволяет легко отделить интерфейсную часть приложения от функциональной, что считается необходимым при проектировании современных приложений. Удалить слушателя определенного события можно с помощью метода **remove**Событие**Listener()**.

Источником событий могут являться элементы управления: кнопки (**JButton**, **JCheckbox**, **JRadioButton**), списки, кнопки-меню. События могут генерироваться фреймами и апплетами, как mouse- и key-события. События генерируются окнами при развертке, сворачивании, выходе из окна. Каждый класс-источник определяет один или несколько методов **add**Событие**Listener()** или наследует эти методы

Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются и принимают копию объекта события. Таким образом источник вызывает метод-обработчик события, определенный в классе, являющемся блоком прослушивания, и передает методу объект события в качестве параметра. В качестве блоков прослушивания на практике используются внутренние классы. В

этом случае в методе, регистрирующем блок прослушивания в качестве параметра, используется объект этого внутреннего класса.

Каждый интерфейс, включаемый в блок прослушивания, наследуется от интерфейса **EventListener** и предназначен для обработки определенного типа событий. При этом он содержит один или несколько методов, которые всегда принимают объект события в качестве единственного параметра и вызываются в определенных ситуациях. В таблице приведены некоторые интерфейсы и их методы, которые должны быть реализованы в классе прослушивания событий, реализующем соответствующий интерфейс:

Интерфейсы	Обработчики 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 определенной ситуации и затем передается зарегистрированному 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()**, возвращающий строчный эквивалент события. Абстрактный класс **AWTEvent** из пакета **java.awt** является суперклассом всех AWT-событий, связанных с компонентами. Метод **getID()** определяет тип события, возникающего вследствие действий пользователя в визуальном приложении. Ниже приведены некоторые из классов событий, производных от **AWTEvent**, и расположенные в пакете **java.awt.event**:

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;

TextEvent – генерируется при изменении текстового поля;

ItemEvent – генерируется при выборе элемента из списка.

Класс **InputEvent** является абстрактным суперклассом событий ввода (для клавиатуры или мыши). События ввода с клавиатуры обрабатывает класс **KeyEvent**, события мыши – **MouseEvent.**

Чтобы реализовать методы-обработчики событий, связанных с клавиатурой, необходимо определить три метода, объявленные в интерфейсе **KeyListener.** 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()**.

Для регистрации события приложение-источник из своего объекта должно вызвать метод **addKeyListener(KeyListener el)**, регистрирующий блок прослушивания этого события. Здесь **el** – ссылка на блок прослушивания события.

```
/* пример # 1 : обработка событий клавиатуры: MyKey.java */
package chapt12;
import java.awt.*;
import java.awt.event.*;
import javax.swing.JApplet;

public class MyKey extends JApplet {
      private String msg = " ";
      private int x = 0, y = 20; // координаты вывода

      private class AppletKeyListener
                        implements KeyListener {
            // 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 init() {
            /* регистрация блока прослушивания */
            addKeyListener(new AppletKeyListener());
            requestFocus(); // запрос фокуса ввода
      }
      public void paint(Graphics g) {
            // значение клавиши в позиции вывода
            g.drawString(msg, x, y);
      }
}
```

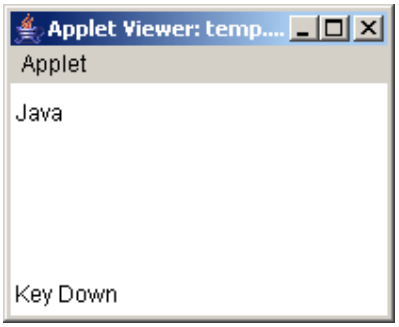


Рис. 12.1. Результат нажатия клавиши отображен в строке состояния

Коды специальных клавиш (перемещение курсора, функциональных клавиш) недоступны через **keyTyped()**, для обработки нажатия этих клавиш используется метод **keyPressed()**.

В качестве блока прослушивания в методе **init()** зарегистрирован внутренний класс **AppletKeyListener**. Затем в блоке прослушивания реализованы все три метода обработки события, объявленные в интерфейсе **KeyListener**.

В следующем апплете проверяется принадлежность прямоугольнику координат нажатия клавиши мыши с помощью реализации интерфейса **MouseListener** и события **MouseEvent**.

```
/* пример # 2 : события нажатия клавиши мыши: MyRect.java */
package chapt12;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class MyRect extends JApplet {
      private Rectangle rect =
                        new Rectangle(20, 20, 100, 60);
      private class AppletMouseListener//блок обработки событий
                        implements MouseListener {
            /* реализация всех пяти методов интерфейса MouseListener */
            public void mouseClicked(MouseEvent me) {
                  int x = me.getX();
                  int y = me.getY();
                  if (rect.contains(x, y)) {
                        showStatus(
                        "клик в синем прямоугольнике");
                  } else {
                        showStatus("клик в белом фоне");
                  }
            }
            // реализация остальных методов интрефейса пустая
            public void mouseEntered(MouseEvent e) {}
            public void mouseExited(MouseEvent e) {}
            public void mousePressed(MouseEvent e) {}
            public void mouseReleased(MouseEvent e) {}
      }
      public void init() {
            setBackground(Color.WHITE);
            /* регистрация блока прослушивания */
            addMouseListener(new AppletMouseListener());
      }
      public void paint(Graphics g) {
            g.setColor(Color.BLUE);
            g.fillRect(rect.x, rect.y,
                              rect.width, rect.height);
      }
}
```

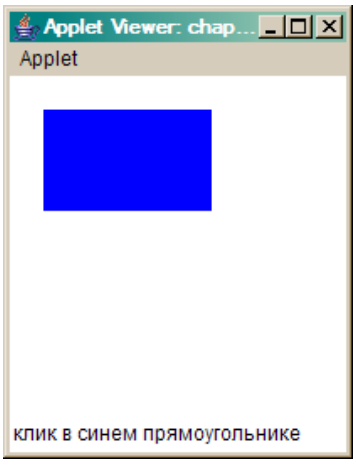


Рис. 12.2. Результат нажатия кнопки отображен в строке состояния

Способ обработки событий в компонентах Swing – это интерфейс (графические компоненты) и реализация (код обработчика события, 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()** объектом класса **JButton**. Интерфейс **ActionListener** содержит единственный метод **actionPerformed()**, который нужно реализовать в блоке обработки в соответствии с поставленной задачей: извлечь числа из двух текстовых полей, сложить их и поместить результат в метку.

```
/* пример # 3 : регистрация, генерация и обработка ActionEvent:
SimpleButtonAction.java */
package chapt12;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleButtonAction extends JApplet {
      private JButton additionBtn = new JButton("Сложить");
      private JTextField txtField1 = new JTextField(3);
      private JTextField txtField2 = new JTextField(3);
      private JLabel answer = new JLabel();

      private class ButtonListener
                        implements ActionListener {
      // реализация класса- обработчика события
      public void actionPerformed(ActionEvent ev) {
            try {
             int t1, t2;
             t1 = Integer.parseInt(txtField1.getText());
             t2 = Integer.parseInt(txtField2.getText());
             answer.setText("Ответ: " + (t1 + t2));
                  showStatus("Выполнено успешно!");
            } catch (NumberFormatException e) {
                  showStatus("Ошибка ввода!");
            }
            /*
             * String s1, s2; извлечение надписи на кнопке из события
             * s1 = ((JButton)ev.getSource()).getText();
             */
            // извлечение команды из события
            // s2 = ev.getActionCommand();
            /*
             * извлечение из события объекта, ассоциированного с кнопкой
             *  if (ev.getSource() == additionBtn)
             *  применяется если обрабатываются
```

```
                * события нескольких кнопок одним обработчиком
                */
                }
        }
        public void init() {
                Container c = getContentPane();
                setLayout(new FlowLayout());/* «плавающее»
                                                размещение компонентов*/
                c.add(txtField1);
                c.add(txtField2);
                // регистрация блока прослушивания события
                additionBtn.addActionListener(
                                                new ButtonListener());
                c.add(additionBtn);
                c.add(answer);
        }
}
```

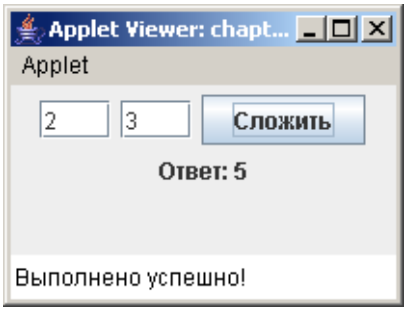


Рис. 12.3. Обработка события кнопки

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

Реализация всех методов–обработчиков событий, объявленных в интерфейсах, не всегда необходима. 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, которые он расширяет. При этом определяется новый класс – 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.

Рассмотрим события, возникающие в приложениях, связанных с консольным или графическим окнами. 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()** и **update()**;
- через возвращаемое значение метода **getGraphics()** класса **Component.**

Событие **FocusEvent** предупреждает программу, что компонент получил или потерял фокус ввода. Событие **WindowEvent** извещает программу, что был активизирован один из системных элементов управления окна.

В следующем примере реализован простейший графический редактор, позволяющий выбрать цвет и рисовать пером в поле приложения. Приложение создает объект **PaintEditor** и передает ему управление сразу же в методе **main()**. Выбор цвета осуществляется с помощью объекта класса **JColorChooser.**

```
/* пример # 4 : применение адаптеров: PaintEditor.java */
package chapt12;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PaintEditor extends JFrame {
      private int prevX, prevY;
      private Color color = Color.BLACK;
      private JButton jButton =
                        new JButton("ColorChooser");

      public PaintEditor() {
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            c.add(jButton);

            jButton.addActionListener(
                        new ButtonActionListener());
            addMouseListener(new PaintMouseAdapter());
```

```
            addMouseMotionListener(
                        new PaintMouseMotionAdapter());
      }
      private class ButtonActionListener
                        implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                  color = JColorChooser.
                  showDialog(((Component) e.getSource())
                              .getParent(), "Demo", color);
            }
      }
      private class PaintMouseAdapter extends MouseAdapter{
            public void mousePressed(MouseEvent ev) {
                  setPreviousCoordinates(
                                    ev.getX(), ev.getY());
            }
      }
      private class PaintMouseMotionAdapter
                        extends MouseMotionAdapter {
            public void mouseDragged(MouseEvent ev) {
                  Graphics g = getGraphics();
                  g.setColor(color);
                  g.drawLine(
                  prevX, prevY, ev.getX(), ev.getY());
                  setPreviousCoordinates(
                                    ev.getX(), ev.getY());
            }
      }
      public void setPreviousCoordinates(
                        int aPrevX, int aPrevY) {
            prevX = aPrevX;
            prevY = aPrevY;
      }
      public static void main(String[] args) {
            PaintEditor pe = new PaintEditor();
            pe.addWindowListener(new WindowAdapter() {
                  public void windowClosing(WindowEvent ev)
                  {
                        System.exit(0);
                  }
            });
            pe.setBounds(200, 100, 300, 200);
            pe.setTitle("MicroPaint");
            pe.setVisible(true);
      }
}
```

Рис. 12.4. Простейший текстовый редактор с классами-адаптерами

Конструктор класса **PaintEditor** использует методы **addMouseListener(new PaintMouseAdapter(this))** и **addMouseMotionListener(new PaintMouseMotionAdapter(this))** для регистрации событий мыши. При создании объекта класса **PaintEditor** эти методы сообщают объекту, что он заинтересован в обработке определенных событий. Однако вместо того, чтобы известить его об этом прямо, конструктор организует посылку ему предупреждений через объекты классов **PaintMouseAdapter** и **PaintMouseMotionAdapter**. 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)**. Абстрактный класс **MouseMotionAdapter** используется для обработки событий, связанных с движениями мыши при создании блока прослушивания, и содержит следующие переопределяемые методы: **mouseDragged(MouseEvent e)**, **mouseMoved(MouseEvent e)**.

Класс **PaintEditor** также обрабатывает событие класса **WindowEvent**. Когда объект генерирует событие **WindowEvent**, объект **PaintEditor** анализирует, является ли оно событием **WindowClosing**. Если это не так, объект **PaintEditor** 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 КОМПОНОВКИ И УПРАВЛЕНИЯ

В первых версиях Java (1.0.x) были созданы элементы управления AWT, такие как метки, кнопки, списки, текстовые поля, предоставляющие пользователю различные способы управления приложением. Эти элементы, наследуемые от абстрактного класса **java.awt.Component** и называемые также компонентами, были частично зависимы от аппаратной платформы и не в полной мере объектно-ориентированы по способу использования. Развитие парадигмы "write once, run everywhere" ("написать однажды, запускать везде") привело к разработке таких компонентов (библиотеки Swing), которые были не привязаны к конкретной платформе. Эти классы доступны разработчикам в составе как JDK, так и отдельного продукта JFC (Java Foundation Classes). Причем для совместимости со старыми версиями JDK 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, библиотека JFC содержит 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. Вместо ручного расположения применяются менеджеры размещения, определяющие 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 один из классов **FlowLayout**, **BorderLayout**, **GridLayout**, **CardLayout**, **BoxLayout**, реализующих интерфейс **LayoutManager**, устанавливающий размещение.

Класс **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 { }
```

В следующем примере демонстрируются возможности поточной компоновки различных элементов управления.

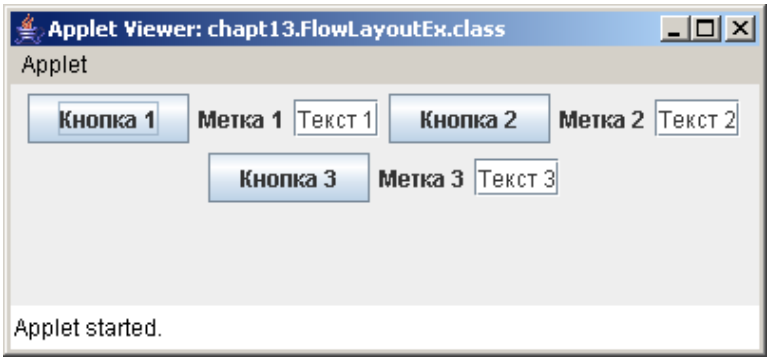
```
/* пример # 1 : поточная компоновка по центру: FlowLayoutEx.java */
package chapt13;
import java.awt.*;
import javax.swing.*;
```

```
public class FlowLayoutEx extends JApplet {
     private Component c[] = new Component[9];

     public void init() {
          String[] msg =
             { "Метка 1", "Метка 2", "Метка 3" };
          String[] str =
             { "Кнопка 1", "Кнопка 2", "Кнопка 3" };
          String[] txt = {"Текст 1", "Текст 2", "Текст 3"};
          //установка менеджера размещений
          setLayout(new FlowLayout());
          setBackground(Color.gray);
          setForeground(Color.getHSBColor(1f, 1f, 1f));
          for (int i = 0; i < c.length/3; i++) {
             c[i] = new JButton(str[i]);
             add(c[i]);
             c[i + 3] = new JLabel(msg[i]);
             add(c[i + 3]);
             c[i+6] = new JTextField(txt[i]);
             add(c[i + 6]);
          }
          setSize(450, 150);
     }
}
```

Перегружаемый метод **add(Component ob)**, определенный в классе **java.awt.Container** (подклассе **Component**), добавляет компоненты **JButton**, **JLabel**, **JTextField** к окну и прорисовывает их всякий раз, когда окно отображается на экран.

Метод **setLayout(LayoutManager mgr)** устанавливает менеджер размещения для данного контейнера. Результаты работы апплета приведены на рисунке.



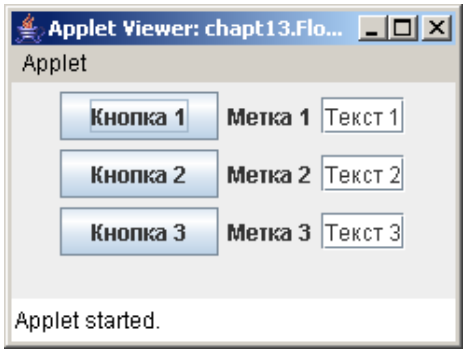


Рис. 13.1. Размещение компонентов **FlowLayout**

Менеджер **GridLayout** разделяет форму 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 формы не влияет.

```
/* пример # 2 : компоновка в табличном виде: GridLayoutEx.java */
package chapt13;
import javax.swing.*;
import java.awt.*;
public class GridLayoutEx extends JApplet {
     private Component b[] = new Component[7];
     public void init() {
          setLayout(new GridLayout(2, 4));/*две строки,
                                          четыре столбца*/
          for (int i = 0; i < b.length; i++)
               add((b[i] = new JButton("(" + i +  ")")));
     }
}
```

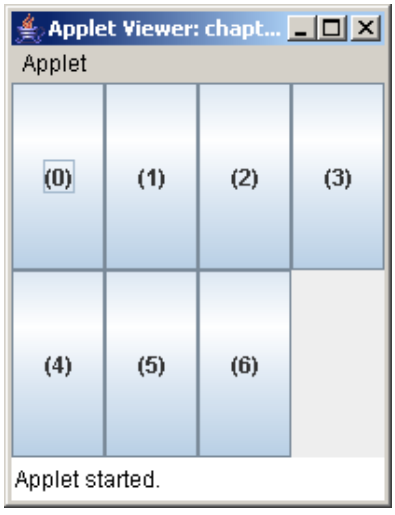


Рис. 13.2. Размещение компонентов **GridLayout**

Менеджер **BorderLayout** позволяет позиционировать элементы и группы из них в областях фиксированного размера, граничащих со сторонами фрейма, которые обозначаются параметрами сторонами света: **NORTH**, **SOUTH**, **EAST**, **WEST**. Остальное пространство обозначается как **CENTER**.

```
/* пример # 3 : фиксированная компоновка по областям:
BorderGridLayoutDemo.java */
package chapt13;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Container;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JToggleButton;// «западающая» кнопка
public class BorderGridLayoutDemo extends JFrame {
      public BorderGridLayoutDemo() {
            Container c = getContentPane();
            c.setLayout(new BorderLayout());
     c.add(new JToggleButton("--1--"), BorderLayout.WEST);
     c.add(new JToggleButton("--2--"), BorderLayout.SOUTH);
     c.add(new JToggleButton("--3--"), BorderLayout.EAST);
            JPanel jPanel = new JPanel();
            c.add(jPanel, BorderLayout.NORTH);
            jPanel.setSize(164, 40);
            jPanel.setLayout(new GridLayout(2, 4));
            for (int i = 0; i < 7; i++)
                  jPanel.add(new JButton("" + i));
      }
      public static void main(String[] args) {
            BorderGridLayoutDemo fr =
                  new BorderGridLayoutDemo();
            fr.setSize(300, 200);
            fr.setTitle("Border & Grid Layouts Example");
            fr.setDefaultCloseOperation(EXIT_ON_CLOSE);
            fr.setVisible(true);
      }
}
```

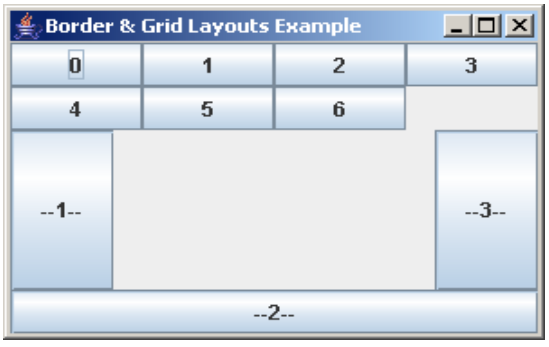


Рис. 13.3. Размещение компонентов **BorderLayout** и **GridLayout**

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 # 4 : компоновка в группах с ориентацией: BoxLayoutDemo.java */
package chapt13;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Font;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JToggleButton;
import javax.swing.border.EtchedBorder;
import javax.swing.border.TitledBorder;

public class BoxLayoutDemo extends JFrame {

      public BoxLayoutDemo() {
            Container c = getContentPane();
            setBounds(20, 80, 300, 300);
            c.setLayout(new BorderLayout());
            Box row = Box.createHorizontalBox();
            for (int i = 0; i < 4; i++) {
                  JButton btn = new JButton("Кн " + i);
            btn.setFont(new Font("Tahoma", 1, 10 + i * 2));
                  row.add(btn);
            }
            c.add(row, BorderLayout.SOUTH);

            JPanel col = new JPanel();
            col.setLayout(
                  new BoxLayout(col, BoxLayout.Y_AXIS));
            col.setBorder(
      new TitledBorder(new EtchedBorder(), "Столбец"));
            for (int i = 0; i < 4; i++) {
                  JToggleButton btn =
                        new JToggleButton("Кнопка " + i);
                  col.add(btn);
            }
            c.add(col, BorderLayout.WEST);
      }
      public static void main(String[] args) {
            BoxLayoutDemo frame = new BoxLayoutDemo();
            frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
            frame.setVisible(true);
      }
}
```

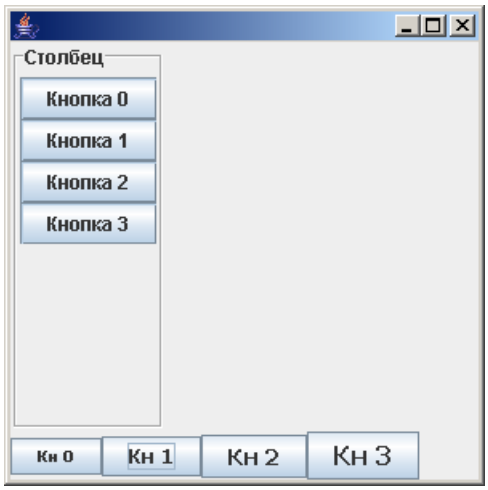


Рис. 13.4. Размещение компонентов **BoxLayout** и **Box**

Для того чтобы располагать компоненты в произвольных областях фрейма, следует установить для менеджера размещений значение **null** и воспользоваться методом **setBounds()**.

```
/* пример # 5 : произвольное размещение: NullLayoutEx.java */
package chapt13;
import java.awt.Container;
import javax.swing.*;

public class NullLayoutEx extends JFrame {
     public NullLayoutEx() {
          Container c = getContentPane();
          //указание размеров фрейма
          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.setDefaultCloseOperation(EXIT_ON_CLOSE);
          nl.setVisible(true);
     }
}
```

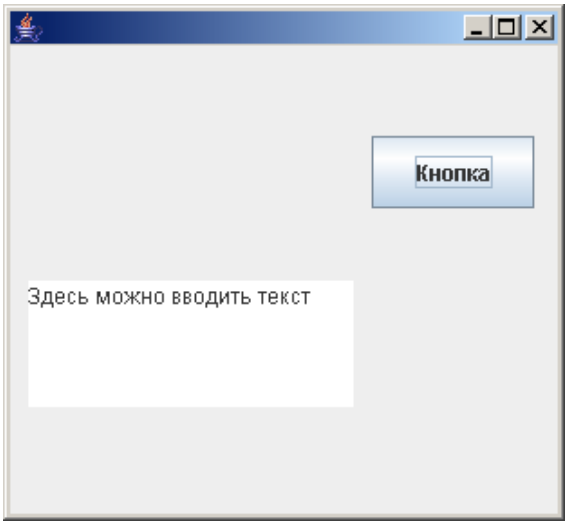


Рис. 13.5. Произвольное размещение компонентов

Элементы управления

Все элементы управления из пакета AWT являются наследниками классов **`Component`** и **`Container`**. При использовании пакета Swing компоненты наследуются от класса **`JComponent`**, производного от класса **`Container`**.

В качестве примеров можно привести текстовые метки **`Label`**, **`JLabel`**, которые создаются с помощью конструкторов, устанавливающих текст метки. Возможность изменения текста метки предоставляет метод **`setText(String txt)`**. Получить значение текста метки можно методом **`getText()`**.

Кнопки **`Button`** и **`JButton`**, **`CheckBox`** и **`JCheckBox`**, **`RadioButton`** и **`JRadioButton`**, **`JToggleButton`** используются для генерации и обработки событий.

Списки **`List`** и **`JList`** позволяют выбирать один или несколько элементов из списка.

Полосы прокрутки **`ScrollBar`** и **`JScrollBar`** используются для облегчения просмотра.

Однострочная область ввода **`TextField`** и **`JTextField`** и многострочная область ввода – **`TextArea`** и **`JTextArea`** позволяют редактировать и вводить текст (см. рис. 13.6).

Суперклассом кнопок является класс **`AbstractButton`**, от которого наследуются два наиболее используемых класса: **`JButton`** и **`JToggleButton`**. Первый предназначен для создания обычных кнопок, а второй – для создания «залипающих» кнопок, радиокнопок (класс **`JRadioButton`**) и отмечаемых кнопок (класс **`JCheckBox`**). Кроме указанных, от **`AbstractButton`** наследуется два класса **`JCheckBoxMenuItem`** и **`JRadioButtonMenuItem`**, применяемых для организации меню с радиокнопками и отмечаемыми кнопками (см. рис. 13.7).

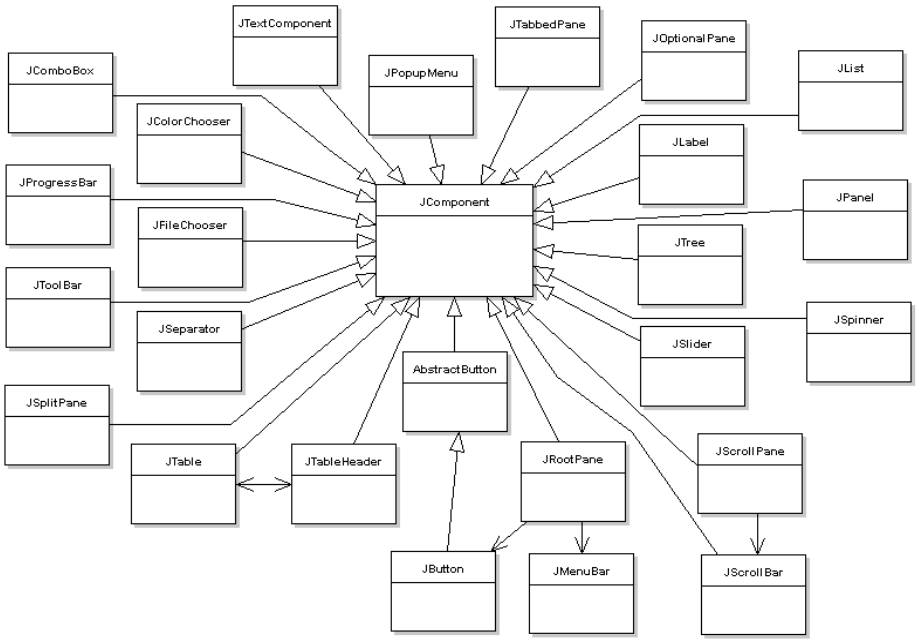


Рис. 13.6. Иерархия наследования компонентов в библиотеке Swing

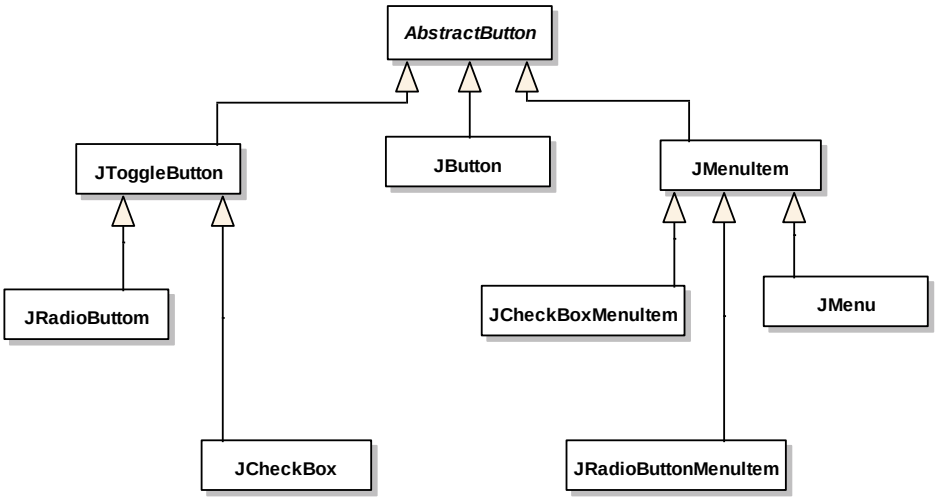


Рис. 13.7. Классы кнопок в Swing

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, достаточно 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 # 6 : кнопка и ее методы: VisualEx.java

```
package chapt13;
import javax.swing.JPanel;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JTextField;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class VisualEx extends JApplet {
      private JPanel jContentPane = null;
      private JButton yesBtn = null;
      private JButton noBtn = null;
      private JLabel label = null;
      private JTextField textField = null;

      public void init() {
            setSize(180, 160);
            setContentPane(getJContentPane());
            setBackground(java.awt.Color.white);
      }
      private JPanel getJContentPane() {
            if (jContentPane == null) {
                  label = new JLabel();
                  label.setText("");
                  jContentPane = new JPanel();
                  jContentPane.setLayout(new FlowLayout());
                  jContentPane.add(getYesBtn(), null);
                  jContentPane.add(getNoBtn(), null);
                  jContentPane.add(label, null);
                  jContentPane.add(getTextField(), null);
            }
            return jContentPane;
      }
      private JButton getYesBtn() {
        if (yesBtn == null) {
            yesBtn = new JButton();
            yesBtn.setText("казнить");
            yesBtn.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                  label.setText("Казнить,");
                  textField.setText("нельзя помиловать");
                        }
                  });
            }
            return yesBtn;
      }
      private JButton getNoBtn() {
            if (noBtn == null) {
                  noBtn = new JButton();
                  noBtn.setText("помиловать");
            noBtn.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                        label.setText("Казнить нельзя,");
                        textField.setText("помиловать");
                        }
                  });
            }
            return noBtn;
      }
      private JTextField getTextField() {
            if (textField == null) {
                  textField = new JTextField();
                  textField.setColumns(12);
                  text-
Field.setHorizontalAlignment(JTextField.CENTER);
                  textField.setEditable(false);
            }
            return textField;
      }
}
```

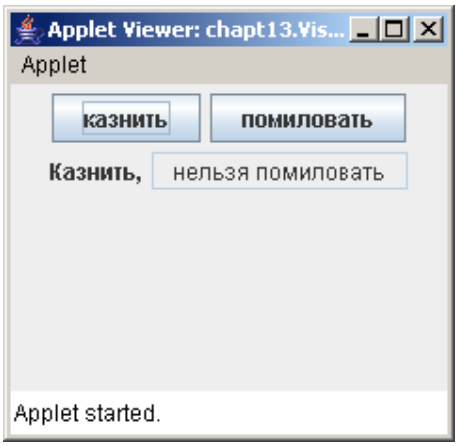


Рис. 13.8. Апплет с кнопками, меткой и текстовым полем

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.

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 альтернативных вариантов, 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()**. Чтобы сделать элемент редактируемым, следует использовать метод **setEditable(boolean editable)**. Существуют методы по вставке и удалению элементов списка во время выполнения программы **insertItemAt(int pos)** и **removeItemAt(int pos)**.

// пример # 7 : простой выпадающий список: ComboBoxEx.java

```
package chapt13;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class ComboBoxEx extends JApplet {
      private final int N = 3;
      private JTextField textField = new JTextField(2);
      private JComboBox comboBox = new JComboBox();
      private Map<String, Integer> exams =
                        new HashMap<String, Integer>(N);
      private class ComboListener implements ItemListener {
            // реакция на изменение текущего значения ComboBox
            public void itemStateChanged(ItemEvent ev) {
                  String name = (String) ev.getItem();
            textField.setText(exams.get(name).toString());
            }
      }
      public void init() {
            exams.put("Программирование", 4);
            exams.put("Алгебра", 7);
            exams.put("Топология", 8);
            // добавление элементов в ComboBox
            Iterator i = exams.entrySet().iterator();
```

```
            while (i.hasNext())
                  comboBox.addItem(
                        ((Map.Entry) i.next()).getKey());
            comboBox.addItemListener(new ComboListener());
            textField.setText(exams.get(
                  comboBox.getSelectedItem()).toString());

            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            c.add(comboBox);
            c.add(textField);
      }
}
```

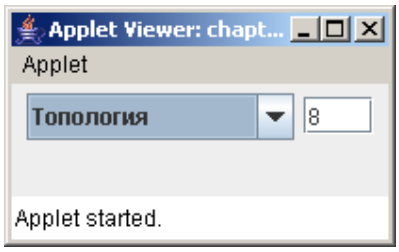


Рис. 13.9. Выпадающий список

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 **`JTextField`**.

В следующем примере приведен способ по организации многоязычного меню. Здесь также используются возможности класса **`ResourceBundle`** по извлечению информации из файла свойств (properties). При разработке больших приложений не рекомендуется помещать в код текстовые сообщения, так как при их изменении программисту потребуется корректировать и затем перекомпилировать большую часть приложения.

```
/* пример # 8 : регистрация, генерация и обработка ActionEvent:
ButtonActionDemo.java, Messages.java */
package chapt13;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.io.UnsupportedEncodingException;
import java.util.Locale;
import javax.swing.JButton;
import javax.swing.JComboBox;
```

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class ButtonActionDemo extends JFrame {
      private static final String EN_LANGUAGE = "English";
      private static final String RU_LANGUAGE = "Русский";
      private JPanel jContentPane = null;
      private JComboBox languageChooser = null;
      private JButton yesBtn = null;
      private JButton noBtn = null;
      private JLabel jLabel = null;

      public ButtonActionDemo() {
            initialize();
      }
      // ActionListener для кнопки 'Yes'
      private class YesButtonListener
                  implements ActionListener {
          public void actionPerformed(ActionEvent event) {
           jLabel.setText(getString("BUTTON_YES_MESSAGE"));
            }
      }
      // ActionListener для кнопки 'No'
      private class NoButtonListener
                  implements ActionListener {
          public void actionPerformed(ActionEvent event) {
            jLabel.setText(getString("BUTTON_NO_MESSAGE"));
            }
      }
      // ItemListener для combobox
      private class LanguageChooserItemListener
                  implements ItemListener {
            public void itemStateChanged(ItemEvent e) {
      if (((String) e.getItem()).equals(EN_LANGUAGE)) {
            Locale.setDefault(Locale.ENGLISH);
      } else {
            Locale.setDefault(new Locale("RU"));
      }
                  yesBtn.setText(getString("BUTTON_YES"));
                  noBtn.setText(getString("BUTTON_NO"));
            }
      }
      private void initialize() {
            setSize(230, 200);
            setContentPane(getJContentPane());
            setTitle("JFrame");
            setVisible(true);
      }
```

```
	private JPanel getJContentPane() {
		if (jContentPane == null) {
			jLabel = new JLabel();
			jLabel.setText("JLabel");
			jContentPane = new JPanel();
			jContentPane.setLayout(new FlowLayout());
		}
		languageChooser = new JComboBox();
		languageChooser.addItem(EN_LANGUAGE);
		languageChooser.addItem(RU_LANGUAGE);
		languageChooser.addItemListener(
			new LanguageChooserItemListener());

		yesBtn = new JButton(getString("BUTTON_YES"));
		yesBtn.addActionListener(
			new YesButtonListener());

		noBtn = new JButton(getString("BUTTON_NO"));
		noBtn.addActionListener(
			new NoButtonListener());

		jContentPane.add(languageChooser);
		jContentPane.add(yesBtn);
		jContentPane.add(noBtn);
		jContentPane.add(jLabel);

		return jContentPane;
	}
	public static void main(String[] args) {
		Locale.setDefault(Locale.ENGLISH);
		ButtonActionDemo ob = new ButtonActionDemo();
		ob.setDefaultCloseOperation(EXIT_ON_CLOSE);
	}
	private String getString(String property) {
		String text = "";
		try {
			text = new String(
Messages.getString(property).getBytes(
					"ISO-8859-1"), "CP1251");
		} catch (UnsupportedEncodingException ex) {
			ex.printStackTrace();
		}
		return text;
	}
}
package chapt13;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
```

```
public class Messages {
      private static final String BUNDLE_NAME =
                                  "chapt13.messages";
      private static final ResourceBundle RESOURCE_BUNDLE =
                  ResourceBundle.getBundle(BUNDLE_NAME);

      public static String getString(String key) {
            try {
                  return RESOURCE_BUNDLE.getString(key);
            } catch(MissingResourceException e) {
                        return '!' + key + '!';
                  }
      }
}
```

Файлы ресурсов **messages.properties** и **messages_ru.properties**, из которых извлекаются сообщения на английском и русском языках соответственно, выглядят следующим образом:

```
BUTTON_YES=yes
BUTTON_NO=no
BUTTON_YES_MESSAGE=Button <yes> is pressed
BUTTON_NO_MESSAGE=Button <no> is pressed

BUTTON_YES=да
BUTTON_NO=нет
BUTTON_YES_MESSAGE=Нажата кнопка <да>
BUTTON_NO_MESSAGE=Нажата кнопка <нет>
```

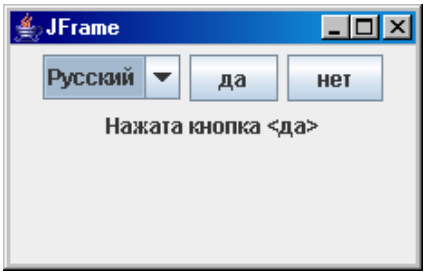


Рис. 13.10. Результат нажатия кнопки «да» отображен в метке

Команда, ассоциированная с кнопкой, возвращается вызовом метода **getActionCommand()** класса **ActionEvent**, экземпляр которого содержит всю информацию о событии и его источнике.

В следующем примере рассмотрен вариант объединения нескольких радиокнопок (**JRadioButton**) в группу и отслеживание изменения их состояния.

```
/* пример # 9 : отслеживание изменения состояния флажка:
 RadioBtnGroupEx.java */
package chapt13;
import java.awt.*;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.*;
```

```
public class RadioBtnGroupEx extends JApplet {
      private ButtonGroup btnGroup = new ButtonGroup();
      private JLabel label = null;
      private class RadioItemListener
                        implements ItemListener {
            public void itemStateChanged(ItemEvent e) {
                  boolean selected =
            (e.getStateChange() == ItemEvent.SELECTED);
                  AbstractButton button =
            (AbstractButton) e.getItemSelectable();
                  if (selected)
                        label.setText("Selected Button: "
                              + button.getActionCommand());
            System.out.println(
            "ITEM Choice Selected: " + selected +
            ", Selection: " + button.getActionCommand());
            }
      }
      public void init() {
            Container c = getContentPane();
            c.setLayout(new FlowLayout());
            JRadioButton red = new JRadioButton("Red");
            red.setSelected(true);
            btnGroup.add(red);
            c.add(red);
            label = new JLabel("Selected Button: Red");
            JRadioButton green = new JRadioButton("Green");
            btnGroup.add(green);
            c.add(green);
            JRadioButton blue = new JRadioButton("Blue");
            btnGroup.add(blue);
            c.add(blue);
      ItemListener itemListener = new RadioItemListener();
            red.addItemListener(itemListener);
            green.addItemListener(itemListener);
            blue.addItemListener(itemListener);
            c.add(label);
      }
}
```

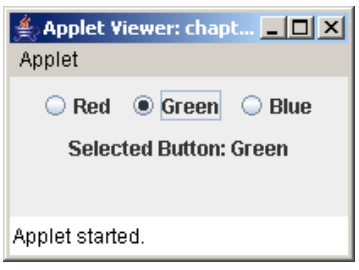


Рис. 13.11. Элемент управления **JRadioButton**

Ниже приведен простейший вариант использования объекта класса **JSlider**, представляющий собой ползунковый регулятор, позволяющий выбрать значение из интервала возможных значений.

```
/* пример # 10 : использование ползункового регулятора: SliderEx.java */
package chapt13;
import java.awt.*;
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class SliderEx extends JApplet {
      private JLabel sliderValue =
                  new JLabel("Slider Value: 25");

      private class SliderListener
                        implements ChangeListener {
            public void stateChanged(ChangeEvent e) {
                  sliderValue.setText("Slider Value: " +
                  ((JSlider) (e.getSource())).getValue());
            }
      }
      public void init() {
            Container c = getContentPane();
            c.setLayout(new FlowLayout());

            JSlider slider =
            new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
            // установка видимости меток и делений
            slider.setPaintLabels(true);
            slider.setPaintTicks(true);
            // установка расстояний между делениями
            slider.setMinorTickSpacing(2);
            slider.setMajorTickSpacing(10);
            slider.setLabelTable(
                        slider.createStandardLabels(10));
            slider.addChangeListener(new SliderListener());
            c.add(slider);
            c.add(sliderValue);
      }
}
```

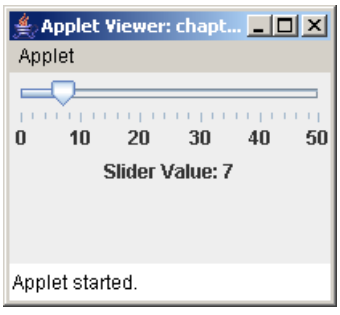


Рис. 13.12. Элемент управления **JSlider**

Блок прослушивания событий в ответ на генерацию объекта события **ItemEvent** вызвал метод **itemStateChange(ItemEvent e)**, извлекающий из объекта класса **ItemEvent** константу состояния, в данном случае **SELECTED**, а в ответ на генерацию объекта события **ChangeEvent** вызывается метод **stateChanged(ChangeEvent e)**.

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 */
package chapt13;
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**.

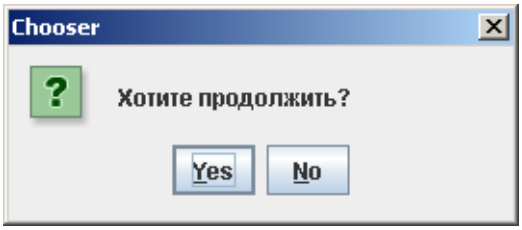


Рис. 13.13. Диалог выбора

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 */
package chapt13;
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
     }
}
```

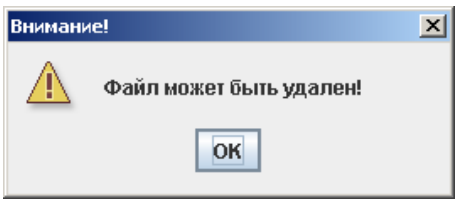


Рис. 13.14. Вывод сообщения

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 */
package chapt13;
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);
     }
}
```

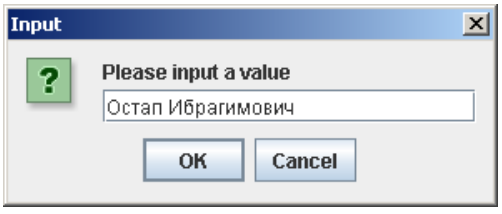


Рис. 13.15. Простейший запрос на ввод строки

Следующий пример предоставляет возможность выбора из заранее определенного списка значений.

```
/* пример # 14 : формирование запроса на выбор из списка:
DemoInputWithOptions.java */
package chapt13;
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);
     }
}
```

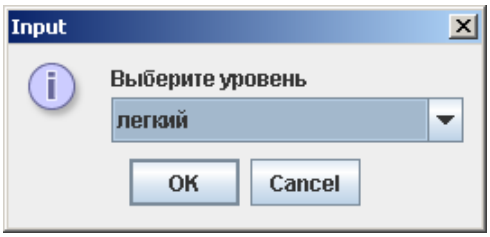


Рис. 13.16. Запрос на выбор из списка

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: MyDialog.java
package chapt13;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MyDialog extends JDialog{
      private final JFrame parent;
      private JCheckBox cb = new JCheckBox();
      private JButton ok = new JButton("Ok");
```

```
      public MyDialog(final 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(
                  new ActionListener(){
                  public void actionPerformed(
                              ActionEvent e) {
                        dispose();
                        if (cb.isSelected())
                              parent.dispose();
                  }
            });
            c.add(cb);
            c.add(ok);

            setSize(200, 100);
            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setVisible(true);
      }
}
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(DemoJDialog.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. 13.17. Произвольный диалог

Для создания пользовательского меню следует воспользоваться возможностями классов **JMenu**, **JMenuBar** и **JMenuItem**.

```
/* пример # 16 : создание меню: SimpleMenu.java */
package chapt13;
import javax.swing.*;
import java.awt.event.*;

public class SimpleMenu extends JApplet {
      private JMenu menu;
      private JMenuItem item1, item2;

      private class MenuItemListener
                        implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                  JMenuItem jmi =
                        (JMenuItem) e.getSource();
                  if (jmi.equals(item2))
                        System.exit(0);
                  else
                        showStatus("My Simple Menu");
            }
      }
      public void init() {
            JMenuBar menubar = new JMenuBar();
            setJMenuBar(menubar);
            menu = new JMenu("Main");

            item1 = new JMenuItem("About");
            item2 = new JMenuItem("Exit");
            item1.addActionListener(
                              new MenuItemListener());
            item2.addActionListener(
                              new MenuItemListener());

            menu.add(item1);
            menu.add(item2);
            menubar.add(menu);
      }
}
```

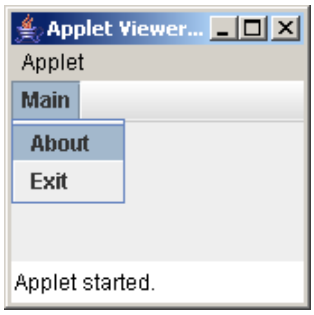


Рис. 13.18. Простейшее меню

Класс **`JTextArea`** позволяет вводить и отображать многострочную информацию. Такая возможность полезна при обработке текстов, двумерных массивов. Пользователь может сам вводить в этот объект информацию, разделяя строки нажатием клавиши <Enter>. Этот объект не имеет полосы прокрутки (ее можно добавить), поэтому если введено строк больше, чем объявлено, но лишний текст будет утрачен. Класс **`JTextArea`** является подклассом **`JTextComponent`**, следовательно, программисту доступны все его возможности. В частности, есть возможность обработки части текста, выделенной пользователем, а также определение количества строк и столбцов (см. рис. 13.19).

В примере решается простая задачи загрузки из файла в объект **`JTextArea`** двумерного массива и его примитивная обработка. При загрузке матрицы из файла для поиска соответствующего дискового файла использованы возможности класса **`JFileChooser`**.

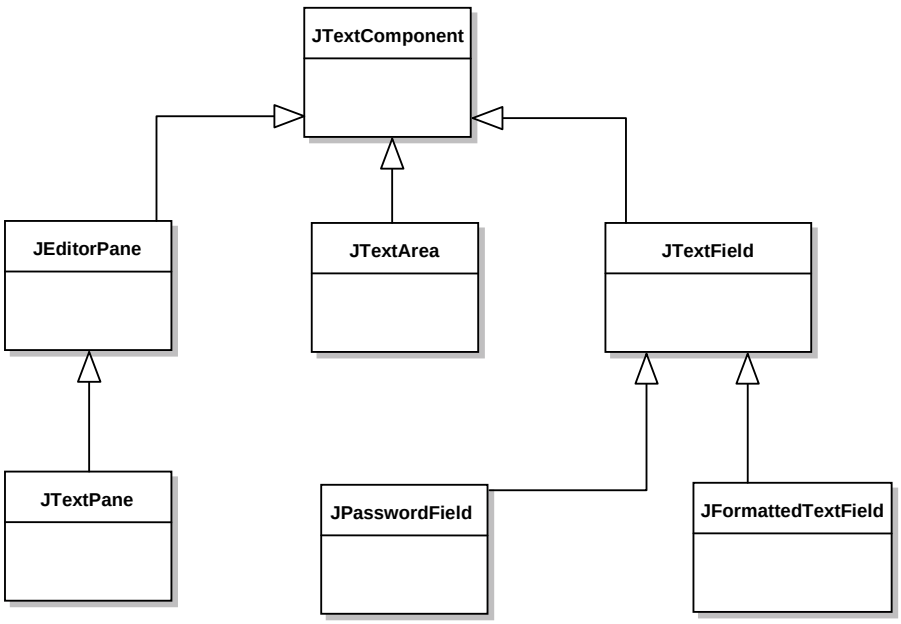


Рис. 13.19. Классы текстовых компонентов

```
/* пример # 17 : вывод двумерного массива в текстовое поле: ArraysWork.java */
package chapt13;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.text.BadLocationException;
import javax.swing.plaf.metal.MetalBorders.TextFieldBorder;

public class ArraysWork extends JFrame {
      private JFileChooser fileChooser =
                              new JFileChooser("D:\\");
      private JButton openBtn = new JButton("Open");
      private JButton resultBtn = new JButton("Result");
      private JButton clearBtn = new JButton("Clear");
      private JTextArea textArea = new JTextArea();
      private ArrayList<int[]> arrays =
                              new ArrayList<int[]>();

      private ArraysWork() {
            Container contentPane = getContentPane();
            contentPane.setLayout(null);
            JPanel col = new JPanel();
            col.setBounds(10, 10, 90, 110);
            col.setLayout(new GridLayout(3, 1, 0, 8));
            openBtn.addActionListener(
                              new ButtonListener());
            resultBtn.addActionListener(
                              new ButtonListener());
            clearBtn.addActionListener(
                              new ButtonListener());
            col.add(openBtn);
            col.add(resultBtn);
            col.add(clearBtn);
            contentPane.add(col, BorderLayout.EAST);
            textArea.setBounds(130, 10, 110, 110);
            textArea.setBorder(new TextFieldBorder());
            textArea.setFont(
                  new Font("Courier New", Font.PLAIN, 12));
            contentPane.add(textArea);
      }
      private class ButtonListener
                              implements ActionListener {
            public void actionPerformed(ActionEvent event)
      {
                  if (event.getSource() == clearBtn)
                        textArea.setText("");
                  else if (event.getSource() == resultBtn)
                        viewArrays(true);
```

```
            else if (event.getSource() == openBtn)
                  readArraysFromFile();
      }
}
public static void main(String[] args) {
      ArraysWork frame = new ArraysWork();

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setBounds(200, 100, 280, 155);
      frame.setVisible(true);
}

private void readArraysFromTextArea() {
      arrays.clear();
      // обработка строк текстового поля
for (int i = 0; i < textArea.getLineCount(); i++)
try {
int startOffset = textArea.getLineStartOffset(i);
int endOffset = textArea.getLineEndOffset(i);
StringTokenizer str =
      new StringTokenizer(textArea.getText(
      startOffset, endOffset - startOffset), " \n");
arrays.add(new int[str.countTokens()]);
int k = 0;
while (str.hasMoreTokens())
arrays.get(i)[k++] =
      new Integer(str.nextToken()).intValue();
} catch (BadLocationException e) {
                  e.printStackTrace();
} catch (NumberFormatException e) {
      JOptionPane.showMessageDialog(
            null, "Enter only numbers!",
      "Fatal error!", JOptionPane.ERROR_MESSAGE);
            }
}

private void readArraysFromFile() {
      String filePath = null;
      int rVal =
            fileChooser.showOpenDialog(fileChooser);
      fileChooser.setVisible(true);
      if (JFileChooser.APPROVE_OPTION == rVal)
            filePath = fileChoos-
            er.getSelectedFile().getPath();
      if (filePath == null)
            return;
      try {
            Scanner sc = new Scanner(
                              new File(filePath));
            int i = 0;
```

```
                while (sc.hasNextLine()) {
                StringTokenizer str =
                      new StringTokenizer(sc.nextLine());
                arrays.add(new int[str.countTokens()]);
                int k = 0;
                while (str.hasMoreTokens())
                arrays.get(i)[k++] =
                new Integer(str.nextToken()).intValue();
                i++;
                }
          } catch (FileNotFoundException e) {
                e.printStackTrace();
          }
          viewArrays(false);
    }

    private void viewArrays(boolean result) {
          if (result) {
                readArraysFromTextArea();
                handlingOfArrays();
          }
          textArea.setText("");
          for (int i = 0; i < arrays.size(); i++) {
                Formatter fmt = new Formatter();
          for (int j = 0; j < arrays.get(i).length; j++)
                fmt.format("%4d", arrays.get(i)[j]);
                textArea.append(fmt.toString() + '\n');
          }
    }

    void handlingOfArrays() {
          for (int i = 0; i < arrays.size(); i++)
          try {
          arrays.get(i)[0] = -arrays.get(i)[0];
          } catch (ArrayIndexOutOfBoundsException e) {
                throw new AssertionError("не матрица!");
          }
    }
}
```

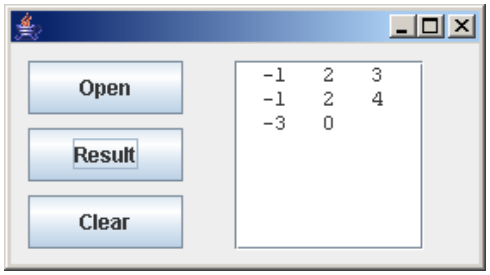


Рис. 13.20. Текстовое поле

Добавить полосу прокрутки можно с помощью следующего кода:

```
JScrollPane sp = new JScrollPane(textarea);
contentPane.add(sp, BorderLayout.CENTER);
```

Компонент **JTable** предназначен для отображения таблицы. Таблицы применяются для отображения связанной информации, например: ФИО, Дата рождения, Адрес, Образование и т. д. Класс **JTable** объявляет конструктор, добавляющий двумерную модель объектов в модель. В примере рассмотрен процесс создания, заполнения и редактирования простой таблицы из двух столбцов.

```
/* пример # 18 : простая таблица:  JTableDemo.java */
package chapt13;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;

public class JTableDemo extends JFrame {

   static ArrayList<Object[]> list =
            new ArrayList<Object[]>();
   JTable table;
   DataModel dataModel = new DataModel();

   JTableDemo() {
      dataModel.addRow("Агеев", 5);
      dataModel.addRow("Смирнов", 2);
      table = new JTable(dataModel);
      getContentPane().add(new JScrollPane(table));
      JButton insertBtn = new JButton("Добавить строку");
      insertBtn.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            dataModel.addRow("", 0);
         }
      });
      getContentPane().add(insertBtn, BorderLayout.SOUTH);
   }
   public static void main(String[] args) {
      JTableDemo frame = new JTableDemo();
      frame.setBounds(100, 100, 300, 300);
      frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

```
class DataModel extends AbstractTableModel {

   private final String[] HEADERS =
      {"Студент", "Оценка"};
   private final int COLS = 2;
   private int rows = 0;

   public String getColumnName(int col) {
      return HEADERS[col];
   }
   public int getRowCount() {
      return rows;
   }
   public int getColumnCount() {
      return COLS;
   }
   public Object getValueAt(int row, int col) {
      return JTableDemo.list.get(row)[col];
   }
   public void setValueAt(Object val, int row, int col) {
      if (col == 1){
            int mark = new Integer(val.toString());
            if (mark > 10 || mark < 0){
JOptionPane.showMessageDialog(null, "Введите число от 0 до
10", "Ошибка ввода!", JOptionPane.ERROR_MESSAGE);
                  return;
            }
      }
      JTableDemo.list.get(row)[col] = val.toString();
      fireTableDataChanged(); //обновление таблицы после изменений
      }
      public boolean isCellEditable(int row, int col) {
            return true;
      }
      public void addRow(String name, int mark) {
            JTableDemo.list.add(new String[COLS]);
            setValueAt(name, rows, 0);
            setValueAt(mark, rows, 1);
            rows++;
      }
      public Class getColumnClass(int col) {
            switch (col) {
            case 0: return String.class;
            case 1: return Integer.class;
            }
            return null;
      }
}
```

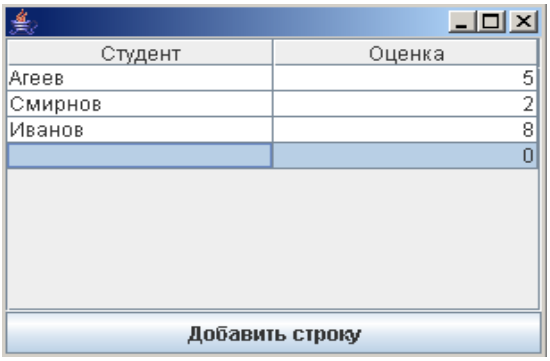


Рис. 13.21. Объект JTable

При попытке добавления в поле «Оценка» значений, выходящих за границы диапазона, генерируется исключительная ситуация, а при попытке внесения нецифровых символов поле подсвечивается красным цветом и добавление не производится.

Компонент **JSplitPane** показывает панель, разделенную на две области либо по вертикали, либо по горизонтали. Разграничительную полосу между ними можно перетаскивать, распределяя долю видимой площади для каждой области. Следующий пример показывает простейшее использование контейнерного класса **JSplitPane**. В верхней части содержится текстовое окно, а в нижней – текстовое поле.

```
/* пример # 19 : простое окно с разграничительной полосой:  DemoJSplit.java */
package chapt13;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class DemoJSplit extends JFrame {
      private JSplitPane tabs;
      private JTextArea area;
      private JPanel tab;
   public DemoJSplit() {
      tabs = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
      area =
          new JTextArea("Текстовое окно в верхней области");
      // установка текстового окна в верхнюю область
      tabs.setTopComponent(area);
      tab = new JPanel();
tab.add(new JTextField("Текстовое поле в нижней области"));
      // установка текстового поля в нижнюю область
      tabs.setBottomComponent(tab);
      tabs.setDividerLocation(130);
      setContentPane(tabs); // установка в окно фрейма компоненты tabs
   }
```

```
    public static void main(String[] args) {
        DemoJSplit dspl = new DemoJSplit();
        dspl.setBounds(200, 200, 250, 200);
        dspl.setVisible(true);
    }
}
```

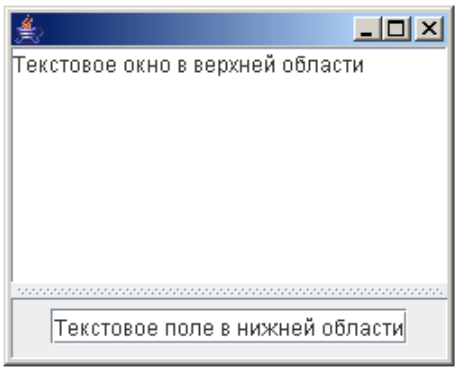


Рис. 13.22. Объект JSplitPane

Контейнерный класс **JTabbedPane** представляет собой окно с закладками, с помощью которых можно выбирать желаемый компонент. Закладки могут иметь всплывающие подсказки, а также содержать как текст, так и значки.

В следующем примере показано простое приложение, использующее окно с закладками.

```
/* пример # 20 : простое окно с двумя закладками:  DemoJTabbed.java */
package chapt13;
import java.awt.Container;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.JToggleButton;

public class DemoJTabbed extends JFrame {
      JTabbedPane tabs;
      JPanel pan1, pan2;
      public DemoJTabbed() {
            Container c = getContentPane();
            tabs = new JTabbedPane();
            pan1 = new JPanel();
            pan1.add(new JToggleButton("Button"));
            tabs.addTab("One", pan1); //добавление первой закладки
            pan2 = new JPanel();
            pan2.add(new JCheckBox("CheckBox"));
            tabs.addTab("Two", pan2);//добавление второй закладки
            c.add(tabs);
      }
```

```
    public static void main(String[] args) {
          DemoJTabbed dt = new DemoJTabbed();
          dt.setSize(250, 150);
          dt.setLocation(200, 200);
          dt.setVisible(true);
    }
}
```

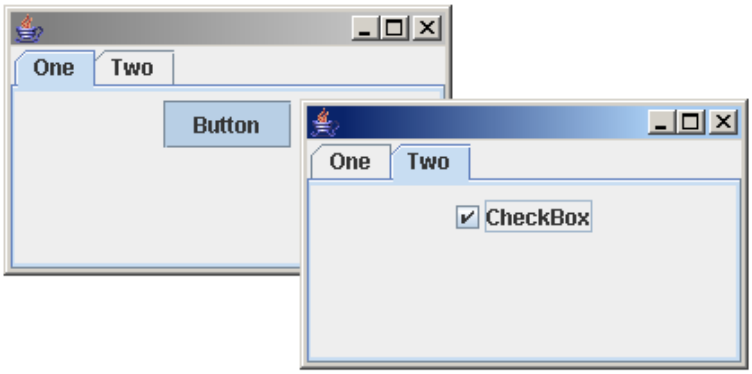


Рис. 13.23. Окно с закладками

В Java 6 стало больше графических возможностей по взаимодействию с рабочим столом Windows и интернетом. Класс **`java.awt.SystemTray`** обеспечивает прямой доступ к рабочему столу пользователя: позволяет добавлять иконки, советы для кнопок и спадающие меню к системным областям состояниям Windows.

```
/* пример # 21 : демонстрация SystemTray:  DemoTray.java */
package chapt13;
import java.awt.*;

public class DemoTray {
   public static void main(String[] args) {
      if(SystemTray.isSupported()) {
         SystemTray tray = SystemTray.getSystemTray();
         Image image =
            Toolkit.getDefaultToolkit().getImage("icon");
         TrayIcon icon = new TrayIcon(image, "Demo Tray");
         try {
            tray.add(icon);
         } catch (AWTException e) {
            System.err.println(e);
         }
      } else {
         System.err.println("Without system tray!");
      }
   }
}
```

Возможности класса **java.awt.Desktop** позволяют запустить браузер с помощью одной строки:

```
/* пример # 22 : запуск браузера:  DesktopTest.java */
package chapt13;
import org.jdesktop.jdic.desktop.*;
import java.net.*;

public class DesktopTest {
  public static void main(String[] args) throws Exception {
    Desktop.browse(new URL("http://java.sun.com"));
  }
}
```

При необходимости можно легко использовать веб-браузер в приложении:

```
/* пример # 23 : использование web- браузера:  BrowserTest.java */
package chapt13;
import org.jdesktop.jdic.browser.WebBrowser;
import java.net.URL;
import javax.swing.JFrame;

public class BrowserTest {
  public static void main(String[] args) throws Exception {

    WebBrowser browser = new WebBrowser();
    browser.setURL(new URL("http://www.netbeans.org"));
    JFrame frame = new JFrame("Использование Browser");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(browser);
    frame.pack();
    frame.setSize(640,480);
    frame.setVisible(true);
  }
}
```

Визуальные компоненты JavaBeans

Визуальные компоненты являются классами языка Java, объекты которых отображаются визуально в проектируемом приложении с помощью средств визуальной разработки. Визуальные компоненты являются удобным средством при создании пользовательских интерфейсов. Обычно часть визуальной разработки приложений состоит в перетаскивании компонентов на форму и в определении их свойств, событий и обработчиков событий. При этом компонент представляет собой объект класса, для которого, кроме данных и методов, дополнительно установлены свойства и события класса. Свойства и события устанавливаются через имена методов на основе соглашения об именовании методов.

JavaBeans (бин) – многократно используемый программный компонент, которым можно манипулировать визуально при создании приложения. Бин реализуется в одном или нескольких взаимосвязанных классах. Основной класс бина должен иметь конструктор по умолчанию, который вызывается визуальной средой при создании экземпляра бина.

Каждый бин должен поддерживать следующие возможности:

- интроспекцию, позволяющую средам разработки анализировать, из чего состоит и как работает данный бин;
- поддержку событий (**events**) ;
- поддержку свойств (**properties**) ;
- сохраняемость (**persistence**).

Каждый бин должен быть сериализуемым. Визуальная среда при сохранении скомпилированного приложения сохраняет настройки компонента, сделанные пользователем в процессе разработки приложения путем сериализации бина. При повторной загрузке приложения эти настройки восстанавливаются. Для этого среда разработки десериализует бины из файла.

Свойства бинов

Каждый бин имеет свойства (**properties**), которые определяют, как он будет работать и как выглядеть. Эти свойства являются **private** или **protected** полями класса, которые доступны через специальные методы **get**Имя**()** и **set**Имя**()** (getters и setters), называемые также аксессорами. Так, утверждение "данный бин имеет свойство **name** типа **String**" означает, что у этого бина

```
//есть поле
   private String name;
//есть get-метод
   public String getName(){
      return name;
   }
//есть set-метод
   public void setString(String name){
      this.name = name;
   }
```

Пусть рассматривается, к примеру, компонент **JLabel**. Во-первых, **JLabel** удовлетворяет интерфейсу **Serializable**, во-вторых, имеет конструктор по умолчанию **public JLabel()** и, в-третьих, имеет ряд методов аксессоров, например **public String getText()**, **public void setText(String text)**. Исходя из этого, можно сделать выводы, что **JLabel** является бином, имеющим свойство **text**. Значение этого свойства отображается как заголовок метки. Кроме того, **JLabel** имеет и другие свойства.

Для свойства типа **boolean** в бинах вместо **get**-методов может быть использован **is**-метод. Например, **JLabel** имеет **boolean**-свойство **enabled**, унаследованное от класса **Component**. Для доступа к этому свойству имеются методы

```
public boolean isEnabled()
public void setEnabled(boolean b)
```

Правила построения методов доступа к атрибутам (аксессоров):

```
public void setИмяСвойства(ТипСвойства value);
public ТипСвойства getИмяСвойства();
public boolean isИмяСвойства().
```

Свойства бинов могут быть как базовых типов, так и объектными ссылками. Свойства могут быть индексированными, если атрибут бина массив. Для индексированных свойств выработаны следующие правила. Они должны быть описаны как поля-массивы, например

```
private String[] messages;
```

и должны быть объявлены следующие методы:

```
public ТипСвойства getИмяСвойства(int index);
public void setИмяСвойства(int index, ТипСвойства value);
public ТипСвойства [] getИмяСвойства();
public void setИмяСвойства(ТипСвойства [] value).
```

Так, для приведенного выше примера должны быть методы

```
public String getMessages(int index);
public void setMessages(int index, String message);
public String[] getMessages();
public void setMessages(String[] messages).
```

Кроме аксессоров, бин может иметь любое колличество других методов.

Интроспекция бинов при помощи Reflection API

Под интроспекцией понимается процесс анализа bean-компонента для установления его возможностей, свойств и методов. Для интроспекции можно воспользоваться классами и методами из библиотеки Reflection API. При использовании бина визуальная среда должна знать полное имя класса бина. По строковому имени класса статический метод **`forName(String className)`** класса **`java.lang.Class`** возвращает объект класса **`Class`**, соответствующий данному бину. Далее с помощью метода класса **`Class getField()`**, **`getMethods()`**, **`getConstructors()`** можно получить необходимую информацию о свойствах и событиях класса.

В частности, можно получить список всех **`public`**-методов данного класса. Исследуя их имена, можно выделить из них аксессоры и определить какие атрибуты (свойства) есть у данного бина и какого они типа. Все остальные методы, не распознанные как аксессоры, являются bean-методами.

В результате соответствующая визуальная разработки может построить диалог, в котором будет предоставлена возможность задавать значения этих атрибутов. Наличие конструктора по умолчанию позволяет построить объект bean-класса, **`set`**-методы позволяют установить в этом объекте значения атрибутов, введенные пользователем, а благодаря сериализации объект с заданными атрибутами можно сохранить в файл и востановить значение объекта при следующем сеансе работы с данной визуальной средой. Более того, можно изобразить на экране внешний вид бина (если это визуальный бин) в процессе разработки и менять этот вид в соответствии с задаваемыми пользователем значениями атрибутов.

События

Еще одним важным аспектом технологии JavaBeans является возможность бинов взаимодествовать с другими объектами, в частности, с другими бинами. JavaBeans реализует такое взаимодействие путем генерации и прослушивания событий.

В приложении к бинам взаимодействие объектов с бином через событийную модель выглядит так. Объект, который интересуется тем, что может произойти во внешнем по отношению к нему бине, может зарегистрировать себя как слушателя (**`Listener`**) этого бина. В результате при возникновении соответствующего события в бине будет вызван определенный метод данного объекта, которому в качестве параметра будет передан объект-событие (**`event`**). Причем если зарегистрировалось несколько слушателей, то эти методы будут последовательно вызваны для каждого слушателя.

Такой механизм взаимодействия является очень гибким, поскольку два объекта – бин и его слушатель – связаны только посредством данного метода и параметра-события.

Одним из способов экспорта событий является использование связанных свойств. Когда значение связанного свойства меняется, генерируется событие и передается всем зарегистрированным слушателям посредством вызова метода **`proertyChange()`**.

Создание и использование связанного свойства

Разберемся практически, как создавать и использовать связанные свойства. Начнем с события, которое должно быть сгенерированно при изменении связанного свойства. Это событие класса **`java.beans.PropertyChangeEvent`** (см. документацию).

Далее можно действовать по следующей инструкции.

1. Для регистрации/дерегистрации слушателя необходимо в бине реализовать два метода:
`addPropertyChangeListener(PropertyChangeListener p)` и **`removePropertyChangeListener(PropertyChangeListener p)`**;
2. Чтобы не реализовывать их вручную, лучше воспользоваться существующим классом **`java.beans.PropertyChangeSupport`** (см. документацию) ;
3. В **`set`**-методе связанного свойства необходимо добавить вызов метода **`firePropertyChange()`** класса **`java.beans.PropertyChangeSupport`**;
4. В классе-слушателе реализовать интерфейс **`PropertyChangeListener`**, т.е. в заголовке класса записать "**`implements PropertChangeListener`**", а в теле класса реализовать метод **`public void propertyChange(PropertyChangeEvent evt)`**;
5. Создать объект-слушатель и зарегистрировать его как слушателя нашего бина при помощи метода **`addPropertyChangeListener()`**, который был реализован в п.1. Лучше всего это сделать сразу после порождения объекта-слушателя, например:

```
MyListener obj = new MyListener();
myBean.addPropertyChangeListener(obj);
```

где **`myBean`** – создаваемый бин (имеется в виду объект, а не класс).

Пункт 4-й должен быть реализован для каждого класса-слушателя, а п.5 – для каждого порожденного объекта-слушателя.

Следует разобрать подробнее пункты 2 и 3.

Сейчас необходимо реализовать генерацию событий. Бин должен генерировать событие **`PropertyChangeEvent`** при изменении связанного свойства (п.3). Кроме того, согласно правилам событийной модели Java он

должен обеспечивать регистрацию/дерегистрацию слушателей при помощи соответствующих методов **add**...**Listener/remove**...**Listener** (п.2).

Т.е. нужно обеспечить наличие в бине некоторого списка слушателей, а также методы **addPropertyChangeListener()** и **removePropertyChangeListener()**.

К счастью, не требуется программировать все это. Соответствующий инструментарий уже подготовлен в пакете **java.beans** – это класс **java.beans.PropertyChangeSupport**. Он обеспечивает регистрацию слушателей и методы **firePropertyChange()**, которые можно использовать в тех местах, где требуется сгенерировать событие, т.е. в **set**-методах, которые изменяют значение связанных атрибутов.

Предложенный механизм будет рассмотрен в следующем примере.

Пусть имеется некоторый бин **SomeBean** с одним свойством **someProperty**:

```
/* пример # 24 : простой bean-класс :  SomeBean.java */
package chapt13;
public class SomeBean{
      private String someProperty = null;
      public SomeBean(){
      }
      public String getSomeProperty(){
            return someProperty;
      }
      public void setSomeProperty(String value){
            someProperty = value;
      }
}
```

Переделаем его так, чтобы свойство **someProperty** стало связанным:

```
/* пример # 25 : bean-класс со связанным свойством:  SomeBean.java */
import java.beans.*;
     public class SomeBean{
            private String someProperty = null;
            private PropertyChangeSupport pcs;
            public SomeBean(){
              pcs = new PropertyChangeSupport(this);
            }
            public void addPropertyChangeListener
            (PropertyChangeListener pcl){
                  pcs.addPropertyChangeListener(pcl);
            }
            public void removePropertyChangeListener
            (PropertyChangeListener pcl){
                 pcs.removePropertyChangeListener(pcl);
            }
            public String getSomeProperty(){
                  return someProperty;
            }
```

```
        public void setSomeProperty(String value){
              pcs.firePropertyChange("someProperty",
              someProperty, value);
              someProperty = value;
        }
}
```

Здесь реализованы пункты 1, 2 и 3 приведенной инструкции. Остальные пункты относятся к использованию связанного свойства, и для их демонстрации потребуется более реальный пример.

Для обеспечения механизма генерации событий в классе **SomeBean** создан объект класса **PropertyChangeSupport** (поле **pcs**). И все действия по регистрации/дерегистрации слушателей по собственно генерации событий "переадресуются" этому объекту, который за нас выполняет всю эту рутинную работу.

Так, например, метод **addPropertyChangeListener(PropertyChangeListener pcl)** созданного класса просто обращается к одноименному методу класса **PropertyChangeSupport**. В методе **setSomeProperty()** перед собственно изменением значения свойства someProperty генерируется событие **PropertyChangeEvent**. Для этого вызывается метод **firePropertyChange()**, который обеспечивает все необходимые для такой генерации действия.

Как видно из кода примера, результат не очень громоздкий, несмотря на то, что наш бин реализует достаточно сложное поведение.

Ограниченные свойства (contrained properties)

Кроме понятия связанных свойств, в JavaBeans есть понятие ограниченных свойств (contrained properties). Ограниченные свойства введены для того, чтобы была возможность запретить изменение свойства бина, если это необходимо. Т.е. бин будет как бы спрашивать разрешения у зарегистрированных слушателей на изменение данного свойства. В случае если слушатель не разрешает ему менять свойство, он генерирует исключение **PropertyVetoException.** Соответственно **set**-метод для ограниченного свойства должен иметь в своем описании **throws PropertyVetoException**, что заставляет перехватывать это исключение в точке вызова данного **set**-метода. В результате прикладная программа, использующая этот бин, будет извещена, что ограниченное свойство не было изменено.

В остальном ограниченные свойства очень похожи на связанные свойства. Как и все свойства, они имеют **get**- и **set**-методы. Но для них **set**-методы могут генерировать исключение **PropertyVetoException** и имеют вид **public void <PropertyName>**(ТипСвойства **param**) **throws PropertyVetoException.**

Второе отличие заключается в именах методов для регистрации/дерегистрации слушателей. Вместо методов **addPropertyChangeListener()** и **removePropertyChangeListener()** для ограниченных свойств применяются методы **addVetoableChangeListener(VetoableChangeListener v)** и

removeVetoableChangeListener(VetoableChangeListener v). Здесь **VetoableChangeListener** – интерфейс с одним методом **void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException()**.

По аналогии со вспомогательным классом **PropertyChangeSupport**, который используется при реализации связанных свойств, для ограниченных свойств в пакете **java.beans** есть вспомогательный класс **VetoableChangeSupport**. В нем реализованы алгоритмы, необходимые для поддержки событий ограниченных свойств.

В качестве примера вспомним класс **SomeBean**, рассмотренный ранее. Его свойство **someProperty()** реализовано как связанное. Переделаем пример и реализуем это свойство как ограниченное.

```
/* пример # 26 : bean-класс с ограниченным свойством :  SomeBean.java */
import java.beans.*;
     public class SomeBean {
            private String someProperty = null;
            private VetoableChangeSupport vcs;
            public SomeBean(){
              vcs = new VetoableChangeSupport(this);
            }
            public void addVetoableChangeListener
            (VetoableChangeListener pcl){
              vcs.addVetoableChangeListener(pcl);
            }
            public void removeVetoableChangeListener
            (VetoableChangeListener pcl){
              pcs.removePropertyChangeListener(pcl);
            }

            public String getSomeProperty(){
                  return someProperty;
            }
            public  void  setSomeProperty(String  value)
            throws
            PropertyVetoException{
                 vcs.fireVetoableChange("someProperty",
                 someProperty, value);
                 someProperty = value;
            }
     }
```

Как видно, принципиально ничего не изменилось. Только вместо **PropertyChangeSupport** использован **VetoableChangeSupport** и в описании **set**-метода добавлено **throws PropertyVetoException**. Теперь **someProperty** является ограниченным свойством, и зарегистрировавшийся слушатель может запретить его изменение.

Рассмотренные возможности организации связи бина с другими компонентами не являются единственно возможными. Бин, как и любой класс,

может быть источником событий и/или слушателем. И эти события могут быть не связаны с изменением свойств бина.

В таких случаях обычно используют существующие события типа **ActionEvent**, хотя можно построить и свои события.

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 переносился в поле вывода, а поле ввода очищалось.
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 он не 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 ВЫПОЛНЕНИЯ

Класс Thread и интерфейс Runnable

К большинству современных распределенных приложений (Rich Client) и Web-приложений (Thin Client) выдвигаются требования одновременной поддержки многих пользователей, каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов. Потоки – средство, которое помогает организовать одновременное выполнение нескольких задач, каждую в независимом потоке. Потоки представляют собой классы, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существуют два способа создания и запуска потока: расширение класса **Thread** или реализация интерфейса **Runnable**.

```
// пример # 1 : расширение класса Thread: Talk.java
package chapt14;

public 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.err.print(e);
               }
          }
     }
}
```

При реализации интерфейса **Runnable** необходимо определить его единственный абстрактный метод **run()**. Например:

```
/* пример # 2 : реализация интерфейса Runnable: Walk.java: WalkTalk.java */
package chapt14;

public class Walk implements Runnable {
     public void run() {
          for (int  i = 0; i < 8; i++) {
               System.out.println("Walking");
               try {
                    Thread.sleep(400);
```

```
                } catch (InterruptedException e) {
                    System.err.println(e);
                }
            }
        }
}
package chapt14;

public class WalkTalk {
    public static void main(String[] args) {
        // новые объекты потоков
        Talk talk  = new Talk();
        Thread walk = new Thread(new Walk());
        // запуск потоков
        talk.start();
        walk.start();

        //Walk w = new Walk(); // просто объект, не поток
        // w.run(); //выполнится метод, но поток не запустится!
    }
}
```

Использование двух потоков для объектов классов **Talk** и **Walk** приводит к выводу строк: `Talking Walking`. Порядок вывода, как правило, различен при нескольких запусках приложения.

Жизненный цикл потока

При выполнении программы объект класса **Thread** может быть в одном из четырех основных состояний: “новый”, “работоспособный”, “неработоспособный” и “пассивный”. При создании потока он получает состояние “новый” (**NEW**) и не выполняется. Для перевода потока из состояния “новый” в состояние “работоспособный” (**RUNNABLE**) следует выполнить метод **start()**, который вызывает метод **run()** – основной метод потока.

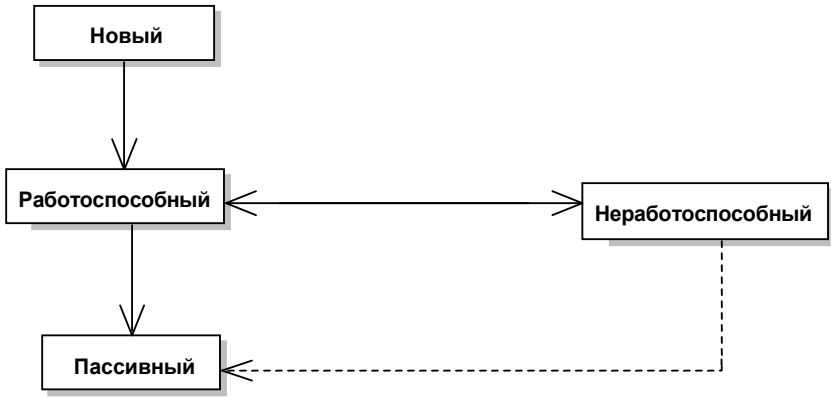


Рис. 14.1. Состояния потока

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления **Thread.State**:

NEW – поток создан, но еще не запущен;

RUNNABLE – поток выполняется;

BLOCKED – поток блокирован;

WAITING – поток ждет окончания работы другого потока;

TIMED_WAITING – поток некоторое время ждет окончания другого потока;

TERMINATED — поток завершен.

Получить значение состояния потока можно вызовом метода **getState()**.

Поток переходит в состояние "неработоспособный" (**WAITING**) вызовом методов **wait()**, **suspend()** (deprecated-метод) или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания (**TIMED_WAITING**) с помощью методов **sleep(long millis)** и **wait(long timeout)**, при выполнении которого может генерироваться прерывание **InterruptedException**. Вернуть потоку работоспособность после вызова метода **suspend()** можно методом **resume()** (deprecated-метод), а после вызова метода **wait()** – методами **notify()** или **notifyAll()**. Поток переходит в "пассивное" состояние (**TERMINATED**), если вызваны методы **interrupt()**, **stop()** (deprecated-метод) или метод **run()** завершил выполнение. После этого, чтобы запустить поток еще раз, необходимо создать новый объект потока. Метод **interrupt()** успешно завершает поток, если он находится в состоянии "работоспособность". Если же поток неработоспособен, то метод генерирует исключительные ситуации разного типа в зависимости от способа остановки 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.

Методы **suspend()**, **resume()** и **stop()** являются deprecated-методами и запрещены к использованию, так как они не являются в полной мере "потокобезопасными".

Управление приоритетами и группы потоков

Потоку можно назначить приоритет от **1** (константа **MIN_PRIORITY**) до **10** (**MAX_PRIORITY**) с помощью метода **setPriority(int prior)**. Получить значение приоритета можно с помощью метода **getPriority()**.

// пример # 3 : демонстрация приоритетов: PriorityRunner.java: PriorThread.java

```
package chapt14;

public class PriorThread extends Thread {
     public PriorThread(String name){
          super(name);
     }
     public void run(){
```

```
            for (int i = 0; i < 71; i++){
                System.out.println(getName() + " " + i);
                try {
                    sleep(1);//попробовать sleep(0);
                } catch (InterruptedException e) {
                    System.err.print("Error" + e);
                }
            }
        }
}
package chapt14;

public class PriorityRunner {
    public static void main(String[] args) {
            PriorThread min = new PriorThread("Min");//1
            PriorThread max = new PriorThread("Max");//10
            PriorThread norm = new PriorThread("Norm");//5
            min.setPriority(Thread.MIN_PRIORITY);
            max.setPriority(Thread.MAX_PRIORITY);
            norm.setPriority(Thread.NORM_PRIORITY);
            min.start();
            norm.start();
            max.start();
    }
}
```

Поток с более высоким приоритетом в данном случае, как правило, монополизирует 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()**. Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения значение его приритета станет равным приоритету группы. Поток же со значением приоритета более низким, чем приоритет группы после включения в оную, значения своего приоритета не изменит.

Управление потоками

Приостановить (задержать) выполнение потока можно с помощью метода **sleep(**время задержки**)** 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 потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока.

// пример # 4 : задержка потока: JoinRunner.java

```
package chapt14;

class Th extends Thread {
     public Th(String str) {
          super();
          setName(str);
     }
     public void run() {
          String nameT = getName();
          System.out.println("Старт потока " + nameT);
          if ("First".equals(nameT)) {
               try {
                    sleep(5000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
               System.out.println("завершение потока "
                                                  + nameT);
          } else if ("Second".equals(nameT)) {
               try {
                    sleep(1000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
             System.out.println("завершение потока "
                                                  + nameT);
          }
     }
}
public class JoinRunner {
     public static void main(String[] args) {
          Th tr1 = new Th("First");
          Th tr2 = new Th("Second");
          tr1.start();
          tr2.start();
          try {
             tr1.join();
             System.out.println("завершение main");
          } catch (InterruptedException e){
                    e.printStackTrace();
          }
          /* join() не дает работать потоку main до окончания выполнения
          потока tr1 */
     }
}
```

Возможно, будет выведено:

```
Старт потока First
```

```
Старт потока Second
завершение потока Second
завершение потока First
завершение main
```

Несмотря на вызов метода **join()** для потока **tr1**, поток **tr2** будет работать, в отличие от потока **main**, который сможет продолжить свое выполнение только по завершении потока **tr1**.

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 # 5 : задержка потока: YieldRunner.java

```
package chapt14;

public class YieldRunner {
     public static void main(String[] args) {
          new Thread() {
             public void run() {
                    System.out.println("старт потока 1");
                    Thread.yield();
                    System.out.println("завершение 1");
               }
          }.start();
          new Thread() {
             public void run() {
                    System.out.println("старт потока 2");
                    System.out.println("завершение 2");
               }
          }.start();
     }
}
```

В результате может быть выведено:

```
старт потока 1
старт потока 2
завершение 2
завершение 1
```

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

Потоки-демоны работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность за-

ключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon(boolean value)**, вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **boolean isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

```
/* пример # 6 : запуск и выполнение потока-демона: DemoDaemonThread.java */
package chapt14;

class T extends Thread {
     public void run() {
          try {
               if (isDaemon()){
                System.out.println("старт потока-демона");
                sleep(10000); // заменить параметр на 1
               } else {
               System.out.println("старт обычного потока");
               }
          } catch (InterruptedException e) {
               System.err.print("Error" + e);
          } finally {
               if (!isDaemon())
                    System.out.println(
                      "завершение обычного потока");
               else
                    System.out.println(
                      "завершение потока-демона");
          }
     }
}
package chapt14;

public class DemoDaemonThread {
     public static void main(String[] args) {
          T usual = new T();
          T daemon = new T();
          daemon.setDaemon(true);
          daemon.start();
          usual.start();
          System.out.println(
                     "последний оператор main");
     }
}
```

В результате компиляции и запуска, возможно, будет выведено:

```
последний оператор main
старт потока-демона
старт обычного потока
завершение обычного потока
```

Поток-демон (из-за вызова метода **sleep(10000)**) 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 графических приложениях

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 графики в новой позиции.

/ пример # 7 : освобождение ресурсов апплетом: GraphicThreadsDemo.java */*

```
package chapt14;
import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GraphicThreadsDemo extends JFrame {
      JPanel panel = new JPanel();
      Graphics g;
      JButton btn = new JButton("Добавить шарик");
      int i;

      public GraphicThreadsDemo() {
            setBounds(100, 200, 270, 350);
            Container contentPane = getContentPane();
            contentPane.setLayout(null);
            btn.setBounds(50, 10, 160, 20);
            contentPane.add(btn);
            panel.setBounds(30, 40, 200, 200);
            panel.setBackground(Color.WHITE);
            contentPane.add(panel);
            btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                        new BallThread(panel).start();
                        i++;
                        repaint();
                  }
```

```
            });
      }
      public static void main(String[] args) {
            GraphicThreadsDemo frame =
                        new GraphicThreadsDemo();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
      }
      public void paint(Graphics g){
            super.paint(g);
         g.drawString("Количество шариков: " + i, 65, 300);
      }
}
class BallThread extends Thread {
      JPanel panel;
      private int posX, posY;
      private final int BALL_SIZE = 10;
      private double alpha;
      private int SPEED = 4;

      BallThread(JPanel p) {
            this.panel = p;
            //задание начальной позиции и направления шарика
            posX = (int)((panel.getWidth() - BALL_SIZE)
                                    * Math.random());
            posY = (int)((panel.getHeight() - BALL_SIZE)
                                    * Math.random());
            alpha = Math.random() * 10;
      }
      public void run() {
            while(true) {
                  posX += (int)(SPEED * Math.cos(alpha));
                  posY += (int)(SPEED * Math.sin(alpha));
            //вычисление угла отражения
            if( posX >= panel.getWidth() - BALL_SIZE )
                  alpha = alpha + Math.PI - 2 * alpha;
            else if( posX <= 0 )
                        alpha = Math.PI - alpha;
            if( posY >= panel.getHeight() - BALL_SIZE )
                        alpha = -alpha;
            else if( posY <= 0 )
                        alpha = -alpha;
                  paint(panel.getGraphics());
            }
      }
      public void paint(Graphics g) {
      //прорисовка шарика
      g.setColor(Color.BLACK);
```

```
        g.fillArc(posX, posY, BALL_SIZE, BALL_SIZE, 0, 360);
        g.setColor(Color.WHITE);
        g.drawArc(posX + 1, posY + 1, BALL_SIZE,
                                BALL_SIZE, 120, 30);
              try {
                    sleep(30);
              } catch(InterruptedException e) {
                    e.printStackTrace();
              }
        //удаление шарика
        g.setColor(panel.getBackground());
        g.fillArc(posX, posY, BALL_SIZE, BALL_SIZE, 0, 360);
        }
}
```

Рис.14.2.Потоки в апплетах

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 несколько потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу; более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток. Для предотвращения такой ситуации может использоваться ключевое слово **`synchronized`**. Синхронизации не требуют только атомарные процессы по записи/чтению, не превышающие по объему 32 бит.

В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе **main()** класса **SynchroThreads** создаются два потока. В этом же методе создается экземпляр класса **Synchro**, содержащий поле типа **FileWriter**, связанное с файлом на диске. Экземпляр **Synchro** передается в качестве параметра обоим потокам. Первый поток записывает строку методом **writing()** в экземпляр класса **Synchro**. Второй поток также пытается сделать запись строки в тот же самый объект **Synchro**. Для избежания одновременной записи такие методы объявляются как **synchronized**. Синхронизированный метод изолирует объект, 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 # 8 : синхронизация записи информации в файл : MyThread.java :
Synchro.java : SynchroThreads.java */
package chapt14;
import java.io.*;

public class Synchro {
      private FileWriter fileWriter;

      public Synchro(String file) throws IOException {
            fileWriter = new FileWriter(file, true);
            }
      public void close() {
            try {
                  fileWriter.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
      public synchronized void writing(String str, int i) {
                  try {
                        System.out.print(str + i);
                        fileWriter.append(str + i);
            Thread.sleep((long)(Math.random() * 50));
                        System.out.print("->" + i + " ");
                        fileWriter.append("->" + i + " ");
                  } catch (IOException e) {
                        System.err.print("ошибка файла");
                        e.printStackTrace();
                  } catch (InterruptedException e) {
                        System.err.print("ошибка потока");
                        e.printStackTrace();
                  }
      }
}
```

```
package chapt14;

public class MyThread extends Thread {
      private Synchro s;

      public MyThread(String str, Synchro s) {
            super(str);
            this.s = s;
      }
      public void run() {
            for (int i = 0; i < 5; i++) {
                  s.writing(getName(), i);
            }
      }
}
package chapt14;
import java.io.*;

public class SynchroThreads {
      public static void main(String[] args) {
            try {
            Synchro s = new Synchro("c:\\temp\\data.txt");

            MyThread t1 = new MyThread("First", s);
            MyThread t2 = new MyThread("Second", s);
            t1.start();
            t2.start();
            t1.join();
            t2.join();
                  s.close();
            } catch (IOException e) {
                  System.err.print("ошибка файла");
                  e.printStackTrace();
            } catch (InterruptedException e) {
                  System.err.print("ошибка потока");
                  e.printStackTrace();
            }
      }
}
```

В результате в файл будет выведено:

First0->0 Second0->0 First1->1 Second1->1 First2->2 Second2->2 First3->3 Second3->3 First4->4 Second4->4

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.

Вывод в этом случае может быть, например, следующим:

First0Second0->0 Second1->0 First1->1 First2->1 Second2->2 First3->3 First4->2 Second3->3 Second4->4 ->4

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 # 9 : блокировка объекта потоком: TwoThread.java */
package chapt14;
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(100);
                          } catch (InterruptedException e) {
                                   System.err.print(e);
                              }
                              System.out.println(s);
                         }
                    }//конец synchronized
               }
          }.start();
          new Thread() {
               public void run() {
                    int j = 0;
                    synchronized (s) {
                         while (j++ < 3) {
                              s.append("B");
                         System.out.println(s);
                    }
                   }//конец synchronized
                   }
          }.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. Эти методы используются для управления потоками в ситуации, когда необходимо задать определенную последовательность действий без повторного запуска потоков.

Метод **wait()**, вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект, в частности объект **lock**. Возвратить блокировку объекта потоку можно вызовом метода **notify()** для конкретного потока или **notifyAll()** для всех потоков. Вызов может быть осуществлен только из другого потока, заблокировавшего, в свою очередь, указанный объект.

```
/* пример # 10 : взаимодействие wait() и notify(): Blocked.java: Runner.java */
package chapt14;

public class Blocked {
      private int i = 1000;

      public int getI() {
            return i;
      }
      public void setI(int i) {
            this.i = i;
      }
      public synchronized void doWait() {
                  try {
                        System.out.print("Не ");
                        this.wait(); /* остановка потока и
                                        освобождение блокировки*/
      System.out.print("сущностей "); // после возврата блокировки
                        Thread.sleep(50);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  for (int j = 0; j < 5; j++) i/=5;
                  System.out.print("сверх ");
            }
      }
package chapt14;

public class Runner {
```

```
    public static void main(String[] args) {
          Blocked lock = new Blocked();
          new Thread() {
                public void run() {
                      lock.doWait();
                }}.start();
          try {
                Thread.sleep(500);
          } catch (InterruptedException e) {
                e.printStackTrace();
          }
          synchronized (lock) {// 1
                lock.setI(lock.getI() + 2);
                System.out.print("преумножай ");
                lock.notify(); // возврат блокировки
          }
          synchronized (lock) {// 2
                lock.setI(lock.getI() + 3);
                //блокировка после doWait()
                System.out.print("необходимого. ");
                try {
                      lock.wait(500);
                } catch (InterruptedException e) {
                      e.printStackTrace();
                }
          }
          System.out.print("=" + lock.getI());
    }
}
```

В результате компиляции и запуска будет выведено следующее сообщение:

Не преумножай сущностей сверх необходимого. =3

Задержки потоков методом **sleep()** используются для точной демонстрации последовательности действий, выполняемых потоками. Если же в коде приложения убрать все блоки синхронизации, а также вызовы методов **wait()** и **notify()**, то вывод может быть следующим:

Не сущностей преумножай необходимого. =1005сверх

Состояния потока

В классе **Thread** объявлено внутреннее перечисление **State**, простейшее применение элементов которого призвано помочь в отслеживании состояний потока в процессе функционирования приложения и, как следствие, в улучшении управления им.

```
/* пример # 11 : состояния NEW, RUNNABLE, TIMED_WAITING, TERMINATED :
ThreadTimedWaitingStateTest.java */
package chapt14;
public class ThreadTimedWaitingStateTest extends Thread {
```

```
   public void run() {
      try {
         Thread.sleep(50);
      } catch (InterruptedException e) {
            System.err.print("ошибка потока");
      }
   }
   public static void main(String [] args){
      try{
         Thread thread = new ThreadTimedWaitingStateTest();
         // NEW – поток создан, но ещё не запущен
         System.out.println("1: " + thread.getState());
         thread.start();
         // RUNNABLE – поток запущен
         System.out.println("2: " + thread.getState());
         Thread.sleep(10);
         // TIMED_WAITING
         // поток ждет некоторое время окончания работы другого потока
         System.out.println("3: " + thread.getState());
         thread.join();
         // TERMINATED –  поток завершил выполнение
         System.out.println("4: " + thread.getState());
      } catch (InterruptedException e) {
            System.err.print("ошибка потока");
      }
   }
}
```

В результате компиляции и запуска будет выведено:

```
1: NEW
2: RUNNABLE
3: TIMED_WAITING
4: TERMINATED
```

```
/* пример # 12 : состояния BLOCKED, WAITING : ThreadWaitingStateTest.java */
package chapt14;
public class ThreadWaitingStateTest extends Thread {

   public void run() {
      try {
         synchronized (this) {
            wait();
         }
      } catch (InterruptedException e) {
            System.err.print("ошибка потока");
      }
   }

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

```
            Thread thread = new ThreadWaitingStateTest();
            thread.start();
            synchronized (thread) {
               Thread.sleep(10);
               // BLOCKED – because thread attempting to acquire a lock
               System.out.println("1: " + thread.getState());
            }
            Thread.sleep(10);
            // WAITING – метод wait() внутри synchronized
            // останавил поток и освободил блокировку
            System.out.println("2: " + thread.getState());
            thread.interrupt();
         } catch (InterruptedException e) {
               System.err.print("ошибка потока");
         }
      }
}
```

В результате компиляции и запуска будет выведено:

```
1: BLOCKED
2: WAITING
```

Потоки в J2SE 5

Java всегда предлагала широкие возможности для мультипрограммирования: потоки – это основа языка. Однако очень часто использование таких возможностей становилось ловушкой: правильно написать и отладить многопоточную программу достаточно сложно.

В версии 1.5 языка добавлены пакеты классов **java.util.concurrent.locks**, **java.util.concurrent.atomic**, **java.util.concurrent**, возможности которых обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков параллельных (concurrent) классов, вызов утилит синхронизации, использование семафоров, ключей и atomic-переменных.

Возможности синхронизации существовали и ранее. Практически это означало, что синхронизированные объекты блокировались, хотя необходимо это было далеко не всегда. Например, поток, изменяющий одну часть объекта **Hashtable**, блокировал работу других потоков, которые хотели прочесть (даже не изменить) совсем другую часть этого объекта. Поэтому введение дополнительных возможностей, связанных с синхронизаций потоков и блокировкой ресурсов довольно логично.

Ограниченно потокобезопасные (thread safe) коллекции и вспомогательные классы управления потоками сосредоточены в пакете **java.util.concurrent**. Среди них можно отметить:

- параллельные классы очередей **ArrayBlockingQueue** (FIFO очередь с фиксированой длиной), **PriorityBlockingQueue** (очередь с приоритетом) и **ConcurrentLinkedQueue** (FIFO очередь с нефиксированой длиной);

- параллельные аналоги существующих синхронизированых классов-коллекций **ConcurrentHashMap** (аналог **Hashtable**) и **CopyOnWriteArrayList** (реализация **List**, оптимизированная для случая, когда количество итераций во много раз превосходит количество вставок и удалений);
- механизм управления заданиями, основанный на возможностях класса **Executor**, включающий пул потоков и службу их планирования;
- высокопроизводительный класс **Lock**, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством класса **Condition**;
- классы атомарных переменных (**AtomicInteger**, **AtomicLong**, **AtomicReference**), а также их высокопроизводительные аналоги **SyncronizedInt** и др.;
- классы синхронизации общего назначения, такие как **Semaphore**, **CountDownLatch** (позволяет потоку ожидать завершения нескольких операций в других потоках), **CyclicBarrier** (позволяет нескольким потокам ожидать момента, когда они все достигнут какой-либо точки) и **Exchanger** (позволяет потокам синхронизироваться и обмениваться информацией);
- обработка неотловленных прерываний: класс **Thread** теперь поддерживает установку обработчика на неотловленные прерывания (подобное ранее было доступно только в **ThreadGroup**).

Хорошим примером новых возможностей является синхронизация коллекции типа **Hashtable**. Объект **Hashtable** синхронизируется целиком, и если один поток занял кэш остальные потоки вынуждены ожидать освобождения объекта. В случае же использования нового класса **ConcurrentHashMap** практически все операции чтения могут проходить одновременно, операции чтения и записи практически не задерживают друг друга, и только одновременные попытки записи могут блокироваться. В случае же использования данного класса как кэша (спецификой которого является большое количество операций чтения и малое – записи), блокироваться многопоточный код практически не будет.

В таблице приведено время выполнения (в миллисекундах) программы, использовавшей в качестве кэша **ConcurrentHashMap** и **Hashtable**. Тесты проводились на двухпроцессорном сервере под управлением Linux. Количество данных для большего количества потоков увеличивалось.

Количество потоков	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

```
// пример # 13 : применение семафора: Sort.java :  ArraySort.java
package chapt14;
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 сортируемого
                * 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(71);
               } catch (InterruptedException e) {
                    System.err.print(e);
               }
          }
```

```
            Sort.sortSemaphore.release();
        }
}
```

Задания к главе 14

Вариант А

1. Создать апплет с использованием потоков: строка движется горизонтально, отражаясь от границ апплета и меняя при этом случайным образом свой цвет.
2. Создать апплет с использованием потоков: 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. Условие предыдущей задачи изменяется 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 два)

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

Язык Java делает сетевое программирование простым благодаря наличию специальных средств и классов. Большинство этих классов находится в пакете **java.net**. Сетевые классы имеют методы для установки сетевых соединений передачи запросов и сообщений. Многопоточность позволяет обрабатывать несколько соединений. Сетевые приложения используют Internet-приложения, к которым относятся Web-браузер, e-mail, сетевые новости, передача файлов. Для создания таких приложений используются сокеты, порты, протоколы TCP/IP, UDP.

Приложения клиент/сервер используют компьютер, выполняющий специальную программу-сервер, которая обычно устанавливается на удаленном компьютере и 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. Запросы и сообщения представляют собой записи, структура которых определяется используемыми протоколами.

В стеке протоколов TCP/IP используются следующие прикладные 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 из подключенных к сети по протоколу TCP/IP имеет уникальный IP-адрес, используемый для идентификации и установки соединения. 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.by`), называемые именами домена. Специальная программа DNS (Domain Name Server), располагаемая на отдельном сервере, проверяет адрес и преобразует имя домена в числовой IP-адрес. 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 **217.21.43.10:443**. Здесь указан номер порта **443**. 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**, **25** – **SMTP**, **110** –

POP3, **119** – **NNTP**. К серверу можно подключиться с помощью различных портов. Каждый порт указывает конкретное место соединения на указанном компьютере и 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://www.bsu.by

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).

Определить IP-адрес в приложеннии можно с помощью объекта класса **InetAddress** из пакета **java.net**.

Класс **InetAddress** не имеет **public**-конструкторов. Создать объект класса можно с помощью статических методов. Метод **getLocalHost()** возвращает объект класса **InetAddress**, содержащий IP-адрес и имя компьютера, на котором выполняется программа. Метод **getByName(String host)** возвращает объект класса **InetAddress**, содержащий IP-адрес по имени компьютера, используя пространство имен DNS. IP-адрес может быть временным, различным для каждого соединения, однако он остается постоянным, если соединение установлено. Метод **getByAddress(byte[] addr)** создает объект класса **InetAddress**, содержащий имя компьютера, по IP-адресу, представленному в виде массива байт. Если компьютер имеет несколько IP, то получить их можно методом **getAllByName(String host)**, возвращающим массив объектов класса **InetAddress**. Если IP для данной машины один, то массив будет содержать один элемент. Метод **getByAddress(String host, byte[] addr)** создает объект класса **InetAddress** с заданным именем и IP-адресом, не проверяя существование такого компьютера. Все эти методы являются потенциальными генераторами исключительной ситуации **UnknownHostException**, и поэтому их вызов должен быть обработан с помощью **throws** для метода или блока **try-catch**. Проверить доступ к компьютеру в данный момент можно с помощью метода **boolean isReachable(int timeout)**, который возвращает **true**, если компьютер доступен, где **timeout** – время ожидания ответа от компьютера в миллисекундах.

Следующая программа демонстрирует при наличии Internet-соединения, как получить IP-адрес текущего компьютера и IP-адрес из имени домена с помощью сервера имен доменов (DNS), к которому обращается метод **getByName()** класса **InetAddress**.

```
/* пример # 1 : вывод IP-адреса компьютера и интернет-ресурса :
InetLogic.java*/
package chapt15;
import java.net.*;
```

```
public class InetLogic {
      public static void main(String[] args) {
            InetAddress myIP = null;
            InetAddress bsuIP = null;
            try {
             myIP = InetAddress.getLocalHost();
                  // вывод IP-адреса локального компьютера
                  System.out.println("Мой IP -> "
                        + myIP.getHostAddress());
             bsuIP = InetAddress.getByName(
                              "www.bsu.by");
                  // вывод IP-адреса БГУ www.bsu.by
                  System.out.println("BSU -> "
                        + ibaIP.getHostAddress());
            } catch (UnknownHostException e) {
                  // если не удается найти IP
                  e.printStackTrace();
            }
      }
}
```

В результате будет выведено, например:

```
Мой IP -> 172.17.16.14
BSU -> 217.21.43.10
```

Метод **getLocalHost()** класса **InetAddress** создает объект **myIP** и возвращает IP-адрес компьютера.

```
/* пример # 2 : присваивание фиктивного имени реальному ресурсу, заданному через IP : UnCheckedHost.java */
package chapt15;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class UnCheckedHost {
  public static void main(String[] args) {
      // задание IP-адреса лаборатории bsu.iba.by в виде массива
      byte ip[] ={(byte)217, (byte)21, (byte)43, (byte)10};
      try {
            InetAddress addr =
               InetAddress.getByAddress("University", ip);
            System.out.println(addr.getHostName()
              + "-> соединение:" + addr.isReachable(1000));
      } catch (UnknownHostException e) {
            System.out.println("адрес недоступен");
            e.printStackTrace();
      } catch (IOException e) {
            System.out.println("ошибка потока");
```

```
            e.printStackTrace();
        }
    }
}
```

В результате будет выведено в случае подключения к сети Интернет:

University-> соединение:true

Для доступа к ресурсам 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 : запуск страницы из апплета: MyShowDocument.java */
package chapt15;
import java.awt.Graphics;
import java.net.MalformedURLException;
import java.net.URL;
import javax.swing.JApplet;

public class MyShowDocument extends JApplet {
      private URL bsu = null;

      public String getMyURL() {
            return "http://www.bsu.by";
      }
      public void paint(Graphics g) {
            int  timer = 0;
            g.drawString("Загрузка страницы", 10, 10);
            try {
                  for (; timer < 30; timer++) {
                    g.drawString(".", 10 + timer * 5, 25);
                    Thread.sleep(100);
                  }
                  bsu = new URL(getMyURL());
      getAppletContext().showDocument(bsu, "_blank");
            } catch (InterruptedException e) {
                  e.printStackTrace();
            } catch (MalformedURLException e) {
            // некорректно задан протокол, доменное имя или путь к файлу
                  e.printStackTrace();
            }
      }
}
```

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 при-

мера апплет следует запускать из браузера, используя следующий HTML-документ:

```
<html>
<body align=center>
<applet code=chapt15.MyShowDocument.class></applet>
</body></html>
```

В следующей программе читается содержимое HTML-файла по указанному адресу и выводится в окно консоли.

```
/* пример # 4 : чтение документа из интернета: ReadDocument.java */
package chapt15;
import java.net.*;
import java.io.*;

public class ReadDocument {
   public static void main(String[] args) {
      try {
            URL lab = new URL("http://www.bsu.by");
            InputStreamReader isr =
                  new InputStreamReader(lab.openStream());
            BufferedReader d = new BufferedReader(isr);
            String line = "";
            while ((line = d.readLine()) != null) {
                  System.out.println(line);
            }
      } catch (MalformedURLException e) {
      // некорректно заданы протокол, доменное имя или путь к файлу
            e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }
   }
}
```

Сокетные соединения по протоколу TCP/IP

Сокеты (сетевые разъёмы) – это логическое понятие, соответствующее разъёмам, к которым подключены сетевые компьютеры и через которые осуществляется двунаправленная поточная передача данных 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 протоколо-ориентированное соединение по протоколу TCP/IP. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Сервер прослушивает сообщение и 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 клиентом с помощью объекта класса **Socket**. При этом указывается IP-адрес сервера и номер порта. Если указано символьное имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу. Например, если сервер установлен на этом же компьютере, соединение с сервером можно установить из приложения клиента с помощью инструкции:

```
Socket socket = new Socket("ИМЯ_СЕРВЕРА", 8030);
```

Сервер ожидает сообщения клиента и должен быть заранее запущен с указанием определенного порта. Объект класса **ServerSocket** создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода **accept()** класса **ServerSocket**, который возвращает сокет клиента:

```
ServerSocket server = new ServerSocket(8030);
Socket socket = server.accept();
```

Таким образом, для установки необходимо установить IP-адрес и номер порта сервера, IP-адрес и номер порта клиента. Обычно порт клиента и сервера устанавливаются одинаковыми. 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 отправки 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 отправляет клиенту строку "привет!", после чего разрывает связь.

```
/* пример # 5 : передача клиенту строки : MyServerSocket.java */
package chapt15;
import java.io.*;
import java.net.*;

public class MyServerSocket {
     public static void main(String[] args) {
          Socket s = null;
          try { // отправка  строки клиенту
               //создание объекта и назначение номера порта
               ServerSocket server = new ServerSocket(8030);
               s = server.accept();//ожидание соединения
               PrintStream ps =
                    new PrintStream(s.getOutputStream());
               // помещение строки "привет!" в буфер
               ps.println("привет!");
               // отправка содержимого буфера клиенту и его очищение
               ps.flush();
               ps.close();
          } catch (IOException e) {
               System.err.println("Ошибка: " + e);
```

```
        } finally {
           if (s != null)
             s.close(); // разрыв соединения
        }
    }
}
```

```
/* пример # 6 : получение клиентом строки: MyClientSocket.java */
package chapt15;
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_СЕРВЕРА", 8030);
                        /* здесь "ИМЯ_СЕРВЕРА" компьютер,
                        на котором стоит сервер-сокет"*/
            BufferedReader br =
                new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
            String msg = br.readLine();
            System.out.println(msg);
            socket.close();
        } catch (IOException e) {
            System.err.println("ошибка: " + e);
        }
     }
}
```

Аналогично клиент может послать данные серверу через поток вывода, полученный с помощью метода **getOutputStream()**, а сервер может получать данные через поток ввода, полученный 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. В этом случае сервер содержит цикл, ожидающий нового клиентского соединения. Каждый раз, когда клиент просит соединения, сервер создает новый поток. В следующем примере создается класс **ServerThread**, расширяющий класс **Thread**, и используется затем для соединений с многими клиентами, каждый в своем потоке.

```
/* пример # 7 : сервер для множества клиентов: NetServerThread.java */
package chapt15;
```

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

public class NetServerThread {
      public static void main(String[] args) {
            try {
            ServerSocket serv = new ServerSocket(8071);
            System.out.println("initialized");
              while (true) {
                  //ожидание клиента
                  Socket sock = serv.accept();
                  System.out.println(
                        sock.getInetAddress().getHostName()
                        + " connected");
                        /*создание отдельного потока для  обмена
                         данными с соединившимся клиентом*/
            ServerThread server = new ServerThread(sock);
            server.start();//запуск потока
                  }
            } catch (IOException e) {
                  System.err.println(e);
            }
      }
}
class ServerThread extends Thread {
      private PrintStream os;//передача
      private BufferedReader is;//чтение
      private InetAddress addr;//адрес клиента

      public ServerThread(Socket s) throws IOException {
            os = new PrintStream(s.getOutputStream());
            is = new BufferedReader(
                        new InputStreamReader(
                              s.getInputStream()));
            addr = s.getInetAddress();
      }
      public void run() {
            int i = 0;
            String str;
            try {
                  while ((str = is.readLine()) != null) {
                        if ("PING".equals(str))
                              os.println("PONG "+ ++i);
                        System.out.println("PING-PONG" + i
                        + " with " + addr.getHostName());
                  }
            } catch (IOException e) {
            //если клиент не отвечает, соединение с ним разрывается
```

```
                System.out.println("Disconnect");
            } finally {
                disconnect();//уничтожение потока
            }
        }
        public void disconnect() {
            try {
                System.out.println(addr.getHostName()
                                    + " disconnected");
                os.close();
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                this.interrupt();
            }
        }
}
```

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 # 8 : получение и отправка сообщения клиентом в потоке:
NetClientThread.java */
package chapt15;
import java.io.*;
import java.net.*;

public class NetClientThread {
        public static void main(String[] args) {
            try {
// установка соединения с сервером
Socket s = new Socket(InetAddress.getLocalHost(), 8071);
//или Socket s = new Socket("ИМЯ_СЕРВЕРА", 8071);
PrintStream ps = new PrintStream(s.getOutputStream());
BufferedReader br = new BufferedReader(
        new InputStreamReader(s.getInputStream()));
                for (int i = 1; i <= 10; i++) {
                    ps.println("PING");
                    System.out.println(br.readLine());
                    Thread.sleep(1000);
                }
                s.close();
            } catch (UnknownHostException e) {
                // если не удалось соединиться с сервером
                System.out.println("адрес недоступен");
                e.printStackTrace();
```

```
            } catch (IOException e) {
                  System.out.println("ошибка I/O потока");
                  e.printStackTrace();
            } catch (InterruptedException e) {
                  System.out.println(
                              "ошибка потока выполнения");
                  e.printStackTrace();
            }
      }
}
```

Сервер должен быть инициализирован до того, как клиент попытается осуществить сокетное соединение. При этом может быть использован IP-адрес локального компьютера.

Датаграммы и протокол UDP

UDP (User Datagram Protocol) не устанавливает виртуального соединения и не гарантирует доставку данных. Отправитель просто посылает пакеты по указанному адресу; если отосланная информация была повреждена или вообще не дошла, отправитель об этом даже не узнает. Однако достоинством UDP является высокая скорость передачи данных. Данный протокол часто используется при трансляции аудио- и видеосигналов, где потеря небольшого количества данных не может привести к серьезным искажениям всей информации.

По протоколу UDP данные передаются пакетами. Пакетом в этом случае UDP является объект класса **DatagramPacket**. Этот класс содержит в себе передаваемые данные, представленные в виде массива байт. Конструкторы класса:

```
DatagramPacket(byte[] buf, int length)
DatagramPacket(byte[] buf, int length,
                  InetAddress address, int port)
DatagramPacket(byte[] buf, int offset, int length)
DatagramPacket(byte[] buf, int offset, int length,
                   InetAddress address, int port)
DatagramPacket(byte[] buf, int offset, int length,
                   SocketAddress address)
DatagramPacket(byte[] buf, int length,
                   SocketAddress address)
```

Первый конструктор используется в тех случаях когда датаграмма только принимает в себя пришедшие данные, так как созданный с его помощью объект не имеет информации об адресе и порте получателя. Остальные конструкторы используются для отправки датаграм.

Класс **DatagramSocket** может выступать в роли клиента и сервера, то есть он способен получать и отправлять пакеты. Отправить пакет можно с помощью метода **send(DatagramPacket pac)**, для получения пакета используется метод **receive(DatagramPacket pac)**.

/ пример # 9 : отправка файла по UDP протоколу : Sender.java */*

```
package chapt15;
import java.io.*;
import java.net.*;
```

```
public class Sender {
      public static void main(String[] args) {
            try {
                  byte[] data = new byte[1000];
                  DatagramSocket s = new DatagramSocket();
                  InetAddress addr =
                        InetAddress.getLocalHost();
            /*файл с именем toxic.mp3 должен лежать в корне проекта*/
                  FileInputStream fr =
                        new FileInputStream(
                              new File("toxic.mp3"));
                  DatagramPacket pac;

                  while (fr.read(data) != -1) {
                  //создание пакета данных
pac = new DatagramPacket(data, data.length, addr, 8033);
                        s.send(pac);//отправление пакета
                  }
                  fr.close();
                  System.out.println("Файл отправлен");
            } catch (UnknownHostException e) {
            // неверный адрес получателя
                  e.printStackTrace();
            } catch (SocketException e) {
            // возникли ошибки при передаче данных
                  e.printStackTrace();
            } catch (FileNotFoundException e) {
            // не найден отправляемый файл
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
/* пример # 10 : прием данных по протоколу UDP : Recipient.java */
package chapt15;
import java.io.*;
import java.net.*;

public class Recipient {
      public static void main(String[] args) {
            File file = new File("toxic2.mp3");
            System.out.println("Прием данных...");
            try { // прием файла
                  acceptFile(file, 8033, 1000);
            } catch (IOException e) {
                  e.printStackTrace();
```

```
            }
        }
        private static void acceptFile(File file, int port,
                    int pacSize) throws IOException {
              byte data[] = new byte[pacSize];
              DatagramPacket pac =
                    new DatagramPacket(data, data.length);
              DatagramSocket s = new DatagramSocket(port);
              FileOutputStream os =
                    new FileOutputStream(file);
              try {
              /* установка времени ожидания: если в течение 10 секунд
              не принято ни одного пакета, прием данных заканчивается*/
                    s.setSoTimeout(60000);
                    while (true) {
                          s.receive(pac);
                          os.write(data);
                          os.flush();
                    }
              } catch (SocketTimeoutException e) {
                    // если время ожидания вышло
                    os.close();
                    System.out.println(
        "Истекло время ожидания, прием данных закончен");
              }
        }
}
```

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.
13. Написать программу, сканирующую сеть в указанном диапазоне IP адресов на наличие активных компьютеров.
14. Прокси. Программа должна принимать сообщения от любого клиента, работающего на протоколе TCP, и отсылать их на соответствующий сервер. При передаче изменять сообщение.
15. Телнет. Создать программу, которая соединяется с указанным сервером по указанному порту и производит обмен текстовой информацией.

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.

Глава 16
XML & JAVA

XML (*Extensible Markup Language* – расширяемый язык разметки) – рекомендован W3C как язык разметки, представляющий свод общих синтаксических правил. XML предназначен для обмена структурированной информацией с внешними системами. Формат для хранения должен быть эффективным, оптимальным с точки зрения потребляемых ресурсов (памяти, и др.). Такой формат должен позволять быстро извлекать полностью или частично хранимые в этом формате данные и быстро производить базовые операции над этими данными.

XML является упрощённым подмножеством языка SGML. На основе XML разрабатываются более специализированные стандарты обмена информацией (общие или в рамках организации, проекта), например XHTML, SOAP, RSS, MathML.

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. С другой стороны, механизм **namespace**, различная интерпретация структуры XML документа (триплеты RDF, microformat) и существование смешанного содержания (mixed content) часто превращают XML в многослойную структуру, в которой отсутствует древовидная организация (разве что на уровне синтаксиса).

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>Java SE 6</name> или **<name/>**.

Далее представлены примеры неправильных написаний тегов:

```
<?xml version="1.0"?>
<book>
   <title>title</title>
</book>
<book/>
```

Каждый XML-документ должен содержать только один корневой элемент (root element или document element). В примере есть два корневых элемента, один из которых пустой. В отличие от файла XML, файл HTML может иметь несколько корневых элементов и не обязательно <HTML>.

```
<?xml version="1.0"?>
<book>
   <caption>C++
</book>
   </caption>
```

Тег должен закрываться в том же теге, в котором был открыт. В данном случае это **caption**. В HTML этого правила не существует.

```
<?xml version="1.0"?>
<book>
   <author>Petrov
</book>
```

Любой открывающий тег должен иметь закрывающий. Если тег не имеет содержимого, можно использовать конструкцию вида **<author/>**. В HTML есть возможность не закрывать теги, и браузер определяет стили по открывающемуся тегу

Наименования тегов являются чувствительные к регистру (case-sensitive), т.е. например теги, **<author>**, **<Author>**, **<AuToR>** будут совершенно разными при работе с XML:

```
<author>Petrov</Author>
```

Программа-анализатор просто не найдет завершающий тег и выдаст ошибку. Язык HTML нетребователен к регистру.

Все атрибуты тегов должны быть заключены либо в одинарные, либо в двойные кавычки:

```
<book dateOfIssue="09/09/2007" title='JAVA in Belarus'/>
```

В HTML разрешено записывать значение атрибута без кавычек.

Например: **<FORM method=POST action=index.jsp>**

Пусть существует XML-документ с данными о студентах:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE students SYSTEM "students.dtd">
<students>
   <student login="mit" faculty="mmf">
      <name>Mitar Alex</name>
      <telephone>2456474</telephone>
      <address>
         <country>Belarus</country>
         <city>Minsk</city>
         <street>Kalinovsky 45</street>
      </address>
   </student>
   <student login="pus" faculty="mmf">
      <name>Pashkun Alex</name>
      <telephone>3453789</telephone>
      <address>
         <country>Belarus</country>
         <city>Brest</city>
         <street>Knorina 56</street>
      </address>
   </student>
</students>
```

Каждый документ начинается декларацией – строкой, указывающей как минимум версию стандарта XML. В качестве других атрибутов могут быть указаны кодировка символов и внешние связи.

После декларации в XML-документе могут располагаться ссылки на документы, определяющие структуру текущего документа и собственно XML-элементы (теги), которые могут иметь атрибуты и содержимое. Открывающий тег состоит из имени элемента, например **<city>**. Закрывающий тег состоит из того же имени, но перед именем добавляется символ '**/**', например **</city>**. Содержимым элемента (content) называется всё, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы.

Инструкции по обработке

XML-документ может содержать инструкции по обработке, которые используются для передачи информации в работающее с ним приложение. Инструкция по обработке может содержать любые символы, находиться в любом месте XML документа и должна быть заключены между **<?** и **?>** и начинаться с идентификатора, называемого **target** (цель).

Например:

```
<?xml-stylesheet type="text/xsl" href="book.xsl"?>
```

Эта инструкция по обработке сообщает браузеру, что для данного документа необходимо применить стилевую таблицу (stylesheet) **book.xsl**.

Комментарии

Для написания комментариев в XML следует заключать их, как и в HTML, между **<!--** и **-->** . Комментарии можно размещать в любом месте документа, но не внутри других комментариев:

```
<!-- комментарий <!-- Неправильный -->  -->
```

Внутри значений атрибутов:

```
<book title="BLR<!-- Неправильный комментарий -->"/>
```

Внутри тегов:

```
<book <!-- Неправильный комментарий -->/>
```

Указатели

Текстовые блоки XML-документа не могут содержать символов, которые служат в написании самого XML: **<**, **>**, **&**.

```
<description>
      в текстовых блоках нельзя использовать символы <,>,&
</description>
```

В таких случаях используются ссылки (указатели) на символы, которые должны быть заключены между символами **&** и **;** .

Особо распространенными указателями являются:

< – символ **<**;

> – символ **>**;

& – символ **&**;

' – символ апострофа **'**;

" – символ двойной кавычки **"**.

Таким образом, пример правильно будет выглядеть так:

```
<description>
      в текстовых блоках нельзя использовать символы
      &lt;, &gt;, &amp;
</description>
```

Раздел CDATA

Если необходимо включить в XML-документ данные (в качестве содержимого элемента), которые содержат символы '**<**', '**>**', '**&**', '**'**' и '**"**', чтобы не заменять их на соответствующие определения, можно все эти данные включить в раздел **CDATA**. Раздел **CDATA** начинается со строки "**<[CDATA[**", а заканчивается "**]]>**", при этом между ними эти строки не должны употребляться. Объявить раздел **CDATA** можно, например, так:

```
<data><[CDATA[ 5 < 7 ]]></data>
```

Корректность XML-документа определяют следующие два компонента:

- синтаксическая корректность (well-formed): то есть соблюдение всех синтаксических правил XML;
- действительность (valid): то есть данные соответствуют некоторому набору правил, определённых пользователем; правила определяют структуру и формат данных в XML. Валидность XML документа определяется наличием DTD или XML-схемы XSD и соблюдением правил, которые там приведены.

DTD

Для описания структуры XML-документа используется язык описания DTD (Document Type Definition). В настоящее время DTD практически не используется и повсеместно замещается XSD. DTD может встречаться в достаточно старых приложениях, использующих XML и, как правило, требующих нововведений (upgrade).

DTD определяет, какие теги (элементы) могут использоваться в XML-документе, как эти элементы связаны между собой (например, указывать на то, что элемент **<student>** включает дочерние элементы **<name>**, **<telephone>** и **<address>**), какие атрибуты имеет тот или иной элемент.

Это позволяет наложить четкие ограничения на совокупность используемых элементов, их структуру, вложенность.

Наличие DTD для XML-документа не является обязательным, поскольку возможна обработка XML и без наличия DTD, однако в этом случае отсутствует средство контроля действительности (validness) XML-документа, то есть правильности построения его структуры.

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" encoding="UTF-8" standalone="yes" ?>
<! DOCTYPE students SYSTEM "students.dtd">
```

Во втором случае описание элемента помещается в XML-документ:

```
<?xml version="1.0" ?>
<! DOCTYPE student [
<!ELEMENT student (name, telephone, address)>
<!--
далее идет описание элементов name, telephone, address
-->
]>
```

Описание элемента

Элемент в DTD описывается с помощью дескриптора **!ELEMENT**, в котором указывается название элемента и его содержимое. Так, если нужно определить элемент **<student>**, у которого есть дочерние элементы **<name>**, **<telephone>** и **<address>**, можно сделать это следующим образом:

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT address (country, city, street)>
```

В данном случае были определены три элемента: **name**, **telephone** и **address** 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** – parsed character data). Есть также маркеры **EMPTY** – элемент пуст и **ANY** – содержимое специально не описывается.

При описании элемента **<student>**, было указано, что он состоит из дочерних элементов **<name>**, **<telephone>** и **<address>**. Можно расширить это описание с помощью символов '**+**'(один или много), '*****'(0 или много), '**?**'(0 или 1), используемых для указания количества вхождений элементов. Так, например,

```
<!ELEMENT student (name, telephone, address)>
```

означает, что элемент **student** содержит один и только один элемент **name**, **telephone** и **address**. Если существует несколько вариантов содержимого элементов, то используется символ '|' (или). Например:

```
<!ELEMENT student (#PCDATA | body)>
```

В данном случае элемент **student** 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 и имеется следующий синтаксис:

```
<!ATTLIST название_елемента название_атрибута тип_атрибута
значение_по_умолчанию >
```

Например:

```
<!ATTLIST student
   login ID #REQUIRED
   faculty CDATA #REQUIRED>
```

В данном случае у элемента **<student>** определяются два атрибута: **login**, **faculty**. 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 (**ENTITIES**) – 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 атрибут должен присутствовать в элементе;

#IMPLIED – означает, что атрибут может отсутствовать, и если указано значение по умолчанию, то анализатор подставит его.

#FIXED – означает, что атрибут может принимать лишь одно значение, то, которое указано в DTD.

defaultValue – значение по умолчанию, устанавливаемое парсером при отсутствии атрибута. Если атрибут имеет параметр **#FIXED**, то за ним должно следовать **defaultValue.**

Если в документе атрибуту не будет присвоено никакого значения, то его значение будет равно заданному в DTD. Значение атрибута всегда должно указываться в кавычках.

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.**

Для повторного использования содержимого внутри описания DTD используются параметрические (параметризованные) сущности.

```
   <!ENTITY   % elementGroup "firstName, lastName,gender,
address, phone">
   <!ELEMENT employee (%elementGroup;)>
   <!ELEMENT contact (%elementGroup)>
```

В XML включены внутренние определения для символов. Кроме этого, есть внешние определения, которые позволяют включать содержимое внешнего файла:

```
<!ENTITY logotype SYSTEM "/image.gif" NDATA GIF87A>
```

Файл DTD для документа **students.xml** будет иметь вид:

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT students (student)+>
<!ELEMENT student (name, telephone, address)>
<!ATTLIST student
   login ID #REQUIRED
   faculty CDATA #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT address (country, city, street)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
```

Схема XSD

Схема XSD представляет собой более строгое, чем DTD, описание XML-документа. XSD-схема, в отличие от DTD, сама является XML-документом и поэтому более гибкая для использования в приложениях, задания правил документа, дальнейшего расширения новой функциональностью. В отличиеи от DTD, эта схема содержит много базовых типов (44 типа) и имеет поддержку пространств имен (namespace). С помощью схемы XSD можно также проверить документ на корректность.

Схема XSD первой строкой должна содержать декларацию XML

```
<?xml version="1.0" encoding="UTF-8"?>
```

Любая схема своим корневым элементом должна содержать элемент **schema**.

Для создания схемы нужно описать все элементы: их тип, количество повторений, дочерние элементы. Сам элемент создается элементом **element**, который может включать следующие атрибуты:

ref – ссылается на определение элемента, находящегося в другом месте;

name – определяет имя элемента;

type – указывает тип элемента;

minOccurs и **maxOccurs** – количество повторений этого элемента (по умолчанию **1**), чтобы указать, что количество элементов неограниченно, в атрибуте **maxOccurs** нужно задать **unbounded**.

Если стандартные типы не подходят, можно создать свой собственный тип элемента. Типы элементов делятся на простые и сложные. Различия заключаются в том, что сложные типы могут содержать другие элементы, а простые – нет.

Простые типы

Элементы, которые не имеют атрибутов и дочерних элементов, называются простыми и должны иметь простой тип данных.

Существуют стандартные простые типы, например **string** (представляет строковое значение), **boolean** (логическое значение), **integer** (целое значение), **float** (значение с плавающей точкой), **ID** (идентификатор) и др. Также простые типы можно создавать на основе существующих типов посредством элемента **simpleType**. Атрибут **name** содержит имя типа.

Все типы в схеме могут быть объявлены как локально внутри элемента, так и глобально с использованием атрибута **name** для ссылки на тип в любом месте схемы. Для указания основного типа используется элемент **restriction**. Его атрибут **base** указывает основной тип. В элемент **restriction** можно включить ряд ограничений на значения типа:

minInclusive – определяет минимальное число, которое может быть значением этого типа;

maxInclusive – максимальное значение типа;

length – длина значения;

pattern – определяет шаблон значения;

enumeration – служит для создания перечисления.

Следующий пример описывает тип **Login**, производный от **ID** и отвечающий заданному шаблону в элементе **pattern**.

```
<simpleType name="Login">
   <restriction base="ID">
      <pattern value="[a-zA-Z]{3}[a-zA-Z0-9_]+"/>
   </restriction>
</simpleType>
```

Сложные типы

Элементы, содержащие в себе атрибуты и/или дочерние элементы, называются сложными.

Сложные элементы создаются с помощью элемента **complexType**. Так же как и в простом типе, атрибут **name** задает имя типа. Для указания, что элементы должны располагаться в определенной последовательности, используется элемент **sequence**. Он может содержать элементы **element**, определяющие содержание сложного типа. Если тип может содержать не только элементы, но и текстовую информацию, необходимо задать значение атрибута **mixed** в **true**. Кроме элементов, тип может содержать атрибуты, которые создаются элементом **attribute**. Атрибуты элемента **attribute**: **name** – имя атрибута, **type** – тип значения атрибута. Для указания, обязан ли использоваться атрибут, нужно использовать атрибут **use**, который принимает значения **required**, **optional**, **prohibited**. Для установки значения по умолчанию используется атрибут **default** , а для фиксированного значения – атрибут **fixed**.

Следующий пример демонстрирует описание типа **Student**:

```
<complexType name="Student">
   <sequence>
      <element name="name" type="string"/>
      <element name="telephone" type="decimal"/>
      <element name="address" type="tns:Address"/>
   </sequence>
   <attribute name="login" type="tns:Login"
      use="required"/>
   <attribute name="faculty" type="string"
      use="required"/>
</complexType>
```

Для объявления атрибутов в элементах, которые могут содержать только текст, используются элемент **simpleContent** и элемент **extension**, с помощью которого расширяется базовый тип элемента атрибутом(ами).

```
<element name="Student">
  <complexType>
   <simpleContent>
      <extension base="string">
         <attribute name="birthday" type="string"/>
      </extension>
   </simpleContent>
  </complexType>
</element>
```

Для расширения/ограничения ранее объявленных сложных типов используется элемент **complexContent**.

```
<complexType name=”personType”>
      <sequence>
            <element name=”firstName” type=”string”/>
            <element name=”lastName” type=”string”/>
            <element name=”address” type=”string”/>
      </sequence>
</complexType>
<complexType name=”studentType”>
   <complexContent>
      <extension base=”personType”>
            <sequence>
                  <element name=”course” type=”integer”/>
                  <element name=”faculty” type=”string”/>
            </sequence>
      </extesion>
   </complexContent>
</complexType>
<element name=”Student” type=”studentType”/>
```

Для задания порядка следования элементов в XML используются такие теги, как **<all>**, который допускает любой порядок.

```
<element name=”person”>
      <complexType>
            <all>
                  <element name=”firstName” type=”string”/>
                  <element name=”lastName” type=”string”/>
            </all>
      </complexType>
</element>
```

Элемент **<choice>** указывает, что в XML может присутствовать только один из перечисленных элементов. Элемент **<sequence>** задает строгий порядок дочерних элементов.

Для списка студентов XML-схема **students.xsd** может выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.example.com/Students"
      xmlns:tns="http://www.example.com/Students">
      <element name="students">
            <complexType>
                  <sequence>
                        <element name="student"
type="tns:Student" minOccurs="1" maxOccurs="unbounded" />
                  </sequence>
            </complexType>
      </element>

      <complexType name="Student">
            <sequence>
```

```
            <element name="name" type="string" />
            <element name="telephone" type="decimal" />
            <element name="address" type="tns:Address" />
          </sequence>
          <attribute name="login" type="tns:Login"
use="required" />
          <attribute name="faculty" type="string"
use="required" />
    </complexType>

    <simpleType name="Login">
          <restriction base="ID">
             <pattern value="[a-zA-Z]{3}[a-zA-Z0-9_]*"/>
          </restriction>
    </simpleType>

    <complexType name="Address">
          <sequence>
                <element name="country" type="string" />
                <element name="city" type="string" />
                <element name="street" type="string" />
          </sequence>
    </complexType>
</schema>
```

В приведенном примере используется понятие пространства имен **namespace**. Пространство имен введено для разделения наборов элементов с соответствующими правилами, описанными схемой. Пространство имен объявляется с помощью атрибута **xmlns** и префикса, который используется для элементов из данного пространства.

Например, **xmlns="http://www.w3.org/2001/XMLSchema"** задает пространство имен по умолчанию для элементов, атрибутов и типов схемы, которые принадлежат пространству имен **"http://www.w3.org/2001/XMLSchema"** и описаны соответствующей схемой.

Атрибут **targetNamespace="http://www.example.com/Students"** задает пространство имен для элементов/атрибутов, которые описывает данная схема.

Атрибут **xmlns:tns="http://www.example.com/Students"** вводит префикс для пространства имен (элементов) данной схемы. То есть для всех элементов, типов, описанных данной схемой и используемых здесь же требуется использовать префикс **tns**, как в случае с типами – **tns:Address**, **tns:Login** и т.д.

Действие пространства имён распространяется на элемент, где он объявлен, и на все дочерние элементы.

Тогда для проверки документа объекту-парсеру следует дать указание использовать DTD или схему XSD, и в XML-документ вместо ссылки на DTD добавить вместо корневого элемента **<students>** элемент **<tns:students>** вида:

```
<tns:students xmlns:tns="http://www.example.com/Students"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="http://www.example.com/Students
students.xsd ">
```

Следующий пример выполняет проверку документа на корректность средствами языка Java.

```
/* пример # 13 : проверка корректности документа XML: XSDMain.java */
package chapt16.xsd;
import java.io.IOException;
import org.xml.sax.SAXException;
import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.SAXNotRecognizedException;
import org.xml.sax.SAXNotSupportedException;
import chapt16.xsd.MyErrorHandler;

public class XSDMain {
      public static void main(String[] args) {

            String filename = "students.xml";
            DOMParser parser = new DOMParser();
            try {
            // установка обработчика ошибок
      parser.setErrorHandler(new MyErrorHandler("log.txt"));

            // установка способов проверки с использованием XSD
                  parser.setFeature(
"http://xml.org/sax/features/validation", true);
                  parser.setFeature(
"http://apache.org/xml/features/validation/schema", true);

                  parser.parse(filename);
            } catch (SAXNotRecognizedException e) {
                  e.printStackTrace();
            System.out.print("идентификатор не распознан");
            } catch (SAXNotSupportedException e) {
                  e.printStackTrace();
            System.out.print("неподдерживаемая операция");

            } catch (SAXException e) {
                  e.printStackTrace();
            System.out.print("глобальная SAX ошибка ");
            } catch (IOException e) {
                  e.printStackTrace();
            System.out.print("ошибка I/O потока");

            }
      System.out.print("проверка " + filename + " завершена");
      }
}
```

Класс обработчика ошибок может выглядеть следующим образом:

```
/* пример # 14 : обработчик ошибок : MyErrorHandler.java */
package chapt16.xsd;
import java.io.IOException;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class MyErrorHandler implements ErrorHandler {
   private Logger logger;

   public MyErrorHandler(String log) throws IOException {
      //создание регистратора ошибок chapt16.xsd
      logger = Logger.getLogger("chapt16.xsd");
      //установка файла и формата вывода ошибок
      logger.addAppender(new FileAppender(
                        new SimpleLayout(), log));
   }
   public void warning(SAXParseException e) {
      logger.warn(getLineAddress(e) + "-" +
         e.getMessage());
   }
   public void error(SAXParseException e) {
      logger.error(getLineAddress(e) + " - "
         + e.getMessage());
   }
   public void fatalError(SAXParseException e) {
      logger.fatal(getLineAddress(e) + " - "
         + e.getMessage());
   }
   private String getLineAddress(SAXParseException e) {
      //определение строки и столбца ошибки
      return e.getLineNumber() + " : "
         + e.getColumnNumber();
   }
}
```

Чтобы убедиться в работоспособности кода, следует внести в исходный XML-документ ошибку. Например, сделать идентичными значения атрибута **login**. Тогда в результате запуска в файл будут выведены следующие сообщения обработчика об ошибках:

```
ERROR - 14 : 41 - cvc-id.2: There are multiple occurrences
of ID value 'mit'.
ERROR - 14 : 41 - cvc-attribute.3: The value 'mit' of at-
tribute 'login' on element 'student' is not valid with re-
spect to its type, 'login'.
```

Если допустить синтаксическую ошибку в XML-документе, например, удалить закрывающую скобку в элементе **telephone**, будет выведено сообщение о фатальной ошибке:

FATAL - 7 : 26 - Element type "telephone2456474" must be followed by either attribute specifications, ">" or "/>".

В Java разработаны серьезные способы взаимодействия с XML. Начиная с версии Java 6, эти механизмы включены в JDK.

Следующий пример на основе внутреннего класса создает структуру документа XML и сохраняет в ней объект.

```
/* пример # 15 : создание XML-документа на основе объекта: DemoJSR.java */
package chapt16;
import java.io.*;
import javax.xml.bind.*;
import javax.xml.bind.annotation.*;

public class DemoJSR {
   public static void main(String[] args) {
      try {
         JAXBContext context =
                  JAXBContext.newInstance(Student.class);
         Marshaller m = context.createMarshaller();
         Student s = new Student(1, "Bender");//объект
            m.marshal(s, new FileOutputStream("stud.xml"));
      } catch (FileNotFoundException e) {
         System.out.println("XMl-файл не найден");
            e.printStackTrace();
      } catch (JAXBException e) {
         System.out.println("JAXB-исключения");
         e.printStackTrace();
      }
   }
   @XmlRootElement
   private static class Student {//внутренний класс
      private int id;
      private String name;

      public Student() {
      }
      public Student(int id, String name) {
         this.id = id;
         this.name = name;
      }
      public int getID() {
         return id;
      }
      public String getName() {
         return name;
      }
      public void setID(int id) {
         this.id = id;
      }
```

```
        public void setName(String name) {
            this.name = name;
        }
    }
}
```

В результате компиляции и запуска программы будет создан XML-документ :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<student>
      <ID>1</ID>
      <name>Bender</name>
</student>
```

Возможно обратное создание на основе XML-схемы классов на языке Java:

/ пример # 16 : описание классов University, Course и перечисления Faculty в XSD-схеме: student.xsd*/*

```
<schema xmlns="http://www.w3c.org/2001/XMLSchema"
xmlns:Revealed="http://www.university.net"
targetNamespace="http://www.university.net">

   <element name="University">
      <complexType>
         <sequence>
          <element name="faculty" type="Revealed:Faculty"/>
          <element name="course" type="Revealed:Course"/>
         </sequence>
      </complexType>
   </element>
   <complexType name="Course">
      <sequence>
         <element name="login" type="string"/>
         <element name="name" type="string"/>
         <element name="telephone" type="string"/>
      </sequence>
   </complexType>
   <simpleType name="Faculty">
      <restriction base="string">
         <enumeration value="FPMI"></enumeration>
         <enumeration value="MMF"></enumeration>
         <enumeration value="Geo"></enumeration>
      </restriction>
   </simpleType>
</schema>
```

Запуск выполняется с помощью командной строки:

```
xjc student.xsd
```

В результате будет сгенерирован следующий код классов:

```
package net.university;
import javax.xml.bind.annotation.XmlEnum;
import javax.xml.bind.annotation.XmlEnumValue;
@XmlEnum
```

```
public enum Faculty {
    FPMI("FPMI"),
    MMF("MMF"),
    @XmlEnumValue("Geo")
    GEO_F("Geo");
    private final String value;

    Faculty(String v) {
        value = v;
    }
    public String value() {
        return value;
    }
    public static Faculty fromValue(String v) {
        for (Faculty c: Faculty.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v.toString());
    }
}
package net.university;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
/**
 * <p>Java class for Course complex type.
*/
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Course", propOrder = {
    "login",
    "name",
    "telephone"
})
public class Course {

    @XmlElement(required = true)
    protected String login;
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected String telephone;
    public String getLogin() {
        return login;
    }
    public void setLogin(String value) {
        this.login = value;
    }
```

```
    public String getName() {
        return name;
    }
    public void setName(String value) {
        this.name = value;
    }
    public String getTelephone() {
        return telephone;
    }
    public void setTelephone(String value) {
        this.telephone = value;
    }
}
package net.university;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
/**
 * <p>Java class for anonymous complex type.
*/
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "faculty",
    "course"
})
@XmlRootElement(name = "University")
public class University {

    @XmlElement(required = true)
    protected Faculty faculty;
    @XmlElement(required = true)
    protected Course course;
    public Faculty getFaculty() {
        return faculty;
    }
    public void setFaculty(Faculty value) {
        this.faculty = value;
    }
    public Course getCourse() {
        return course;
    }
    public void setCourse(Course value) {
        this.course = value;
    }
}
package net.university;
import javax.xml.bind.annotation.XmlRegistry;
```

```
@XmlRegistry
public class ObjectFactory {
    public ObjectFactory() {
    }
    public Course createCourse() {
        return new Course();
    }
    public University createUniversity() {
        return new University();
    }
}
```

XML-анализаторы

XML как набор байт в памяти, запись в базе или текстовый файл представляет собой данные, которые еще предстоит обработать. То есть из набора строк необходимо получить данные, пригодные для использования в программе. Поскольку XML представляет собой универсальный формат для передачи данных, существуют универсальные средства его обработки – XML-анализаторы (парсеры).

Парсер – это библиотека (в языке Java: класс), которая читает XML-документ, а затем предоставляет набор методов для обработки информации этого документа.

Валидирующие и невалидирующие анализаторы

Как было выше упомянуто, существует два вида корректности XML-документа: синтаксическая (well-formed) – документ сформирован в соответствии с синтаксическими правилами построения, и действительная (valid) – документ синтаксически корректен и соответствует требованиям, заявленным в DTD.

Соответственно есть невалидирующие и валидирующие 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 XSD или 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 три подхода (API) к обработке XML-документов:

- DOM (Document Object Model – объектная модель документов) – платформенно-независимый программный интерфейс, позволяющий программам и скриптам управлять содержимым документов HTML и XML, а также изменять их структуру и оформление. Модель DOM не накладывает ограничений на структуру документа. Любой документ известной структуры с помощью DOM может быть представлен в виде дерева узлов, каждый узел которого содержит элемент, атрибут, текстовый, графический или любой другой объект. Узлы связаны между собой отношениями родитель-потомок.

- SAX (Simple API for XML) базируется на модели последовательной одноразовой обработки и не создает внутренних деревьев. При прохождении по XML вызывает соответствующие методы у классов, реализующих интерфейсы, предоставляемые SAX-парсером.
- StAX (Streaming API for XML) не создает дерево объектов в памяти, но, в отличие от SAX-парсера, за переход от одной вершины XML к другой отвечает приложение, которое запускает разбор документа.

Анализаторы, которые строят древовидную модель, – это DOM-анализаторы. Анализаторы, которые генерируют события, – это SAX-анализаторы.

Анализаторы, которые ждут команды от приложения для перехода к следующему элементу XML – StAX-анализаторы.

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 вызывает методы, связанные с различными участками XML-файла, а программа, использующая анализатор, решает, как реагировать на тот или иной элемент XML-документа. Так, аннализатор 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.

StAX работает как **`Iterator`**, который указывает на наличие элемента с помощью метода **`hasNext()`** и для перехода к следующей вершине использует метод **`next()`.**

Когда следует использовать DOM-, а когда – SAX, StAX -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/StAX-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 всё, что необходимо, – это просто посчитать точное количество элементов в файле.

SAX-анализаторы

SAX API определяет ряд методов, используемых при разборе документа:

`void startDocument()` – вызывается на старте обработки документа;

`void endDocument()` – вызывается при завершении разбора документа;

`void startElement(String uri, String localName, String qName, Attributes attrs)` – будет вызван, когда анализатор полностью

обработает содержимое открывающего тега, включая его имя и все содержащиеся атрибуты;

void endElement(String uri, String localName, String qName) – сигнализирует о завершении элемента;

void characters(char[] ch, int start, int length) – вызывается в том случае, если анализатор встретил символьную информацию внутри элемента (тело тега);

warning(SAXParseException e), **error(SAXParseException e)**, **fatalError(SAXParseException e)** – вызываются в ответ на возникающие предупреждения и ошибки при разборе XML-документа.

В пакете **org.xml.sax в SAX2 API** содержатся интерфейсы **org.xml.sax.ContentHandler**, **org.xml.sax.ErrorHandler**, **org.xml.sax.DTDHandler**, и **org.xml.sax.EntityResolver**, которые необходимо реализовать для обработки соответствующего события.

Для того чтобы создать простейшее приложение, обрабатывающее XML-документ, достаточно сделать следующее:

1. Создать класс, который реализует один или несколько интерфейсов (**ContentHandler**, **ErrorHandler**, **DTDHandler**, **EntityResolver**) и реализовать методы, отвечающие за обработку интересующих событий.
2. Используя SAX2 API, поддерживаемое всеми SAX парсерами, создать **org.xml.sax.XMLReader**, например для Xerces:
   ```
   XMLReader reader =
   XMLReaderFactory.createXMLReader(
       "org.apache.xerces.parsers.SAXParser");
   ```
3. Передать в **XMLReader** объект класса, созданного на шаге 1 с помощью соответствующих методов:
   setContentHandler(), **setErrorHandler()**, **setDTDHandler()**, **setEntityResolver()**.
4. Вызвать метод **parse()**, которому в качестве параметров передать путь (URI) к анализируемому документу либо **InputSource**.

Следующий пример выводит на консоль содержимое XML-документа.

```
/* пример # 1 : чтение и вывод XML-документа : SimpleHandler.java */
package chapt16.analyzer.sax;
import org.xml.sax.ContentHandler;
import org.xml.sax.Attributes;

public class SimpleHandler implements ContentHandler {

   public void startElement(String uri, String localName,
         String qName, Attributes attrs) {
      String s = qName;
      //получение и вывод информации об атрибутах элемента
      for (int i = 0; i < attrs.getLength(); i++) {
         s += " " + attrs.getQName(i) + "="
            + attrs.getValue(i) + " ";
      }
```

```
      System.out.print(s.trim());
   }
   public void characters(char[] ch,
         int start, int length) {
      System.out.print(new String(ch, start, length));
   }
   public void endElement(String uri,
      String localName, String qName) {
      System.out.print(qName);
   }
}
```

/ пример # 2 : создание и запуск парсера : SAXSimple.java*/*

```
package chapt16.main;
import org.xml.sax.XMLReader;
import org.xml.sax.XMLReaderFactory;
import org.xml.sax.SAXException;
import javax.xml.parsers.ParserConfigurationException;
import java.io.IOException;
import chapt16.analyzer.sax.SimpleHandler;

public class SAXSimple {
   public static void main(String[] args) {
      try {
         //создание SAX-анализатора
XMLReader reader = XMLReaderFactory.createXMLReader();
       SimpleHandler contentHandler = new SimpleHandler();
       reader.setContentHandler(contentHandler);
       reader.parse("students.xml");
      } catch (SAXException e) {
         e.printStackTrace();
         System.out.print("ошибка SAX парсера");
      } catch (ParserConfigurationException e) {
         e.printStackTrace();
         System.out.print("ошибка конфигурации");
      } catch (IOException e) {
         e.printStackTrace();
         System.out.print("ошибка I/O потока");
      }
   }
}
```

В результате в консоль будет выведено (если убрать из XML-документа ссылку на DTD):

```
students
      student login=mit faculty=mmf
            name Mitar Alex name
            telephone 2456474 telephone
            address
                  country Belarus country
                  city Minsk city
```

```
                street Kalinovsky 45 street
            address
        student
       student login=pus faculty=mmf
             name Pashkun Alex name
             telephone 3453789 telephone
             address
                   country Belarus country
                   city Brest city
                   street Knorina 56 street
             address
        student
students
```

В следующем приложении производятся разбор документа **students.xml** и инициализация на его основе коллекции объектов класса **Student**.

/ пример # 3 : формирование коллекции объектов на основе XML-документа : StudentHandler.java */*

```
package chapt16.analyzer.sax;

enum StudentEnum {
   NAME, TELEPHONE, STREET, CITY, COUNTRY
}
package chapt16.analyzer.sax;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import java.util.ArrayList;
import chapt16.entity.Student;

public class StundentHandler implements ContentHandler {
   ArrayList<Student> students = new ArrayList<Student>();
   Student curr = null;
   StudentEnum currentEnum = null;

   public ArrayList<Student> getStudents() {
      return students;
   }
   public void startDocument() {
      System.out.println("parsing started");
   }
   public void startElement(String uri, String localName,
         String qName, Attributes attrs) {
      if (qName.equals("student")) {
         curr = new Student();
         curr.setLogin(attrs.getValue(0));
         curr.setFaculty(attrs.getValue(1));
      }
      if(!"address".equals(qName) &&
         !"student".equals(qName) &&
         !qName.equals("students"))
         currentEnum =
```

```
            StudentEnum.valueOf(qName.toUpperCase());
    }
    public void endElement(String uri, String localName,
          String qName) {
       if(qName.equals("student"))
          students.add(curr);
       currentEnum = null;
    }
    public void characters(char[] ch, int start,
                                          int length) {
       String s = new String(ch, start, length).trim();
       if(currentEnum == null) return;
       switch (currentEnum) {
          case NAME:
             curr.setName(s);
             break;
          case TELEPHONE:
             curr.setTelephone(s);
             break;
          case STREET:
             curr.getAddress().setStreet(s);
             break;
          case CITY:
             curr.getAddress().setCity(s);
             break;
          case COUNTRY:
             curr.getAddress().setCountry(s);
             break;
       }
    }
}
```

/ пример # 4 : создание и запуск парсера : SAXStudentMain.java */*

```
package chapt16.main;
import org.xml.sax.XMLReader;
import org.xml.sax.XMLReaderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import java.util.ArrayList;
import chapt16.analyzer.sax.StundentHandler;
import chapt16.entity.Student;
import java.io.IOException;

public class SAXStudentMain {
   public static void main(String[] args) {
      try {
         //создание SAX-анализатора
        XMLReader reader =
            XMLReaderFactory.createXMLReader();
```

```
         StundentHandler sh = new StundentHandler();
         reader.setContentHandler(sh);
          ArrayList <Student> list;
          if(sh != null) {
             //разбор XML-документа
             parser.parse("students.xml");
                System.out.println(sh.getStudents());
             }
       } catch (SAXException e) {
          e.printStackTrace();
          System.out.print("ошибка SAX парсера");
       } catch (ParserConfigurationException e) {
          e.printStackTrace();
          System.out.print("ошибка конфигурации");
       } catch (IOException e) {
          e.printStackTrace();
          System.out.print("ошибка I/O потока");
       }
   }
}
```

В результате на консоль будет выведена следующая информация:

```
parsing started
Login: mit
Name: Mitar Alex
Telephone: 2456474
Faculty: mmf
Address:
        Country: Belarus
        City: Minsk
        Street: Kalinovsky 45
Login: pus
Name: Pashkun Alex
Telephone: 3453789
Faculty: mmf
Address:
        Country: Belarus
        City: Brest
        Street: Knorina 56
```

Класс, объект которого формируется на основе информации из XML-документа, имеет следующий вид:

```
/* пример # 5 : класс java bean : Student.java */
package chapt16.entity;

public class Student {
   private String login;
   private String name;
   private String faculty;
```

```
private String telephone;
private Address address = new Address();
public String getLogin() {
   return login;
}
public void setLogin(String login) {
   this.login = login;
}
public String getName() {
   return name;
}
public void setName(String name) {
   this.name = name;
}
public String getFaculty() {
   return faculty;
}
public void setFaculty(String faculty) {
   this.faculty = faculty;
}
public String getTelephone() {
   return telephone;
}
public void setTelephone(String telephone) {
   this.telephone = telephone;
}
public Address getAddress() {
   return address;
}
public void setAddress(Address address) {
   this.address = address;
}
public String toString() {
   return "Login: " + login
      + "\nName: " + name
      + "\nTelephone: " + telephone
      + "\nFaculty: " + faculty
      + "\nAddress:"
      + "\n\tCountry: " + address.getCountry()
      + "\n\tCity: " + address.getCity()
      + "\n\tStreet: " + address.getStreet()
      + "\n";
}
public class Address {//внутренний класс
   private String country;
   private String city;
   private String street;

   public String getCountry() {
```

```
            return country;
        }
        public void setCountry(String country) {
            this.country = country;
        }
        public String getCity() {
            return city;
        }
        public void setCity(String city) {
            this.city = city;
        }
        public String getStreet() {
            return street;
        }
        public void setStreet(String street) {
            this.street = street;
        }
    }
}
```

Древовидная модель

Анализатор DOM представляет собой некоторый общий интерфейс для работы со структурой документа. При разработке DOM-анализаторов различными вендорами предполагалась возможность ковариантности кода.

DOM 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 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.

Существуют различные общепризнанные DOM-анализаторы, которые в настоящий момент можно загрузить с указанных адресов:

`Xerces – http://xerces.apache.org/xerces2-j/;`

`JAXP –` входит в JDK.

Существуют также библиотеки, предлагающие свои структуры объектов XML с API для доступа к ним. Наиболее известные:

`JDOM – http://www.jdom.org/dist/binary/jdom-1.0.zip.`

`dom4j – http://www.dom4j.org`

Xerces

В стандартную конфигурацию Java входит набор пакетов для работы с XML. Но стандартная библиотека не всегда является самой простой в применении, поэтому часто в основе многих проектов, использующих XML, лежат библиотеки сторонних производителей. Одной из таких библиотек является Xerces, замечательной особенностью которого является использование части стандартных воз-

можностей XML-библиотек JSDK с добавлением собственных классов и методов, упрощающих и облегчающих обработку документов XML.

org.w3c.dom.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.

`Element getDocumentElement()` – возвращает корневой элемент документа.

org.w3c.dom.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`**.

Интерфейс **`Node`** определяет ряд методов, которые используются для работы с деревом:

`short getNodeType()` – возвращает тип объекта (элемент, атрибут, текст, **`CDATA`** и т.д.);

`String getNodeValue()` – возвращает значение **`Node`**;

`Node getParentNode()` – возвращает объект, являющийся родителем текущего узла **`Node`**;

`NodeList getChildNodes()` – возвращает список объектов, являющихся дочерними элементами;

`Node getFirstChild()`, **`Node getLastChild()`** – возвращает первый и последний дочерние элементы;

`NamedNodeMap getAttributes()` – возвращает список атрибутов данного элемента.

У интерфейса **`Node`** есть несколько важных наследников – **`Element`**, **`Attr`**, **`Text`**. Они используются для работы с конкретными объектами дерева.

org.w3c.dom.Element

Интерфейс предназначен для работы с содержимым элементов XML-документа. Некоторые методы:

`String getTagName(String name)` – возвращает имя элемента;

`boolean hasAttribute()` – проверяет наличие атрибутов;

`String getAttribute(String name)` – возвращает значение атрибута по его имени;

`Attr getAttributeNode(String name)` – возвращает атрибут по его имени;

`void setAttribute(String name, String value)` – устанавливает значение атрибута, если необходимо, атрибут создается;

`void removeAttribute(String name)` – удаляет атрибут;

`NodeList getElementsByTagName(String name)` – возвращает список дочерних элементов с определенным именем.

org.w3c.dom.Attr

Интерфейс служит для работы с атрибутами элемента XML-документа.

Некоторые методы интерфейса **`Attr`**:

String getName() – возвращает имя атрибута;

Element getOwnerElement – возвращает элемент, который содержит этот атрибут;

String getValue() – возвращает значение атрибута;

void setValue(String value) – устанавливает значение атрибута;

boolean isId() – проверяет атрибут на тип ID.

org.w3c.dom.Text

Интерфейс **Text** необходим для работы с текстом, содержащимся в элементе.

String getWholeText() – возвращает текст, содержащийся в элементе;

void replaceWholeText(String content) – заменяет строкой **content** весь текст элемента.

В следующих примерах производятся разбор документа **students.xml** с использованием DOM-анализатора и инициализация на его основе набора объектов.

```
/* пример # 6 : создание анализатора и загрузка XML-документа:
DOMLogic.java*/
package chapt16.main;
import java.util.ArrayList;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
//import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.xml.sax.SAXException;
import chapt16.analyzer.dom.Analyzer;
import chapt16.entity.Student;

public class DOMLogic {
      public static void main(String[] args) {
            try {
            // создание DOM-анализатора(JSDK)
            DocumentBuilderFactory dbf=
                  DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            // распознавание XML-документа
            Document document = db.parse("students.xml");

             // создание DOM-анализатора (Xerces)
            /* DOMParser parser = new DOMParser();
            parser.parse("students.xml");
            Document document = parser.getDocument();*/

            Element root = document.getDocumentElement();
ArrayList<Student> students = Analyzer.listBuilder(root);
```

```
            for (int i = 0; i < students.size(); i++) {
                  System.out.println(students.get(i));
                  }
      } catch (SAXException e) {
         e.printStackTrace();
         System.out.print("ошибка SAX парсера");
      } catch (ParserConfigurationException e) {
         e.printStackTrace();
         System.out.print("ошибка конфигурации");
      } catch (IOException e) {
         e.printStackTrace();
         System.out.print("ошибка I/O потока");
            }
      }
}
```

/ пример # 7 : создание объектов на основе объекта типа Element : Analyzer.java */*

```
package chapt16.analyzer.dom;
import java.util.ArrayList;
import java.io.IOException;
import org.xml.sax.SAXException;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import chapt16.entity.Student;

public class Analyzer {
 public static ArrayList<Student> listBuilder(Element root)
                  throws SAXException, IOException {
            ArrayList<Student> students
                              = new ArrayList<Student>();
                  // получение списка дочерних элементов <student>
                  NodeList studentsNodes =
                      root.getElementsByTagName("student");
                  Student student = null;
      for (int i = 0; i < studentsNodes.getLength(); i++) {
            student = new Student();
            Element studentElement =
                        (Element) studentsNodes.item(i);
                        // заполнение объекта student
student.setFaculty(studentElement.getAttribute("faculty"));
student.setName(getBabyValue(studentElement, "name"));
student.setLogin(studentElement.getAttribute("login"));
student.setTelephone(
            getBabyValue(studentElement, "telephone"));

            Student.Address address = student.getAddress();
                        // заполнение объекта address
            Element addressElement =
```

```
                getBaby(studentElement, "address");
            address.setCountry(
                getBabyValue(addressElement, "country"));
            address.setCity(
                getBabyValue(addressElement, "city"));
            address.setStreet(
                getBabyValue(addressElement, "street"));

                students.add(student);
                }
            return students;
        }
        // возвращает дочерний элемент по его имени и родительскому элементу
        private static Element getBaby(Element parent,
                                       String childName) {
            NodeList nlist =
                parent.getElementsByTagName(childName);
            Element child = (Element) nlist.item(0);
            return child;
        }
        // возвращает текст, содержащийся в элементе
        private static String getBabyValue(Element parent,
                                       String childName) {
            Element child = getBaby(parent, childName);
            Node node = child.getFirstChild();
            String value = node.getNodeValue();
            return value;
        }
}
```

JDOM

JDOM не является анализатором, он был разработан для более удобного, более интуитивного для Java-программист, доступа к объектной модели XML-документа. JDOM представляет свою модель, отличную от DOM. Для разбора документа JDOM использует либо SAX-, либо DOM-парсеры сторонних производителей. Реализаций JDOM немного, так как он основан на классах, а не на интерфейсах.

Разбирать XML-документы с помощью JDOM проще, чем с помощью Xerces. Иерархия наследования объектов документа похожа на Xerces.

Content

В корне иерархии наследования стоит класс **`Content`**, от которого унаследованы остальные классы (**`Text`**, **`Element`** и др.).

Основные методы класса **`Content`**:

`Document getDocument()` – возвращает объект, в котором содержится этот элемент;

`Element getParentElement()` – возвращает родительский элемент.

Document

Базовый объект, в который загружается после разбора XML-документ. Аналогичен **Document** из Xerces.

Element getRootElement() – возвращает корневой элемент.

Parent

Интерфейс **Parent** реализуют классы **Document** и **Element**. Он содержит методы для работы с дочерними элементами. Интерфейс **Parent** и класс **Content** реализуют ту же функциональность, что и интерфейс **Node** в Xerces.

Некоторые из его методов:

List getContent() – возвращает все дочерние объекты;

Content getContent(int index) – возвращает дочерний элемент по его индексу;

int getContentSize() – возвращает количество дочерних элементов;

Parent getParent() – возвращает родителя этого родителя;

int indexOf(Content child) – возвращает индекс дочернего элемента.

Element

Класс **Element** представляет собой элемент XML-документа.

Attribute getAttribute(String name) – возвращает атрибут по его имени;

String getAttributeValue(String name) – возвращает значение атрибута по его имени;

List getAttributes() – возвращает список всех атрибутов;

Element getChild(String name) – возвращает дочерний элемент по имени;

List getChildren() – возвращает список всех дочерних элементов;

String getChildText(String name) – возвращает текст дочернего элемента;

String getName() – возвращает имя элемента;

String getText() – возвращает текст, содержащийся в элементе.

Text

Класс **Text** содержит методы для работы с текстом. Аналог в Xerces – интерфейс **Text**.

String getText() – возвращает значение содержимого в виде строки;

String getTextTrim() – возвращает значение содержимого без крайних пробельных символов.

Attribute

Класс **Attribute** представляет собой атрибут элемента XML-документа. В отличие от интерфейса **Attr** из Xerces, у класса **Attribute** расширенная функциональность. Класс **Attribute** имеет методы для возвращения значения определенного типа.

int getAttributeType() – возвращает тип атрибута;

тип **get**Тип**Type()** – (**Int**, **Double**, **Boolean**, **Float**, **Long**) возвращает значение определенного типа;

String getName() – возвращает имя атрибута;

Element getParent() – возвращает родительский элемент.

Следующие примеры выполняют ту же функцию, что и предыдущие, только с помощью JDOM.

```
/* пример # 8 : запуск JDOM : JDOMStudentMain.java */
package chapt16.main;
import java.util.List;
import org.jdom.*;
import org.jdom.input.SAXBuilder;
import java.io.IOException;
import chapt16.analyzer.dom.JDOMAnalyzer;
import chapt16.entity.Student;

public class JDOMStudentMain {
   public static void main(String[] args) {
      try {
      //создание JDOM
      SAXBuilder builder = new SAXBuilder();
      //распознавание XML-документа
      Document document = builder.build("students.xml");
         List<Student> list =
            JDOMAnalyzer.listCreator(document);

      for (Student st : list) System.out.println(st);
      } catch(IOException e) {
         e.printStackTrace();
      } catch(JDOMException e) {
         e.printStackTrace();
      }
   }
}
/* пример # 9 : создание объектов с использованием JDOM: JDOMAnalyzer.java */
package chapt16.analyzer.dom;
import java.util.*;
import java.io.IOException;
import org.jdom.Element;
import org.jdom.Document;
import org.jdom.JDOMException;
import chapt16.entity.Student;

public class JDOMAnalyzer {
   public static List<Student> listCreator(Document doc)
         throws JDOMException, IOException {
      //извлечение корневого элемента
      Element root = doc.getRootElement();
      //получение списка дочерних элементов <student>
      List studElem = root.getChildren();
      Iterator studentIterator = studElem.iterator();
      //создание пустого списка объектов типа Student
```

```
      ArrayList<Student> students =
                  new ArrayList<Student>();
      while(studentIterator.hasNext()) {
         Element studentElement =
            (Element)studentIterator.next();
         Student student = new Student();
         //заполнение объекта student
         student.setLogin(
      studentElement.getAttributeValue("login"));
         student.setName(
      studentElement.getChild("name").getText());
         student.setTelephone(
      studentElement.getChild("telephone").getText());
         student.setFaculty(
      studentElement.getAttributeValue("faculty"));

         Element addressElement =
            studentElement.getChild("address");
         Student.Address address = student.getAddress();
         //заполнение объекта address
address.setCountry(addressElement.getChild("country")
                                          .getText());
address.setCity(addressElement.getChild("city").getText());
address.setStreet(addressElement.getChild("street")
                                          .getText());
         students.add(student);
      }
      return students;
   }
}
```

Создание и запись XML-документов

Документы можно не только читать, но также модифицировать и создавать совершенно новые.

Для создания документа необходимо создать объект каждого класса (**Element**, **Attribute**, **Document**, **Text** и др.) и присоединить его к объекту, который в дереве XML-документа находится выше. В данном разделе будет рассматриваться только анализатор JDOM.

Element

Для добавления дочерних элементов, текста или атрибутов в элемент XML-документа нужно использовать один из следующих методов:

Element addContent(Content child) – добавляет дочерний элемент;

Element addContent(int index, Content child) – добавляет дочерний элемент в определенную позицию;

Element addContent(String str) – добавляет текст в содержимое элемента;

Element setAttribute(Attribute attribute) – устанавливает значение атрибута;

Element setAttribute(String name, String value) – также устанавливает значение атрибута;

Element setContent(Content child) – заменяет содержимое этого элемента на элемент, переданный в качестве параметра;

Element setContent(int index, Content child) – заменяет дочерний элемент на определенной позиции элементом, переданным как параметр;

Element setName(String name) – устанавливает имя элемента;

Element setText(String text) – устанавливает текст содержимого элемента.

Text

Класс **Text** также имеет методы для добавления текста в элемент XML-документа:

void append(String str) – добавляет текст к уже имеющемуся;

void append(Text text) – добавляет текст из другого объекта **Text**, переданного в качестве параметра;

Text setText(String str) – устанавливает текст содержимого элемента.

Attribute

Методы класса **Attribute** для установки значения, имени и типа атрибута:

Attribute setAttributeType(int type) – устанавливает тип атрибута;

Attribute setName(String name) – устанавливает имя атрибута;

Attribute setValue(String value) – устанавливает значение атрибута.

Следующий пример демонстрирует создание XML-документа и запись его в файл. Для записи XML-документа используется класс **XMLOutputter**.

```
/* пример # 10 : создание и запись документа с помощью JDOM:
JDOMLogic.java */
package chapt16.saver.dom;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.List;
import java.util.Iterator;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.XMLOutputter;
import chapt16.entity.Student;

public class JDOMLogic {
      public static Document create(List<Student> list) {
//создание корневого элемента <studentsnew>
      Element root = new Element("studentsnew");
      Iterator<Student> studentIterator =
         list.iterator();
      while(studentIterator.hasNext()) {
         Student student = studentIterator.next();
         //создание элемента <student> и его содержимого
```

```
            Element studentElement = new Element("student");
            //создание атрибутов  и передача им значений
            studentElement.setAttribute("login",
               student.getLogin());
            studentElement.setAttribute("phone",
               student.getTelephone());

               Element faculty = new Element("faculty");
               faculty.setText(student.getFaculty());
            //«вложение»  элемента <faculty> в элемент <student>
               studentElement.addContent(faculty);

               Element name = new Element("name");
               name.setText(student.getName());
               studentElement.addContent(name);
            //создание элемента <address>
            Element addressElement = new Element("address");
            Student.Address address = student.getAddress();

            Element country = new Element("country");
            country.setText(address.getCountry());
            addressElement.addContent(country);

            Element city = new Element("city");
            city.setText(address.getCity());
            addressElement.addContent(city);

            Element street = new Element("street");
            street.setText(address.getStreet());
            // «вложение» элемента <street> в элемент <address>
            addressElement.addContent(street);
            //«вложение»  элемента <address> в элемент <student>
            studentElement.addContent(addressElement);
            //«вложение»  элемента <student> в элемент <students>
            root.addContent(studentElement);
         }
            //создание основного дерева XML-документа
         return new Document(root);
      }
public static boolean saveDocument(String fileName,
                                   Document doc) {
            boolean complete = true;
            XMLOutputter outputter = new XMLOutputter();
            // запись XML-документа
            try {
outputter.output(doc, new FileOutputStream(fileName));
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
                  complete = false;
```

```
            } catch (IOException e) {
                  e.printStackTrace();
                  complete = false;
            }
            return complete;
      }
}
```

/ пример # 11 : создание списка и запуск приложения : JDOMMainSaver.java*/*

```
package chapt16.main;
import java.io.IOException;
import java.util.ArrayList;
import chapt16.entity.Student;
import chapt16.saver.dom.JDOMLogic;

public class JDOMMainSaver {
   public static void main(String[] args) {
         //создание списка студентов
ArrayList<Student> students = new ArrayList<Student> ();
         for(int j = 1; j < 3; j++) {
            Student st = new Student();
            st.setName("Petrov" + j);
            st.setLogin("petr" + j);
            st.setFaculty("mmf");
            st.setTelephone("454556"+ j*3);
          Student.Address adr = st.getAddress();
            adr.setCity("Minsk");
            adr.setCountry("BLR");
            adr.setStreet("Gaja, " + j);
                  st.setAddress(adr);
                        students.add(st);
         }
         //создание «дерева» на основе списка студентов
         Document doc = JDOMLogic.create(students);
         //сохранение «дерева» в XML-документе
      if(JDOMLogic.saveDocument("studentsnew.xml", doc))
            System.out.println("Документ создан");
      else
            System.out.println("Документ НЕ создан");
   }
}
```

В результате будет создан документ **studentsnew.xml** следующего содержания:

```
<?xml version="1.0" encoding="UTF-8"?>
<studentsnew>
      <student login="petr1" phone="4545563">
            <faculty>mmf</faculty>
            <name>Petrov1</name>
            <address>
                  <country>BLR</country>
```

```
                <city>Minsk</city>
                <street>Gaja, 1</street>
            </address>
        </student>
        <student login="petr2" phone="4545566">
            <faculty>mmf</faculty>
            <name>Petrov2</name>
            <address>
                <country>BLR</country>
                <city>Minsk</city>
                <street>Gaja, 2</street>
            </address>
        </student>
    </studentsnew>
```

В этом примере был использован JDOM, основанный на идее "if something doesn't work, fix it".

StAX

StAX (Streaming API for XML), который еще называют pull-парсером, включен в JDK, начиная с версии Java SE 6. Он похож на SAX отсутствием объектной модели в памяти и последовательным продвижением по XML, но в StAX не требуется реализация интерфейсов, и приложение само командует StAX-парсеру перейти к следующему элементу XML. Кроме того, в отличие от SAX, данный парсер предлагает API для создания XML-документа.

Основными классами StAX являются **`XMLInputFactory`**, **`XMLStreamReader`** и **`XMLOutputFactory`**, **`XMLStreamWriter`**, которые соответственно используются для чтения и создания XML-документа. Для чтения XML надо получить ссылку на **`XMLStreamReader`**:

```
StringReader stringReader = new StringReader(xmlString);
XMLInputFactory inputFactory=XMLInputFactory.newInstance();
XMLStreamReader reader = inputFactory
                    .createXMLStreamReader(stringReader);
```

после чего **`XMLStreamReader`** можно применять аналогично интерфейсу **`Iterator`**, используя методы **`hasNext()`** и **`next()`**:

`boolean hasNext()` – показывает, есть ли еще элементы;

`int next()` – переходит к следующей вершине XML, возвращая ее тип.

Возможные типы вершин:

```
XMLStreamConstants.START_DOCUMENT
XMLStreamConstants.END_DOCUMENT
XMLStreamConstants.START_ELEMENT
XMLStreamConstants.END_ELEMENT
XMLStreamConstants.CHARACTERS
XMLStreamConstants.ATTRIBUTE
XMLStreamConstants.CDATA
XMLStreamConstants.NAMESPACE
XMLStreamConstants.COMMENT
XMLStreamConstants.ENTITY_DECLARATION
```

Далее данные извлекаются применением методов:

String getLocalName() – возвращает название тега;
String getAttributeValue(NAMESPACE_URI, ATTRIBUTE_NAME) – возвращает значение атрибута;
String getText() – возвращает текст тега.

Пусть дан XML-документ с описанием медиатехники.

```
<?xml version="1.0" encoding="UTF-8"?>
<products xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation=" products.xsd">
      <category name="Audio And Video">
            <subcategory name="Audio">
                  <product>
                        <producer>Samsung</producer>
                        <model>NV678</model>
                        <year>12-12-2006</year>
                        <color>White</color>
                        <notAvailable />
                  </product>
            </subcategory>
            <subcategory name="Video">
                  <product>
                        <producer>Samsung</producer>
                        <model>VH500</model>
                        <year>12-12-2004</year>
                        <color>Black</color>
                        <cost>200</cost>
                  </product>
                  <product>
                        <producer>Samsung</producer>
                        <model>VH500</model>
                        <year>12-12-2004</year>
                        <color>White</color>
                        <notAvailable />
                  </product>
            </subcategory>
      </category>
      <category name="Computers">
            <subcategory name="Pocket">
                  <product>
                        <producer>HP</producer>
                        <model>rx371</model>
                        <year>31-01-2006</year>
                        <color>Black</color>
                        <notAvailable />
                  </product>
            </subcategory>
      </category>
</products>
```

Организация процесса разбора документа XML с помощью StAX приведена в следующем примере:

```
/* пример # 12 : реализация разбора XM-документа : StAXProductParser.java :
ProductParser.java: ParserEnum.java */
package chapt16;
public enum ParserEnum {
      PRODUCTS, CATEGORY, SUBCATEGORY, PRODUCT, PRODUCER,
MODEL, YEAR, COLOR, NOTAVAILABLE, COST, NAME
}
package chapt16;
import java.io.InputStream;

public abstract class ProductParser {
      public abstract void parse(InputStream input);

      public void writeTitle() {
            System.out.println("Products:");
      }
      public void writeCategoryStart(String name) {
            System.out.println("Category: " + name.trim());
      }
      public void writeCategoryEnd() {
            System.out.println();
      }
      public void writeSubcategoryStart(String name) {
         System.out.println("Subcategory: " + name.trim());
      }
      public void writeSubcategoryEnd() {
            System.out.println();
      }
      public void writeProductStart() {
            System.out.println("  Product Start  ");
      }
      public void writeProductEnd() {
            System.out.println("   Product End   ");
      }
      public void writeProductFeatureStart(String name) {
         switch (ParserEnum.valueOf(name.toUpperCase())) {
            case PRODUCER:
                  System.out.print("Provider: ");
                  break;
            case MODEL:
                  System.out.print("Model: ");
                  break;
            case YEAR:
                  System.out.print("Date of issue: ");
                  break;
```

```
            case COLOR:
                  System.out.print("Color: ");
                  break;
            case NOTAVAILABLE:
                  System.out.print("Not available");
                  break;
            case COST:
                  System.out.print("Cost: ");
                  break;
            }
         }
      public void writeProductFeatureEnd() {
            System.out.println();
      }

      public void writeText(String text) {
            System.out.print(text.trim());
      }
}
package chapt16;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
import java.io.InputStream;

public class StAXProductParser extends ProductParser {
      // реализация абстрактного метода из суперкласса для разбора потока
      public void parse(InputStream input) {
            XMLInputFactory inputFactory =
                        XMLInputFactory.newInstance();
            try {
               XMLStreamReader reader =
                 inputFactory.createXMLStreamReader(input);
               process(reader);
            } catch (XMLStreamException e) {
               e.printStackTrace();
            }
      }
      // метод, управляющий разбором потока
      public void process(XMLStreamReader reader)
                              throws XMLStreamException {
        String name;

        while (reader.hasNext()) {
       // определение типа "прочтённого" элемента (тега)
          int type = reader.next();
```

```
    switch (type) {
      case XMLStreamConstants.START_ELEMENT:
        name = reader.getLocalName();

switch (ParserEnum.valueOf(name.toUpperCase())) {
      case PRODUCTS:
            writeTitle();
            break;
      case CATEGORY:
writeCategoryStart(reader.getAttributeValue(null,
      ParserEnum.NAME.name().toLowerCase()));
      break;
      case SUBCATEGORY:
writeSubcategoryStart(reader.getAttributeValue(null,
      ParserEnum.NAME.name().toLowerCase()));
      break;
      case PRODUCT:
            writeProductStart();
            break;
      default:
            writeProductFeatureStart(name);
            break;
}
break;

      case XMLStreamConstants.END_ELEMENT:
            name = reader.getLocalName();

switch (ParserEnum.valueOf(name.toUpperCase())) {
      case CATEGORY:
            writeCategoryEnd();
            break;
      case SUBCATEGORY:
            writeSubcategoryEnd();
            break;
      case PRODUCT:
            writeProductEnd();
            break;
      default:
            writeProductFeatureEnd();
            break;
}
break;

      case XMLStreamConstants.CHARACTERS:
            writeText(reader.getText());
            break;
```

```
            default:
                  break;
            }
         }
      }
}
```

Для запуска приложения разбора документа с помощью StAX ниже приведен достаточно простой код:

```
/* пример # 13 : запуск приложения : StreamOutputExample.java*/
package chapt16;
import java.io.FileInputStream;
import java.io.InputStream;

public class StreamOutputExample {
  public static void main(String[] args) throws Exception {
      ProductParser parser = new StAXProductParser();
      // создание входного потока данных из xml-файла
      InputStream input =
            new FileInputStream("chapt16\\mediatech.xml");
      // разбор файла с выводом результата на консоль
      parser.parse(input);
  }
}
```

XSL

Документ XML используется для представления информации в виде некоторой структуры, но он никоим образом не указывает, как его отображать. 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 документа нужно обработать XML-файл согласно этим инструкциям.

Существует два стандарта стилевых таблиц, опубликованных W3C. Это CSS (Cascading Stylesheet) и XSL (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 три части: XSLT (XSL Transformation), XPath и XSLFO (XSL Formatting Objects).

XSL Processor необходим для преобразования XML-документа согласно инструкциям, находящимся в файле таблицы стилей.

XSLT

Этот язык для описания преобразований XML-документа применяется не только для приведения XML-документов к некоторому “читаемому” виду, но и для изменения структуры XML-документа.

К примеру, XSLT можно использовать для:

- удаления существующих или добавления новых элементов в XML-документ;
- создания нового XML-документа на основании заданного;
- извлечения информации из XML-документа с разной степенью детализации;
- преобразования XML-документа в документ HTML или документ другого типа.

Пусть требуется построить новый XML-файл на основе файла **students.xml**, у которого будет удален атрибут **login**. Элементы **country**, **city**, **street** станут атрибутами элемента **address** и элемент **telephone** станет дочерним элементом элемента **address**. Следует воспользоваться XSLT для решения данной задачи. В следующем коде приведено содержимое файла таблицы стилей **students.xsl**, решающее поставленную задачу.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" />

   <xsl:template match="/">
      <students>
         <xsl:apply-templates />
      </students>
   </xsl:template>

   <xsl:template match="student">
      <xsl:element name="student">
         <xsl:attribute name="faculty">
            <xsl:value-of select="@faculty"/>
         </xsl:attribute>
         <name><xsl:value-of select="name"/></name>
         <xsl:element name="address">
            <xsl:attribute name="country">
               <xsl:value-of select="address/country"/>
            </xsl:attribute>
            <xsl:attribute name="city">
               <xsl:value-of select="address/city"/>
            </xsl:attribute>
            <xsl:attribute name="street">
               <xsl:value-of select="address/street"/>
            </xsl:attribute>
            <xsl:element name="telephone">
               <xsl:attribute name="number">
                  <xsl:value-of select="telephone"/>
               </xsl:attribute>
            </xsl:element>
         </xsl:element>
      </xsl:element>
```

```
   </xsl:template>
</xsl:stylesheet>
```

Преобразование XSL лучше сделать более коротким, используя ATV (attribute template value), т.е «**{}**»

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" />
   <xsl:template match="/">
      <students>
         <xsl:apply-templates />
</students>
   </xsl:template>
   <xsl:template match="student">
<student faculty="{@faculty}">
         <name><xsl:value-of select="name"/></name>
      <address country="{address/country}"
               city="{address/city}"
               street="{address/street}">
               <telephone number="{telephone}"/>
      </address>
</student>
   </xsl:template>
</xsl:stylesheet>
```

Для трансформации одного документа в другой можно использовать, например, следующий код.

```
/* пример # 14 : трансформация XML : 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 tf =
            TransformerFactory.newInstance();

         //установка используемого XSL-преобразования
         Transformer transformer =
tf.newTransformer(new StreamSource("students.xsl"));

         //установка исходного XML-документа и конечного XML-файла
         transformer.transform(
            new StreamSource("students.xml"),
               new StreamResult("newstudents.xml"));
```

```
            System.out.print("complete");
        } catch(TransformerException e) {
            e.printStackTrace();
        }
    }
}
```

В результате получится XML-документ **newstudents.xml** следующего вида:

```
<?xml version="1.0" encoding="UTF-8"?>
<students>
   <student faculty="mmf">
      <name>Mitar Alex</name>
      <address country="Belarus" city="Minsk"
            street="Kalinovsky 45">
         <telephone number="3462356"/>
      </address>
   </student>
   <student faculty="mmf">
      <name>Pashkun Alex</name>
      <address country="Belarus" city="Brest"
            street="Knorina 56">
         <telephone number="4582356"/>
      </address>
   </student>
</students>
```

Элементы таблицы стилей

Таблица стилей представляет собой well-formed XML-документ. Эта таблица описывает изначальный документ, конечный документ и то, как трансформировать один документ в другой.

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="student">
   <lastname>
      <xsl:apply-templates/>
   </lastname>
</xsl:template>
```

Инструкция **<xsl:template...>** задает шаблон преобразования. Набор шаблонов преобразования составляет основную часть таблицы стилей. В предыдущем примере приводится шаблон, который преобразует элемент **student** в элемент **lastname**.

Шаблон состоит из двух частей:

1. параметр **match**, который задает элемент или множество элементов в исходном дереве, где 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. Эта запись означает в расширенном виде:

```
<xsl:apply-templates select="child::node()" />
```

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.

Задания к главе 16

Вариант А

Создать файл XML и соответствующее ему DTD-определение. Задать схему XSD. Определить класс Java, соответствующий данному описанию. Создать Java-приложение для инициализации массива объектов информацией из XML-файла. Произвести проверку XML-документа с привлечением DTD и XSD. Определить метод, производящий преобразование данного XML-документа в документ, указанный в задании.

1. **Оранжерея**.
   Растения, содержащиеся в оранжерее, имеют следующие характеристики:
   - Name – название растения.
   - Soil – почва для посадки, которая может быть следующих типов: подзолистая, грунтовая, дерново-подзолистая.
   - Origin – место происхождения растения.
   - Visual parameters (должно быть несколько) – внешние параметры: цвет стебля, цвет листьев, средний размер растения.

- Growing tips (должно быть несколько) – предпочитаемые условия произрастания: температура (в градусах), освещение (светолюбиво либо нет), полив (мл в неделю).
- Multiplying – размножение: листьями, черенками либо семенами.

Корневой элемент назвать Flower.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 растений. С помощью XSL преобразовать данный файл в формат HTML, где отобразить растения по предпочитаемой температуре (по возрастанию).

2. **Алмазный фонд.**

Драгоценные и полудрагоценные камни, содержащиеся в павильоне, имеют следующие характеристики:

- Name – название камня.
- Preciousness – может быть драгоценным либо полудрагоценным.
- Origin – место добывания.
- Visual parameters (должно быть несколько) – могут быть: цвет (зеленый, красный, желтый и т.д.), прозрачность (измеряется в процентах 0-100%), способы огранки (количество граней 4-15).
- Value – вес камня (измеряется в каратах).

Корневой элемент назвать Gem.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 камней. С помощью XSL преобразовать данный файл в формат XML , где корневым элементом будет место происхождения.

3. **Тарифы мобильных компаний.**

Тарифы мобильных компаний могут иметь следующую структуру:

- Name – название тарифа.
- Operator name – название сотового оператора, которому принадлежит тариф.
- Payroll – абонентская плата в месяц (0 – n рублей).
- Call prices (должно быть несколько) – цены на звонки: внутри сети (0 – n рублей в минуту), вне сети (0 – n рублей в минуту), на стационарные телефоны (0 – n рублей в минуту).
- SMS price – цена за смс (0 – n рублей).
- Parameters (должно быть несколько) – наличие любимого номера (0 – n), тарификация (12-секундная, минутная), плата за подключение к тарифу (0 – n рублей).

Корневой элемент назвать Tariff.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 тарифов. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать тарифы по абонентской плате.

4. **Лекарственные препараты.**

Лекарственные препараты имеют следующие характеристики.

- Name – название препарата.
- Price – цена за упаковку (0 – n рублей).
- Dosage – дозировка препарата (мг/день).

- Visual (должно быть несколько) – визуальные характеристики препарата: цвет (белый, желтый, зеленый, красный), консистенция (жидкий, порошкообразный, твердый), показания к применению (респираторные заболевания, расстройства организма, психические заболевания, общеукрепляющее).

Корневой элемент назвать Medicine.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 лекарств. С помощью XSL преобразовать данный файл в формат HTML , при выводе отсортировать лекарства по цене.

5. **Компьютер.**

Компьютерные комплектующие имеют следующие характеристики:

- Name – название комплектующего.
- Origin – страна производства.
- Price – цена (0 – n рублей).
- Type (должно быть несколько) – периферийное либо нет, энергопотребление (ватт), наличие кулера (есть либо нет), группа комплектующих (устройства ввода-вывода, мультимедийные), порты (COM, USB, LPT).
- Critical – критично ли наличие комплектующего для работы компьютера.

Корневой элемент назвать Device.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 устройств. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать Critical.

6. **Огнестрельное оружие.**

Огнестрельное оружие можно структурировать по следующей схеме:

- Model – название модели.
- Handy – одно- или двуручное.
- Origin – страна производства.
- TTC (должно быть несколько) – тактико-технические характеристики: дальнобойность (близкая [0 – 500м], средняя [500 – 1000 м], дальняя [1000 – n метров]), прицельная дальность (в метрах), наличие обоймы, наличие оптики.
- Material – материал изготовления.

Корневой элемент назвать Gun.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 видов. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать страну производства.

7. **Холодное оружие.**

Холодное оружие можно структурировать по следующей схеме:

- Type – тип (нож, кинжал, сабля и т.д.).
- Handy – одно или двуручное.
- Origin – страна производства.
- Visual (должно быть несколько) – визуальные характеристики: клинок (длина клинка [10 – n см], ширина клинка [10 – n мм]), материал (клинок [сталь, чугун, медь и т.д.]), рукоять (деревянная [если да, то указать тип дерева], пластик, металл), наличие кровостока (есть либо нет).

- Value – коллекционный либо нет.

Корневой элемент назвать Knife.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 видов. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать по длине клинка.

8. **Военные самолеты.**

Военные самолеты можно описать по следующей схеме:

- Model – название модели.
- Origin – страна производства.
- Chars (должно быть несколько) – характеристики, могут быть следующими: тип (самолет поддержки, сопровождения, истребитель, перехватчик, разведчик), кол-во мест (1 либо 2), боекомплект (есть либо нет [разведчик], если есть, то: ракеты [0 – 10]), наличие радара.
- Parameters – длина (в метрах), ширина (в метрах), высота (в метрах).
- Price – цена (в долларах).

Корневой элемент назвать Plane.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 типов самолетов. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать по стоимости.

9. **Конфеты**.

- Name – название конфеты.
- Energy– калорийность (ккал).
- Type (должно быть несколько) – тип конфеты (карамель, ирис, шоколадная [с начинкой либо нет]).
- Ingredients (должно быть несколько) – ингредиенты: вода, сахар (в мг), фруктоза (в мг), тип шоколада (для шоколадных), ванилин (в мг)
- Value – пищевая ценность: белки (в гр.), жиры (в гр.) и углеводы (в гр.).
- Production – предприятие-изготовитель.

Корневой элемент назвать Candy.

Создать XML файл, отображающий заданную тему, привести примеры 4-5 конфет. С помощью XSL преобразовать данный файл в формат HTML, при выводе отсортировать по месту изготовления.

10. **Пиво**.

- Name – название пива.
- Type – тип пива (темное, светлое, лагерное, живое).
- Al – алкогольное либо нет.
- Manufacturer – фирма-производитель.
- Ingredients (должно быть несколько) – ингредиенты: вода, солод, хмель, сахар и т.д.
- Chars (должно быть несколько) – характеристики: кол-во оборотов (если алкогольное), прозрачность (в процентах), фильтрованное либо нет, пищевая ценность (ккал), способ разлива (объем и материал емкостей)
- Корневой элемент назвать Beer.

Создать XML-файл, отображающий заданную тему, привести примеры 4-5 сортов пива. С помощью XSL преобразовать данный

файл в формат XML, при выводе корневым элементом сделать производителя.

11. **Периодические издания**.
    - Title – название издания.
    - Type – тип издания (газета, журнал, буклет).
    - Monthly – ежемесячное либо нет.
    - Chars (должно быть несколько) – характеристики: цветное (да либо нет), объем (n страниц), глянцевое (да [только для журналов и буклетов] либо нет [для газет]), имеет подписной индекс (только для газет и журналов).

    Корневой элемент назвать Paper.

    Создать XML файл, отображающий заданную тему, привести примеры 4-5 типов периодики. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать тип (Type).
12. **Интернет-страницы**.
    - Title – название страницы.
    - Type – тип страницы (рекламный, страница новостей, портал, зеркало).
    - Chars (должно быть несколько) – наличие электронного ящика (только для порталов, зеркал и страниц новостей), наличие новостей (только для страниц новостей), наличие архивов для выкачивания (только для зеркал), наличие голосования (есть[если есть, то анонимное либо с применением авторизации] либо нет), платный (информация, доступная для выкачивания, бесплатна либо нет).
    - Authorize – необходима либо нет авторизация.

    Корневой элемент назвать Site.

    Создать XML файл, отображающий заданную тему, привести примеры 4-5 типов периодики. С помощью XSL преобразовать данный файл в формат XML, при выводе корневым элементом сделать тип (Type).

Тестовые задания к главе 16

Вопрос 16.1.

Какой существует способ описания данных в XML? (выберите два)

1. XML использует DTD для описания данных
2. XML использует XSL для описания данных
3. XML использует XSD для описания данных
4. XML использует CSS для описания данных

Вопрос 16.2.

В каких строках XML документа есть ошибки? (выберите два)

```
1 <?xml version="1.0"?>
2 <folder>
3   <file><name><contents></contents></name></file>
4   <file><name/><contents></contents><name/></file>
5   <file><name/><contents></contents></name></file>
```

6 <file><name><contents/><name/></file>

7 </folder>

1. 1;
2. 2;
3. 3;
4. 4;
5. 5;
6. 6;
7. 7;
8. нет ошибок.

Вопрос 16.3.

Какое из данных имен не является корректным именем для XML элемента? (выберите 2)

1. <hello_dolly>;
2. <big bang>;
3. <xmldocument>;
4. <7up>;
5. только одно имя некорректно.

Вопрос 16.4.

Значения атрибутов XML всегда должны помещаться в …? (выберите два)

1. двойные кавычки “ ”;
2. апострофы ‘ ’;
3. фигурные скобки { };
4. квадратные скобки [];
5. могут обходиться без ограничивающих символов.

Вопрос 16.5.

Какие виды событий нельзя обрабатывать с помощью SAX-анализатора?

1. события документа;
2. события загрузки DTD-описаний;
3. события при анализе DTD-описаний;
4. ошибки;
5. все перечисленные события можно обработать.

Часть 3.
ТЕХНОЛОГИИ РАЗРАБОТКИ WEB-ПРИЛОЖЕНИЙ

В третьей части даны основы программирования распределенных информационных систем с применением сервлетов, JSP и баз данных, а также основные принципы создания собственных библиотек тегов.

Глава 17
ВВЕДЕНИЕ В СЕРВЛЕТЫ И JSP

Согласно заявлению Sun Microsystems, на настоящий момент более 90% корпоративных систем поддерживают платформу Java Enterprise Edition.

Первый сервлет

Сервлеты – это компоненты приложений Java Enterprise Edition, выполняющиеся на стороне сервера, способные обрабатывать клиентские запросы и динамически генерировать ответы на них. Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP.

Все сервлеты реализуют общий интерфейс **`Servlet`** из пакета **`javax.servlet`**. Для обработки HTTP-запросов можно воспользоваться в качестве базового класса абстрактным классом **`HttpServlet`** из пакета **`javax.servlet.http`**.

Жизненный цикл сервлета начинается с его загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый запрос. Далее производятся 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 программист помещает код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом **`service(HttpServletRequest req, HttpServletResponse res)`** сервлета, а все параметры запроса упаковываются в объект **`req`** класса **`HttpServletRequest`**, передаваемый в сервлет. Еще одним параметром этого метода является объект **`res`** класса **`HttpServletResponse`**, в который загружается информация для передачи клиенту. Для каждого нового клиента при обращении к сервлету создается независимый поток, в котором производится вызов метода **`service()`**. Метод **`service()`** предназначен для одновременной обработки множества запросов.

После завершения выполнения сервлета контейнер сервлетов вызывает метод **destroy()**, в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

При разработке сервлетов в качестве базового класса в большинстве случаев используется не интерфейс **Servlet**, а класс **HttpServlet**, отвечающий за обработку запросов HTTP. Этот класс уже имеет реализованный метод **service()**.

Метод **service()** класса **HttpServlet** служит 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 : простейший сервлет : MyServlet.java
package chapt17;
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();
      }
      public void init() throws ServletException {
      }
```

```
        protected void doGet(
              HttpServletRequest request,
              HttpServletResponse response)
              throws ServletException, IOException {
              response.setContentType("text/html");
              PrintWriter out = response.getWriter();
              out.print("This is ");
              out.print(this.getClass().getName());
              out.print(", using the GET method");
              }
        protected void doPost(
              HttpServletRequest request,
              HttpServletResponse response)
              throws ServletException, IOException {
              response.setContentType("text/html");
              PrintWriter out = response.getWriter();
              out.print("This is ");
              out.print(this.getClass().getName());
              out.print(", using the POST method");
              }
        public void destroy() {
              super.destroy(); // Just puts "destroy" string in log
              }
}
```

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 возможности JSP, JSPX или JSF. Признаки наличия анти-шаблонов все же будут встречаться ниже, но это отступление сделано только с точки зрения компактности примеров.

Сервлет является компонентом Web-приложения, который будет называться **`FirstProject`** и размещен в папке **`/WEB-INF/classes`** проекта.

Запуск контейнера сервлетов и размещение проекта

Здесь и далее применяется контейнер сервлетов Apache Tomcat в качестве обработчика страниц JSP и сервлетов. Последняя версия может быть загружена с сайта **`jakarta.apache.org`**.

При установке Tomcat предложит значение порта по умолчанию **`8080`**, но во избежание конфликтов с иными Application Server рекомендуется присвоить другое значение, например **`8082`**.

Ниже приведены необходимые действия по запуску сервлета из предыдущего примера с помощью контейнера сервлетов Tomcat 5.5.20, который установлен в ката-

логе **/Apache Software Foundation/Tomcat5.5**. В этом же каталоге размещаются следующие подкаталоги:

/bin – содержит файлы запуска контейнера сервлетов **tomcat5.exe**, **tomcat5w.exe** и некоторые необходимые для этого библиотеки;

/common – содержит библиотеки служебных классов, в частности Servlet API;

/conf – содержит конфигурационные файлы, в частности конфигурационный файл контейнера сервлетов **server.xml**;

/logs – помещаются log-файлы;

/webapps – в этот каталог помещаются папки, содержащие сервлеты и другие компоненты приложения.

В каталог **/webapps** необходимо поместить папку **/FirstProject** с вложенным в нее сервлетом **MyServlet**. Кроме того, папка **/FirstProject** должна содержать каталог **/WEB-INF**, в котором помещаются подкаталоги:

/classes – содержит класс сервлета **chapt17.MyServlet.class**;

/lib – содержит используемые внешние библиотеки (если они есть), упакованные в JAR-файлы (архивы java);

/src – содержит исходный файл сервлета **MyServlet.java** (опционально);

а также **web.xml** – дескриптор доставки приложения располагается в каталоге **/WEB-INF**.

В файле **web.xml** необходимо прописать имя и путь к сервлету. Кроме того, в дескрипторном файле можно определять параметры инициализации, MIME-типы, mapping сервлетов и JSP, стартовые страницы и страницы с сообщениями об ошибках, а также параметры для безопасной авторизации и аутентификации. Этот файл можно сконфигурировать так, что имя сервлета в браузере не будет совпадать с истинным именем сервлета. Например:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
      <display-name>FirstProject</display-name>
<servlet>
  <display-name>MyServletdisplay</display-name>
  <servlet-name>MyServletname</servlet-name>
  <servlet-class>chapt17.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MyServletname</servlet-name>
  <url-pattern>/MyServlettest</url-pattern>
</servlet-mapping>
<session-config>
        <session-timeout>30</session-timeout>
</session-config>
      <welcome-file-list>
```

```
            <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
 <login-config>
  <auth-method>BASIC</auth-method>
 </login-config>
</web-app>
```

Здесь указано имя сервлета **MyServletname**, путь к откомпилированному классу сервлета **MyServlet.class**, а также URL-имя сервлета, по которому происходит его вызов **MyServlettest**.

Таким образом, требуется выполнить следующие действия:

1. Компиляцию сервлета с указанием в `-cp` пути к архиву
2. `servlet-api.jar`;
3. Полученный файл класса **MyServlet.class** поместить в папку **/FirstProject/WEB-INF/classes/chapt18**;
4. В папку **/MyProject/WEB-INF** поместить файл конфигурации **web.xml**;
5. Переместить папку **/FirstProject** в каталог **/webapps** контейнера сервлетов Tomcat;
6. Стартовать Tomcat;
7. Запустить браузер и ввести адрес:
   http://localhost:8082/FirstProject/MyServlettest
   При обращении к сервлету из другого компьютера вместо **localhost** следует указать IP-адрес или имя компьютера.
8. Если вызывать сервлет из **index.jsp**, то тег **FORM** должен выглядеть следующим образом:

```
<FORM action="MyServlettest">
      <INPUT type="submit" value="Execute">
</FORM>
```

Файл **index.jsp** помещается в папку **/webapps/FirstProject** и в браузере набирается строка:

http://localhost:8082/FirstProject/index.jsp

Сервлет будет вызван из JSP-страницы по URL-имени **MyServlettest**, и в результате в браузер будет выведено:

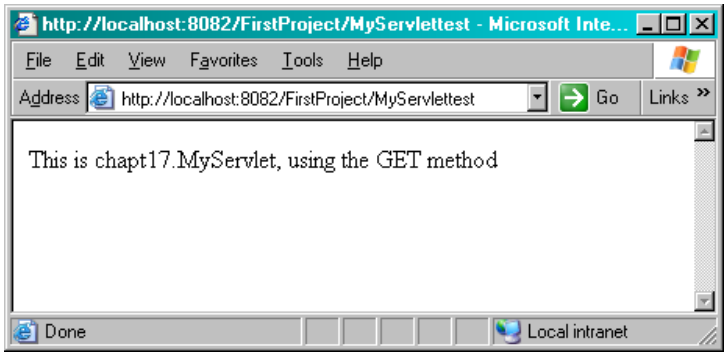


Рис. 17.1. Вывод сервлета после вызова метода doGet()

Первая JSP

Java Server Pages (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.

Процессы, выполняемые с файлом JSP при первом вызове:

d) Браузер делает запрос к странице JSP.
e) JSP-engine анализирует содержание файла JSP.
f) 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.
g) Полученный код компилируется в файл **`*.class`**.
h) Вызываются методы **`init()`** и **`_jspService()`**, и сервлет логически исполняется.
i) Сервлет на основе JSP установлен. Комбинация статического HTML и графики вместе с динамическими элементами, определенными в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа **`ServletResponse`**.

Последующие обращения к файлу JSP просто вызовут метод **`_jspService()`** сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен и сервлет не будет выгружен вручную либо пока не будет изменен файл JSP. Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности в его **`_jspService()`**-метод.

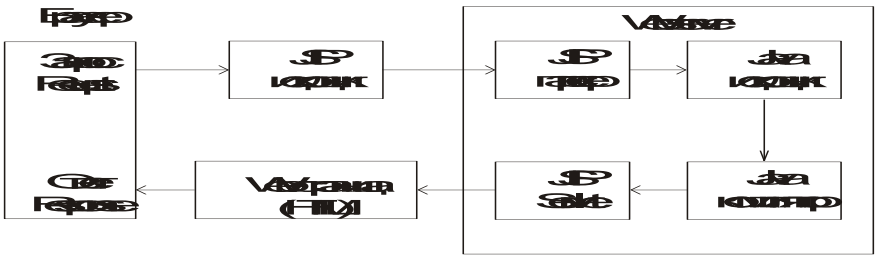


Рис. 17.2. Рабочий цикл JSP

Если рассмотреть преобразование в сервлет простейшей JSP, отправляющей в браузер приветствие:

```
<!--пример # 2 : простейшая страница JSP : simple.jsp -->
<html><head>
<title>Simple</title>
</head>
<body>
<jsp:text>Hello, Bender</jsp:text>
</body></html>
```

то в результате запуска в браузер будет выведено:

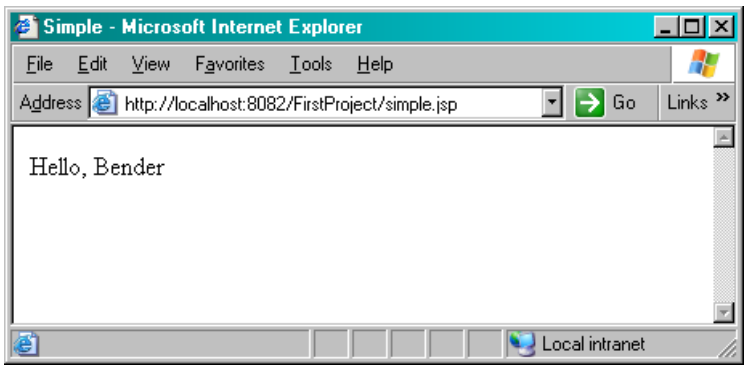


Рис. 17.3. Вывод после вызова index.jsp

А код сервлета будет получен следующий:

```
// пример # 3 : сгенерированный сервлет : simple_jsp.java
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class simple_jsp
   extends org.apache.jasper.runtime.HttpJspBase
   implements org.apache.jasper.runtime.JspSourceDependent{
  private static java.util.List _jspx_dependants;

  public Object getDependants() {
    return _jspx_dependants;
  }
  public void _jspService(HttpServletRequest request,
                          HttpServletResponse response)
        throws java.io.IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;
    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this,
            request, response, null, true, 8192, true);
```

```
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("<html><head>\r\n");
      out.write("<title>Simple</title>\r\n");
      out.write("</head>\r\n");
      out.write("<body>\r\n");
      out.write("Hello, Bender\r\n");
      out.write("</body></html>");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null)
_jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

JSP-код 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 статическую страницу, и теги Java, которые управляют 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.

Взаимодействие сервлета и JSP

Страницы JSP и сервлеты никогда не используются в информационных системах друг без друга. Причиной являются принципиально различные роли, которые играют данные компоненты в приложении. Страница JSP ответственна за формирование пользовательского интерфейса и отображение информации, переданной с сервера. Сервлет выполняет роль контроллера запросов и ответов, то есть принимает запросы от всех связанных с ним JSP-страниц, вызывает соответствующую бизнес-логику для их (запросов) обработки и в зависимости от результата выполнения решает, какую JSP поставить этому результату в соответствие.

Ниже приведен пример вызова сервлета из JSP с последующим вызовом другой JSP.

```
<!--пример # 4 : страница JSP с вызовом сервлета : index.jsp -->
<%@ page language="java" contentType="text/html; char-
set=ISO-8859-5" pageEncoding="ISO-8859-5"%>
<html><body>
<jsp:useBean id="gc" class="java.util.GregorianCalendar"/>
<jsp:getProperty name="gc" property="time"/>
<FORM action="serv" method="POST">
          <INPUT type="submit" value="Вызвать сервлет">
</FORM>
</body></html>
```

В результате запуска проекта в браузер будет выведено:

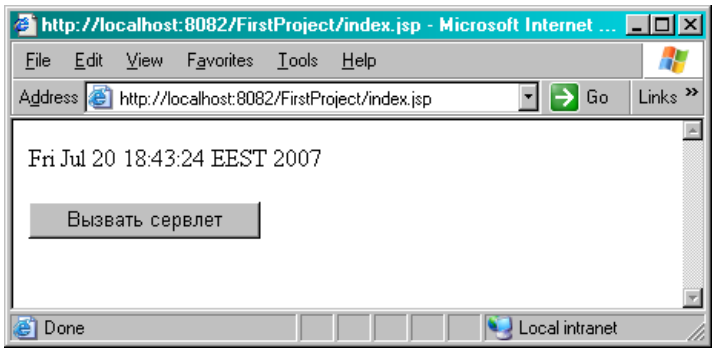


Рис. 17.4. Запуск index.jsp

Кодировка для символов кириллицы задана с помощью директивы **page**. Action-теги **useBean** и **getProperty** используются для создания объекта класса **GregorianCalendar** в области видимости JSP и вывода его значения. Сервлет **ContServlet** вызывается методом **POST**.

```
// пример # 5 : простой контроллер : ContServlet.java
package chapt17;
import java.io.IOException;
import java.util.Calendar;
import java.util.Locale;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ContServlet
          extends javax.servlet.http.HttpServlet {

     protected void doPost(HttpServletRequest request,
                    HttpServletResponse response)
               throws ServletException, IOException {
     //добавление атрибута к запросу
     request.setAttribute("loc", Locale.getDefault());
     //добавление атрибута к сессии
     request.getSession().setAttribute("calend",
                         Calendar.getInstance());
```

```
        //получение объекта RequestDispatcher и вызов JSP
request.getRequestDispatcher("/main.jsp").forward(request,
                                                   response);
        }
}
```

Передачу информации между JSP и сервлетом можно осуществлять, в частности, с помощью добавления атрибутов к объектам **HttpServletRequest**, **HttpSession**, **HttpServletContext**. Вызов **main.jsp** из сервлета в данном случае производится методом **forward()** интерфейса **RequestDispatcher**.

```
<!--пример # 6 : страница, вызванная сервлетом : main.jsp -->
<%@ page language="java"
contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>
<html><body>
<h3>Региональные установки и Время</h3>
<c:out value="Locale from request: ${loc}"/><br>
<c:out value="Time from Servlet: ${calend.time}"/>
</body></html>
```

После вызова сервлета и последующего вызова **main.jsp** будет выведено:

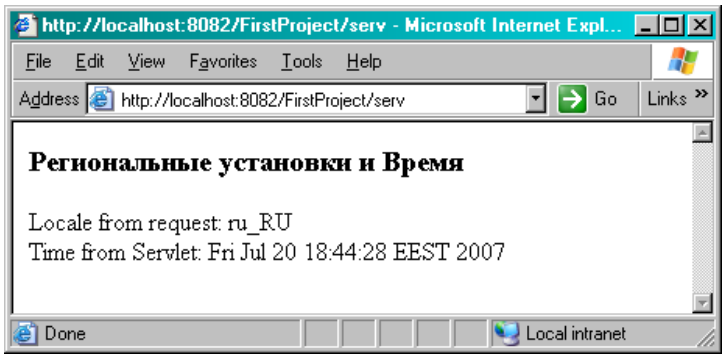


Рис. 17.5. Вывод информации страницей main.jsp

В данном коде директива **taglib** подключает JSP Standard Tag Library (JSTL), и становится возможным вызов тега **<c:out>**, а также использование Expression Language (EL) в виде **${loc}**.

Конфигурационный файл **web.xml** для данной задачи должен содержать следующую информацию:

```
<servlet>
  <display-name>Controller</display-name>
  <servlet-name>controller</servlet-name>
  <servlet-class>chapt17.ContServlet</servlet-class>
 </servlet>
  <servlet-mapping>
  <servlet-name>controller</servlet-name>
```

```
  <url-pattern>/serv</url-pattern>
 </servlet-mapping>
```

В этой главе была дана общая информация о взаимодействии различных компонентов Web-приложения.

Задания к главе 17

Вариант А

Создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы, способные выполнить следующие действия:

8. Подсчет суммы случайным образом выбранных элементов массива.
9. Вывести полное название страны и языка.
10. Подсчитать время между выполнением сервлета и JSP в наносекундах.
11. Создать массив дат и вывести самую позднюю дату.
12. Задать температуру. Если она меньше нуля, вывести значение температуры синим цветом, если больше, то красным.
13. Создать приложение, выводящее фамилию разработчика, дату и время получения задания, а также дату и время его выполнения.

Вариант В

Задания варианта В главы 1 выполнить на основе сервлетов. Число **n** генерировать с помощью методов класса **java.util.Random**.

Тестовые задания к главе 17

Вопрос 17.1.

Укажите стандартный путь к сервлету `com.example.MyServlet`, чтобы Web-приложение могло к нему обратиться.

1) /lib/MyServlet.class
2) /com/example/MyServlet.class
3) /WEB-INF/lib/MyServlet.class
4) classes/com/example/MyServlet.class
5) /servlets/com/example/MyServlet.class
6) /WEB-INF/classes/com/example/MyServlet.class

Вопрос 17.2.

Дано:

```
public void service(ServletRequest request,
                           ServletResponse response) {
         ServletInputStream sis =
             //1
         }
```

Какой код инициализирует ссылку на байтовый поток в строке 1?

1) `request.getWriter();`
2) `request.getReader();`
3) `request.getInputStream();`
4) `request.getResourceAsStream();`
5) `request.getResourceAsStream(ServletRequest.REQUEST);`

Вопрос 17.3.

На странице JSP необходимо создать объект JavaBean для использования только на этой странице. Какие два атрибута `jsp:useBean` должны применяться для этого?

1) id
2) type
3) name
4) class
5) create

Вопрос 17.4.

Какой стиль комментариев используется в страницах JSP?

1) <!--this is a comment-->
2) <%// this is a comment %>
3) <%-- this is a comment --%>
4) <%/** this is a comment **/%>

Вопрос 17.5.

Определен метод `doGet()` интерфейса `HttpServlet`:

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException {
            HttpSession s = request.getSession();
            // insert code here
}
```

С помощью какого кода можно удалить атрибут из объекта сессии?

1) `session.unbind("key");`
2) `session.remove("key");`
3) `session.removeAttribute("key");`
4) `session.unbindAttribute("key");`
5) `session.deleteAttribute("key");`

Вопрос 17.6.

Какая из записей указывает правильное имя и расположение файла-дескриптора Web-приложения относительно его корневой директории?

1) /conf/web.xml
2) /WEB-INF/web.xml
3) /conf/server.xml
4) /META-INF/web.xml
5) /WEB-INF/rules.xml
6) /META-INF/server.xml

Глава 18
СЕРВЛЕТЫ

Сервлет применяется для создания серверного приложения, получающего от клиента запрос, анализирующего его, делающего выборку данных из базы, а затем пересылающего клиенту страницу HTML, сгенерированную с помощью JSP на основе полученных данных.

Преимуществом сервлетов перед CGI или ASP является быстродействие, переносимость на различные платформы, использование объектно-ориентированного языка высокого уровня Java, который расширяется большим числом классов и программных интерфейсов.

Сервлеты поддерживаются большинством Web-серверов и являются частью платформы J2EE. Сервлеты реализуют интерфейс **`Servlet`**, в котором, кроме рассмотренных выше методов **`service()`**, **`init()`**, **`destroy()`**, предусмотрена реализация еще двух методов:

`ServletConfig getServletConfig()` – возвращает объект, содержащий параметры конфигурации сервлета;

`String getServletInfo()` – определение информации о назначении сервлета.

Интерфейс ServletContext

Интерфейс **`ServletContext`** объявляет методы, которые сервлет применяет для связи с контейнером сервлетов и позволяет получать информацию о среде выполнения, а также использовать ресурсы совместно с другими объектами приложения. Каждому сервлету ставится в соответствие единственный объект, реализующий **`ServletContext`**. Контекст выполнения сервлета дает средства для общения с сервером. В частности, можно получить информацию о MIME-типе файла, добавить/удалить атрибуты контекста или записать информацию в log-файл. Получить ссылку на объект **`ServletContext`** можно вызовом метода **`getServletContext()`**.

Следующие методы позволяют получить из контекста сервлета базовую информацию:

`String getMimeType(String filename)` – определение MIME-типа файла или документа. По умолчанию MIME-типом для сервлетов является **`text/plain`**, но используется обычно **`text/html`**;

`String getRealPath(String filename)` – определение истинного маршрута файла относительно каталога, в котором сервер хранит документы;

`String getServerInfo()` – предоставляет информацию о самом сервере.

Ряд методов предназначен для управления атрибутами, с помощью которых передается информация между различными компонентами приложения (JSP, сервлетами):

Object getAttribute(String name) – получает значение атрибута по имени;

Enumeration getAttributeNames() – получает список имен атрибутов;

void setAttribute(String name, Object object) – добавляет атрибут и его значение в контекст;

void removeAttribute(String name) – удаляет атрибут из контекста;

ServletContext getContext(String uripath) – позволяет получить доступ к контексту других ресурсов данного контейнера сервлетов;

String getServletContextName() – возвращает имя сервлета, которому принадлежит данный объект интерфейса **ServletContext**.

Используя объект **ServletContext**, можно регистрировать события сервлета, сессии и запроса.

Интерфейс ServletConfig

Ранее уже упоминался метод **getServletConfig()**, но не было сказано об интерфейсе **ServletConfig**, с помощью которого контейнер сервлетов передает информацию сервлету в процессе его инициализации.

Некоторые методы класса:

String getServletName() – определение имени сервлета;

Enumeration getInitParameterNames() – определение имен параметров инициализации сервлета из дескрипторного файла **web.xml**;

String getInitParameter(String name) – определение значения конкретного параметра по его имени.

Чтобы задать параметры инициализации сервлета **MyServlet**, необходимо в тег <**servlet**> его описания вложить тег <**init-param**> с описанием имени и значения параметра в виде:

```
<servlet>
  <servlet-name>MyServletname</servlet-name>
  <servlet-class>chapt18.MyServlet</servlet-class>
            <init-param>
                <param-name>mail.smtphost</param-name>
                <param-value>mail.bsu</param-value>
            </init-param>
            <init-param>
                <param-name>mail.smtpport</param-name>
                <param-value>25</param-value>
            </init-param>
</servlet>
```

Тогда для доступа к параметрам инициализации сервлета и их дальнейшего использования можно применить следующую реализацию метода **init()** сервлета:

```
public void init() throws ServletException {
    ServletConfig sc = getServletConfig();

      // определение набора имен параметров инициализации
         Enumeration e = sc.getInitParameterNames();
```

```
        while(e.hasMoreElements()) {
    // определение имени параметра инициализации
            String name = (String)e.nextElement();
    // определение значения параметра инициализации
            String value = sc.getInitParameter(name);
            //...
        }
}
```

Такими же возможностями обладает и объект **ServletContext**, который содержит практически всю информацию о среде, в которой запущен и выполняется сервлет, например:

```
getServletContext().getInitParameter("mail.smtpport");
```

Интерфейсы ServletRequest и HttpServletRequest

Информация от компьютера клиента отправляется серверу в виде объекта запроса типа **HttpServletRequest**. Данный интерфейс является производным от интерфейса **ServletRequest**. Используя методы интерфейса **ServletRequest**, можно получить много дополнительной информации, в том числе и о сервлете и деталях протокола HTTP, закодированной и упакованной в запрос:

String getCharacterEncoding() – определение символьной кодировки запроса;

String getContentType() – определение MIME-типа (Multipurpose Internet Mail Extension) пришедшего запроса;

String getProtocol() – определение названия и версии протокола;

String getServerName(), **getServerPort()** – определение имени сервера, принявшего запрос, и порта, на котором запрос был принят сервером соответственно;

String getRemoteAddr(), **getRemoteHost()** – определение IP-адреса клиента, от имени которого пришел запрос, и его имени соответственно;

String getRemoteUser() – определение имени пользователя, выполнившего запрос;

ServletInputStream getInputStream(), **BufferedReader getReader()** – получение ссылки на поток, ассоциированный с содержимым полученного запроса. Первый метод возвращает ссылку на байтовый поток **ServletInputStream**, а второй – на объект **BufferedReader**. В результате можно прочитать любой байт из полученного объекта-запроса. Если метод **getReader()** был вызван после вызова **getInputStream()** для этого запроса, то генерируется исключение **IllegalStateException** и наоборот.

При обращении к серверу, как правило, передаются параметры и их значения. Для разбора параметров и извлечения их значений применяются методы:

String getParameter(String name) – определение значения параметра по его имени или **null**, если параметр с таким именем не задан;

String[] getParameterValues(String name) – определение всех значений параметра по его имени;

Enumeration getParameterNames() – определение ссылки на список имен всех параметров через объект типа **Enumeration**.

Непосредственно в интерфейсе **HttpServletRequest** объявлен ряд методов, позволяющих манипулировать содержимым запросов:

void setAttribute(String name, Object ob) – установка значения атрибута компонента, являющегося внутренним параметром для передачи информации между компонентами приложения, например от сервлета к странице JSP или другому сервлету;

Enumeration getAttributeNames() – извлечение перечисления имен атрибутов;

Object getAttribute(String name) – извлечение значения переданного атрибута по имени;

Cookie[] getCookies() – извлечение массива cookie, полученного с запросом. Файл cookie – маленький файл, сохраняемый приложением на стороне клиента;

String getMethod() – определение имени метода доступа к ресурсам, на основе которого построен запрос;

String getQueryString() – извлечение строки HTTP-запроса.

В следующем примере рассматривается применение некоторых методов интерфейса **HttpServletRequest** для получения данных о запросе, посылаемом методом **GET** и генерации ответа клиенту.

```
/* пример # 1 : извлечение информации из запроса и счетчик посещений : Request-
Servlet.java : RequestInfo.java : ClickOutput.java */
package chapt18;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;

public class RequestServlet extends HttpServlet {
      // счётчик подключений к сервлету
      private int count = 0;

      public void doGet(HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException {
                  performTask(req, resp);
      }
      public void doPost(HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException {
                  performTask(req, resp);
      }
      private void performTask(HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException {
```

```
      try {
            // установка MIME-типа содержания ответа
resp.setContentType("text/html; charset=Windows-1251");

            // поток для данных ответа
            PrintWriter out = resp.getWriter();
            count = ClickOutput.printClick(out, count);
            //обращение к классу бизнес-логики
            if(count == 1)
                  RequestInfo.printToBrowser(out, req);
            // закрытие потока
            out.close();
      } catch (UnsupportedEncodingException e) {
            System.err.print("UnsupportedEncoding");
      } catch (IOException e) {
            System.err.print("IOErroR");
      }
      }
}
package chapt18;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;

public class RequestInfo {
      static String br = "<br>";

      public static void printToBrowser(
PrintWriter out, HttpServletRequest req) {
out.println("Method: " + req.getMethod());
out.print(br + "Request URI: " + req.getRequestURI());
out.print(br + "Protocol: " + req.getProtocol());
out.print(br + "PathInfo: " + req.getPathInfo());
out.print(br + "Remote Address: " + req.getRemoteAddr());
                  // извлечение имен заголовочной информации запроса
Enumeration e = req.getHeaderNames();
out.print(br + "Header INFO: ");
            while (e.hasMoreElements()) {
                  String name = (String) e.nextElement();
                  String value = req.getHeader(name);
                  out.print(br + name + " = " + value);
                  }
      }
}
package chapt18;
import java.io.PrintWriter;

public class ClickOutput {
```

```
      public static int printClick(
                  PrintWriter out, int count) {

      out.print(++count + " -е обращение." + "<br>");
      return count;
            }
}
```

Приведенный выше сервлет вызывается нажатием кнопки “Выполнить” формы из документа **index.jsp** по адресу URL – **RequestServlet**.

```
<!-- пример # 2 :  стартовая страница вызова сервлета : index.jsp-->
<%@ page language="java" contentType="text/html; char-
set=ISO-8859-1"  pageEncoding="ISO-8859-5"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional
//EN">
<html><head>
<meta http-equiv="Content-Type" content="text/html; char-
set=ISO-8859-5">
<title>Info about Request </title>
</head><body>
 <FORM action="RequestServlet" method="GET">
      <INPUT type="submit"  value="Выполнить">
 </FORM>
</body></html>
```

В результате выполнения в браузер будет выведено:

1-е обращение.
Method: GET
Request URI: /FirstProject/RequestServlet
Protocol: HTTP/1.1
PathInfo: null
Remote Address: 127.0.0.1
Header INFO:
accept = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
referer = http://localhost:8080/FirstProject/index.jsp
accept-language = ru
content-type = application/x-www-form-urlencoded
accept-encoding = gzip, deflate
user-agent = Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
host = localhost:8080
content-length = 0
connection = Keep-Alive
cache-control = no-cache
cookie = JSESSIONID=91014EB2B2208BCA18AE898424B71CEF

Сервлет **RequestServlet** следует попробовать вызывать с различных компьютеров локальной сети или вызвать несколько раз сервлет из нескольких браузеров, запущенных на одном компьютере.

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.

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.

Можно получить ссылки на потоки вывода одним из двух методов:

ServletOutputStream getOutputStream() – извлечение ссылки на поток **ServletOutputStream** для передачи бинарной информации;

PrintWriter getWriter() – извлечение ссылки на поток типа **PrintWriter** для передачи символьной информации;

Если метод **getOutputStream()** уже был вызван для этого ответа, то генерируется **IllegalStateException**. Обратное также верно.

В интерфейсе **HttpServletResponse**, наследующем интерфейс **ServletResponse**, есть еще несколько полезных методов:

void setContentType(String type) – установка MIME-типа генерируемых документов;

void addCookie(Cookie c) – добавление cookie к объекту ответа для последующей пересылки на клиентский компьютер;

void sendError(int sc, String msg) – сообщение о возникших ошибках, где **sc** – код ошибки, **msg** – текстовое сообщение;

void setDateHeader(String name, long date) – добавление даты в заголовок ответа;

void setHeader(String name, String value) – добавление параметров в заголовок ответа. Если параметр с таким именем уже существует, то он будет заменен.

Обработка 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**.

Приведенная на рисунке 18.1 форма является результатом отображения JSP-страницы **index.jsp**, находящейся в папке **/webapps/FirstProject3**.

В форме имеется текстовое поле с именем **name** и значением по умолчанию «**Название проекта**». Значение поля можно изменить непосредственно на странице.

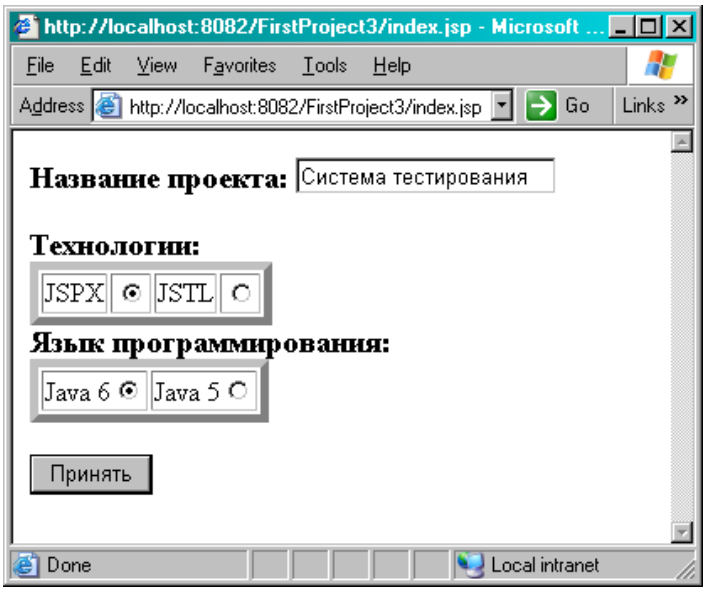


Рис. 18.1. JSP-форма

В форме заданы две группы по два элемента ввода типа **`radio`**, каждая из которых имеет свое имя. При наличии на странице нескольких полей, имеющих одно имя, можно выбрать только один из них. Им задаются соответствующие значения, и при выборе одной из кнопок значение, заданное соответствующей кнопке, заносится в значение своего элемента. По умолчанию для радиогрупп принято задавать одно из значений при помощи свойства **`checked`**.

В итоге пользователь может изменить значения текстового поля и радиогрупп. При нажатии кнопки типа происходит подтверждение формы и вызывается сервлет.

В форме задан метод **`POST`**, при помощи которого происходит передача данных формы в виде отдельных заголовков. Если не задавать этот метод, то по умолчанию будет использоваться метод **`GET`**, и данные формы будут передаваться через универсальный запрос (URL), в который к адресу будут добавлены значения соответствующих элементов.

```
<!-- пример # 3 :  стартовая страница : index.jsp-->
<%@ page language="java" contentType=
      "text/html; charset=utf-8" pageEncoding="utf-8"%>
<html><body>
<FORM action="testform" method=POST>
<H3>Название проекта:
<INPUT type="text" name="Имя проекта" value="-задать!-">
Технологии:
<TABLE BORDER=5> <tr>
<td>JSPX</td><td><INPUT type="radio"
            name="Технология"
            value="JSP в формате XML"></td>
<td>JSTL</td><td><INPUT type="radio"
```

```
            name="Технология"
            value="Библиотека тегов JSTL"></td>
            </tr></TABLE>
Язык программирования:
<TABLE BORDER=5> <tr>
 <td>Java 6<INPUT type="radio"
            name="Язык"
            value="Java SE 6"></td>
<td>Java 5<INPUT type="radio"
            name="Язык"
            value="Java 1.5.0" checked></td>
</tr></TABLE></H3>
      <INPUT type="submit" value="Принять"> <BR>
</FORM>
</body></html>
```

При подтверждении из формы вызывается сервлет **FormRequest**. Сервлет получает и извлекает значения всех переменных формы и отображает их вместе с именами переменных. Для обработки данных, полученных из полей формы, используется приведенный ниже сервлет.

/ пример # 4 : обработка запроса клиента : FormRequest.java : ParameterOutput.java */*

```
package chapt18;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FormRequest extends HttpServlet {
      protected void doPost(HttpServletRequest req,
                        HttpServletResponse resp)
                        throws ServletException {
            performTask(req, resp);
      }
      private void performTask(HttpServletRequest req,
                              HttpServletResponse resp) {
            RequestOutput.generate(resp, req);
      }
}
```

В методе **performTask()** происходит обращение к другому классу-обработчику запроса пользователя с передачей ему объектов **HttpServletRequest req** и **HttpServletResponse resp**.

/ пример # 5 : извлечение информации из запроса клиента : RequestOutput.java */*

```
package chapt18;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class RequestOutput {
      public static void generate(HttpServletResponse resp,
                                  HttpServletRequest req) {
            try {
                  String name, value;
resp.setContentType("text/html; charset=utf-8");
PrintWriter out = resp.getWriter();

      out.print("<HTML><HEAD>");
      out.print("<TITLE>Результат</TITLE>");
      out.print("</HEAD><BODY>");
      out.print("<TABLE BORDER=3>");
      Enumeration names = req.getParameterNames();
            while (names.hasMoreElements()) {
                  name = (String) names.nextElement();
                  value = req.getParameterValues(name)[0];
      /*
name = new String(name.getBytes("ISO-8859-1"), "utf-8");
value = new String(value.getBytes("ISO-8859-1"), "utf-8");
       */
                  out.print("<TR>");
                  out.print("<TD>" + name + "</TD>");
                  out.print("<TD>" + value + "</TD>");
                  out.print("</TR>");
                  }
                  out.print("</TABLE></BODY></HTML>");
                  out.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

В классе в объекте **resp** задается тип содержимого **text/html** и кодировка **UTF-8,** 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 **UTF-8**. Для каждой из переменных извлекаются из запроса соответствующие им значения при помощи метода **getParameterValues(name)**. Тем же способом их кодировка может быть изменена и добавлена в выходной поток.

Класс сервлета относится к пакету **chapt18**, поэтому файл **FormRequest.class** должен быть размещен в папке

/webapps/FirstProject3/WEB-INF/classes/chapt18

и обращение к этому классу, например из документа HTML, должно производиться как **chapt18.FormRequest**. В файле **web.xml** должны находиться строки:

```
<servlet>
    <servlet-name>MyForm</servlet-name>
    <servlet-class>chapt18.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:

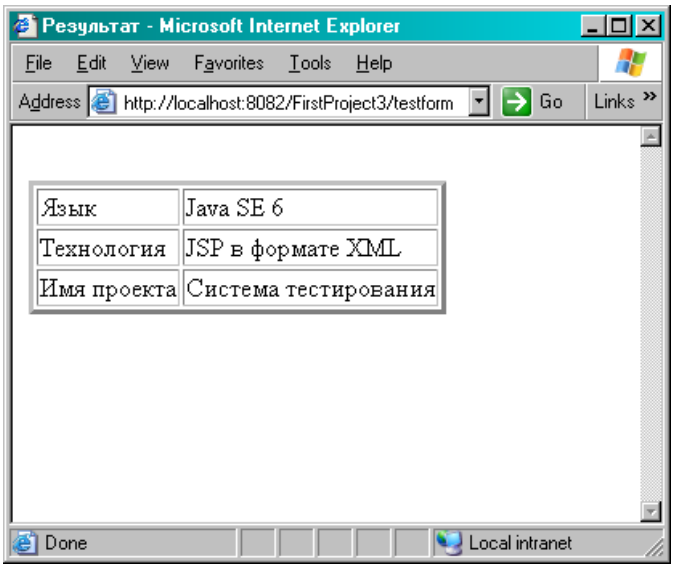


Рис. 18.2. Результат выполнения запроса

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, извлеченных из формы.

Многопоточность

Контейнер сервлетов будет иметь несколько потоков выполнения, распределяемых согласно запросам клиентов. Вероятна ситуация, когда два клиента одно-

временно вызовут методы **doGet()** или **doPost()**. Метод **service()** должен быть написан с учетом вопросов многопоточности. 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 # 6 : доступ к синхронизированным ресурсам :
ServletSynchronization.java */
package chapt18;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
public class ServletSynchronization extends HttpServlet {
    // синхронизируемый объект
  private StringBuffer locked = new StringBuffer();

  protected void doGet(HttpServletRequest req,
                       HttpServletResponse res)
                throws ServletException, IOException {
                performTask(req, res);
  }
  private 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("</BODY></HTML>");
      out.flush();
      out.close();
    } catch (IOException e) {
      throw new RuntimeException(
        "Failed to handle request: " + e);
    }
  }
  protected String createString() {
      // оригинал строки
      final String SYNCHRO = "SYNCHRONIZATION";

    synchronized (locked) {
      try {
        for (int i = 0; i < SYNCHRO.length(); i++) {
          locked.append(SYNCHRO.charAt(i));
          Thread.sleep(500);
        }
```

```
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        String result = locked.toString();
        locked.delete(0, SYNCHRO.length() - 1);
        return result;
      }
    }
}
```

Результаты работы сервлета при наличии и отсутствии синхронизации представлены на рисунках.

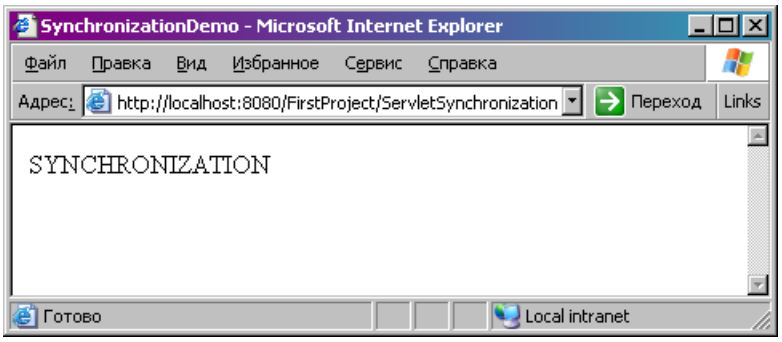
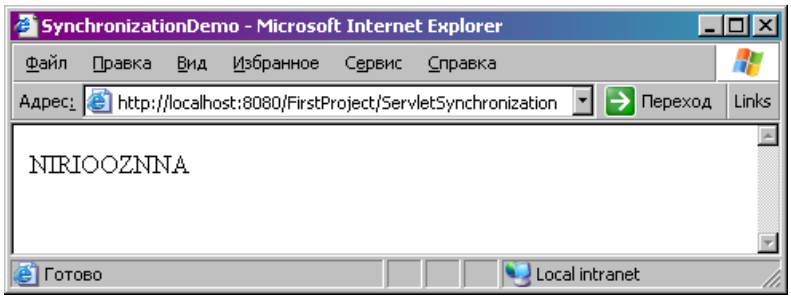


Рис. 18.3. Результат работы сервлета **Synchronization** с блоком синхронизации



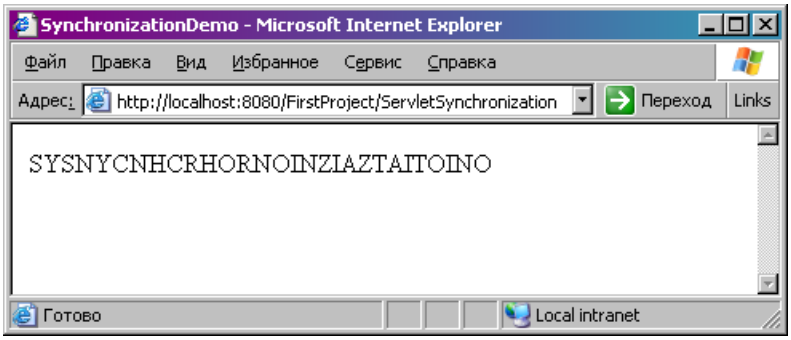


Рис. 18.4. Результат работы сервлета **Synchronization** без синхронизации

Можно синхронизировать и весь сервлет целиком, но причиной, почему это не делается, является возможность нахождения критической секции вне основного пути выполнения программы.

Электронная почта

Рассылка электронной почты, в том числе и автоматическая, является стандартным родом деятельности при использовании Web-приложений. Собственный почтовый сервер создать достаточно легко, только необходимо указать адрес почтового севера, который будет использован в качестве транспорта.

Следующий пример использует интерфейсы API JavaMail для работы с электронной почтой в сервлетах и JSP. API JavaMail содержит классы, с помощью которых моделируется система электронной почты. Класс **`javax.mail.Session`** представляет сеанс почтовой связи, класс **`javax.mail.Message`** – почтовое сообщение, класс **`javax.mail.internet.InternetAddress`** – адреса электронной почты.

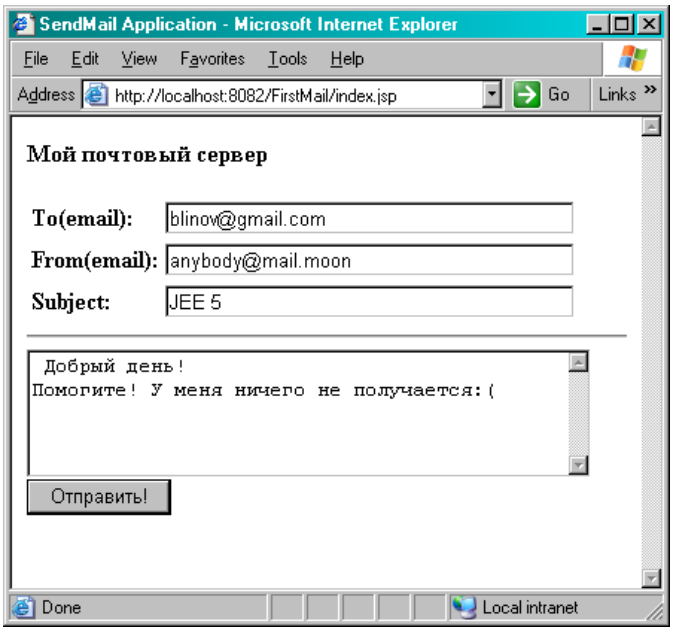


Рис. 18.5. Формирование запроса на отправку письма

Для работы с данной частью платформы J2EE необходимо скачать zip-файлы, расположенные по адресу **`http://java.sun.com/products/javamail/`**, содержащие архивы **`mail.jar`** и **`activation.jar`**. И добавить эти файлы в каталог jar-файлов серверам приложений (**`common/lib`** для Tomcat). Также необходимо запустить почтовую программу James, являющуюся также одним из проектов **`apache.org`**.

Ниже приведена страница JSP, содержащая форму для заполнения основных полей: «**`Кому`**» – «**`to`**», «**`От кого`**» – «**`from`**», «**`Тема сообщения`**» – «**`subj`**»

```
<!-- пример # 7 : страница создания электронного письма : index.jsp-->
<%@ page language="java" contentType=
```

```
"text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>SendMail Application</title></head>
      <b>Мой почтовый сервер</b>
      <form method="post" action="sendmail">
      <table>
            <tr><td><b>To(email):</b></td><td>
<input name="to" type="text" size=40></td></tr>
            <tr><td><b>From(email):</b></td><td>
<input name="from" type="text" size=40></td></tr>
            <tr><td><b>Subject:</b></td><td>
<input name="subj" type="text" size=40></td></tr>
      </table>
            <hr>
<textarea name="body" type="text" rows=5 cols=45>
Добрый день!</textarea>
            <br>
            <input type="submit" value="Отправить!">
      </form>
</body></html>
```

Параллельные процессы по отправке письма и предложению пользователю в это же самое время создать новое письмо организуются с применением потока в следующем сервлете.

```
/* пример # 8 : доступ к синхронизированным ресурсам :
SendMailServlet.java */
package chapt18;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.activation.*;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

public class SendMailServlet
      extends javax.servlet.http.HttpServlet
            implements javax.servlet.Servlet {
      //объект почтовой сессии
       private Session mailSession = null;

            public void init(ServletConfig config)
                  throws ServletException {
            //mailSession = Session.getDefaultInstance(System.getProperties());
```

```
            final String host = "mail.smtphost";
            final String port = "mail.smtpport";
      //запрос параметров почтового сервера из web.xml
      String hostvalue = config.getInitParameter(host);
      String portvalue = config.getInitParameter(port);
java.util.Properties props = new java.util.Properties();
      //загрузка параметров почтового сервера в объект свойств
        props.put(host, hostvalue);
        props.put(port, portvalue);
      //загрузка параметров почтового сервера в объект почтовой сессии
  mailSession = Session.getDefaultInstance(props, null);
      }
      protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
            throws ServletException, IOException {
      //извлечение параметров письма из запроса
      String from = request.getParameter("from");
      String to = request.getParameter("to");
      String subject = request.getParameter("subj");
      String content = request.getParameter("body");
            if((from == null) || (to == null)
      || (subject == null) || (content == null)) {
      /*при отсутствии одного из параметров предлагается повторить
                                                        ввод*/
                  response.sendRedirect("index.jsp");
                  return;
            }
//запуск процесса отправки почты в отдельном потоке
(new MailSender(to, from, subject, content)).start();
//формирование страницы с предложение о создании нового письма
response.setContentType("text/html; charset=CP1251");
PrintWriter out = response.getWriter();
out.println("<html><head><title>");
out.println("SendMail Application</title></head>");
out.println("Message to <b>" + to
            + "</b> sending in progress");
out.println("<a href = \"index.jsp\">New message</a>");
out.println("</body></html>");
      }
      private class MailSender extends Thread {
            private String mailTo;
            private String mailFrom;
            private String mailSubject;
            private String mailContent;
          MailSender(String mailTo, String mailFrom,
            String mailSubject, String mailContent) {
            setDaemon(true);
            this.mailTo = mailTo;
```

```
                this.mailFrom = mailFrom;
                this.mailSubject = mailSubject;
                this.mailContent = mailContent;
                }
                public void run() {
                      try {
          //создание объекта почтового сообщения
          Message message = new MimeMessage(mailSession);
          //загрузка параметров в объект почтового сообщения
          message.setFrom(new InternetAddress(mailFrom));
          message.setRecipient(Message.RecipientType.TO,
                      new InternetAddress(mailTo));
          message.setSubject(mailSubject);
          message.setContent(mailContent, "text/plain");
          //отправка почтового сообщения
                Transport.send(message);
                      } catch(AddressException e) {
                            e.printStackTrace();
                          System.err.print("Ошибка адреса");
                      } catch(MessagingException e) {
                            e.printStackTrace();
                          System.out.print("Ошибка сообщения");
                      }
                }
          }
    }
```

В результате в браузер будет выведено:

Message to blinov@gmail.com sending in progress New message

где **New message** представляет собой активную ссылку, перенаправляющую при запуске на **index.jsp** для создания еще одного письма. Процесс же отправки письма будет функционировать независимо от дальнейшей работы приложения.

Файл **web.xml** для данного приложения имеет вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
      <display-name>FirstMail</display-name>
      <servlet>
            <display-name>
            SendMailServlet</display-name>
            <servlet-name>SendMailServlet</servlet-name>
            <servlet-class>
            SendMailServlet</servlet-class>
            <init-param>
                  <param-name>mail.smtphost</param-name>
```

```
                <param-value>mail.bsu.bsu</param-value>
            </init-param>
            <init-param>
                <param-name>mail.smtpport</param-name>
                <param-value>25</param-value>
            </init-param>
        </servlet>
        <servlet-mapping>
         <servlet-name>chapt18.SendMailServlet</servlet-name>
         <url-pattern>/sendmail</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
</web-app>
```

В качестве значения параметра **mail.smtphost** можно попробовать использовать адрес почтового сервера **mail.attbi.com**.

Задания к главе 18

Вариант А

Создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы, 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 0ºС; 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 задаются 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 В

Для заданий варианта В 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 18

Вопрос 18.1.

Каким образом в методе **`init()`** сервлета получить параметр инициализации сервлета с именем “URL”? (выберите два)

1) `ServletConfig.getInitParameter("URL");`
2) `getServletConfig().getInitParameter("URL");`
3) `this.getInitParameter("URL");`
4) `HttpSerlet.getInitParameter("URL");`
5) `ServletContext.getInitParameter("URL").`

Вопрос 18.2.

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 18.3.

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 18.4.

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 18.5.

Какие интерфейсы находятся в пакете **`javax.servlet`**?

1) `ServletRequest`;
2) `ServletOutputStream`;
3) `PageContext`;
4) `Servlet`;
5) `ServletContextEvent`;
6) ни один из перечисленных.

Вопрос 18.6.

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 19

JAVA SERVER PAGES

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems, чтобы облегчить создание страниц с динамическим содержанием.

В то время как сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения в виде обработки запросов и определения вида ответа, страницы JSP выполняют функцию формирования текстовых документов типа HTML, XML, WML и некоторых других.

Под терминами “динамическое/статическое содержание” обычно понимаются не части JSP, а содержание Web-приложения:

1) динамические ресурсы, изменяемые в процессе работы: сервлеты, JSP, а также java-код;
2) статические ресурсы, не изменяемые в процессе работы – HTML, JavaScript, изображения 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.

Рекомендуется разделить и разрабатывать параллельно две части приложения: Web-приложение, состоящее только из динамических ресурсов, и Web-приложение, состоящее только из статических ресурсов.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

d) *Разделение динамического и статического содержания.*
Возможность разделить логику приложения и дизайн Web-страницы снижает сложность разработки Web-приложений и упрощает их поддержку.

d) *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.

e) *Многократное использование компонентов.*
Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание Web-сайтов.

f) *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.

Чтобы облегчить внедрение динамической структуры, JSP использует ряд тегов, которые дают возможность проектировщику страницы вставить значение полей объекта JavaBean в файл JSP.

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.

JSP составляется из стандартных HTML-тегов, JSP-тегов, action-тегов, JSTL и пользовательских тегов. В спецификации JSP 2.0 существует пять основных тегов:

<%@ *директива* **%>** – используются для установки параметров серверной страницы JSP.

<%! *объявление* **%>** – содержит переменные Java и методы, которые вызываются в expression-блоке и являются полями генерируемого сервлета. Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптлетах и выражениях.

<% *скриптлет* **%>** – вживление Java-кода в JSP-страницу. 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 **_jspService()** на этапе трансляции.

<%= *вычисляемое выражение* **%>** – операторы языка Java, которые вычисляются, после чего результат вычисления преобразуется в строку **String** и посылается в поток **out**.

<%-- *JSP-комментарий* --%> – комментарий, который не отображается в исходных кодах JSP-страницы после этапа выполнения.

Стандартные элементы action

Большинство тегов, объявленных выше, применяются не так уж часто. Наиболее используемыми являются стандартные действия версии JSP 2.0. Они позволяют создавать правильные JSP –документы с помощью следующих тегов:

1) **jsp:declaration** – объявление, аналогичен тегу **<%! ... %>**;
2) **jsp:scriptlet** – скриптлет, аналогичен тегу **<% ... %>**;
3) **jsp:expression** – скриптлет, аналогичен тегу **<%= ... %>**;
4) **jsp:text** – вывод текста;
2 **jsp:useBean** – позволяет использовать экземпляр компонента Java Bean. 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 этот объект можно вызывать доступные методы класса. Специфика компонентов JavaBean в том, что если компонент имеет поле **field**, экземпляр компонента имеет параметр **field**, а метод, устанавливающий значение, должен называться **setField(type value)**, возвращающий значение – **getField()**.

```
package test;
public class MyBean {
     private String field = "нет информации";
     public String getField() {
           return info;
     }
     public void setField(String f) {
           field = f;
     }
}
```

5) **jsp:setProperty** – позволяет устанавливать значения полей указанного в атрибуте **name** объекта. Если установить значение **property** в «*», то значения свойств компонента JavaBean будут установлены таким образом, что будет определено соответствие между именами параметров и именами методов-установщиков (setter-ов) компонента:

```
<jsp:setProperty name="ob"
                        property="field"
                        value="привет" />
```

e) **jsp:getProperty** – получает значения поля указанного объекта, преобразует его в строку и отправляет в неявный объект **out**:

```
<jsp:getProperty name="ob" property="field" />
```

f) **jsp:include** – позволяет включать файлы в генерируемую страницу при запросе страницы:

```
<jsp:include page="относительный URL"
                        flush="true"/>
```

g) **jsp:forward** – позволяет передать запрос другой странице:

```
<jsp:forward page="относительный URL"/>
```

h) **jsp:plugin** – замещается тегом **<OBJECT>** или **<EMBED>**, в зависимости от типа браузера, в котором будет выполняться подключаемый апплет или Java Bean.

i) **jsp:params** – группирует параметры внутри тега **jsp:plugin**.

6) **jsp:param** – добавляет параметры в объект запроса, например в элементах **forward**, **include**, **plugin**.

7) **jsp:fallback** – указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься. Используется внутри элемента **plugin**.

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 11, 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.

JSP-документ

Предпочтительно создавать JSP-страницу в виде JSP-документа – корректного XML-документа, который ссылается на определенное пространство имен, содержит стандартные действия JSP, пользовательские теги и теги ядра JSTL, XML-эквиваленты директив JSP. В JSP-документе вышеперечисленные пять тегов неприменимы, поэтому их нужно заменять стандартными действиями и корректными тегами. JSP-документы необходимо сохранять с расширением **.jspx**.

Директива **taglib** для обычной JSP:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>
```

для JSP-документа:

```
<jsp:root xmlns:c="http://java.sun.com/jsp/jstl/core"/>
```

Директива **page** для обычной JSP:

```
<%@ page contentType="text/html"%>
```

для JSP-документа:

```
<jsp:directive.page contentType="text/html" />
```

Директива **include** для обычной JSP:

```
<%@ include file="file.jspf"%>
```

для JSP-документа:

```
<jsp:directive.include file="file.jspf" />
```

Ниже приведены два примера, демонстрирующие различие применения стандартных действий и тегов при создании JSP-страниц и JSP-документов.

```
<!--пример # 1 : обычная jsp-страница: page.jsp -->
<%@ page contentType="text/html; charset=Cp1251" %>
<html><head><title>JSP-страница</title></head>
<%! private int count = 0;
      String version = new String("J2EE 1.5");
      private String getName(){return "J2EE 1.6";} %>
```

```
<% out.println("Значение count: "); %>
<%= count++ %>
<br/>
<% out.println("Значение count после инкремента: " +
count); %>
<br/>
<% out.println("Старое значение version: "); %>
<%= version %>
<br/>
<% version=getName();
out.println("Новое значение version: " + version); %>
</html>
```

Версия в виде JSP-документа несколько более громоздка, но читать и искать ошибки в таком документе проще, нежели в предыдущем.

```
<!--пример # 2 : правильный jsp-документ : page.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
                                        version="2.0">
      <jsp:directive.page contentType=
                           "text/html; charset=UTF-8" />
      <html><body>
      <jsp:declaration>
            private int count = 0;
            String version = new String("J2EE 1.5");
            private String getName(){return "J2EE 1.6";}
      </jsp:declaration>
      <jsp:scriptlet>
            out.println("Значение count: ");
      </jsp:scriptlet>
      <jsp:expression>
            count++
      </jsp:expression>
      <br />
      <jsp:scriptlet>
            out.println("Значение count после инкремента:"
                                        + count);
      </jsp:scriptlet>
      <br/>
      <jsp:scriptlet>
      out.println("Старое значение version: ");
      </jsp:scriptlet>
      <jsp:expression> version </jsp:expression>
      <br/>
      <jsp:scriptlet> version=getName();
      out.println("Новое значение version: " + version);
      </jsp:scriptlet>
      </body></html>
</jsp:root>
```

Далее в главе примеры будут приведены в виде JSP-документов.

JSTL

JSP-страницы, включающие скриптлеты, элементы action (стандартные действия) и пользовательские теги, не могут быть технологичными без использования JSTL (JSP Standard Tag Library). Создание страниц с применением JSTL позволяет упростить разработку и отказаться от вживления Java-кода в JSP. Как было показано ранее, страницы со скриптлетами трудно читаемы, что вызывает проблемы как у программиста, так и у веб-дизайнера, не обладающего глубокими познаниями в Java.

Библиотеку JSTL версии 1.1.2 (**jstl-1.1.2.jar** и **standard-1.1.2.jar**) или более позднюю версию можно загрузить с сайта apache.org. Библиотеки следует разместить в каталоге **/lib** проекта. При указании значения параметра **xmlns** элемента **root** (для JSP-страницы значение параметра **taglib uri=""**) необходимо быть внимательным, так как если адрес будет указан неправильно, JSP-страница не сможет иметь доступ к тегам JSTL. Проверить правильность значение параметра **uri** (оно же справедливо и для параметра **xmlns**) можно в файле подключаемой библиотеки (например **c.tld**). Простейшая JSP с применением тега JSTL, выводящим в браузер приветствие будет выглядеть следующим образом:

```
<!--пример # 3  : правильный jsp-документ: simple.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c=http://java.sun.com/jsp/jstl/core
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8"/>
<html><body>
<c:out value="Welcome to JSTL"/>
</body></html>
</jsp:root>
```

Тег **<c:out/>** отправляет значение параметра **value** в поток **JspWriter**.

JSTL предоставляет следующие возможности:

- поддержку Expression Language, что позволяет разработчику писать простые выражения внутри атрибутов тега и предоставляет “прозрачный” доступ к переменным в различных областях видимости страницы;
- организацию условных переходов и циклов, основанную на тегах, а не на скриптовом языке;
- простое формирование доступа (URL) к различным ресурсам;
- простую интернационализацию JSP;
- взаимодействие с базами данных;
- обработку XML, а также форматирование и разбор строк.

Expression Language

В JSTL вводится понятие Expression Language (EL). EL используется для упрощения доступа к данным, хранящимся в различных областях видимости (**page**, **request**, **application**) и вычисления простых выражений.

EL вызывается при помощи конструкции **“${**имя**}”**.

Начиная с версии спецификации JSP 2.0 / JSTL 1.1, EL является частью JSP и поддерживается без всяких сторонних библиотек. С версии web-app 2.4 атрибут

isELIgnored по умолчанию имеет значение **true**. В более ранних версиях необходимо указывать его в директиве **page** со значение **true.**

EL-идентификатор ссылается на переменную, возвращаемую вызовом **PageContext.findAttribute(**имя**)**. В общем случае переменная может быть сохранена в любой области видимости: **page**(**PageContext**), **request**(**HttpServletRequest**), **session** (**HttpSession**), **application** (**ServletContext**). В случае если переменная не найдена, возвращается **null**. Также возможен доступ к параметрам запроса через предопределённый объект **paramValues** и к заголовкам запроса через **requestHeaders**.

Данные приложения, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой коллекции, такие как **List**, **Map**, **Array** и др. EL предоставляет доступ к этим объектам при помощи операторов **"."** и **"[]"**. Применение этих операторов зависит от типа объекта. Например:

```
<c:out value="${student.name}"/>
```

```
<!--пример # 4 : правильный jsp-документ : simple2.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
                              "text/html; charset=Utf-8"/>
<html>
<head><title>Простое использование EL</title></head>
<body>
<c:set var="login" value="Бендер" scope="page"/>
<c:out value="${login} in Rio"/>
<br/>
<c:out value="Бендер в байтовом виде: ${login.bytes} "/>
</body></html>
</jsp:root>
```

С помощью оператора **"."** можно вызывать некоторые методы класса, к которому принадлежит объект. Вызов **login.bytes** в переводе на обычную Java означает **login.getBytes()**.

В результате запуска этого документа в браузер будет выведено:

Бендер in Rio
Бендер в байтовом виде: [B@edf730

Операторы в EL поддерживают наиболее часто используемые манипуляции данными.

Типы операторов:

Стандартные операторы отношения:

== (или **eq**), **!=** (или **neq**), **<** (или **lt**), **>** (или **gt**), **<=** (или **le**), **>=** (или **ge**).

Арифметические операторы: **+**, **-**, *****, **/** (или **div**), **%** (или **mod**).

Логические операторы: **&&** (или **and**), **||** (или **or**), **!** (или **not**).

Оператор **empty** – используется для проверки переменной на **null**, или "пустое значение". Термин "пустое значение" зависит от типа проверяемого объекта. Например, нулевая длина для строки или нулевой размер для коллекции.

Например:

```
<c:if test="${not empty user and user.name neq 'guest'}>
  User is Customer.
</c:if>
```

Автоматическое приведение типов

Данные не всегда имеют тот же тип, который ожидается в EL-операторе. EL использует набор правил для автоматического приведения типов. Например, если оператор ожидает параметр типа **Integer**, то значение идентификатора будет приведено к типу **Integer** (если это возможно).

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 в соответствии с протоколом 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.

JSTL core

Библиотека тегов JSTL состоит из четырёх групп тегов: основные теги – **core**, теги форматирования – **fomatting**, теги для работы с SQL – **sql**, теги для обработки XML – **xml**.

Library	Actions	Description
core	14	Основные: **if**/**then** выражения и **switch** конструкции; вывод; создание и удаление контекстных переменных; управление свойствами JavaBeans компонентов; перехват исключений; итерирование коллекций; создание URL и импортирование их содержимого.
formatting	12	Интернационализация и форматирование: установка локализации; локализация текста и структуры сообщений; форматирование и анализ чисел, процентов, денег и дат.
sql	6	Доступ к БД: описание источника данных; выполнение запросов, обновление данных и транзакций; обработка результатов запроса.
xml	10	XML-анализ и преобразование: преобразование XML; доступ и преобразование XML через XPath и XSLT.

Библиотека **core** содержит в себе наиболее часто используемые теги.

<%@taglib uri="http://java.sun.com/jstl/core" prefix="c" %> – для обычной JSP.

<jsp:root version="1.2" xmlns:c=
"http://java.sun.com/jstl/core"**> ...</jsp:root>** – для XML формата JSP.

Теги общего назначения:

<c:out /> — вычисляет и выводит значение выражения;

<**c:set />** — устанавливает переменную в указанную область видимости;

<c:remove /> — удаляет переменную из указанной области видимости;

<c:catch /> — перехватывает обработку исключения.

Теги условного перехода:

<c:if /> — тело тега вычисляется только в том случае, если значение выражения **true**;

<c:choose /> (**<c:when />**, **<c:otherwise />**) – то же что и **<c:if />** с поддержкой нескольких условий и действия, производимого по умолчанию.

Итераторы:

<c:forEach /> – выполняет тело тега для каждого элемента коллекции;

<c:forTokens /> – выполняет тело тега для каждой лексемы в строке.

Теги обработки URL:

<c:redirect /> – перенаправляет запрос на указанный **URL**;

<c:import/> – добавляет на JSP содержимое указанного WEB-ресурса;

<c:url /> – формирует адрес с учётом контекста приложения **request.getContextPath()**.

<c:param /> – добавляет параметр к запросу, сформированному при помощи **<c:url/>**.

Ниже приведено несколько примеров, иллюстрирующих применение основных тегов из группы **core**.

```
<!--пример #5 : демонстрация работы тегов  c:set, c:remove, c:if, c:out: core1.jspx
-->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      version="2.0">
<jsp:directive.page contentType=
                              "text/html; charset=Utf-8" />
<html><head><title>Демонстрация тегов core</title></head>
<h3> Демонстрация работы тегов c:set, c:remove, c:if, c:out
<br/></h3>
      <form>
Новое значение переменной:<input type="text" name="set" />
<br/>
Удалить переменную:<input type="checkbox" name="del" />
<br/>
<input type="submit" name="send" value="принять"/><br/>
      </form>
<c:if test="${not empty param.send }">
      <c:if test="${not empty param.set }">
            <c:set var="item" value="${param.set}"
                  scope="session"></c:set>
      </c:if>
      <c:if test="${not empty param.del }">
            <c:remove var="item"/>
      </c:if>
</c:if>
<c:if test="${not empty item }">
      <jsp:text>Значение переменной :</jsp:text>
      <c:out value="${item }"/>
</c:if>
```

```
<c:if test="${empty item and not empty param.send}">
      <jsp:text>Значение переменной: </jsp:text>
      <c:out value="пусто"/>
</c:if>
</html>
</jsp:root>
```

<!--пример # 6 : демонстрация работы тегов c:forEach, c:choose, c:when, c:otherwise : core2.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<html><head>
<title>Демонстрация тегов core</title>
</head>
<h3>Демонстрация работы тегов <br/>c:forEach, c:choose,
c:when, c:otherwise</h3>
<jsp:text>Ниже приведены случайно сгенерированные элементы
массива <br/> и сделана их оценка по отношению к числу 50:
<br/></jsp:text>
<jsp:useBean id="arr" class="chapt21.Arr" />
<c:set var="items" value="${arr.fillMap}"
       scope="session" />
    <c:forEach var="id" items="${items}">
        <c:out value="${id}" />
            <c:choose>
                  <c:when test="${id > 50}" >
                        <c:out value=" - число больше 50"/>
                  </c:when>
                  <c:otherwise>
                        <c:out value=" - число меньше 50"/>
                  </c:otherwise>
            </c:choose>
            <br/>
    </c:forEach>
</html>
</jsp:root>
```

Элемент JavaBean, используемый в данном примере, – класс **Arr**, генерирующий массив из пяти случайных чисел. Его описание хранится в файле **Arr.java**. Исходный Java-файл хранится в каталоге на верхнем уровне приложения (например каталог **build**, **src** или **JavaSourse**) в зависимости от настроек web-приложения. Значение атрибута **class** равно **chapt21.Arr** тега **jsp:useBean** и означает, что файл находится в пакете **chapt21**, а имя класса – **Arr**. EL-выражение **arr.fillMap** означает вызов метода **getfillMap()**.

// пример # 7 : javabean класс : Arr.java

```
package chapt21;
public class Arr  {
      public Arr(){}
      public  String[] getfillMap(){
      String str[] = new String[5];
      for (int i =0; i < str.length ; i++){
String r = Integer.toString((int)(Math.random()*100));
            str[i] = r;
      }
      return str;
      }
}
```

В результате в браузер будет выведено:

1) Демонстрация работы тегов
c:forEach, c:choose, c:when, c:otherwise

Ниже приведены случайно сгенерированные элементы массива
и сделана их оценка по отношению к числу 50:
8 - число меньше 50
68 - число больше 50
84 - число больше 50
5 - число меньше 50
36 - число меньше 50

Следующие примеры показывают работу тегов по взаимодействию с другими документами и ресурсами.

<!--пример # 8 : демонстрация работы тегов c:import, c:url, c:redirect, c:param : url.jsp -->

```
<%@ page language="java" contentType=
            "text/html; charset=Cp1251"
            pageEncoding="Cp1251"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
            prefix="c" %>
<html><head><title>Переход по ссылке</title></head>
<body>
<h3>Данная страница демонстрирует работу тегов
<br/>c:import, c:url, c:param, c:redirect</h3><br/>
<c:import url="\WEB-INF\jspf\imp.jspf"
      charEncoding="Cp1251"/>
<c:url value="redirect.jspx" var="myUrl" />
<a href='<c:out value="${myUrl}"/>' />Перейти</a>
</body></html>
```

<!--пример # 9 : фрагмент, включаемый с помощью тега c:import (находится в каталоге WEB-INF/jspf/) : imp.jspf -->

```
<h5>importing by using c:import from jspf</h5>
```

В результате запуска страницы **url.jsp** в браузер будет выведено:

Данная страница демонстрирует работу тегов
c:import, c:url, c:param, c:redirect.
2) importing by using c:import from jspf
Перейти

В **url.jsp** был импортирован фрагмент **imp.jspf**, а также с помощью тега **c:url** была задана активная ссылка, при активации которой будет осуществлен переход на страницу **redirect.jspx**.

```
<!--пример # 10 : демонстрация работы тегов  c:redirect, c:param: redirect.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<html><head><title>Демонстрация тегов core</title></head>
<body>
<c:redirect url="urldestination.jspx">
      <c:param name="fname" value="Ostap"/>
      <c:param name="lname" value="Bender"/>
</c:redirect>
</body></html>
</jsp:root>
```

В документе были объявлены два параметра и заданы их значения, а также был автоматически выполнен переход на документ **urldestination.jspx**.

```
<!--пример # 11 : конечная страница, на которую был перенаправлен запрос
и переданы данные: urldestination.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head>
      <title>Демонстрация работы тега c:url</title>
</head>
<body>
<jsp:text>
      Ваш запрос был перенаправлен на эту страницу<br/>
      Параметры, переданные с помощью тега c:param:<br/>
</jsp:text>
<c:forEach var="ps" items="${param}">
<c:out value="${ps.key} - ${ps.value}"/><br/>
</c:forEach>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Ваш запрос был перенаправлен на эту страницу.
Параметры, переданные с помощью тега c:param:

lname - Ostap
fname - Bender

JSTL fmt

Библиотека содержит теги форматирования и интернационализации.
<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
– для обычной страницы JSP;
<jsp:root version="1.2" xmlns:fmt=
"http://java.sun.com/jstl/fmt">...</jsp:root> – для JSP-документа.

Теги интернационализации:

<fmt:setLocale/> – устанавливает объект класса **Locale**, используемый на странице;

<fmt:setBundle/>, **<fmt:bundle/>** – устанавливают объект **ResourceBundle**, используемый на странице. В зависимости от установленной локали выбирается **ResourceBundle**, соответствующий указанному языку, стране и региону;
<fmt:message/> – выводит локализованное сообщение.

Теги форматирования:

<fmt:timeZone/>, **<fmt:setTimeZone/>** – устанавливает часовой пояс, используемый для форматирования;

<fmt:formatNumber/>, **<fmt:formatDate/>** – форматирует числа/даты с учётом установленной локали (региональных установок) либо указанного шаблона;

<fmt:parseNumber/>, **<fmt:parseDate/>** – переводит строковое представление числа/даты в объекты подклассов **Number** / **Date**.

Ниже приведены три примера на использование тегов из группы **fmt**.

Документ **formatdatejsp.jspx** выводит на экран текущую дату и время с учётом установленного объекта класса **Locale**.

```
<!--пример # 12 : вывод даты и времени :  formatdate.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:fmt="http://java.sun.com/jsp/jstl/fmt" version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head><title>Формат даты</title></head>
<body>
<jsp:useBean id="now" class="java.util.Date" />
<fmt:setLocale value="en-EN"/>
<jsp:text>Вывод даты в формате English</jsp:text><br/>
Сегодня: <fmt:formatDate value="${now}" /><br/>
<fmt:setLocale value="ru-RU"/>
<jsp:text>Вывод даты в формате Russian</jsp:text><br/>
Сегодня: <fmt:formatDate value="${now}" /><br/>
```

```
Время(стиль-short): <fmt:formatDate value="${now}"
type="time" timeStyle="short" /><br/>
Время(стиль-medium): <fmt:formatDate value="${now}"
type="time" timeStyle="medium" /><br/>
Время(стиль-long): <fmt:formatDate value="${now}"
type="time" timeStyle="long" /><br/>
Время(стиль-full): <fmt:formatDate value="${now}"
type="time" timeStyle="full" /><br/>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Вывод даты в формате English
Сегодня: Aug 14, 2007
Вывод даты в формате Russian
Сегодня: 14.08.2007
Время (стиль-short): 23:23
Время (стиль-medium): 23:23:02
Время (стиль-long): 23:23:02 EEST
Время (стиль-full): 23:23:02 EEST

В следующем примере реализован ещё один способ вывода времени и даты

<!--пример # 13 : полный вывод даты и времени : timezone.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
      version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head><title>timezone</title></head>
<body>
<jsp:useBean id="now" class="java.util.Date" />
<jsp:text>
Вывод даты и времени с помощью тега<br/> fmt:formatDate
и установки TimeZone
</jsp:text><br/>
<fmt:setLocale value="ru-RU"/>
<fmt:timeZone value="GMT+4:00">
<fmt:formatDate value="${now}" type="both"
                dateStyle="full" timeStyle="full"/><br/>
</fmt:timeZone>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Вывод даты и времени с помощью тега
fmt:formatDate и установки TimeZone
15 Август 2007 г. 0:26:38 GMT+04:00

Страница **formatnumber.jspx** выводит формат числа в соответствии с установленными региональными установками.

<!--пример # 14 : формат чисел : formatnumber.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:fmt="http://java.sun.com/jsp/jstl/fmt" version="2.0">
<jsp:directive.page contentType=
      "text/html; charset=Utf-8"/>
<html><head><title>fomatnumber</title></head>
<body>
Вывод формата числа 9876543.21: <br/>
<jsp:text>Обычный формат - </jsp:text>
<fmt:formatNumber value="9876543.21" /><br/>
<jsp:text>Процентный формат - </jsp:text>
<fmt:formatNumber value="9876543.21" type="percent"/><br/>
<fmt:setLocale value="ru-RU"/>
<jsp:text>Русская валюта - </jsp:text>
<fmt:formatNumber value="9876543.21" type="currency"/><br/>
<fmt:setLocale value="en-EN"/>
<jsp:text>Английская валюта - </jsp:text>
<fmt:formatNumber value="9876543.21" type="currency"/><br/>
<jsp:text>Французская валюта - </jsp:text>
<fmt:setLocale value="fr-FR"/>
<fmt:formatNumber value="9876543.21" type="currency"/><br/>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Вывод формата числа 9876543.21:
Обычный формат - 9 876 543,21
Процентный формат - 987 654 321%
Русская валюта - 9 876 543,21 руб.
Английская валюта - ¤9,876,543.21
Французская валюта - 9 876 543,21 €

JSTL sql

Используется для выполнения запросов SQL непосредственно из JSP и обработки результатов запроса в JSP.

<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%> — для обычной страницы JSP;

<jsp:root version="1.2" xmlns:sql=
"http://java.sun.com/jstl/sql">...</jsp:root> — для JSP-документа.

Теги:

<sql:dateParam> – определяет параметры даты для **<sql:query>** либо **<sql:update>**;

<sql:param> – определяет параметры **<sql:query>** либо **<sql:update>**;

<sql:query> – выполняет запрос к БД;

<sql:setDataSource> – усанавливает data source для **<sql:query>**, **<sql:update>**, и **<sql:transaction>** тегов;

<sql:transaction> – объединяет внутренние теги **<sql:query>** и **<sql:update>** в одну транзакцию;

<sql:update> - выполняет преобразование БД.

В промышленном программировании данная библиотека не используется из-за прямого доступа из JSP в СУБД, что является явным нарушением шаблона MVC.

JSTL xml

Используется для обработки данных XML в JSP-документе.

<% @taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>

– для обычной JSP-страницы,

<jsp:root version="1.2" xmlns:x=

"http://java.sun.com/jstl/xml">...</jsp:root> – для XML формата JSP.

Теги:

<x:forEach> – XML-версия тега **<c:choose>**;

<x:choose> – XML-версия тега **<c:forEach>**;

<x:if> – XML-версия тега **<c:if>**;

<x:otherwise> – XML-версия тега **<c:otherwise>**;

<x:out> – XML-версия тега **<c:out>**;

<x:param> – XML-версия тега **<c:param>**, определяющая параметры для другого тега **<x:transform>**;

<x:parse> – разбор XML-документа;

<x:set> – XML-версия тега **<c:set>**;

<x:transform> – трансформация XML-документа;

<x:when> – XML-версия тега **<c:when>**;

<x:choose> – XML-версия тега **<c:choose>**;

<x:forEach> – XML-версия тега **<c:forEach>**.

Включение ресурсов

В реальных проектах JSP-страницы часто состоят из статических элементов. Для этого используется директива **include**, а файл, содержащий необходимый статичный элемент, сохраняется с расширением **.jspf**, что означает «фрагмент JSP». При необходимости включения содержимого в JSP-страницу каждый раз, когда та получает запрос, используется стандартное действие **jsp:include**. В этом случае включаемые сегменты имеют доступ к объектам **request**, **session** и **application** исходной страницы и ко всем атрибутам, которые имеют эти объекты. Если использовать директиву **include**, то изменения включаемого сегмента отразятся только после изменения исходной страницы (контейнер JSP перекомпилирует исходную страницу). Для включения содержимого в JSP-документ также используется стандартное действие **jsp:include**. При этом не обязательно, чтобы включаемый JSP-фрагмент был правильным XML-документом. Главное, чтобы он возвращал текст в виде правильного XML и не нарушал структуру исходного JSP-документа.

```
<!--пример #15 : включение в код статического содержимого :  incl_title.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
```

```
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<html><head>
<jsp:directive.include file="\WEB-INF\jspf\title.jspf" />
</head>
<body>
<h1>JSP-страница, использующая директиву include</h1>
<h3>Директива include используется для включения статиче-
ского содержимого, например заголовка страницы.</h3>
</body></html>
</jsp:root>
```

<!-- пример # 16 : код включаемого фрагмента : title.jspf -->

```
<title>Title from title.jspf</title>
```

Ниже приведен пример включения динамического содержимого. Включаемый фрагмент получает данные из объектов **request** и **session**. Для передачи значения параметра можно использовать строку запроса. Запрос может выглядеть следующим образом:

http://localhost:8082/home/thanks.jspx?lname=username.

Установка кодировки в фрагменте необходима для того, чтобы устранить неполадки при включении русского текста.

<!-- пример # 17 : использование действия include для динамического включения : thanks.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" ver-
sion="2.0">
<jsp:directive.page contentType="text/html;charset=Utf-8"/>
<html><head><title>Действие include</title></head>
<body>
      Данные, сформированные динамически при включении jsp-
фрагмента<br/>
      Включаемый фрагмент получает данные из объекта
session <br/>
      <jsp:include page="\WEB-INF\jspf\params.jsp"/>.
</body></html>
</jsp:root>
```

<!-- пример #18 : включаемый фрагмент : params.jsp -->

```
<jsp:directive.page contentType="text/html;charset=Utf-8"/>
ID сессии -
   <jsp:expression>session.getId()</jsp:expression>
```

В результате работы документа в браузер будет выведено:

Данные, сформированные динамически при включении jsp-фрагмента.
Включаемый фрагмент получает данные из объектов request, session
ID сессии - 08C51EEC60A97E90C734101F54EA310E .

Также для включения содержимого можно использовать тег **<c:import>**. Его использование уже было приведено выше.

Обработка ошибок

При выполнении web-приложений, как и любых других, могут возникать ошибки и исключительные ситуации. Три основных типа исключительных ситуаций:

- код «`404 Not Found`». Возникает при неправильном наборе адреса или обращении к станице, которой не существует;
- код «`500 Internal Server Error`». Возникает при вызове сервлетом метода **`sendError(500)`** для объекта **`HttpServletResponse`**;
- исключения времени исполнения. Исключения, генерируемые web-приложением и не перехватываемые фильтром, сервлетом или JSP.

Для обработки исключений в зависимости от типа в приложении может существовать несколько JSP-страниц, сервлетов или обычных HTML-страниц. Для настройки соответствия ошибок и обработчиков используется элемент **`error-page`** файла **`web.xml`**. Например:

```
<error-page>
   <error-code>404</error-code>
   <location>/error404</location>
</error-page>
```

или

```
<error-page>
   <exception-type>java.io.IOException</exception-type>
   <location>/errorIo</location>
</error-page>
```

В элементе **`error-code`** указывается код ошибки, в элементе **`exception-type`** – тип исключения.

Для указания страницы, обрабатывающей ошибки, возникающие при выполнении текущей страницы, можно использовать директиву

`<jsp:directive.page errorPage="path" />`, где **`path`** – эту путь к странице-обработчику. Ниже приведен пример, использующий именно такой способ. При нажатии на кнопку генерируется ошибка **`java.lang.NullPointerException`**, и управление передается странице **`error_hand.jsp`**

```
<!--пример # 19 : генерация ошибки : gen_error.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            xmlns:c="http://java.sun.com/jsp/jstl/core"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8"/>
<jsp:directive.page errorPage="/error_hand.jspx" />
<html><head><title>Генерация исключения</title></head>
<body>
<h2>При нажатии кнопки будет сгенерирована ошибка!</h2>
<form>
<input type="submit" name="gen"
       value="Сгенерировать ошибку"/>
</form>
```

```
<c:if test="${not empty param.gen}">
 <jsp:declaration>String str;</jsp:declaration>
 <jsp:scriptlet>str.length();</jsp:scriptlet>
</c:if>
</body></html>
</jsp:root>
```

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 # 20 : ERROR PAGE : error_hand.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<jsp:directive.page isErrorPage="true" />
<html><head><title>Сообщение об ошибке</title></head>
<body>
<p>Сгенерирована ошибка! <br/>
<jsp:expression>exception.toString()</jsp:expression>
</p></body></html>
</jsp:root>
```

Извлечение значений полей

Библиотеки JSLT и EL позволяют легко обрабатывать данные, полученные из форм, так как JSP-страница имеет доступ к неявному объекту **param**, который состоит из объектов типа **java.util.Map.Entry**, что позволяет обращаться к данным как к парам «ключ-значение».

В следующем примере в документе **params.jspx** производится извлечение значений параметров, передаваемых из страницы **form.jspx**.

<!--пример # 21 : страница, которая выводит форму и передает данные странице param.jspx: form.jspx -->

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<html><head><title>Форма для заполнения</title></head>
<body>
<form action="params.jspx">
Введите, пожалуйста, ваши данные: <br/>
Фамилия: <input type="text" name="fname" value="" /><br/>
Имя:     <input type="text" name="lname" value="" /><br/>
E-mail:  <input type="text" name="e-mail" value="" /><br/>
<input type="submit" value="Отправить" /><br/>
</form>
</body></html>
</jsp:root>
```

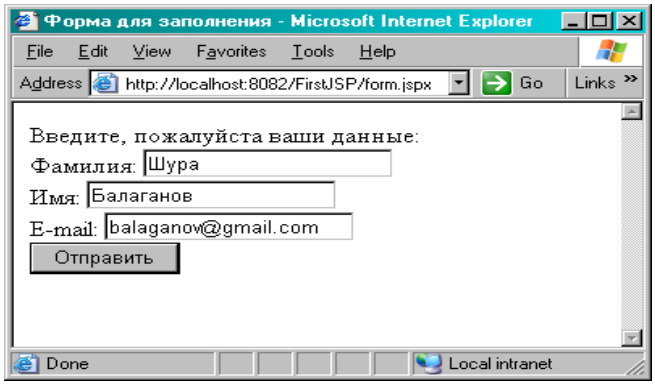


Рис. 19.1. Документ для задания и передачи параметров

```
<!--пример # 22 : считывание информации и генерация ответа : params.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          version="2.0">
<jsp:directive.page contentType=
          "text/html; charset=Utf-8" />
<html><head><title>Обработка данных</title></head>
<body>
Вывод данных с помощью JSTL и EL<br/>
<c:forEach var="items" items="${param}">
<b><c:out value="${items.key}"></c:out></b>:
<c:out value="${items.value}"></c:out><br/>
</c:forEach>
<c:if test="${not empty param.fname}">
<b>Имя:</b><c:out value="${param.fname}"/><br/>
</c:if>
<c:if test="${not empty param.lname}">
<b>Фамилия:</b><c:out value="${param.lname}"/>
</c:if>
</body></html>
</jsp:root>
```

В результате работы документа в браузер будет выведено:

Вывод данных с помощью JSTL и EL
lname: Балаганов
fname: Шура
e-mail: balaganov@gmail.com
Имя: Шура
Фамилия: Балаганов

В вышеприведенном примере с помощью тега **`c:forEach`** перебираются все данные, полученные из формы. Так же можно выводить отдельные параметры, обращаясь к ним с помощью EL. Конструкция **`${param.lname}`** возвращает значение параметра **`lname`**.

С помощью тега **`jsp:forward`** можно добавлять данные к запросу.

```
<!--пример # 23: добавление параметра add_param и перенаправление запроса
к странице form.jspx: forward.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
            version="2.0">
<jsp:directive.page contentType=
            "text/html; charset=Utf-8" />
<html><head><title>Добавление параметра</title></head>
<body>
<jsp:forward page="params.jspx">
<jsp:param name="addparam" value="added"/>
</jsp:forward>
</form>
</body></html>
</jsp:root>
```

Если обратиться к этой странице, передавая в строке запроса параметры (например **http://localhost:8082/FirstJSP/forward.jspx?name=UserName**), то, кроме этих параметров, странице **param.jspx** будет передан параметр **addparam** со значением **added**.

Технология взаимодействия JSP и сервлета

В большинстве приложений 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.

Реализацию достаточно простой, но эффективной технологии построения распределенного приложения можно рассмотреть на примере решения задачи проверки логина и пароля пользователя с выводом приветствия в случае положительного результата. Схематично организацию данного приложения можно представить в виде следующей диаграммы:

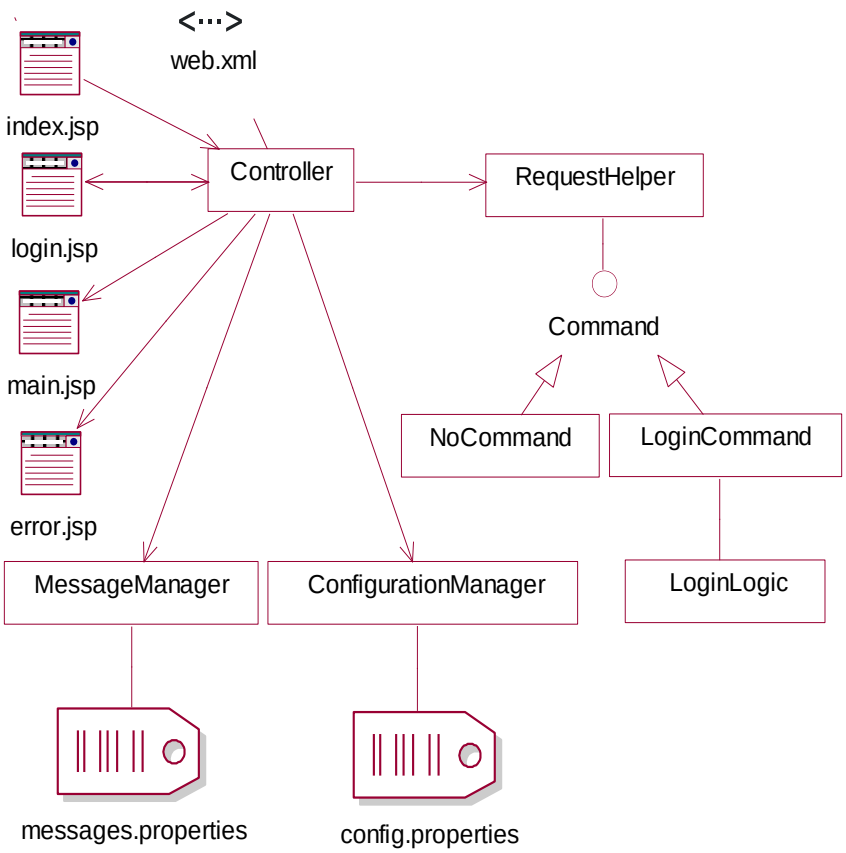


Рис. 19.2. Диаграмма взаимодействия классов и страниц JSP приложения.

```
<!--пример # 24 : прямой вызов контроллера : index.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" ver-
sion="2.0">
<jsp:directive.page contentType="text/html; charset=Utf-8"
/>
<html><head><title>Index JSP</title></head>
<body>
<a href="controller">Main Controller</a>
</body></html>
</jsp:root>
```

Следующая страница **`login.jsp`** содержит форму для ввода логина и пароля для аутентификации в системе:

```
<!--пример # 25 : форма ввода информации и вызов контроллера : login.jsp -->
<%@ page language="java" contentType="text/html;
  charset=ISO-8859-1"  pageEncoding="ISO-8859-1"%>
    <html><head><title>Login</title></head>
    <body><h3>Login</h3>
    <hr/>
```

```
<form name="loginForm" method="POST"
action="controller">
<input type="hidden" name="command" 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 **Controller**:

```
/* пример # 26 : контроллер запросов : Controller.java */
package by.bsu.famcs.jspservlet;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.famcs.jspservlet.commands.Command;
import by.bsu.famcs.jspservlet.manager.MessageManager;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;

public class Controller extends HttpServlet
        implements  javax.servlet.Servlet {
//объект, содержащий список возможных команд
    RequestHelper requestHelper =
        RequestHelper.getInstance();
    public Controller() {
        super();
    }
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException{
        processRequest(request, response);
    }
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException{
        processRequest(request, response);
    }
    private void processRequest(HttpServletRequest
        request, HttpServletResponse response)
    throws ServletException, IOException {

        String page = null;
```

```
            try {
//определение команды, пришедшей из JSP
                  Command command =
                        requestHelper.getCommand(request);
/*вызов реализованного метода execute() интерфейса Command и передача
параметров классу-обработчику конкретной команды*/
            page = command.execute(request, response);
// метод возвращает страницу ответа
            } catch (ServletException e) {
                  e.printStackTrace();
//генерация сообщения об ошибке
request.setAttribute("errorMessage",
      MessageManager.getInstance().getProperty(
      MessageManager.SERVLET_EXCEPTION_ERROR_MESSAGE));
//вызов JSP-страницы с сообщением об ошибке
page = ConfigurationManager.getInstance()
      .getProperty(ConfigurationManager.ERROR_PAGE_PATH);
            } catch (IOException e) {
                  e.printStackTrace();
                  request.setAttribute("errorMessage",
 MessageManager.getInstance()
.getProperty(MessageManager.IO_EXCEPTION_ERROR_MESSAGE));

            page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.ERROR_PAGE_PATH);
            }
//вызов страницы ответа на запрос
            RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher(page);
            dispatcher.forward(request, response);
      }
}
/* пример # 27 : класс контейнер команд : RequestHelper.java */
package by.bsu.famcs.jspservlet;
import java.util.HashMap;
import javax.servlet.http.HttpServletRequest;
import by.bsu.famcs.jspservlet.commands.Command;
import by.bsu.famcs.jspservlet.commands.LoginCommand;
import by.bsu.famcs.jspservlet.commands.NoCommand;

public class RequestHelper {
      private static RequestHelper instance = null;

      HashMap<String, Command> commands =
                  new HashMap<String, Command>();

      private RequestHelper() {
//заполнение таблицы командами
            commands.put("login", new LoginCommand());
      }
```

```
   public Command getCommand(HttpServletRequest request) {
//извлечение команды из запроса
      String action = request.getParameter("command");
//получение объекта, соответствующего команде
      Command command = commands.get(action);
            if (command == null) {
//если команды не существует в текущем объекте
                  command = new NoCommand();
            }
            return command;
      }
//создание единственного объекта по шаблону Singleton
   public static RequestHelper getInstance() {
            if (instance == null) {
                  instance = new RequestHelper();
            }
            return instance;
      }
}
```

/ пример # 28 : интерфейс, определяющий контракт и его реализации : Command.java: LoginCommand.java : NoCommand.java */*

```
package by.bsu.famcs.jspservlet.commands;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;

public interface Command {
      public String execute(HttpServletRequest request,
                              HttpServletResponse response)
                  throws ServletException, IOException;
}
```

```
package by.bsu.famcs.jspservlet.commands;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.famcs.jspservlet.logic.LoginLogic;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;
import by.bsu.famcs.jspservlet.manager.MessageManager;

public class LoginCommand implements Command {

private static final String PARAM_NAME_LOGIN = "login";
private static final String PARAM_NAME_PASSWORD
                                          = "password";
```

```
      public String execute(HttpServletRequest request,
                                 HttpServletResponse response)
                     throws ServletException, IOException {

            String page = null;
//извлечение из запроса логина и пароля
String login = request.getParameter(PARAM_NAME_LOGIN);
String pass = request.getParameter(PARAM_NAME_PASSWORD);
//проверка логина и пароля
            if (LoginLogic.checkLogin(login, pass)) {
                  request.setAttribute("user", login);
//определение пути к main.jsp
                  page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.MAIN_PAGE_PATH);
            } else {
                  request.setAttribute("errorMessage",
 MessageManager.getInstance()
.getProperty(MessageManager.LOGIN_ERROR_MESSAGE));
                  page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.ERROR_PAGE_PATH);
            }
            return page;
      }
}
package by.bsu.famcs.jspservlet.commands;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;
public class NoCommand implements Command {

      public String execute(HttpServletRequest request,
             HttpServletResponse response)
             throws ServletException, IOException {
/*в случае прямого обращения к контроллеру переадресация на страницу ввода
логина*/
      String page = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.LOGIN_PAGE_PATH);
            return page;
      }
}
/* пример # 29 : служебные классы, извлекающие из properties-файлов
необходимую для функционирования приложения информацию :
ConfigurationManager.java: MessageManager.java */
package by.bsu.famcs.jspservlet.manager;
```

```
import java.util.ResourceBundle;

public class ConfigurationManager {
      private static ConfigurationManager instance;
      private ResourceBundle resourceBundle;

//класс извлекает информацию из файла config.properties
      private static final String BUNDLE_NAME = "config";

      public static final String DATABASE_DRIVER_NAME =
                                    "DATABASE_DRIVER_NAME";
      public static final String DATABASE_URL =
                                        "DATABASE_URL";
      public static final String ERROR_PAGE_PATH =
                                    "ERROR_PAGE_PATH";
      public static final String LOGIN_PAGE_PATH =
                                    "LOGIN_PAGE_PATH";
      public static final String MAIN_PAGE_PATH =
                                    "MAIN_PAGE_PATH";

      public static ConfigurationManager getInstance() {
            if (instance == null) {
                  instance = new ConfigurationManager();
                  instance.resourceBundle =
                  ResourceBundle.getBundle(BUNDLE_NAME);
            }
            return instance;
      }
      public String getProperty(String key) {
            return (String)resourceBundle.getObject(key);
      }
}
package by.bsu.famcs.jspservlet.manager;
import java.util.ResourceBundle;
public class MessageManager {
      private static MessageManager instance;
      private ResourceBundle resourceBundle;
//класс извлекает информацию из файла messages. properties
      private static final String BUNDLE_NAME = "messages";
      public static final String LOGIN_ERROR_MESSAGE =
"LOGIN_ERROR_MESSAGE";
      public static final String
SERVLET_EXCEPTION_ERROR_MESSAGE =
            "SERVLET_EXCEPTION_ERROR_MESSAGE";
      public static final String IO_EXCEPTION_ERROR_MESSAGE
= "IO_EXCEPTION_ERROR_MESSAGE";

      public static MessageManager getInstance() {
```

```
            if (instance == null) {
                  instance = new MessageManager();
                  instance.resourceBundle =
                  ResourceBundle.getBundle(BUNDLE_NAME);
            }
            return instance;
      }

      public String getProperty(String key) {
            return (String)resourceBundle.getObject(key);
      }
}
```

Далее приведено содержимое файла *config.properties:*

```
##############################
## Application configuration ##
##############################
DATABASE_DRIVER_NAME=com.mysql.jdbc.Driver
DATABASE_URL=jdbc:mysql://localhost:3306/db1?user=
root&password=root
ERROR_PAGE_PATH=/jsp/error.jspx
LOGIN_PAGE_PATH=/jsp/login.jspx
MAIN_PAGE_PATH=/jsp/main.jspx
```

Далее приведено содержимое файла *messages.properties:*

```
##############################
##         Messages      ##
##############################
LOGIN_ERROR_MESSAGE=Incorrect login or password
SERVLET_EXCEPTION_ERROR_MESSAGE=ServletException: Servlet
encounters difficulty
IO_EXCEPTION_ERROR_MESSAGE=IOException: input or output er-
ror while handling the request
```

Ниже приведен код класса бизнес-логики **LoginLogic**, выполняющий проверку правильности введенных логина и пароля с помощью запроса в БД:

/ пример # 30 : бизнес-класс проверки данных пользователя : LoginLogic.java */*

```
package by.bsu.famcs.jspservlet.logic;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.DriverManager;
import
by.bsu.famcs.jspservlet.manager.ConfigurationManager;

public class LoginLogic {
      public static boolean checkLogin(
            String login, String password) {
```

```
            // проверка логина и пароля
            try {
//организация простейшего соединения с базой данных
String driver = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.DATABASE_DRIVER_NAME);

                  Class.forName(driver);
                  Connection cn = null;
                  try {
String url = ConfigurationManager.getInstance()
.getProperty(ConfigurationManager.DATABASE_URL);
            cn = DriverManager.getConnection(url);
                        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, существует ли пользователь
       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 (SQLException e) {
                  e.printStackTrace();
                  return false;
            } catch (ClassNotFoundException e) {
                  e.printStackTrace();
                  return false;
            }
      }
}
```

Страница **main.jsp** показывается пользователю в случае успешной аутентификации в приложении:

```
<!--пример # 31 : сообщение о входе : main.jsp -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c=http://java.sun.com/jsp/jstl/core
      version="2.0">
<jsp:directive.page contentType="text/html;
      charset=Utf-8" />
      <html><head><title>Welcome</title></head>
      <body><h3>Welcome</h3>
      <hr />
      <c:out value="${user}, Hello!"/>
      <hr />
      <a href="controller">Return to login page</a>
      </body></html>
</jsp:root>
```

Страница **error.jsp** загружается пользователю в случае возникновения ошибок (например, если неправильно введены логин и пароль):

```
<!-- пример # 32 : страница ошибок, предлагающая повторить процедуру ввода
информации : error.jspx -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:c=
      "http://java.sun.com/jsp/jstl/core" version="2.0">
      <jsp:directive.page contentType=
                  "text/html; charset=Utf-8" />
      <html><head><title>Error</title></head>
      <body>
      <h3>Error</h3>
      <hr />
      <jsp:expression>
(request.getAttribute("errorMessage") != null)
? (String) request.getAttribute("errorMessage")
: "unknown error"</jsp:expression>
      <hr />
      <a href="controller">Return to login page</a>
      </body></html>
</jsp:root>
```

И последнее, что надо сделать в приложении, – это настроить файл **web.xml**, чтобы можно было обращаться к сервлету-контроллеру по имени **controller**, т.е. необходимо настроить mapping.

```
<!--пример # 33 : имя сервлета и путь к нему : web.xml -->
   <?xml version="1.0" encoding="UTF-8"?>
   <web-app id="WebApp_ID" version="2.4" >
      <display-name>Project</display-name>
      <servlet>
            <description>
            </description>
            <display-name>
```

```
        Controller</display-name>
        <servlet-name>Controller</servlet-name>
        <servlet-class>
    by.bsu.famcs.jspservlet.Controller</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Controller</servlet-name>
        <url-pattern>/controller</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

В данном случае в поле **<servlet-name>** было занесено имя **controller,** а в поле **<url-pattern>** – соответственно **/controller**.

Запуск примера производится из командной строки Web-браузера при запущенном контейнере сервлетов Tomcat 5.5.*, например в виде:

http://localhost:8082/Project/index.jspx

В этом случае при вызове сервлета в браузере будет отображен путь и имя в виде:

http://localhost:8082/Project/controller

Задания к главе 19

3) Вариант A

Реализовать приложение, используя технологию взаимодействия JSP и сервлетов. Вся информация должна храниться в базе данных.

d) **Банк.** 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.

e) **Регистрация пользователя**. 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.

f) **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.

g) **Склад**. 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.

h) **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.

i) **Каталог библиотеки**. Выдается список книг, наличествующих в библиотеке. Запрос на заказ отправляется пометкой требуемой книги. Система проверяет в БД, свободна книга или нет. В случае занятости выводится информация о сроках возвращения книги в фонд.

j) **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.

4) Вариант В

Для заданий варианта В главы 4 создать информационную систему, использующую страницы JSP на стороне клиента, сервлет в качестве контроллера и БД для хранения информации.

Тестовые задания к главе 19

5) 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; %>.`

6) 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.`

7) 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.`

8) *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.`

9) *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

JDBC

Драйверы, соединения и запросы

JDBC (Java DataBase Connectivity) – 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. Драйвер, использующий другой прикладной интерфейс взаимодействия с СУБД, в частности ODBC (так называемый JDBC-ODBC – мост). Стандартный драйвер первого типа **`sun.jdbc.odbc.JdbcOdbcDriver`** входит в JDK.
2. Драйвер, работающий через внешние (native) библиотеки (т.е. клиента СУБД).
3. Драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером, который, в свою очередь, подключается к нужной СУБД.
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.

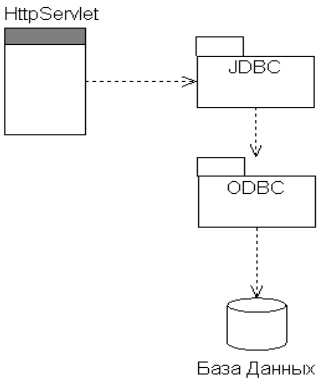


Рис. 20.1. Доступ к БД с помощью JDBC-ODBC-моста

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.

Последовательность действий

1. *Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса.*

Например:

```
String driverName = "org.gjt.mm.mysql.Driver";
```

для СУБД MySQL,

```
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
```

для СУБД MSAccess или

```
String driverName = "org.postgreesql.Driver";
```

для СУБД PostgreeSQL.

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. Например, для СУБД DB2 от IBM объект-драйвер можно создать следующим образом:

```
new com.ibm.db2.jdbc.net.DB2Driver();
```

2. *Установка соединения с БД.*

Для установки соединения с БД вызывается статический метод **`getConnection()`** класса **`DriverManager`**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект **`Connection`**. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Например:

```
Connection cn = DriverManager.getConnection(
"jdbc:mysql://localhost/my_db", "root", "pass");
```

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. С помощью метода **`registerDriver()`** драйверы регистрируются, а методом **`getDrivers()`** можно получить список всех драйверов.

3. *Создание объекта для передачи запросов.*

После создания объекта **`Connection`** и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект **`Statement`**, создаваемый вызовом метода **`createStatement()`** класса **`Connection`**.

```
Statement st = cn.createStatement();
```

Объект класса **Statement** используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов **PreparedStatement** и **CallableStatement** для выполнения подготовленных запросов и хранимых процедур. Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов **executeQuery(String sql)** или **executeUpdate(String sql)**.

4. *Выполнение запроса*.

Результаты выполнения запроса помещаются в объект **ResultSet**:

```
ResultSet rs = st.executeQuery(
"SELECT * FROM my_table");//выборка всех данных таблицы my_table
```

Для добавления, удаления 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(int pos)** а также аналогичные методы, начинающиеся с **get**Тип**(int pos)** (**getInt(int pos)**, **getFloat(int pos)** и др.) и **update**Тип**()**. Среди них следует выделить методы **getClob(int pos)** и **getBlob(int pos)**, 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 СУБД может быть загружена с сайта **www.mysql.com**. Для корректной установки необходимо следовать инструкциям мастера установки. Каталог лучше выбирать по умолчанию. В процессе установки следует создать администратора СУБД с именем `root` и паролем `pass`. Если планируется разворачивать реально работающее приложение, необходимо исключить тривиальных пользователей сервера БД (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-3.1.12.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 chapt20;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletToBase extends HttpServlet {
      public void doGet(HttpServletRequest req,
                        HttpServletResponse resp)
                        throws ServletException {
            performTask(req, resp);
      }
      public void doPost(HttpServletRequest req,
                         HttpServletResponse resp)
                         throws ServletException {
            performTask(req, resp);
      }

      public void showInfo(PrintWriter out, ResultSet rs)
                                    throws SQLException {
            out.print("From DataBase:");
        while (rs.next()) {
            out.print("<br>Name:-> " + rs.getString(1)
                        + " Phone:-> " + rs.getInt(2));
            }
      }
      public void performTask(HttpServletRequest req,
                              HttpServletResponse resp) {
      resp.setContentType("text/html; charset=Cp1251");
      PrintWriter out = null;
            try {//1
                  out = resp.getWriter();
                  try {//2

      Class.forName("org.gjt.mm.mysql.Driver");
            // для MSAccess
```

```
            /* return "sun.jdbc.odbc.JdbcOdbcDriver" */
            // для PostgreeSQL
            /* return " org.postgreesql.Driver " */
      Connection cn = null;
                        try {//3
      cn =
DriverManager.getConnection("jdbc:mysql://localhost/db2",
                                                  "root", "pass");
            // для MSAccess
            /* return "jdbc:odbc:db2"; */
            // для PostgreeSQL
            /* return "jdbc:postgreesql://localhost/db2"; */

            Statement st = null;
                              try {//4
            st = cn.createStatement();
                        ResultSet rs = null;
                                    try {//5
            rs = st.executeQuery("SELECT * FROM users");
      out.print("From DataBase:");
        while (rs.next()) {
            out.print("<br>Name:-> " + rs.getString(1)
                        + " Phone:-> " + rs.getInt(2));
                              }
                              } finally {// для 5-го блока try
      /*    закрыть ResultSet, если он был открыт и ошибка
            произошла во время чтения из него данных  */
                  // проверка успел ли создаться ResultSet
                  if (rs != null)   rs.close();
                  else
      out.print("ошибка во время чтения данных из БД");
                                    }
                              } finally {// для 4-го блока try
      /*    закрыть Statement, если он был открыт и ошибка
            произошла во время создания ResultSet    */
                  // проверка успел ли создаться Statement
                  if (st != null)   st.close();
                  else  out.print("Statement не создан");
                              }
                        } finally {// для 3-го блока try
      /*    закрыть Connection, если он был открыт и ошибка произошла
            во время создания ResultSet или создания и использования
            Statement  */
            // проверка - успел ли создаться Connection
            if (cn != null) cn.close();
            else  out.print("Connection не создан");
                        }
```

```
        } catch (ClassNotFoundException e) {// для 2-го блока try
        out.print("ошибка во время загрузки драйвера БД");
                    }
              }
        /* вывод сообщения о всех SQLException и IOException в блоках finally, */
              /* поэтому следующие блоки catch оставлены пустыми */
              catch (SQLException e) {
              }// для 1-го блока try
              catch (IOException e) {
              }// для 1-го блока try
              finally {// для 1-го блока try
        /*    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.print("PrintWriter не проинициализирован");
              }
        }
}
```

В несложном приложении достаточно контролировать закрытие соединения, так как незакрытое (“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
driver=org.gjt.mm.mysql.Driver
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 SQLException {

      ResourceBundle resource =
            ResourceBundle.getBundle("database");
            String url = resource.getString("url");
            String driver = resource.getString("driver");
            String user = resource.getString("user");
            String pass = resource.getString("password");
            try {
                  Class.forName(driver).newInstance();
            } catch (ClassNotFoundException e) {
      throw new SQLException("Драйвер не загружен!");
            } catch (InstantiationException e) {
```

```
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
return DriverManager.getConnection(url, user, pass);
}
```

Объект класса **ResourceBundle**, содержащий ссылки на все внешние ресурсы проекта, создается с помощью вызова статического метода **getBundle(String filename)**, 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(String name)**, 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(String key)** и **getStringArray(String key)** 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, созданных средствами самой СУБД.

Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод **prepareStatement(String sql)** интерфейса **Connection**, возвращающий объект **PreparedStatement**. Установка входных значений конкретных параметров этого объекта 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 chapt20;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
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();
      }
}
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();
      }
}
```

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 именованная последовательность команд SQL, рассматриваемых как единое целое, и выполняющаяся 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**.

Интерфейс **CallableStatement** позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса в том, что **CallableStatement** способен обрабатывать не только входные (**IN**) параметры, но и выходящие (**OUT**) и смешанные (**INOUT**) параметры. Тип выходного параметра должен быть зарегистрирован методом **regis-**

terOutParameter(). После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Пусть в БД существует хранимая процедура **getempname**, которая по уникальному для каждой записи в таблице **employee** числу SSN будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getempname
(emp_ssn IN INT, emp_name OUT VARCHAR) AS
BEGIN
SELECT name
INTO emp_name
FROM employee
WHERE SSN = EMP_SSN;
END
```

Тогда для получения имени служащего **employee** через вызов данной процедуры необходимо исполнить java-код вида:

```
String SQL = "{call getempname (?,?)}";
CallableStatement cs = conn.prepareCall(SQL);
cs.setInt(1,822301);
//регистрация выходящего параметра
cs.registerOutParameter(2,java.sql.Types.VARCHAR);
cs.execute();
String empName = cs.getString(2);
System.out.println("Employee with SSN:" + ssn
+ " is " + empName);
```

В результате будет выведено:

```
Employee with SSN:822301 is Spiridonov
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
// turn off autocommit
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO employee VALUES
                                    (10, 'Joe ')");
stmt.addBatch("INSERT INTO location VALUES
                                    (260, 'Minsk')");
stmt.addBatch("INSERT INTO emp_dept VALUES
                                    (1000, 260)");
// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

Если используется объект **PreparedStatement**, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод **PreparedStatement.executeBatch()** возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует массив объектов типа **Employee** со стандартным набором методов **getТип()/setТип()** для каждого из его полей, и необходимо внести их значения в БД. Многократное выполнение методов **execute()** или **exe-**

cuteUpdate() становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
   Employee[] employees = new Employee[10];
   PreparedStatement statement =
   con.prepareStatement("INSERT INTO employee VALUES
                                        (?,?,?,?,?)");
   for (int i = 0; i < employees.length; i++) {
         Employee currEmployee = employees[i];
         statement.setInt(1, currEmployee.getSSN());
         statement.setString(2, currEmployee.getName());
   statement.setDouble(3, currEmployee.getSalary());
   statement.setString(4,currEmployee.getHireDate());
         statement.setInt(5, currEmployee.getLoc_Id());
         statement.addBatch();
   }
   updateCounts = statement.executeBatch();
   } catch (BatchUpdateException e) {
         e.printStackTrace();
   }
```

Транзакции

При проектировании распределенных систем часто возникают ситуации, когда сбой в системе или какой-либо ее периферийной части может привести к потере информации или 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. Такие две операции трактуют как одну и называют транзакцией.

Транзакцию (деловую операцию) 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 – при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении 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 API JDBC эта операция выполняется по умолчанию после каждого вызова методов **executeQuery()** и **executeUpdate()**. Если же необходимо сгруппировать запросы и только после этого выполнить операцию COMMIT, сначала вызывается метод **setAutoCommit(boolean param)** 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 : вызов сервлета : index.jsp -->
<%@ page language="java" contentType="text/html; char-
set=windows-1251" pageEncoding="windows-1251"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional
//EN">
<html><head>
<meta http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
<title>Simple Transaction Demo</title>
</head>
<body>
<form name="students" method="POST"
      action="SQLTransactionServlet">

      id:<br/>
      <input type="text" name="id" value=""><br/>
      Name:<br/>
      <input type="text" name="name" value=""><br/>

      Course:<br/>
      <select name="course">
        <option>Java SE 6
        <option>XML
        <option>Struts
      </select><br/>

      <input type="submit" value="Submit">
</form>
</body></html>
```

```
/* пример # 4 : выполнение транзакции : метод perform() сервлета
SQLTransactionServlet.java */
public void taskPerform(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
 response.setContentType("text/html; charset=Cp1251");
 PrintWriter out = null;
 Connection cn = null;
 try {
      out = response.getWriter();
      String id = request.getParameter("id");
      String name = request.getParameter("name");
      String course = request.getParameter("course");
      out.print("ID студента: " + id + ", " + name +<br>");
            cn = getConnection();
            cn.setAutoCommit(false);
            Statement st = cn.createStatement();
      try {
            String upd;
            upd =
      "INSERT INTO student (id, name) VALUES ('"
                              + id  + "', '" + name + "')";
            st.executeUpdate(upd);
            out.print("Внесены данные в students: "
                              + id + ", " + name + "<br>");

            upd =
 "INSERT INTO course(id_student, name_course) VALUES('"
                              + id + "','" + course + "')";
            st.executeUpdate(upd);
            out.print("Внесены данные в course: " + id
                              + ", " + course + "<br>");

            cn.commit();// подтверждение
 out.print("<b>Данные внесены - транзакция завершена"
                                    + "</b><br>");
      } catch (SQLException e) {
            cn.rollback();// откат
            out.println("<b>Произведен откат транзакции:"
                        + e.getMessage()  + "</b>");
      } finally {
            if (cn != null)
            cn.close();
      }
 } catch (SQLException e) {
      out.println("<b>ошибка при закрытии соединения:"
                                    + e.getMessage());
 }
}
```

Если таблицы `student` и `course` базы данных **`db1`** до изменения выглядели, например, следующим образом,

id	name
71	Goncharenko

id_student	name_course
71	Java SE 6

Рис. 20.2. Таблицы до выполнения запроса

то после внесения изменений и их подтверждения они примут вид:

id	name
71	Goncharenko
83	Petrov

id_student	name_course
71	Java SE 6
83	XML

Рис. 20.3. Таблицы после подтверждения выполнения запросов

ID студента: 83, Petrov
Внесены данные в students: 83, Petrov
Внесены данные в course: 83, XML
Данные внесены - транзакция завершена

Приведенный пример в полной мере не отражает принципы транзакции, но демонстрирует способы ее поддержки методами языка Java.

Для транзакций существует несколько типов чтения:

- Грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;
- Непроверяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;
- Фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие **`WHERE`**-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие **`WHERE`**-условию, уже вместе с новой строкой или недосчитавшись старой.

JDBC удовлетворяет четырем уровням изоляции транзакций, определенным в стандарте SQL:2003.

Уровни изоляции транзакций определены в виде констант интерфейса **`Connection`** (по возрастанию уровня ограничения):

- **`TRANSACTION_NONE`** – информирует о том, что драйвер не поддерживает транзакции;
- **`TRANSACTION_READ_UNCOMMITTED`** – позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, непроверяющееся и фантомное чтения;
- **`TRANSACTION_READ_COMMITTED`** – означает, что любое изменение, сделанное в транзакции, не видно вне неё, пока она не сохранена. Это предотвращает грязное чтение, но разрешает непроверяющееся и фантомное;

- **TRANSACTION_REPEATABLE_READ** – запрещает грязное и непроверяющееся, но фантомное чтение разрешено;
- **TRANSACTION_SERIALIZABLE** – определяет, что грязное, непроверяющееся и фантомное чтения запрещены.

Метод **boolean supportsTransactionIsolationLevel(int level)** интерфейса **DatabaseMetaData** определяет, поддерживается ли заданный уровень изоляции транзакций.

В свою очередь, методы интерфейса **Connection** определяют доступ к уровню изоляции:

int getTransactionIsolation() – возвращает текущий уровень изоляции;

void setTransactionIsolation(int level) – устанавливает нужный уровень.

Точки сохранения

Точки сохранения дают дополнительный контроль над транзакциями. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных. Таким образом, если произойдет ошибка, можно вызвать метод **rollback()** для отмены всех изменений, которые были сделаны после точки сохранения.

Метод **boolean supportsSavepoints()** интерфейса **DatabaseMetaData** используется для того, чтобы определить, поддерживает ли точки сохранения драйвер JDBC и сама СУБД.

Методы **setSavepoint(String name)** и **setSavepoint()** (оба возвращают объект **Savepoint**) интерфейса **Connection** используются для установки именованной или неименованной точки сохранения во время текущей транзакции. При этом новая транзакция будет начата, если в момент вызова **setSavepoint()** не будет активной транзакции.

/ пример # 5 : применение точек сохранения : SavepointServlet.java */*

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.*;

public class SavepointServlet 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; charset=Cp1251");
                  PrintWriter out = resp.getWriter();
                  try {
                  Class.forName("org.gjt.mm.mysql.Driver");
                        Connection cn = null;
                        Savepoint savepoint = null;
                        try {
                        cn = DriverManager.getConnection(
      "jdbc:mysql://localhost/db3","root","pass");
                  cn.setAutoCommit(false);
                  out.print("<b>Соединение с БД...</b>");
                  out.print("<br>");
                  Statement stmt = cn.createStatement();
                  String trueSQL =
            "INSERT INTO emp (id,name,surname,salary) "
                        + "VALUES(2607,'Петя','Иванов',540)";
                  stmt.executeUpdate(trueSQL);
                              //установка точки сохранения
                  savepoint =
                        cn.setSavepoint("savepoint1");
                              //выполнение некорректного запроса
                  String wrongSQL =
                  "INSERT INTO (id,name,surname,salary) "
                  + "VALUES(2607,'Петя','Иванов',540)";
                  stmt.executeUpdate(wrongSQL);
                        } catch (SQLException ex) {
                              out.print(ex + "<br>");
                              cn.rollback(savepoint);
                  out.print("<b>Откат к точке сохранения: "
                                    + savepoint + "</b>");
                        } finally {
                              if (cn != null) cn.close();
                        }
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
                  out.close();
            }
}
```

В результате в браузер будет выведено:

```
Соединение с БД...
java.sql.SQLException: You have an error in your SQL syntax
...
Откат к точке сохранения: savepoint1
```

При этом результаты выполнения запроса **trueSQL** будут сохранены перед попыткой повторения транзакции.

Пул соединений

При большом количестве клиентов, работающих с приложением, к его базе данных выполняется большое количество запросов. Установление соединения 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).

Пул соединений можно организовать с помощью **server.xml** дескрипторного файла Apache Tomcat в виде:

```
<Context docBase="FirstProject" path="/FirstProject"
reloadable="true" source="com.ibm.etools.webtools.server:
FirstProject">
<!—создание пул соединений для СУБД MySQL -->
<Resource auth="Container" name="jdbc/db1"
                           type="javax.sql.DataSource"/>
<ResourceParams name="jdbc/db1">
      <parameter>
            <name>factory</name>
      <val-
ue>org.apache.commons.dbcp.BasicDataSourceFactory</value>
      </parameter>
      <parameter>
            <name>driverClassName</name>
            <value>org.gjt.mm.mysql.Driver</value>
      </parameter>
<!—url-адрес соединения JDBC для конкретной базы данных db1
Аргумент autoReconnect=true, заданный для url-адреса драйвера JDBC, должен
автоматически разорвать соединение, если mysqld его закроет. По умолчанию
mysqld закроет соединение через 8 часов.-->
      <parameter>
            <name>url</name>
<value>jdbc:mysql://localhost:3306/db1?autoReconnect=true
</value>
      </parameter>
      <parameter>
            <name>username</name>
            <value>root</value>
      </parameter>
      <parameter>
            <name>password</name>
            <value>pass</value>
      </parameter>
      <parameter>
            <name>maxActive</name>
```

```
            <value>500</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>10</value>
        </parameter>
        <parameter>
            <name>maxWait</name>
            <value>10000</value>
        </parameter>
        <parameter>
            <name>removeAbandoned</name>
            <value>true</value>
        </parameter>
        <parameter>
            <name>removeAbandonedTimeout</name>
            <value>60</value>
        </parameter>
        <parameter>
            <name>logAbandoned</name>
            <value>true</value>
        </parameter>
</ResourceParams>
</Context>
```

Найти и запустить данный пул соединений можно с помощью JNDI.

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/db1");
Connection cn = ds.getConnection("root", "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:50000/db1");
      Connection cn = ds.getConnection("db2inst1", "pass");
```

Драйвер определяется автоматически в объекте **DB2DataSource**.

Задания к главе 20

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;
   – имена 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 N фильмах.
- Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.
- Удалить все фильмы, дата выхода которых была более заданного числа 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 заданное количество занятий.
- Вывести дни недели, в которых занято заданное 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 адресатам.

5. **Сувениры.** В БД хранится информация о сувенирах и их производителях.

Для сувениров необходимо хранить:
– название;
– реквизиты 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 заданной.
- Вывести информацию о производителях заданного сувенира, произведенного в заданном 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 заданную, и количество различных товаров равно заданному.
- Вывести номера заказов, содержащих заданный товар.
- Вывести номера заказов, не содержащих заданный товар и поступивших 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 заданной отрицательной.
- 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 больше заданной.

9. **Магазин часов.** В БД хранится информация о часах, продающихся в магазина.
Для часов необходимо хранить:
– марку;
– тип (кварцевые, механические);
– цену;
– количество;
– реквизиты производителя.
Для производителей необходимо хранить:
– название;
– страну.

- Вывести марки заданного типа часов.
- Вывести информацию о механических часах, цена на которые не превышает заданную.
- Вывести марки часов, изготовленных в заданной стране.

- Вывести производителей, общая сумма часов которых в магазине не превышает заданную.

10. **Города.** В БД хранится информация о городах и их жителях.
    Для городов необходимо хранить:
    – название;
    – год основания;
    – 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 заданным 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 приближена к заданной.
    - Вывести треугольники, сумма площадей которых наиболее приближена к заданной.
    - Вывести треугольники, которые помещаются 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 20

Вопрос 20.1.

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 20.2.

Какой интерфейс из пакета **`java.sql`** должен реализовывать каждый драйвер JDBC?

1) `Driver;`
2) `DriverManager;`
3) `Connection;`
4) `DriverPropertyInfo;`
5) `ResultSet.`

Вопрос 20.3.

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 20.4.

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 20.5.

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 21
СЕССИИ, СОБЫТИЯ И ФИЛЬТРЫ

Сеанс (сессия)

При посещении клиентом Web-ресурса и выполнении вариантов запросов, контекстная информация о клиенте не хранится. В протоколе HTTP нет возможностей для сохранения и изменения информации о предыдущих посещениях клиента. 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. В иных случаях необходима информация о предыдущих запросах клиента. Существует несколько способов хранения текущей информации о клиенте или о нескольких соединениях клиента с сервером.

Сеанс (сессия) – соединение между клиентом и сервером, устанавливаемое на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и Web-сервером. Каждый клиент устанавливает с сервером свой собственный сеанс.

Сеансы используются для обеспечения хранения данных во время нескольких запросов Web-страницы или на обработку информации, введенной в пользовательскую форму в результате нескольких HTTP-соединений (например, клиент совершает несколько покупок в интернет-магазине; студент отвечает на несколько тестов в системе дистанционного обучения). Как правило, при работе с сессией возникают следующие проблемы:

- поддержка распределенной сессии (синхронизация/репликация данных, уникальность идентификаторов и т.д.);
- обеспечение безопасности;
- проблема инвалидации сессии (expiration), предупреждение пользователя об уничтожении сессии и возможность ее продления (watchdog).

Чтобы открыть новый сеанс, используется метод **`getSession()`** интерфейса **`HttpServletRequest`**. Метод извлекает из переданного в сервлет запроса объект сессии класса **`HttpSession`**, соответствующий данному пользователю. Сессия содержит информацию о дате и времени создания последнего обращения к сессии, которая может быть извлечена с помощью методов **`getCreationTime()`** и **`getLastAccessedTime()`**.

Если для метода **`getSession(boolean param)`** 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 se = request.getSession(true);
```

после чего начинается сбор информации о клиенте.

Чтобы сохранить значения переменной в текущем сеансе, используется метод **setAttribute()** класса **HttpSession**, прочесть – **getAttribute()**, удалить – **removeAttribute()**. Список имен всех переменных, сохраненных в текущем сеансе, можно получить, используя 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.

Если требуется сохранить для использования одну из переменных сеанса, представляющего собой целое число, то:

```
se.setAttribute("teacherId", new Integer(71));
```

После этого любой подключившийся к текущему сеансу сервлет сможет прочесть значение переменной **teacherId** следующим образом:

```
Integer testId = (Integer)se.getAttribute("teacherID");
```

Завершить сеанс можно методом **invalidate()**. Сеанс уничтожает все связи с объектами, и данные, сохраненные в старом сеансе, будут потеряны для всех приложений.

/ пример # 1 : добавление информации в сессию : SessionServlet.java */*

```
package chapt21;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SessionServlet extends HttpServlet {
      protected void doGet(
            HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException {
            performTask(req, resp);
      }
      private void performTask(
            HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException {
            SessionLogic.printToBrowser(resp, req);
      }
}
```

```
package chapt21;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```

```
public class SessionLogic {

      public static void printToBrowser(
      HttpServletResponse resp, HttpServletRequest req) {
            try {
/* возвращается ссылка на сессию для текущего пользователя (если сессия еще не
существует, то она при этом создается) */
            HttpSession session = req.getSession(true);

                  PrintWriter out = resp.getWriter();

                  StringBuffer url = req.getRequestURL();
                  session.setAttribute("URL", url);

                  out.write("My session counter: ");
/* количество запросов, которые были сделаны к данному сервлету текущим
пользователем в рамках текущей пользовательской сессии (следует приводить
значение к строковому виду для корректного отображения в результате) */
out.write(String.valueOf(prepareSessionCounter(session)));
                  out.write("<br> Creation Time : "
            + new Date(session.getCreationTime()));
                  out.write("<br> Time of last access : "
            + new Date(session.getLastAccessedTime()));
                  out.write("<br> session ID : "
            + session.getId());
                  out.write("<br> Your URL: " + url);
                  int timeLive = 60 * 30;
                  session.setMaxInactiveInterval(timeLive);
            out.write("<br>Set max inactive interval : "
            + timeLive + "sec");

                  out.flush();
                  out.close();
            } catch (IOException e) {
                  e.printStackTrace();
            throw new RuntimeException("Failed : " + e);
            }
      }
/* увеличивает счетчик обращений к текущему сервлету и кладет его в сессию */
      private static int prepareSessionCounter(
                        HttpSession session) {
            Integer counter =
            (Integer)session.getAttribute("counter");

            if (counter == null) {
               session.setAttribute("counter", 1);
                  return 1;
            } else {
                  counter++;
```

```
            session.setAttribute("counter", counter);
                  return counter;
            }
      }
}
```

В результате в браузер будет выведено:

My session counter: 3
Creation Time : Sun Jan 29 16:02:30 EET 2006
Time of last access : Sun Jan 29 16:02:38 EET 2006
session ID : 314A546CD9270A840E0BDA3286636B20
Your URL: http://localhost:8080/FirstProject/SessionServlet
Set max inactive interval : 1800sec

В качестве данных сеанса выступают: счетчик кликов – объект типа **Integer** и URL запроса, сохраненный в объекте **StringBuffer**. В ответ на пользовательский запрос сервлет **SessionServlet** возвращает страницу HTML, на которой отображаются все атрибуты сессии, время создания и последнего доступа, идентификационный номер сессии и время инвалидации (жизни) сессии. Это время можно задать с помощью тега **session-config** в **web.xml** в виде:

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

где время ожидания задается в минутах.

В следующем примере рассмотрен процесс ликвидации сессии при отсутствии активности за определенный промежуток времени.

```
/* пример # 2 : инвалидация и ликвидация сессии : TimeSessionServlet.java */
package chapt21;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class TimeSessionServlet extends HttpServlet {
      boolean flag = true;

      protected void doGet(HttpServletRequest req,
HttpServletResponse resp)
                  throws ServletException {
            performTask(req, resp);
      }
      private void performTask(HttpServletRequest req,
 HttpServletResponse resp) throws ServletException {

            HttpSession session = null;
            if (flag) {
            //создание сессии и установка времени инвалидации
                  session = req.getSession();
```

```
                int timeLive = 10; //десять секунд!
                session.setMaxInactiveInterval(timeLive);
                flag = false;
            } else {
            //если сессия не существует, то ссылка на нее не будет получена
                session = req.getSession(false);
                    }
            TimeSession.go(resp, req, session);
      }
}
package chapt21;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class TimeSession {

      public static void go(HttpServletResponse resp,
HttpServletRequest req, HttpSession session ) {
            PrintWriter out = null;
            try {
                out = resp.getWriter();
                out.write("<br> Creation Time : "
            + new Date(session.getCreationTime()));
                out.write("<br> Session alive! ");

                out.flush();
                out.close();
            } catch (NullPointerException e) {
            //если сессия не существует, то генерируется исключение
                if (out != null)
                out.print("Session disabled!");
            } catch (IOException e) {
                e.printStackTrace();
            throw new RuntimeException("i/o failed: " + e);
            }
      }
   }
```

При первом запуске в браузер будет выведено:

Creation Time : Tue Aug 14 17:54:23 EEST 2007
Session alive!

Если повторить запрос к сервлету менее чем за 10 секунд, вывод будет повторен. Если же запрос повторить более через десять секунд, сессия будет автоматически уничтожена, и в браузер будет выведено следующее:

Session disabled!

Cookie

Для хранения информации на компьютере клиента используются возможности класса **Cookie.**

Cookie – это небольшие блоки текстовой информации, которые сервер посылает клиенту для сохранения в файлах cookies. Клиент может запретить браузеру прием файлов cookies. Браузер возвращает 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 серверы указанного домена. Этот принцип лежит в основе одного из протоколов обеспечения единой идентификации пользователя (Single Signon), где серверы 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 объект класса **Cookie**, указав конструктору имя и значение блока, и добавить их в объект-response. Конструктор использует имя блока в качестве первого параметра, а его значение – в качестве второго.

```
Cookie cookie = new Cookie("myid", "007");
response.addCookie(cookie);
```

Извлечь информацию cookie из запроса можно с помощью метода **getCookies()** объекта **HttpServletRequest**, который возвращает массив объектов, составляющих этот файл.

```
Cookie[] cookies = request.getCookies();
```

После этого для каждого объекта класса **Cookie** можно вызвать метод **getValue()**, который возвращает строку **String** с содержимым блока cookie. В данном случае этот метод вернет значение "007".

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. Для запуска следующего приложения можно использовать сервлет из примера # 1 этой главы, вставив в метод **performTask()** следующий код:

```
CookieWork.setCookie(resp);  // добавление cookie
CookieWork.printToBrowser(resp, req); // извлечение cookie
```

Класс **CookieWork** имеет вид:

```
/* пример # 3 : создание и чтение cookie : CookieWork.java */
package chapt16;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.Cookie;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class CookieWork {

public static void setCookie(HttpServletResponse resp) {
            String name = "Spiridonov";
            String role = "MegaAdmin";
            Cookie c = new Cookie(name, role);
            c.setMaxAge(3600);// время жизни файла
            resp.addCookie(c);
      }
public static void printToBrowser(
HttpServletResponse response, HttpServletRequest request) {
            try {
                  response.setContentType("text/html");
                  PrintWriter out = response.getWriter();

                  Cookie[] cookies = request.getCookies();
            if (cookies != null) {
                  out.print("Number cookies :"
                              + cookies.length + "<BR>");

            for (int i = 0; i < cookies.length; i++) {
            Cookie c = cookies[i];
            out.print("Secure :" + c.getSecure() + "<br>");
            out.print(c.getName() + " = " + c.getValue()
                                                + "<br>");
                                          }// end for
            }// end if
                  out.close();
            } catch (IOException e) {
                  e.printStackTrace();
                  throw new RuntimeException(e.toString());
            }
      }
}
```

В результате в файле cookie будет содержаться следующая информация:

Number cookies :1
Secure :false
Spiridonov = MegaAdmin

Файл cookie можно изменять. Для этого следует воспользоваться сервлетом из примера #1 и в метод **`performTask()`** добавить следующий код:

```
CookieCounter.printToBrowser(resp, req);
```

В классе **`CookieCounter`** производится модификация файла cookie, хранимого на компьютере клиента.

```
/* пример # 4 : создание cookie и чтение количества вызовов сервлета из cookie :
CookieCounter.java */
package chapt16;
import java.io.IOException;
import java.io.Writer;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class CookieCounter {
/* константа, которая будет использована для установки максимального
времени жизни cookie (здесь указано 30 дней) */
public static final int MAX_AGE_COOKIE = 3600 * 24 * 30;

      public static void printToBrowser(
HttpServletResponse response, HttpServletRequest request) {
            try {
                  Writer out = response.getWriter();
                  out.write("My Cookie counter: ");
/* устанавливает счетчик количества вызовов сервлета пользователем    */
out.write(String.valueOf(prepareCookieCounter(
                                          request,response)));
                  out.flush();
                  out.close();
            } catch (IOException e) {
                  e.printStackTrace();
            throw new RuntimeException("Failed: " + e);
            }
      }
      // обновляет в cookie счетчик обращений пользователя к сервлету
      private static int prepareCookieCounter(
HttpServletRequest request, HttpServletResponse response) {
            Cookie[] cookies = request.getCookies();
            Cookie counterCookie;
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++;
      counterCookie = new Cookie("counter",
                  String.valueOf(counterValue));
      counterCookie.setMaxAge(MAX_AGE_COOKIE);
      response.addCookie(counterCookie);
```

```
                        return counterValue;
                    }//end if
                }//end for
            } //end if
            counterCookie = new Cookie("counter", "1");
            counterCookie.setMaxAge(MAX_AGE_COOKIE);
            response.addCookie(counterCookie);
            return 1;
        }
    }
```

В результате в файле cookie будет содержаться следующая информация:

```
counter
1
localhost/FirstProject/
1024
939371136
29793584
1067152496
29787549
*
```

В браузер будет выведено следующее сообщение:
My session counter: 1

Обработка событий

Существует несколько интерфейсов, которые позволяют следить за событиями, связанными с сеансом, контекстом и запросом сервлета, генерируемыми во время жизненного цикла Web-приложения:

- **javax.servlet.ServletContextListener** – обрабатывает события создания/удаления контекста сервлета;
- **javax.servlet.http.HttpSessionListener** – обрабатывает события создания/удаления HTTP-сессии;
- **javax.servlet.ServletContextAttributeListener** – обрабатывает события создания/удаления/модификации атрибутов контекста сервлета;
- **javax.servlet.http.HttpSessionAttributeListener** – обрабатывает события создания/удаления/модификации атрибутов HTTP-сессии;
- **javax.servlet.http.HttpSessionBindingListener** – обрабатывает события привязывания/разъединения объекта с атрибутом HTTP-сессии;
- **javax.servlet.http.HttpSessionActivationListener** – обрабатывает события связанные с активацией/дезактивацией HTTP-сессии;
- **javax.servlet.ServletRequestListener** – обрабатывает события создания/удаления запроса;
- **javax.servlet.ServletRequestAttributeListener** – обрабатывает события создания/удаления/модификации атрибутов запроса сервлета.

Интерфейсы и их методы
ServletContextListener **contextDestroyed(ServletContextEvent e)** **contextInitialized(ServletContextEvent e)**
HttpSessionListener **sessionCreated(HttpSessionEvent e)** **sessionDestroyed(HttpSessionEvent e)**
ServletContextAttributeListener **attributeAdded(ServletContextAttributeEvent e)** **attributeRemoved(ServletContextAttributeEvent e)** **attributeReplaced(ServletContextAttributeEvent e)**
HttpSessionAttributeListener **attributeAdded(HttpSessionBindingEvent e)** **attributeRemoved(HttpSessionBindingEvent e)** **attributeReplaced(HttpSessionBindingEvent e)**
HttpSessionBindingListener **valueBound(HttpSessionBindingEvent e)** **valueUnbound(HttpSessionBindingEvent e)**
HttpSessionActivationListener ***sessionWillPassivate(HttpSessionEvent e)*** **sessionDidActivate(HttpSessionEvent e)**
ServletRequestListener **requestDestroyed(ServletRequestEvent e)** **requestInitialized(ServletRequestEvent e)**
ServletRequestAttributeListener **attributeAdded(ServletRequestAttributeEvent e)** **attributeRemoved(ServletRequestAttributeEvent e)** **attributeReplaced(ServletRequestAttributeEvent e)**

Регистрация блока прослушивания производится в дескрипторном файле приложения.

Демонстрация обработки событий изменения состояния атрибутов сессии будет рассмотрена на примере №1 данной главы. Обрабатываться будут события добавления атрибута **URL**, а также добавления и изменения атрибута счетчика **counter**. Для этого необходимо создать класс, реализующий интерфейс **HttpSessionAttributeListener** и реализовать необходимые для выполнения поставленной задачи методы.

/ пример # 5 : обработка событий добавления и изменения атрибута сессии : MyAttributeListener.java */*

```
package chapt21;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;

public class MyAttributeListener
            implements HttpSessionAttributeListener {
    private String counterAttr = "counter";

public void attributeAdded(HttpSessionBindingEvent ev) {
    String currentAttributeName = ev.getName();
    String urlAttr = "URL";

    if (currentAttributeName.equals(counterAttr)) {

       Integer currentValueInt = (Integer) ev.getValue();
       System.out.println("в Session добавлен счетчик="
                          + currentValueInt);
    } else if (currentAttributeName.equals(urlAttr)) {

StringBuffer currentValueStr = (StringBuffer)ev.getValue();
       System.out.println("в Session добавлен URL="
                          + currentValueStr);
     } else System.out.println("добавлен новый атрибут");
}
public void attributeRemoved(HttpSessionBindingEvent ev) {
}
public void attributeReplaced(HttpSessionBindingEvent ev) {
    String currentAttributeName = ev.getName();
    // в случае  изменений, произведенных со счетчиком,
    // выводит соответствующее сообщение
    if (currentAttributeName.equals(counterAttr)) {
       Integer currentValueInt = (Integer) ev.getValue();
       System.out.println("В Session заменен счетчик="
                          + currentValueInt);
    }
  }
}
```

Чтобы события обрабатывались, необходимо включить упоминание об обработчике событий в элемент **<web-app>** дескрипторного файла **web.xml**.

```
<listener>
            <listener-class>chapt21.MyAttributeListener
            </listener-class>
</listener>
```

Тогда в результате запуска сервлета **SessionServlet** и нескольких обращений к нему в консоль будет выведено:

```
в Session добавлен URL=
http://localhost:8080/FirstProject/SessionServlet
в Session добавлен счетчик=1
в Session заменен счетчик=1
в Session заменен счетчик=2
```

в Session заменен счетчик=3

Интерфейс **ServletRequestListener** добавлен в Servlet API начиная с версии 2.4. С его помощью можно отследить события создания запроса при обращении к сервлету и его уничтожении.

```
/* пример # 6 : обработка событий создания и уничтожения запроса к сервлету :
MyRequestListener.java */
package chapt21;
import javax.servlet.ServletContext;
import javax.servlet.ServletRequest;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.http.HttpServletRequest;

public class MyRequestListener
            implements ServletRequestListener {
      // счетчик числа обращений к сервлету
      private static int reqCount;

    public void requestInitialized(ServletRequestEvent e) {
            //будет использован для доступа к log-файлу
            ServletContext context = e.getServletContext();
            //будет использован для получения информации о запросе
            ServletRequest req = e.getServletRequest();

            synchronized (context) {
                  String name = "";
      name = ((HttpServletRequest) req).getRequestURI();
                  // сохранение значения счетчика в log-файл
            context.log("Request for " + name
                              + "; Count=" + ++reqCount);
            }
      }
      public void requestDestroyed(ServletRequestEvent e) {
            // вызывается при уничтожении запроса
            System.out.println("Request уничтожен");
      }

}
```

В результате запуска сервлета **SessionServlet**, в log-файл, расположенный в каталоге **/logs** контейнера сервлетов, будет выведено:

```
28.02.2006 23:59:53 org.apache.catalina.core.ApplicationContext log
INFO: Request for /FirstProject/SessionServlet; Count=1
Request уничтожен
```

При этом класс «обработчик событий» должен быть зарегистрирован в файле **web.xml** следующим образом:

```
<listener>
      <listener-name>MyRequestListener</listener-name>
<listener-class>chapt21.MyRequestListener</listener-class>
```

</listener>

Фильтры

Реализация интерфейса **Filter** позволяет создать объект, который может трансформировать заголовок и содержимое запроса клиента или ответа сервера. Фильтры не создают запрос или ответ, а только модифицируют его. Фильтр выполняет предварительную обработку запроса, прежде чем тот попадает в сервлет, с последующей (если необходимо) обработкой ответа, исходящего из сервлета. Фильтр может взаимодействовать с разными типами ресурсов, в частности и с сервлетами, и с JSP-страницами.

Основные действия, которые может выполнить фильтр:

- перехват инициализации сервлета и определение содержания запроса, прежде чем сервлет будет инициализирован;
- блокировка дальнейшего прохождения пары request-response;
- изменение заголовка и данных запроса и ответа;
- взаимодействие с внешними ресурсами;
- построение цепочек фильтров;
- фильтрация более одного сервлета.

При программировании фильтров следует обратить внимание на интерфейсы **Filter**, **FilterChain** и **FilterConfig** из пакета **javax.servlet**. Сам фильтр определяется реализацией интерфейса **Filter**. Основным методом этого интерфейса является метод

void doFilter(ServletRequest req, ServletResponse res, FilterChain chain),

которому передаются объекты запроса, ответа и цепочки фильтров. Он вызывается каждый раз, когда запрос/ответ проходит через список фильтров **FilterChain**. В данный метод помещается реализация задач, обозначенных выше.

Кроме того, необходимо реализовать метод **void init(FilterConfig config)**, который принимает параметры инициализации и настраивает конфигурационный объект фильтра **FilterConfig**. Метод **destroy()** вызывается при завершении работы фильтра, в тело которого помещаются команды освобождения используемых ресурсов.

Жизненный цикл фильтра начинается с однократного вызова метода **init()**, затем контейнер вызывает метод **doFilter()** столько раз, сколько запросов будет сделано непосредственно к данному фильтру. При отключении фильтра вызывается метод **destroy()**.

С помощью метода **doFilter()** каждый фильтр получает текущий запрос и ответ, а также список фильтров **FilterChain**, предназначенных для обработки. Если в **FilterChain** не осталось необработанных фильтров, то продолжается передача запроса/ответа. Затем фильтр вызывает **chain.doFilter()** для передачи управления следующему фильтру.

В следующем примере рассматривается обращение к объекту **request** JSP-страницы **demofilter.jsp** и изменение значения атрибута запроса **testName**.

<!--пример # 7 : обращение к атрибуту : demofilter.jsp -->

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-5"  pageEncoding="ISO-8859-5"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" pre-
fix="c"%>
<html><head><title>Demo Filter</title></head>
<body>
<c:out value="Info from filter: ${info}"/><br>
<P>Дублирование действий фильтра смотреть в консоли</P>
</body></html>
```

Если фильтр не подключать, то переменная **info** значения не получит.

Реализация интерфейса **Filter** для поставленной задачи выглялит следующим образом:

```
/* пример # 8 : простая фильтрация значения атрибута : MyFilter.java */
package chapt21;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class MyFilter implements Filter {
      private FilterConfig filterConfig;

      public void init(final FilterConfig filterConfig) {
            this.filterConfig = filterConfig;
      }
      public void doFilter(final ServletRequest request,
      final ServletResponse response, FilterChain chain)
                  throws java.io.IOException,
                        javax.servlet.ServletException {
            System.out.println("Вход в фильтр");
            String value = "Simple Filter";

            request.setAttribute("info", value);

            chain.doFilter(request, response);
            System.out.println("info = " + value);
            System.out.println("Окончание фильтра");
      }
      public void destroy() {
            System.out.println("Уничтожение фильтра");
      }
}
```

Чтобы к фильтру происходило обращение, необходимо включить упоминание о фильтре и обрабатываемом ресурсе в элемент **<web-app>** дескрипторного файла **web.xml** в виде:

```
<filter>
    <filter-name>simplefilter</filter-name>
```

```
    <filter-class>chapt21.MyFilter</filter-class>
 </filter>
 <filter-mapping>
    <filter-name>simplefilter</filter-name>
    <url-pattern>/demofilter.jsp</url-pattern>
 </filter-mapping>
```

Фильтр может модифицировать ответ сервера клиенту. Одним из распространенных приемов использования фильтра является модификация кодировки ответа. Когда сервлет посылает ответ клиенту через поток **PrintWriter**, используется установленная в сервлете кодировка. В следующем примере рассматривается фильтр, изменяющий кодировку ответа на кириллицу UTF-8.

// пример # 9 : фильтр, устанавливающий кодировку запроса : SetCharFilter.java

```
package chapt21;
import java.io.IOException;
import javax.servlet.*;

public class SetCharFilter implements Filter {
      private FilterConfig filterConfig = null;

      public void init(FilterConfig config)
throws ServletException {
            this.filterConfig = config;

      }
public void doFilter(ServletRequest request,
ServletResponse response, FilterChain next)
throws IOException, ServletException {
      // чтение кодировки из запроса
      String encoding = request.getCharacterEncoding();
      System.out.println(encoding);
      // установка UTF-8, если не установлена
      if (!"UTF-8".equalsIgnorCase(encoding))
            response.setCharacterEncoding("UTF-8");
      next.doFilter(request, response);
      }
      public void destroy() {
      }
}
```

И его конфигурации в **web.xml**:

```
<filter>
      <filter-name>setCharFilter</filter-name>
      <filter-class>chapt21.SetCharFilter</filter-class>
</filter>
<filter-mapping>
      <filter-name>setCharFilter</filter-name>
      <url-pattern>/DemoCharServlet</url-pattern>
</filter-mapping>
```

Таким образом, ответ сервлета **DemoCharServlet** будет в необходимой кодировке.

*/*пример # 10 : без установки кодировки ответ сервлета будет нечитаем : DemoCharServlet.java*/*

```
package chapt21;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DemoCharServlet extends HttpServlet {
      public void init() throws ServletException {
      }
      public void doGet(HttpServletRequest request,
            HttpServletResponse response)
                  throws ServletException, IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.print("Кодировка установлена успешно!");
      }
      public void destroy() {
            super.destroy();
      }
}
```

В результате в браузер будет выведено:

Кодировка установлена успешно!

Задания к главе 21

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 хранится несколько стихотворений, которые разделяются строкой, состоящей из одних звездочек. В каком из стихотворений 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. Подсчитать 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. Подсчитать, 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 файле cookie.
10. В файле хранится текст. Найти три предложения, содержащие наибольшее количество знаков препинания, и сохранить их в файле 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, которые 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 файле cookie.
18. Выбрать из файла имена зон (*.by, *.ru и т.д.), вводимые пользователем, и сохранить их в файле 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 21

Вопрос 21.1.

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 21.2.

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 21.3.

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 21.4.

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 21.5.

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 21.6.

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 22
ПОЛЬЗОВАТЕЛЬСКИЕ ТЕГИ

Начиная с версии JSP 1.1 у разработчиков появилась возможность определения собственных тегов. Это значительно упростило жизнь Web-дизайнерам, которым привычнее использовать теги, а не код на языке Java. Если один и тот же скриптлет используется на разных страницах, то он явный кандидат для переноса кода в пользовательский тег. Фактически последний представляет собой перенос Java-кода из страницы JSP в Java-класс, что можно считать продолжением идеи о необходимости отделения логики от представления. JSP-страница должна содержать как можно меньше логики.

Для создания пользовательских тегов необходимо определить класс обработчика тега, определяющий его поведение, а также дескрипторный файл библиотеки тегов (файл **.tld**), в которой описываются один или несколько тегов, устанавливающих соответствия между именами XML-элементов и реализацией тегов.

При определении нового тега создается класс Java, который должен реализовывать интерфейс **javax.servlet.jsp.tagext.Tag**. Обычно создается класс, который наследует один из классов **TagSupport** или **BodyTagSupport** (для тегов без тела и с телом соответственно). Указанные классы реализуют интерфейс **Tag** и содержат стандартные методы, необходимые для базовых тегов. Класс для тега должен также импортировать классы из пакетов **javax.servlet.jsp** и, если необходима передача информации в поток вывода, то **java.io** или другие классы.

Простой тег

Для создания тега без атрибутов или тела необходимо переопределить метод **doStartTag()**, определяющий код, который вызывается во время запроса, если обнаруживается начальный элемент тега.

В качестве примера можно привести следующий класс тега, с помощью которого клиенту отправляется информация о размере некоторой коллекции объектов.

```
/ пример # 1 : простейший тег без тела и атрибутов : GetInfoTag.java */
package test.mytag;
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import java.io.IOException;
// класс бизнес-логики (см. пример #7 этой главы)
import test.my.MySet;

public class GetInfoTag extends TagSupport {
      public int doStartTag() throws JspException {
//получение информации, передаваемой на страницу
```

```
        int size = new Integer(new MySet().getSize());
        String str = "Size =<B>( " + size + " )</B>";
              try {
                    JspWriter out = pageContext.getOut();
                    out.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.

Чтобы сгенерировать вывод, следует использовать метод **write()** класса **JspWriter**, который выводит на страницу содержимое объекта **str**. Объект **pageContext** класса **PageContext** – это атрибут класса, унаследованный от класса **TagSupport**, обладающий доступом ко всей области имен, ассоциированной со страницей JSP. Метод **getOut()** этого класса возвращает ссылку на поток **JspWriter**, с помощью которой осуществляется вывод. С помощью методов класса **PageContext** можно получить:

getRequest() – объект запроса;
getResponse() – объект ответа;
getServletContext() – объект контекста сервлета;
getServletConfig() – объект конфигурации сервлета;
getSession() – объект сессии;
ErrorData getErrorData() – информацию об ошибках.

Кроме этого:

– 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**, определяемые относительным адресом.

Следующей задачей после создания класса обработчика тега является идентификация этого класса для сервера и связывание его с именем XML-тега. Эта задача выполняется в формате XML с помощью дескрипторного файла библиотеки тегов.

Файл дескриптора **.tld** 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 должен присутствовать стандартный заголовок XML-файла с указанием версии и адреса ресурса для схемы XSD, который определяет допустимый формат тега **<taglib>**.

Перед списком тегов, сразу после открывающего тега **<taglib>**, указываются следующие параметры:

- **tlib-version** – версия пользовательской библиотеки тегов;
- **short-name** – краткое имя библиотеки тегов. В качестве него принято указывать рекомендуемое сокращение для использования в JSP-страницах;
- **uri** – уникальный идентификатор ресурса, определяющий 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.

Основным в элементе **<taglib>** является элемент **<tag>**. 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**;
- **tag-class** – полное имя класса-обработчика тега;
- **info** – краткое описание тега;
- **body-content** – 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 обрабатываются, используют значение **tagdependent**.

Вся эта информация помещается в файл **mytaglib.tld**, который для JSP версии 2.0 имеет вид:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib
      xmlns="http://java.sun.com/JSP/TagLibraryDescriptor"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/
web-jsptaglibrary_2_0.xsd"
      version="2.0"><!--дескриптор библиотеки тегов -->
      <tlib-version>1.0</tlib-version>
      <short-name>mytag</short-name>
      <uri>/WEB-INF/mytaglib.tld</uri>
      <tag>
            <name>getinfo</name>
            <!--класс обработки тега -->
            <tag-class>test.mytag.GetInfoTag</tag-class>
            <body-content>empty</body-content>
      </tag>
</taglib>
```

Для JSP версии 2.1 тег **taglib** записывается в виде:

```
<taglib version="2.1"
  xmlns="http://java.sun.com/xml/ns/javaee"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      web-jsptaglibrary_2_1.xsd">
```

Зарегистрировать адрес URI библиотеки пользовательских тегов **mytaglib.tld** для приложения можно двумя способами:

1. Указать доступ к ней в файле **web.xml**, для чего следует указать после **<welcome-file-list\>**:

```
<jsp-config>
   <taglib>
      <taglib-uri>/WEB-INF/mytaglib.tld</taglib-uri>
      <taglib-location>/WEB-INF/mytaglib.tld
      </taglib-location>
   </taglib>
</jsp-config>
```

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 адресом URI. Естественно, в этом случае TLD-файл должен размещаться не в локальной папке проекта, а, например, в сети Интернет как независимый файл.

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:

```
Size = (3)
```

Тег с атрибутами

Тег может содержать параметры и передавать их значения для обработки в соответствующий ему класс. Для этого при описании тега в файле ***.tld** используются атрибуты, которые должны объявляться внутри элемента **tag** с помощью элемента **attribute**. Внутри элемента **attribute** между тегами **<attribute>** и **</attribute>** могут находиться следующие элементы:

- **name** – имя атрибута (обязательный элемент);
- **required** – указывает на то, всегда ли должен присутствовать данный атрибут при использовании тега, который принимает значение **true** или **false** (обязательный элемент);

- **rtexprvalue** – показывает, может ли значение атрибута быть JSP-выражением вида **${expr}** или **<%=expr%>** (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 (необязательный элемент).

Соответственно для каждого из атрибутов тега класс, его реализующий, должен содержать метод **set**ИмяАтрибута**()**.

В следующем примере рассматривается простейший тег с атрибутом **firstname**, 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 String firstname;

     public void setFirstname(String firstname) {
        this.firstname = firstname;
    }
     public int doStartTag() {
        try {
        pageContext.getOut().write("Hello, " + firstname);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return SKIP_BODY;
    }
}
```

В файл **mytaglib.tld** должна быть помещена следующая информация о теге:

```
<tag>
  <name>hello</name>
  <tag-class>test.mytag.HelloTag</tag-class>
  <body-content>empty</body-content>
  <attribute>
      <name>firstname</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

Использовать созданный тег в файле **demotag2.jsp** можно следующим образом:

пример # 4 : вызов тега с передачей ему значения : demotag2.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"
                  prefix="c" %>
```

```
<%@ taglib uri="/WEB-INF/mytaglib.tld" prefix="mytag"%>
<%@ page
            language="java"
            contentType="text/html; charset=CP1251"
            pageEncoding="CP1251"
      %>
<HTML><HEAD>
      <TITLE>demotag2.jsp</TITLE>
      </HEAD>
            <BODY>
      <c:set var="login" value="Bender"/>
      <mytag:hello firstname="${login}" />
            </BODY>
</HTML>
```

При обращении по адресу:

http://localhost:8080/FirstProject/demotag2.jsp

в браузер будет выведено:

Hello, Бендер

Тег с телом

Как и в обычных тегах, между открывающим и закрывающим пользовательскими тегами может находиться тело тега, или **body**. Пользовательские теги могут использовать содержимое элемента **body-content.** На данный момент поддерживаются следующие значения для **body-content**:

- **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 отработаны 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.

В следующем примере рассматривается 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 = new Integer(num);
      }
      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) {
                  try {
pageContext.getOut().write("</TD></TR><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("</TD></TR>");
            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>
      <tag-class>test.mytag.AttrTag</tag-class>
      <body-content>JSP</body-content>
      <attribute>
            <name>num</name>
            <required>false</required>
            <rtexprvalue>true</rtexprvalue>
      </attribute>
</tag>
```

При использовании в файле JSP тег **bodyattr** может вызываться с параметрами и без них:

пример # 6 : тег с телом : demotag3.jsp

```
<%@ page language="java" contentType="text/html; char-
set=ISO-8859-5" pageEncoding="ISO-8859-5"%>
<%@ taglib uri="/WEB-INF/mytaglib.tld" prefix="mytag" %>
<HTML><HEAD>
            <TITLE>Example</TITLE>
</HEAD><BODY>
<jsp:useBean id="rw" scope="request" class=
                                          "test.my.MySet"/>
      <mytag:bodyattr num="${rw.size}">
            ${rw.element}
      </mytag:bodyattr>
      <mytag:bodyattr> Просто текст </mytag:bodyattr>
</BODY></HTML>
```

В результате запуска этой JSP клиенту будет возвращено:

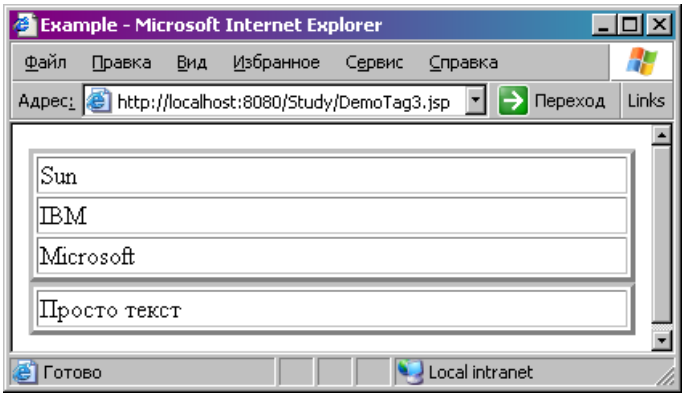


Рис. 22.1. Выполнение тега с телом

В примерах данной главы были использованы методы класса **Rows**, который приведен ниже:

/ пример # 7 : примитивный класс бизнес-логики : MySet.java */*

```
package test.my;
```

```
public class MySet extends java.util.HashSet {
      private java.util.Iterator it;

      public MySet() {
//переписать этот класс на чтение информации из БД
            this.add("Sun");
            this.add("Microsoft");
            this.add("IBM");
      }
      public String getSize() {
            it = this.iterator();
            return Integer.toString(this.size());
      }
      public String getElement() {
            return it.next().toString();
      }
}
```

Элементы action

Элемент **jsp:attribute** позволяет определить значение атрибута тега в теле XML-элемента, а не через значение атрибута стандартного или пользовательского тега:

```
<%@ taglib uri="/WEB-INF/mytaglib.tld" prefix="mytag" %>
<HTML>
      <mytag:hello>
            <jsp:attribute name="firstname">
                  Bender
            </jsp:attribute>
      </mytag:hello>
</HTML>
```

в браузер будет выведено:

Hello, Bender

Если в теле тега имеются элементы **jsp:attribute**, то тело тега нужно указать явно при помощи стандартного действия **jsp:body**:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>
<%@ taglib uri="/WEB-INF/mytaglib.tld" prefix="mytag" %>
<HTML>
      <jsp:useBean id="rw" scope="request"
                           class="test.my.MySet"/>
      <mytag:bodyattr>
            <jsp:attribute name="num">
                  <c:out value="${requestScope.rw.size}"/>
            </jsp:attribute>
            <jsp:body>
                <c:out value="${requestScope.rw.element}"/>
            </jsp:body>
      </mytag:bodyattr>
</HTML>
```

В результате в браузер будет выведено:

IBM
Sun
Microsoft

Элемент **jsp:element** с обязательным атрибутом **name** используется для динамического определения элемента XML и дополнительно может содержать действия **jsp:attribute** и **jsp:body**:

```
<jsp:element name="H2" >
      <jsp:attribute name="Style">
            color:red
      </jsp:attribute>
      <jsp:body>
            Simple Text
      </jsp:body>
</jsp:element>
```

в результате должно быть сгенерировано:

<H2 Style="color:red">Simple Text</H2>

Стандартные действия **jsp:doBody** и **jsp:invoke** используются только в тег-файлах. Тег **jsp:doBody** вызывает тело тега, выводя результат в **JspWriter** или в атрибут области видимости. Действие **jsp:invoke** подобно действию **jsp:doBody** и используется для вызова атрибута-фрагмента. Например, поведение тега **bodyattr** можно воспроизвести так:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<tags:actiondemo>
      <jsp:attribute name="num">
            <c:out val-
ue="${sessionScope.mysetInstance.size}"/>
      </jsp:attribute>
      <jsp:body>
            <c:out val-
ue="${sessionScope.mysetInstance.element}"/>
      </jsp:body>
</tags:actiondemo>
```

Файл **actiondemo.tag** помещен в каталог **/WEB-INF/tags**:

```
<%@ tag import="test.my.MySet" %>
<%@ attribute name="num" fragment="true" %>
<%@ variable name-given="mysetInstance" %>
<%session.setAttribute("mysetInstance", new MySet());%>
<TABLE border=1>
<TR><TD>Rows number: <jsp:invoke fragment="num"</TD></TR>
      <TR><TD><jsp:doBody /></TD></TR>
      <TR><TD><jsp:doBody /></TD></TR>
      <TR><TD><jsp:doBody /></TD></TR>
</TABLE>
```

Здесь директива **tag** схожа с директивой **page** для страниц JSP. Директива **attribute** декларирует атрибут тега **actiondemo**, и если **fragment="true"**, то этот атрибут можно использовать совместно с **jsp:invoke**. Директива **variable** – для передачи переменной обратно в вызывающую JSP-страницу.
В браузер будет выведено:

Rows number: 3
IBM
Sun
Microsoft

Задания к главе 22

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 $a_1, a_2, \ldots, a_n$. Вычислить суммы вида $S_i = a_i + a_{i+1} + \ldots + a_j$ 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 В

Для заданий варианта В предыдущей главы применить пользовательские теги для визуализации работы приложения.

Тестовые задания к главе 22

Вопрос 22.1.

Какой элемент тега **<attribute>** определяет имя атрибута, которое должно быть передано обработчику тегов?

1) `<attribute-name>;`
2) `<name>;`
3) `<attributename>;`
4) `<param-name>.`

Вопрос 22.2.

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 22.3.

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 22.4.

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 22.5.

Какие элементы описывают характеристики пользовательского тега в файле **.tld**?

1) value;
2) name;
3) rtexprvalue;
4) class.

Вопрос 22.6.

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 22.7.

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.

Приложение 1

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.

Существует несколько версий HTML. В 1999 году вышла последняя редакция языка – HTML 4.01. С тех пор развиваются языки разметки, базирующиеся на XML. Среди них – XHTML 1.0/ XHTML 2.0, MathML (язык разметки математических формул) и некоторые другие.

Каждый HTML-документ, отвечающий спецификации HTML какой-либо версии, должен начинаться со строки объявления версии HTML, которая выглядит так:

Строгий (Strict) – не содержит элементов, помеченных как «устаревшие» или «не одобряемые» (deprecated):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
```

Переходный (Transitional) – содержит устаревшие теги в целях совместимости и упрощения перехода со старых версий HTML:

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
```

С фреймами (Frameset) – аналогичен переходному, но содержит также теги для создания наборов фреймов:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
    "http://www.w3.org/TR/html4/frameset.dtd">
```

Документ, соответствующий XHTML 1.0, автоматически соответствует также и HTML 4.0. XHTML 2.0 обеспечивает полное отделение содержания документа от его представления. В нем нет ничего отвечающего за внешний вид документа – нет элементов **font**, **i**, **b**, **center**, нет атрибутов **align**, **size** и **bgcolor**. Предполагается, что вся информация об оформлении документа содержится в отдельной таблице стилей. Кроме того, в XHTML 2.0 вводится много новых элементов – например элемент **section**, обозначающий структурную часть документа, и элемент **h** – заголовок общего вида.

Таким образом, если предполагается использование документа вместе с расширениями в XHTML 2.0, базирующемся на XML, следует учитывать, что в документе нельзя использовать устарелые или нежелательные элементы. При этом:

- верхний и нижний регистры различаются, все теги следует писать с маленькой буквы;
- теги **center**, **font**, **s**, **u**, атрибуты **align**, **bgcolor**, **background**, **clear**, **color**, **face**, **size**, т.е. почти все теги и атрибуты, определяющие представление документа HTML (цвета, выравнивание, шрифты, графика и т.д.), являются нежелательными, взамен рекомендуется использовать таблицы стилей;

- если пишется атрибут, то обязательно должно быть указано его значение. Все значения атрибутов должны писаться в кавычках.
- обязательно прописывать и закрывающие теги! (например у тега **li**). Примеры тегов, у которых отсутствуют закрывающие теги: **
, **<img> и т.д. В случае если закрывающий тег отсутствует, то в конце тега пишется «**/**».

HTML-документ создается с помощью HTML-редактора 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 (в XML используются строчные буквы).

Заголовочная часть документа <HEAD>

Тег заголовочной части документа используется сразу после тега **<HTML>**. Данный тег содержит общее описание 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.

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 и смысловую часть документа.

```
<!--пример # 1: простой HTML документ-->
<HTML>
<HEAD><TITLE>Домашняя страница </TITLE></HEAD>
<BODY>
<!-- Это комментарий-->
<H1>Пример заголовка, размер 1</H1>
<H6>Пример заголовка, размер 6</H6>
<ADDRESS>Романчик - e-mail:rom@bsu.by</ADDRESS> <P>
</BODY></HTML>
```

В этом примере использовались теги: **<P>** – тег нового абзаца. Тег **<ADDRESS>** позволяет сформировать информацию о связи с автором документа и определяет вид сообщения. HTML позволяет вставлять в тело документа комментарии. 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, и т.д. до шести. Синтаксис заголовка уровня 1 следующий:

```
<H1> Заголовок первого уровня </H1>
```

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, хотя может его иметь. Если браузер не понимает тега, то он его просто пропускает.

Разбивка текста на смысловые группы

В HTML-документе браузер игнорирует 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. Дополнительный параметр **CLEAR** тега **
** позволяет не просто выполнить перевод строки, но и разместить следующую строку, начиная с чистой левой (**left**), правой (**right**) или обеих (**all**) границ окна браузера:

```
 <BR CLEAR=left|right|all>
```

Например, если рядом с текстом слева встречается рисунок, то можно использовать тег **
** для смещения текста ниже рисунка:

```
<!--пример # 2: разбивка 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>
</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 пере-

носить 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 какой-либо разбивки где бы то ни было: **</NOBR>.** Если необходимо всё же позволить разбивку данной строки на две, но в строго запланированном месте, то следует вставить тег **<WBR>** в это место.

Линии

Тег **<HR>** проводит контурную горизонтальную линию (опция **SHADE** по умолчанию). Например:

<HR NOSHADE> – горизонтальная линия с тенью;
<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>
</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 ** **, **<font>**,**<s>**,**<u>** для оформления стиля текста применяются в настоящее время крайне редко и являются нежелательными элементами. Вместо них широко используются таблицы стилей CSS.

Тег **<CENTER>** позволяет центрировать все элементы документа в окне браузера. 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>**,

<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> </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 -->
</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 </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>
</BODY></HTML>
```

Работа с тегами FONT

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>
```

Кроме размера, могут устанавливаться цвет и тип шрифта, например:

```
<FONT COLOR=RED SIZE=6 FACE="Arial"> Пример шрифта </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=1> Шрифт размера 1</font> <HR>
<H3>Задание относительных размеров шрифтов</h3>
<P><FONT size=+4> Шрифт размера +4</font>
<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="red">
<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>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">
<HR color="lime" size=15 width=195 align="left">
</BODY></HTML>
```

Специальные символы

Символы, которые не могут быть введены в текст документа непосредственно через клавиатуру, называются специальными символами. Для них 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. Для использования одного из этих символов введите одну из следующих 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>
```

Атрибуты ненумерованного списка: **type="disk | circle | square".** Атрибут нумерованного списка **start** - устанавливает число, с которого будет начинаться отсчет. 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>"Lamer"</B>
<DD><I> Юзер-идиот</I>
      <DT><B>"Cookies "</B>
<DD><I>Технология, позволяющая сохранять индивидуальную ин-
формацию о пользователе сети</I>    </DL>
</BODY></HTML>
```

Для создания таблиц используются следующие теги **HTML: <TABLE>** и **</TABLE>** – охватывают таблицу. Для того чтобы была видна сетка, разделяющая строки и столбцы, 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>

</TABLE></BODY></HTML>
```

Чтобы ячейка занимала две строки вместо одной, можно заменить тег на следующий: **<TD ROWSPAN=2> </TD>**. Аналогично два столбца можно объединить 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. Рисунки можно изготовить с помощью графических редакторов.

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">
```

В последних версиях HTML атрибут **alt** обязателен для тега img.

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>
```

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**.

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.

Для внедрения в документ таких объектов, как видеоклипы, музыкальные файлы, flash-анимация могут использоваться теги **<OBJECT>** и **<EMBED>.** Например, подгрузить картинку в документ можно следующим образом:

```
<object data="fract.jpg" type="image/jpeg"> текст
фрактала</object>
```

Атрибут **data** указывает местоположение данных для объекта, атрибут **type** указывает тип содержимого **image** – изображение, **jpeg** – тип изображения. Текст, заключенный в тег **<OBJECT></OBJECT>**, отобразится, если браузер не сможет отобразить объект-картинку.

Чтобы подгрузить видеоклип, запишем аналогично:

```
<object data="film.mpeg" type="application/mpeg"> фильм
</object>
```

Тэг **<EMBED>** поддерживается на данный момент почти всеми браузерами и используется, как и тег **<OBJECT>**, для внедрения объектов в документ. Рассмотрим внедрение музыкального файла в документ:

```
<embed src="music.mid" width="40" height="30"
       autostart="false" loop="true"
       play_loop="5" hidden="false">
</embed>
```

В теге могут использоваться атрибуты: **src** (путь к объекту), **align** (выравнивание), **width** (ширина), **height** (высота), **hspace** (расстояние до текста или других объектов по горизонтали), **vspace** (расстояние до текста или других объектов по вертикали), **hidden="true" (false)** – отображать указанный объект, **type=" image/jpeg"** – тип подгружаемого объекта, **pluginspage="http://…"** – путь к плагину (plugin – небольшая программка, выполняющая какие-либо дополнительные функции. Например, чтобы проиграть flash-анимацию, браузеру нужен плагин, к которому мы указываем путь на случай отсутствия в браузере).

Рассмотрим пример вставки flash-анимации в документ:

```
<ENBED src="someMovie.swf"  quality=high  bgcolor=#FFFFFF
WIDTH=500 HEIGHT=400 swLiveConnect=true NAME=yourMovie
TYPE="application/x-shockwave-flash"
PLUGINSPAGE="http://www.macromedia.com/Shockwave/download/
index.cgi?Pl_Prod_Version=ShockwaveFlash">
</EMBED>
```

Для тега **<OBJECT>** четыре параметра (**HEIGHT**, **WIDTH**, **CLASSIC** и **CODEBASE**) являются атрибутами и записываются внутри тега **<OBJECT>**, а все остальные записываются отдельно в виде тегов **<PARAM>**. Например:

```
<OBJECT CLASSID="clsid:D27CDB6E-AE6D-llcf-96B8-
444553540000"
WIDTH="100"
HEIGHT="100"CODEBASE="http://active.macromedia.com/flashS/
cabs/swflash.cab#version=5,0,0,0">
<PARAM NAME="MOVIE"VALUE="moviename .swf">
<PARAM NAME="PLAY"VALUE="true">
<PARAM NAME="LOOP"VALUE="true">
<PARAM NAME="QUALITY"VALUE="high">
</OBJECT>
```

Фреймы

Технология фреймирования в HTML позволяет просматривать в одном окне обозревателя несколько гипертекстовых документов. Однако в последние годы фреймы практически не используются.

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 # 15: 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 # 16: 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 # 17: 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 # 18: 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. 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; char-
set=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
словарь 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) позволяют 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 устанавливаются 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 подчеркиванием </H1>
<H2 style="color: red">
Текст заголовка красным цветом</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 # 19: 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 # 20: 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 # 21: 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 # 22: 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 # 23: 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

JAVASCRIPT

Язык сценариев JavaScript создал сотрудник компании Netscape Communication Брендан Эйх для разработки Web-приложений на качественно новом уровне. JavaScript принес на клиентскую Web-страницу динамику и интерактивность и заменил Java-апплеты на клиентской странице. Синтаксис языка JavaScript очень похож на синтаксис языка Java, однако это полностью самостоятельный язык скиптов. Язык JavaScript используется совместно с HTML, XML и может использовать объекты языка Java.

Включение скриптов на языке JavaScript в HTML-код

Традиционное вкючение скрипта тег **<script>**, имело вид:

```
<!-- пример # 1: шаблон HTML для скрипта -->
<html>
<head>
<title> Шаблон HTML </title>
<script language="javascript">
<!--Маскируемся, начало JavaScript
 //...код скрипта
// снятие маскировки; конец JavaScript -->
</script>
</head>
<body >
</body></html>
```

В настоящее время использование атрибута **language** является устаревшим. Вместо него используется атрибут **type**, значением **type** является **"text/javascript"**:

```
<script type="text/javascript">
//JavaScript код
</script>
```

Предпочтительно располагать тег **script** внутри тега **head**, т.к. это гарантирует выполнение скрипта до начала загрузки основного HTML-кода страницы. Тег **script** может располагаться в любом месте HTML. Код, содержащийся внутри, будет выполнен незамедлительно, если этот код не функция.

Кроме того, код скрипта можно хранить в отдельном файле с расширением **.js** – в этом случае в HTML-коде скрипт объявляется с помощью тега **script** с атрибутом **src**, в котором прописывается путь к файлу.

```
<script type="text/javascript"
src="/jspr/pr.js"></script>
```

В этом варианте в директории **/jspr/** должен находиться файл **pr.js**, который содержит код.

Отладка скриптов JavaScript

Наиболее распространенный способ отладки заключается в многократном вызове метода **alert()** объекта **window**, который выводит стандартное окно

с текстом и кнопкой OK или других методов, выводящих окна, например **confirm()** и **prompt():**

```
<!-- пример # 2: Вывод текста в окно -->
<html> <head>
<title> вывод окна</title>
<script type="text/javascript">
alert("Вас приветствует JavaScript и метод alert()!");
confirm("Метод confirm, выберите вариант !");
prompt("Метод prompt(),Введите Ваше имя");
</script>
</head>
<body >
<P> <CENTER>
<H1 style="color:blue">Вывод  окон: alert(), confirm() и
prompt() </h1>
<HR><P><P>Страница документа </center>
</body> </html>
```

Помимо этого, существует более продвинутый способ отладки.

Если используется MS Visual Studio, потребуется отключить в браузере Internet Explorer параметр `Disable script debugging`, который находится в настройках браузера на закладке Advanced. После этого можно использовать ключевое слово `debugger`; – что вызовет Visual Studio, позволив шаг за шагом пройти по скрипту и просмотреть значения любых переменных.

Кроме этого, в браузере Internet Explorer можно воспользоваться пунктом меню View > Script Debugger > Break at Next Statement, в этом случае Visual Studio будет запущена, как только будет выполнена какая-либо команда JavaScript.

Для других браузеров (не от компании Microsoft) такой способ не подойдет, однако практически для каждого браузера написаны специальные компоненты, которые также позволяют производить отладку скриптов.

Типы данных

Переменные в JavaScript объявляются с помощью ключевого слова **var**, например:

```
var x;
```

После этого можно задать значение переменной:

```
x = "Sample string";
```

Язык JavaScript позволяет создавать переменные «на лету» без их объявления, например:

```
y = "Second string";
```

При этом переменная `y` будет создана, и в памяти будет отведено для нее место, однако такая практика затрудняет чтение и отладку скрипта.

В языке JavaScript переменные не имеют строго закрепленного типа, тип переменной определяется данными, которые она хранит. Например, можно объявить переменную и присвоить ей число, а затем присвоить строку.

JavaScript поддерживает пять базовых типов данных: **Number** – числа; **String** – строки; **Boolean** – Булев тип; **Undefined** – неопределенный; **Null** - пустой. Эти пять типов данных называются базовыми, т.к. на основе их строятся более сложные типы данных. Фактический интерес с точки зрения хранения дан-

ных представляют три: числа, строки и логические значения, неопределенный и пустой типы представляют интерес в определенных ситуациях.

Числа

В языке JavaScript численный тип данных включает целые и вещественные числа. Целые числа могут принимать значения от -2^{53} до 2^{53}, вещественные могут принимать большие значения в пределах $\pm 1.7976 \times 10^{308}$ или быть точными в пределах $\pm 2.2250 \times 10^{-308}$.

Числа также могут записываться в экспоненциальной форме.

Для записи целого шестнадцатеричного числа в начале ставится ноль, затем буква **x**, затем само число, которое может содержать цифры от **0** до **9** и буквы от **A** до **F**.

Числа в шестнадцатеричной системе счисления могут пригодиться при использовании битовых операций, а также для хранения цветов – для Web все цвета хранятся в шестнадцатеричном виде.

В языке JavaScript также имеется возможность записи чисел в восьмеричной системе счисления: для записи числа в начале ставится **0**, затем идут цифры от **0** до **7**.

Специальные числа

Если результат математической операции выходит за допустимые пределы, переменная принимает значение **Infinity** – бесконечность. При совершении любых операций над таким числом результатом будет бесконечность. При сравнении положительная бесконечность всегда больше любого действительного числа, и наоборот, отрицательная бесконечность всегда меньше любого действительного числа.

Еще одним важным специальным значением является **NaN** – «не число» (not a number). Типичным примером операции, которая возвращает **NaN**, является деление на ноль. Для определения, является ли значение переменной **NaN**, используется функция **isNaN()**, которая возвращает **true**, если число является действиительным (включая бесконечность), и **false**, если значение переменной **NaN**.

К специальным числам относятся максимальное и минимальное значения, которые может принимать переменная. Все специальные числа приведены в таблице:

Специальное число	Название
Number.MAX_VALUE	Максимальное значение числа
Number.MIN_VALUE	Минимальное значение числа
Number.NaN	Не число
Number.POSITIVE_INFINITY	Положительная бесконечность
Number.NEGATIVE_INFINITY	Отрицательная бесконечность

Строки

Строка – это последовательность символов огрниченная двойными или одинарными кавычками.

После создания строки она имеет одно свойство – **length**, возвращающее длину строки, и большое количество методов:

charAt(*index* : Number) : String – возвращает символ, находящийся на определенной позиции;

concat([*string1* : String [, ... [, *stringN* : String]]]) : String – соединяет строки (аналогично оператору «+»);

indexOf(*subString* : String [, *startIndex* : Number]) : Number – возвращает номер вхождения подстроки в строку, необязательным параметром является номер символа, с которого начинается поиск. Если подстрока не найдена, возвращается **-1**. Поиск выполняется слева направо, для поиска справа налево используется метод **lastIndexOf()**, который имеет аналогичный синтаксис;

replace(*rgExp* : RegExp, *replaceText* : String) : String – выполняет замену регулярного выражения строкой;

split([*separator* : { String | RegExp } [, *limit* : Number]]) : Array – разбивает строку на массив подстрок. В качестве первого параметра передается разделитель, на основе которого производится разбиение, если разделитель не указан, возвращается массив, содержащий один элемент с исходной строкой. Второй параметр определяет максимальное количество элементов в возвращаемом массиве;

substr(*start* : Number [, *length* : Number]) : String – возвращает подстроку, которая начинается с определенной позиции и имеет определенную длину;

substring(*start* : Number, *end* : Number) : String – возвращает подстроку, которая начинается и заканчивается в позициях, определенных параметрами.

Булев тип

Переменные булевого типа могут принимать одно из двух значения: **true** – истина; **false** – ложь. Переменные булевого типа часто используются в условном операторе **if**. Пример:

```
var doAlert = true;
if (doAlert) { alert("Hello, World!"); }
```

Переменные типа Undefined и Null

Тип **Undefined** используется для несуществующих переменных или переменных, значения которых еще не определены. В следующем примере переменная **x** будет иметь значение **Undefined** – то есть не определена.

```
var x;
```

Тип **Null** означает пустое значение. Пример объявления пустой переменной:

```
var x = null;
```

Массивы

В JavaScript массив – это упорядоченная коллекция различных данных. Все элементы массива пронумерованы, первый элемент массива имеет нулевой индекс. Доступ к определенному элементу массива осуществляется с помощью квадратных скобок и номера этого элемента. Пример:

```
myArr[3] = "Hello!";
var x = myArr[3];
```

Объявление пустого массива и массива, содержащего данные:

```
var emptyArr = [];
var filledArr = ["Hello", "World"];
```

Массивы могут быть многомерными. Объявление и обращение к многомерному массиву выглядит так:

```
var myArr = [[1,2],[3,4],[5,6]];
alert(myArr[2][1]);
```

//создается массив размером 3x2 и выводится элемент, содержащийся в третьей строке, во втором столбце, равный 6.

Так как по своей сути массив представляет собой объект, его можно объявлять следующим образом:

```
var myArr = new Array();
```

Операторы и выражения

Выражения в языке JavaScript разделяются точкой с запятой; если выражения находятся на разных строках, то в конце выражения точка с запятой могут не ставиться.

Несколько выражений могут объединяться в один блок с помощью фигурных скобок **{ }**.

Оператор присваивания

Значение присваивается одной переменной или сразу нескольким переменным:

```
var n = j = k = 2;
```

Арифметические операторы

В языке JavaScript поддерживаются те же арифметические операторы, что и в языке Java**: «+», «–», «*», «/», «%», <<+=>>, << –=>>, ...**

При этом оператор сложения "**+**" при работе со строками означает конкатенацию последних, например

```
s = "str1" + "str2"
```

Операция инкремента «**++**», служит для прибавления 1 к операнду, соответственно декремент «--» – используется для вычитания 1 от операнда.

Операторы сравнения

В языке JavaScript поддерживаются следующие операторы сравнения:

«<», «<=», «>», «>=», «!=», «==» – равно; **«===»** – равно и операнды одинакового типа (строгое сравнение); **«!==»** – не равно или операнды разных типов.

Для иллюстрации оператора строгого сравнения приведем пример:

```
var x = 0;
var y = false;
alert(x==y);
```

//выдаст на экран true из-за автоматического приведения операндов к одному типу

```
alert(x===y);
```

//выдаст на экран false, так как операнды разных типов.

Логические операторы

В языке JavaScript поддерживаются следующие логические операторы:

«**&&**» – логическое И; «**||**» – логическое ИЛИ; «**!**» – логическое НЕ.

Оператор «?»

Оператор «**?**» возвращает значение первого выражения, если условие истинно, и второго выражения, если условие ложно при синтаксисе:

```
условие ? выражение1 : выражение2;
```

Пример использования:

```
var x = 6; var y = 9;
var res = x < y ? "x меньше y" : "x больше или равно y";
alert(res);
```

//выведет на экран фразу «x меньше y»

Оператор typeof()

Оператор **typeof()** возвращает строковое значение, которое определяет тип операнда: "**number**", "**string**", "**boolean**", "**object**", "**function**" и "**undefined**".

Условный оператор if

Блок имеет следующий вид:

```
if (условие)
    выражение или блок выражений
else
    выражение или блок выражений
```

Ветка **else** может отсутствовать.

Операторы организации циклов

Ниже приведен синтаксис для организации циклов:

```
while (условие)
    выражение или блок выражений
do
    выражение или блок выражений
while (условие);
```

Цикл **for** имеет синтаксис:

```
for (начальные значения; условие; изменение начальных переменных)
    выражение или блок выражений
```

Пример цикла **for**:

```
<script type="text/javascript">
for (var i = 0, j = 7; i < j; i++, j--) {
    alert(i + ' ' + j);
}
```

//на экран последовательно будут выведены пары чисел: 0 и 7, 1 и 6, 2 и 5, 3 и 4

```
</script>
```

Оператор цикла **for**…**in** служит для перебора всех свойств в объекте:

```
for (имя_переменной in объект)
    выражение или блок выражений
```

Примером может служить перебор всех стилей какого-либо элемента:

```
<p id="myP">test</p>
<script type="text/javascript">
var objProp;
for (objProp in document.getElementById("myP").style) {
    alert(objProp + ' = ' + document.getElementById("myP").style[objProp]);
}
```

```
</script>
```

Цикл выведет все свойства объекта **style** элемента параграфа.

Во всех циклах могут применяться операторы **continue** и **break**, первый служит для перехода к следующей итерации в цикле, второй – для выхода из цикла.

Оператор with

С помощью оператора **with** можно обращаться к свойствам объекта в сокращенном виде:

```
with (object) {
    выражения
}
```

Пример оператора **with**:

```
<p id="myP">test</p>
<script type="text/javascript">
with (document.getElementById("myP").style) {
    color = "red";
    fontSize = "20px";
    fontFamily = "Arial";
    letterSpacing = "5px";
}
</script>
```

Оператор switch

```
switch (переменная) {
    case условие1: выражение
        break;
    case условие2: выражение
        break;
    ...
    case условиеN: выражение
        break;
    default: выражение
}
```

Ветка **default** выполняется, если ни одно из предыдущих условий не выполняется и может отсутствовать.

Метод eval()

Метод выполняет JavaScript код, переданный ему строкой в качестве параметра.

Пример реализации простейшего калькулятора приведен ниже:

```
<body>
<input type="text" id="calc" />
<input type="button" value="Calculate!"
onclick=
"alert(eval(document.getElementById('calc').value))" />
</body>
```

Функции

В языке JavaScript описание функций имеет следующий синтаксис:

```
function имя_функции(список_параметров) {
    список выражений;
    return (значение)
}
```

Параметры в списке разделяются запятыми и могут отсутствовать. Например:

```
function hello() {
    alert("Hello, World!")
}
hello();
```

//выведет на экран фразу «Hello, World!»

Если в описании функции определено несколько параметров, а при вызове эти параметры не передаются, то JavaScript автоматически присваивает неопределенным параметрам значение **undefined**.

Для выхода из функции и возврата значения в вызвавшее эту функцию выражение используется оператор **return**.

```
function sumIt(arg1, arg2, arg3) {
    var res = arg1 + arg2;
    if (arg3) res = res + arg3;
    return res;
}
var x = 1; var y = 2; var z = 3;
alert(sumIt(x,y,z));
```

//выведет на экран 6

```
alert(sumIt(x,y));
```

//выведет на экран 3, прибавление третьего параметра к результату не производится, так как проверяется его наличие. Если такой проверки нет, то возникнет ошибка

Передача параметров по значению и по ссылке

Базовые типы данных (числа, строки, булевы переменные) передаются по значению. Это значит, что значение переменной, переданной как параметр в функцию, не изменится во время выполнения функции. Для всех остальных типов данных параметры передаются по ссылке и могут изменяться во время выполнения функции. Пример, иллюстрирующий такое поведение, приведен ниже:

```
function passByVal(arg1) {
   var arg1 = 10;
}
var x = 5;
passByVal(5);
alert(x);
```

//выведет на экран число 5, так как параметр передавался по значению

```
function passByRef(arr1) {
   arr1[0] = "NEW";
}
var y = ["first", "second"];
passByRef(y);
alert(y);
```

//выведет на экран «NEW, second», так как параметр передавался по ссылке, и значение первого элемента массива изменилось в основной части скрипта

Глобальные и локальные переменные

В языке JavaScript существует два набора переменных: глобальные, которые видны во всем документе во время выполнения скрипта, и локальные, которые объявляются внутри функций. Глобальные переменные также видны внутри функций. Глобальные переменные могут объявляться внутри функций, при этом перед именем переменной ключевое слово var не ставится.

Все глобальные переменные принадлежат объекту **window.** Если переменная не определена и к ней обращаться напрямую, это вызовет ошибку выполнения скрипта, а если к переменной обращаться как к свойству объекта **window**, то будет возвращено значение **undefined**.

```
function myFunc() {
   var x = "local";
   y = "global";
}
myFunc();
alert(window.x);
//вернет undefined
alert(y);
//вернет слово global
alert(x);
//вызовет ошибку выполнения скрипта
```

Объектная модель

В JavaScript используются следующие виды объектов:

пользовательские объекты – это объекты, которые создаём мы сами с помощью конструктора объекта Object;

встроенные объекты языка JavaScript – **String** – строка текста; **Array** – массив; **Date** – дата и время; **Math** – математические функции; **Object** – содержит конструктор для создания пользовательских объектов;

объекты браузера – создаются автоматически при загрузке документа в браузер:

window – объект верхнего уровня в иерархии объектов браузера;

document – содержит свойства, которые относятся к текущему HTML-документу;

location – содержит свойства, описывающие местонахождение текущего документа, например адрес URL;

navigator – содержит информацию о версии браузера;

history – содержит информацию обо всех ресурсах, к которым пользователь обращался во время текущего сеанса;

объекты, связанные с тегами HTML и стилями CSS – в JavaScript большинству тэгов HTML и стилей CSS соответствуют свойства объекта document, которые сами также являются объектами;

Пользовательские объекты

Для создания объекта используется конструктор, который определяет вид объекта и его поведение. Конструктор описывает шаблон вида реализованного объекта. Конструктор объекта может иметь вид:

```
function Book() {
}
```

Создание экземпляра объекта типа **book** будет иметь вид:

```
var myBook = new Book();
```

Конструктор может обращаться к создаваемому объекту, используя ключевое слово **this**. Таким образом можно добавить свойство создаваемому объекту:

```
function Book() {
    this.paper = true;
}
var myBook = new Book();
alert(myBook.paper);
```

//выведет на экран true, т.к. все объекты типа Book будут иметь свойство paper со значением true

В конструктор можно передавать параметры, чтобы задать начальные свойства создаваемого объекта.

```
function Book(isPaper) {
    if (isPaper) this.paper = true;
        else this.paper = false;
}
var myBook = new Book(false);
alert(myBook.paper);
```

//выведет на экран false

Прототипы

Каждый объект имеет свойство **prototype**, которое определяет его структуру. Пример использования прототипирования объектов:

```
Book.prototype.paper = false;
Book.prototype.isPaperBook = function() {
    if (this.paper) alert("This is a paper book");
        else alert("This is not a paper book");
}
function Book(isPaper) {
    if (isPaper) this.paper = true;
}
var myBook = new Book(true);
myBook.isPaperBook();
```

//выведет на экран фразу «This is a paper book»

В этом примере показано создание метода объекта и свойства, причем в прототипе описано, что свойство paper имеет значение **false**. Однако в конструкторе это свойство переопределяется, если при создании объекта конструктору передается параметр со значением **true**.

Хеш-таблицы в JavaScript

В языке JavaScript можно реализовать хеш-таблицу (то есть таблицу пар ключ-значение) несколькими способами, которые описаны ниже.

Объекты позволяют хранить и получать значения по ключам:

```
var h = new Object();
```

```
h.property = 10;
alert(h.property);
```

Следующий код будет выполнен, как ожидалось, и выведет на экран число **110**:

```
var h = new Object();
h.property = 10;
h[0] = 100;
alert(h.property + h[0]);
```

Следует заметить, что свойства не конфликтуют между собой, то есть `h.property` и `h[0]` – два разных свойства объекта.

В квадратные скобки можно заключать не только числа:

```
var h = new Object();
h.property = 10;
var s = 'ty';
alert(h['proper' + s]);
```

То есть `h['property']` то же самое, что и `h.property`.

Удаление свойств осуществляется с помощью оператора **`delete`**:

```
var h = new Object();
h['property'] = 10;
alert(delete h['property']);
```
//выведет на экран true
```
alert(h['property']);
```
//выведет на экран undefined

Последний пример – перебор всех свойств объекта:

```
var h = new Object();
...
var props = '';
for(var i in h)
      props += i + ' : ' + h[i] + '\n';
alert(props);
```

Следует заметить, что переменная **`i`** внутри цикла содержит имя свойства, а не его значение.

Встроенные объекты Array, Date, Math

Объект Array

Массив является объектом и имеет ряд свойств и методов. Единственным свойством массива является **`length`**, которое показывает количество элементов, содержащихся в массиве.

Ниже описаны основные методы массивов:

`concat``([item1 : { Object | Array } [, ... [, itemN : { Object | Array }]]]]) : Array` – возвращает массив, состоящий из исходного массива и любого количества новых элементов;

`join``(separator : String) : String` – возвращает строку, содержащую все элементы массива, разделенные строкой;

`pop``() : Object` – удаляет последний элемент массива и возвращает его;

push([item1 : Object [, ... [, itemN : Object]]]) : Number – добавляет в конец массива новые элементы; если один из аргументов – массив, он добавляется как один элемент;

shift() : Object – удаляет первый элемент массива и возвращает его;

sort(sortFunction : Function) : Array – возвращает отсортированный массив;

unshift([item1 : Object [, ... [, itemN : Object]]]) : Array – добавляет один или больше новых элементов в начало массива.

Объект Date

Объект **Date** предоставляет набор методов для работы с датой и временем.

Текущее время в объекте Date берется из операционной системы. Дата и время в объекте **Date** хранятся в виде числа, показывающего количество миллисекунд, прошедшее с 1 января 1970 года. Также следует помнить, что нумерация месяцев и дней недели начинается также с нуля, нулю соответствует воскресенье.

У объекта **Date** два конструктора с разными наборами параметров:

function Date([dateVal : { Number | String }]) – если тип параметра – число, оно воспринимается как количество миллисекунд с 1 января 1970 года; если строка, то она преобразуется в соответствии с шаблоном "month dd, yyyy hh:mm:ss";

function Date(year : int, month : int, date : int[, hours : int [, minutes : int [, seconds : int [, ms : int]]]]) – в качестве параметров передается год, месяц и день, в качестве необязательных параметров может передаваться время.

Если при создании экземпляра объекта **Date** никаких параметров не передается, то этому экземпляру присваивается текущая дата.

Приведем некоторые методы объекта **Date**. Они начинаются с префикса **set** и устанавливают определенные значения. Существует также аналогичный набор методов с префиксом **get**, который возвращает атрибуты объекта **Date**.

setDate(numDate : Number) – устанавливает день в месяце, если устанавливаемое значение больше количества дней в месяце, то устанавливается следующий месяц и день, равный **numDate** минус количество дней в текущем месяце. Например, если текущая дата 10 января 2008 года и в **setDate()** передается число 33, то дата станет 2 февраля 2008 года. Если в **setDate()** передается отрицательное значение, то поведение в этом случае аналогично;

setFullYear(numYear : Number [, numMonth Number [, numDate Number]]) – устанавливает год, в качестве необязательных параметров можно указать месяц и день;

setHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli : Number]]]) – устанавливает часы, необязательными параметрами являются минуты, секунды и миллисекунды;

setMilliseconds(numMilli : Number) – устанавливает миллисекунды;

setMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number]]) – устанавливает минуты, необязательными параметрами являются секунды и миллисекунды;

setMonth(numMonth : Number [, dateVal : Number]) – устанаваливает месяц, необязательный параметр – день от начала месяца;

setSeconds(numSeconds : Number [, numMilli : Number]) – устанавливает секунды, необязательный параметр – миллисекунды;

setTime(milliseconds : Number) – устанавливает время в миллисекундах, прошедших с 1 января 1970 года.

Важное значение имеет метод parse, который преобразует дату в строку по заданному шаблону. Для более подробной информации по преобразованию даты в строку смотрите MSDN Library.

```
<!-- пример # 3: часы. Использование методов объекта Date -->
<html> <head>
<title>Clock</title>
<script type="text/JavaScript">
function clockform()
         {d=new Date();

time=d.getHours()+":"+d.getMinutes()+":"+d.getSeconds();
       document.formcl.fclock.value=time;
            setTimeout("clockform()",100); }
</script>
</head>
<body  onLoad="clockform()">
<center>
<form name=formcl metod="get">
<input name=fclock maxlength=8 size=8>
</form> </center>
</body></html>
```

Объект Math

Объект **Math** содержит в себе набор констант и методов, расширяющих базовые арифметические операции.

Невозможно создать экземпляр объекта **Math**, так как он является статическим и создается автоматически во время выполнения скрипта.

Приведем пример возведения числа в степень:

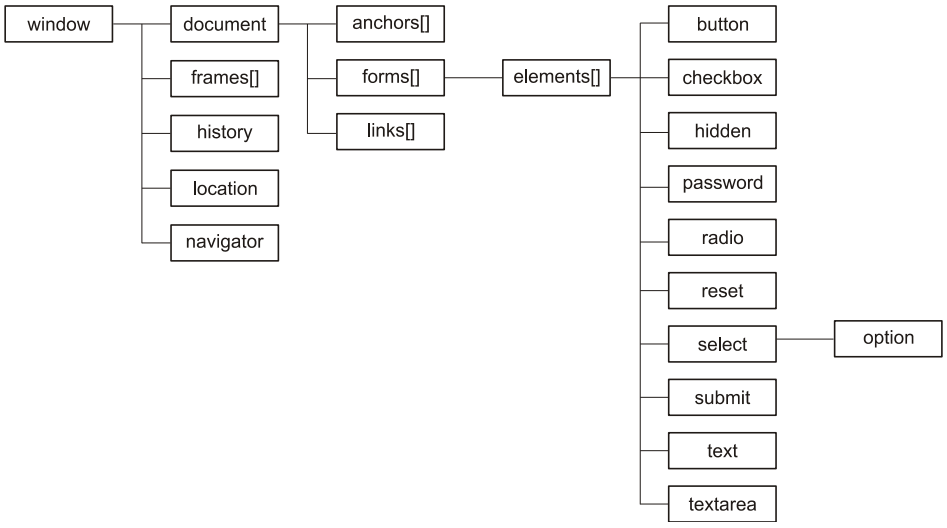
```
alert(Math.pow(10,3));
//выведет 1000
<!--пример # 4: функция вычисления площади круга  -->
<html>
<head>
<title>Функция вычисления площади круга </title>
</head>
<body>
<script type="text/javascript">
  function sq(r) {
    document.write("Задали r= ", r ,
                    " для функции.","<BR>")
    return Math.PI*r * r;
  }
```

```
  var r=2;
  document.write("Площадь круга равна ",sq(r),".")
</script>
 </body> </html>
```

Объекты window и document

Объектная модель JavaScript предоставляет возможность работы с объектами, зависящими от браузера (**window**, **navigator**, **location** и т.д.) и объектами, относящимися к HTML-документу (**document**, **forms**, **frames** и т.д.);

Ниже представлена схема документа, которая позволяет манипулировать свойствами и структурой документа.



Объект **window** является объектом верхнего уровня в иерархии JavaScript. Ссылки **self** и **window** являются синонимами для текущего окна. Вы можете закрыть текущее окно, используя **window.close()** или **self.close().** Ссылка **top** указывает на самое верхнее окно, а **parent** ссылается на окно, содержащее **frameset.** Когда вы открываете или закрываете окно внутри события, необходимо определить **window.open()** или **window.close()** вместо **open()** или **close()**, так как вызов **close()**

В следующем примере рассмотрим использование метода **window.open()** для открытия минимизированного окна.

```
<!-- пример # 5: минимизация окна и его удаление-->
<html> <head>
<title> Минимизация окна </title>
<script type="text/JavaScript">
   function makeicon()
                {
window.open("pr1.htm","icon", //открытие окна
"resizable=yes,scrollbars=yes,width=50,height=70");
window.close(); //закрытие старого окна
```

```
                        }
      </script>
      </head><body>
      <h1>minimize page</h1>
   <form name=formicon>
   <input name=ibutton type=button value=mini on-
Click="makeicon()">
   </form>
   </body></html>
```

Метод **open()** открывает новое окно web-браузера. Синтаксис:

```
[windowVar=][window].open("URL", "winName", "windowFea-
tures"])
```

windowVar – имя нового окна; URL определяет URL, открываемый в новом окне; **winName** – имя окна; **windowFeatures** – список через запятую любых из следующих опций или значений:

toolbar[=yes | no] – создает стандартные рабочие инструменты с такими кнопками, как "Back" и "Forward";

location[=yes | no] – создает поле ввода Location;

directories status[=yes | no] – создает строку состояния внизу окна;

menubar[=yes | no] – создает меню вверху окна;

scrollbars[=yes | no] – создает горизонтальную и вертикальную прокрутки, когда документ больше, чем размер окна;

resizable[=yes | no] – позволяет пользователю изменять размер окна;

width=pixels,**height**=pixels – размеры окна в пикселях.

Опции разделяются запятой. Не делайте пробелов между опциями!

Следующие объекты являются наследниками объекта **window**: **document**, **frame**, **location**.

Методы: **alert()**, **confirm()**, **prompt()**, **open()**, **close()**, **setTimeout()**, **ClearTimeout()**. События: **onLoad**, **onUnload**.

В систему введено свойство **opener**, которое определено для текущего окна или фрейма, а методы **blur** и **focus** распространены на работу с окнами. Свойство **opener** определяет окно документа, который вызвал открытие окна текущего документа. Свойство определено для любого окна и фрейма. Если нужно выполнить некоторые действия по отношению к окну, открывшему данное окно, то можно использовать выражение типа: `window.opener.[method]`. Например, если требуется закрыть окно-предшественник, то можно выполнить метод **close()**: `window.opener.close()`

Можно менять и другие свойства объектов в окне-предшественнике. Например, для окна-предшественника определить голубой цвет в качестве цвета фона: `window.opener.document.bgColor ='cyan'`

Следующий пример переназначает для текущего окна окно-предшественник:

```
window.opener= new_window
```

Объект Document

Объект **Document** содержит ряд свойств и методов, позволяющих изменять сам документ. В следующем примере используется массив **all**, содержащий все элементы документа для вывода списка используемых на странице тегов.

<!--пример # 6: список используемых на странице тегов -->

```
<html> <head>
<title> Список используемых на странице тегов </title>
<script type="text/javascript">
function findtags()
{var tag;
var tags="страница содержит следующие теги";
for(i=0;i<document.all.length;i++)
{tag=document.all(i).tagName;
tags=tags+"\r"+tag;}
alert(tags);
return tags;}
</script>
</head>
<body onload="findtags()">
<h2 onMouseover="findtags()">Вывод списка используемых на
странице тегов в окно предупреждений </h2>
 </body> </html>
```

<!-- пример # 7: изменение фона документа при выборе кнопки - цвета с помощью свойства bgColor объекта Document -->

```
<html> <head>
<title>bgcolor</title>
</head>
<body text=000000 bgcolor=ffffff>
<table border=0 align=center>
<tr><td><form>
<input type=button value="красный"
   onClick= "document.bgColor ='ff0000'" >
<input type=button value="желтый"
    onClick="document.bgColor ='ffff00'">
<input type=button value="синий"
   onClick="document.bgColor= '0000ff'">
<input type=button value="голубой"
   onClick="document.bgColor= '87ceeb'">
</form></td>
</table>
</body> </html>
```

Document Object Model (DOM)

Структура любого HTML-документа представляет собой дерево, в корне которого расположен тег (точнее сказать, узел) HTML. Дочерними узлами HTML являются узлы **HEAD** и **BODY**, у которых, в свою очередь, есть собственные дочерние узлы.

В структуре дерева могут существовать узлы разных типов, они представлены в таблице:

Описание	Пример
Определяет тип HTML документа	`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">`
Верхний элемент в структуре HTML-дерева	`<html>`
HTML комментарий	`<!-- this is comment -->`
HTML элемент (тег)	`<p> … </p>`
Атрибут HTML элемента	`bgcolor="red"`
Текст, содержащийся внутри HTML элемента	`Content of paragraph`

Каждый HTML-тег может иметь атрибут **id**, который позволяет быстро получить доступ к этому элементу с помощью метода **getElementById** объекта **document**. Приведем пример кода внутри тега **BODY**:

```
<p id="myP">Hello, World!</p>
<script type="text/javascript">
alert(document.getElementById("myP").innerHTML);
//выведет на экран фразу Hello World!
var str = "";
str += document.getElementById("myP").nodeName + '\n';
str += document.getElementById("myP").nodeValue + '\n';
str += document.getElementById("myP").nodeType + '\n';
alert(str);
//выведет на экран «P null 1»
</script>
```

На экран вывелось *nodeValue*, равное *null*, потому что на самом деле внутри узла **<p>** есть еще один текстовый узел, в котором содержится искомый текст «`Hello, World!`». Для того чтобы получить доступ к этому значению, используется следующая строка:

```
document.getElementById("myP").childNodes[0].nodeValue;
```

Каждый узел DOM имеет ряд свойств:

nodeName – имя узла;

nodeValue – значение узла;

nodeType – номер, соответствующий типу узла;

parentNode – ссылка на родительский узел, если он существует;

childNodes – список дочерних узлов;

firstChild – первый дочерний элемент;

lastChild – последний дочерний элемент;

previousSibling – указывает на предыдущий соседний узел, если у родительского узла несколько дочерних и текущий узел не первый дочерний;

nextSibling – указывает на следующий соседний узел, если у родительского узла несколько дочерних и текущий узел не последний дочерний;

attributes – список атрибутов;

ownerDocument – указатель на объект **document**, которому принадлежит текущий узел.

Чтобы проиллюстрировать иерархию узлов DOM, приведем пример:

```
<!-- пример # 8: иерархия узлов DOM-->
<table>
<tr id="firstRow">
   <td id="firstCell"></td>
   <td id="currentNode" width="10">
<span id="spanNode">
text</span>
<p id="pNode">text</p>
   </td>
   <td id="lastCell"></td>
</tr>
</table>
<script type="text/javascript">
alert(document.getElementById("currentNode").parentNode.id);
   //выведет на экран firstRow
alert(document.getElementById("currentNode").childNodes[0].id);
   //выведет на экран spanNode
alert(document.getElementById("currentNode").firstChild.id);
   //выведет на экран spanNode
alert(document.getElementById("currentNode").lastChild.id);
   //выведет на экран pNode
alert(document.getElementById("currentNode").previousSibling.id);
   //выведет на экран firstCell
alert(document.getElementById("currentNode").nextSibling.id);
   //выведет на экран lastCell
alert(document.getElementById("currentNode")
                        .attributes["width"].value);
   //выведет на экран 10
alert(document.getElementById("currentNode")
                        .ownerDocument.nodeName);
   //выведет на экран #document
</script>
```

Помимо метода **getElementById()**, существует несколько других, облегчающих доступ к необходимым элементам документа. Метод **getElementsByName()** возвращает коллекцию элементов с определенным атрибутом name, а метод **getElementsByTagName()** возвращает коллекцию элементов (тегов) с одинаковым именем. Оба метода принадлежат объекту **document**.

Создание новых узлов

DOM поддерживает следующие методы, связанные с созданием новых узлов:

createAttribute(name) – создает атрибут с именем, переданным в параметре;

createComment(string) – создает HTML-комментарий в виде `<!--string -->`, текст комментария передается в параметре;

createDocumentFragment() – создает новый документ для хранения новых узлов;

createElement(tagName) – создает узел (тег) с именем, переданным в параметре;

createTextNode(string) – создает текстовый узел с содержанием, переданным в параметре.

Все методы принадлежат объекту **document**.

Добавление новых узлов в документ

Для добавления новых узлов в текущий документ используются следующие методы:

```
insertBefore(newChild, referenceChild);
appendChild(newChild).
```

Оба эти метода добавляют новый узел **newChild** к существующему в документе, **appendChild()** добавляет новый узел после всех дочерних, **insertBefore()** добавляет новый узел перед дочерним узлом, указанным в параметре **referenceChild.**

Кроме этого, существует метод для копирования существующего узла. В качестве параметра можно указать, должны ли копироваться все дочерние узлы, по умолчанию значение параметра **false**:

```
cloneNode(deepClone).
```

Удаление и замена узлов в документе

Для удаления узла используются методы:

`currentNode.`**`removeChild`**`(child)` – в качестве параметра принимает узел для удаления.

`currentNode.`**`replaceChild`**`(newChild, oldChild)` – заменяет узел `oldChild` на `newChild`.

Оба метода должны принадлежать узлу **currentNode**, у которого удаляются или заменяются дочерние узлы.

Использование каскадных таблиц стилей в DOM

Каждый узел DOM имеет объект **style**, который описывает применяемые стили. Например, можно изменить цвет шрифта тега **<p>**, `document.getElementById("samplePtag").style.color = «red».`

Более подробно обо всех свойствах можно узнать в спецификации CSS (http://www.w3.org/Style/CSS/#specs).

Свойство элемента innerHTML и outerHTML

Кроме методов, описанных выше, для изменения структуры документа используется более простой метод, основанный на использовании свойств элементов DOM – **innerHTML** и **outerHTML**. **innerHTML** содержит HTML-код между открывающим и закрывающим тегом. С помощью этого свойства можно работать с кодом внутри тега, как со строкой – считывать и записывать. Однако для следующей группы элементов это свойство доступно только для чтения: **COL**, **COLGROUP**, **FRAMESET**, **HTML**, **STYLE**, **TABLE**, **TBODY**, **TFOOT**, **THEAD**, **TITLE**, **TR**.

По определению свойство **innerHTML** не существует у элементов, которые не имеют одновременно открывающего и закрывающего тега (например **
**).

Отличие свойства **outerHTML** в том, что это свойство включает в себя HTML-код между открывающим и закрывающим тегом, а также открывающий и закрывающий тег этого элемента.

Для следующих элементов это свойство доступно только при чтении: **CAPTION**, **COL**, **COLGROUP**, **FRAMESET**, **HTML**, **TBODY**, **TD**, **TFOOT**, **TH**, **THEAD**, **TR**.

Свойство **outerHTML** доступно для записи только после того, как весь документ будет загружен, т.е. произойдет событие **window.onload**.

Ниже приведен пример использования свойств **innerHTML** и **outerHTML**:

```
<!-- пример # 9: использование свойств -->
<html><head>
<script type="text/javascript">
function transformBody() {
   var myPar = document.getElementById("myP");
   myP.innerHTML = "<i>Hello World!</i>";
   myP.outerHTML = "<strong>" + myP.innerHTML
                                        + "</strong>";
}
</script>
</head>
<body onload="transformBody();">
<p id="myP">sample text</p>
</body>
</html>
<!-- после выполнения функции структура элемента body будет:
<BODY><STRONG><I>Hello World!</I></STRONG></BODY>
 -->
```

Свойства **innerHTML** и **outerHTML** не являются официально поддерживаемыми стандартами, однако они поддерживаются всеми современными браузерами. Свойством **outerHTML** следует пользоваться с большой осторожностью, так как оно поддерживается в меньшем количестве браузеров (например, оно не работает в браузере Firefox).

Event Model

Event Model, или модель событий – это способность языка JavaScript реагировать на изменение состояния документа в браузере, например нажатие на ссылку или отправка заполненной формы.

Ниже приведены основные события, которые поддерживаются всеми современными браузерами:

onblur – происходит, когда пользователь убирает фокус с элемента, например элемента формы;

onchange – происходит, когда элемент формы теряет фокус, и его значение при этом изменилось;

onclick – происходит, когда пользователь нажимает на любой визуальный элемент;

ondblclick – аналогично предыдущему, но при двойном нажатии;

onfocus – происходит, когда элемент получает фокус, то есть пользователь выбирает этот элемент;

onkeydown – происходит, когда пользователь нажимает клавишу на клавиатуре;

onkeypress – происходит, когда пользователь нажимает и отпускает клавишу на клавиатуре;

onkeyup – происходит, когда пользователь отпускает нажатую клавишу;

onload – происходит, когда документ (или фрейм) загружен;

onmousedown – происходит, когда пользователь нажимает клавишу мыши;

onmousemove – происходит, когда пользователь двигает курсором мыши над элементом;

onmouseout – происходит, когда указатель мыши покидает область элемента;

onmouseover – происходит, когда указатель мыши попадает в область элемента;

onmouseup – происходит, когда пользователь отпускает нажатую клавишу мыши;

onreset – происходит, когда значения элементов формы сбрасываются;

onselect – происходит, когда пользователь выделяет текст в элементе формы;

onsubmit – происходит, когда пользователь отсылает форму;

onunload – происходит, когда пользователь покидает текущий документ, то есть закрывает браузер или переходит к другой странице.

События в HTML прописываются следующим способом:

```
<a href=”sample.html”
onclick=”alert(‘Links clicked’)”>click me</a>
```

```
<!-- пример # 10: получение типа броузера -->
<html> <head>
<title>Test of Browser name</title>
</head>
<body>
<h1 align=center>Проверка  типа браузера</h1>
<hr>
```

Чтобы получить имя вашей программы просмотра, следует выбрать кнопку "Browser"

```
<P><hr><form name=fr>
<input type=button name=browser value=Browser
 onClick="window.alert(window.navigator.appName)">
</form>
</body> </html>
```

Динамическое назначение событий

Кроме объявления событий в HTML, JavaScript позволяет назначать события во время выполнения скрипта:

```
<a href="sample.html" id="lnk1">click me</a>
<script type="text/javascript">
document.getElementById("lnk1").
onclick = function(){alert('Links clicked')};
</script>
```

Ключевое слово this

Зачастую обработчику события необходимо передать элемент, который это событие вызвал. Пример приведен ниже:

```
<a href="sample.html" onclick="showInfo(this);">
click me</a>
< type="text/javascript">
function showInfo(_obj) {
    alert(_obj.innerHTML);
  //при нажатии на ссылку выводится  на экран «click me»
}
</script>
```

По умолчанию после нажатия на ссылку происходит переход на другую страницу, однако это действие можно отменить, если обработчик события вернет значение **`false`**. Например:

```
<a href="sample.html"
onclick="return showInfo(this);">click me</a>
<script type="text/javascript">
function showInfo(_obj) {
    return confirm("Do you want go to another page?");
//при нажатии на ссылку будет выведен стандартный диалог с кнопками OK и
Cancel; если будет нажата Cancel, то браузер не перейдет по адресу, на ко-
торый указывает ссылка
}
</script>
```

Примеры на JavaScript:

```
<!-- пример # 11: открытие документа в новом окне -->
<html>
<head>
<script type="text/javascript">
function openStaticWin() {
    window.open("test.html", "_blank",
 "height=200,width=400,status=yes,toolbar=no,menubar=no,
   location=no");
}
function openDynamicWin() {
    var newWin = window.open();
    newWin.document.open();
    newWin.document.write("<html><head></head><body>"
         + new Date() + "</body></html>");
    newWin.document.close();
}
</script>
</head>
<body>
<a href="#" onclick="openStaticWin();
   return false;">open existing html</a>
```

```
<br>
<a href="#" onclick="openDynamicWin();
   return false;">open dynamic html</a>
</body>
</html>
```

Первая функция открывает в новом окне существующий документ, вторая задает HTML-код нового документа динамически.

Следующий пример:

```
<!-- пример # 12: создание динамического меню после загрузки страницы -->
<html><head>
<style>
.menuContainer {
   border: 1px solid #123456;
}
.menuContainer .menuItem {
   float: left;
   margin: 2px;
   padding: 10px;
}
.menuContainer .menuOver {
   background-color: #808080;
}
.menuContainer .menuOver a {
   color: #800000;
   font-weight: bold;
}
.menuContainer .clear {
   clear: left;
}
</style>
<script type="text/javascript">
function processMenu() {
   var allDiv = document.getElementsByTagName("DIV");
   for (var i = 0; i < allDiv.length; i++) {
         if (allDiv[i].className == "menuContainer")
processMenuItem(allDiv[i]);
   }
}
function processMenuItem(_obj) {
   var allChilds = _obj.childNodes;
   for (var i = 0; i < allChilds.length; i++) {
         if (allChilds[i].className == "menuItem") {
               allChilds[i].onmouseover = function() {
                     this.className += " menuOver";
               }
               allChilds[i].onmouseout = function() {
                     this.className =
```

```
   this.className.replace(" menuOver","");
                  }
            }
      }
   }
   </script>
   </head>
   <body onload="processMenu();">
   <div class="menuContainer">
      <div class="menuItem"><a href="#">first</a></div>
      <div class="menuItem"><a href="#">second</a></div>
      <div class="menuItem"><a href="#">third</a></div>
      <div class="menuItem"><a href="#">fourth</a></div>
      <div class="clear"></div>
   </div>
   </body></html>
   <!-- пример # 13: валидация формы -->
   <script type="text/javascript">
   function submitForm(_form) {debugger;
      markErrorField(false);
      var errMess = "";
      if (!_form.elements["login"].value) {
            errMess += "Your Login is empty.\n";
            markErrorField(_form.elements["login"]);
      }
      if (!_form.elements["mail"].value) {
            errMess += "Your Email is empty\n";
            markErrorField(_form.elements["mail"]);
      }
      if (!_form.elements["pass"].value) {
            errMess += "Your Password is empty.\n";
            markErrorField(_form.elements["pass"]);
      } else if (!_form.elements["confpass"].value) {
            errMess +=
   "Your Password confirmation is empty.\n";
            markErrorField(_form.elements["confpass"]);
      } else if
   (_form.elements["pass"].value !=
                  _form.elements["confpass"].value) {
   errMess +=
"Your Password confirmation does not equal to your Pass-
word.\n";
            markErrorField(_form.elements["pass"]);
            markErrorField(_form.elements["confpass"]);
      }
      if (errMess) {
            alert(errMess + "");
            return false;
      }
```

```
}
function markErrorField(_element) {
   var allLabels =
          document.getElementsByTagName("LABEL");
   if (_element) {
         for (var i = 0; i < allLabels.length; i++) {
               if (allLabels[i].htmlFor == _element.id)
 allLabels[i].style.color = "red"; }
   } else {
         for (var i = 0; i < allLabels.length; i++) {
               allLabels[i].style.color = "black";
         }
   }
}
</script>
<form onsubmit="return submitForm(this);">
<table>
<tr>
<td><label for="login">Your Login</label></td>
<td><input type="text" name="login" id="login" /></td>
</tr>
<tr>
<td><label for="mail">Your Email</label></td>
<td><input type="text" name="mail" id="mail" /></td>
</tr>
<tr>
<td><label for="pass">Your Password</label></td>
<td><input type="password" name="pass" id="pass" /></td>
</tr>
<tr>
<td><label for="confpass">
Confirm Your Password</label></td>
<td><input type="password" name="confpass"
          id="confpass" /></td>
</tr>
<tr>
<td colspan="2"><input type="submit"
                      name="Register" /></td>
</tr>
</table>
</form>
```

Приложение 3

UML

Создаваемое программное обеспечение (ПО) постоянно усложняется. При работе над любыми информационными (распределенными) системами в первую очередь возникает проблема взаимопонимания программиста и заказчика уже на стадии обсуждения структуры системы. До начала кодирования программы предлагаемая концепция предусматривает решение двух предварительных задач: проанализировать поставленную перед разработчиком задачу и разработать проект будущей системы. Программа разбивается на отдельные модули, взаимодействующие между собой с помощью механизмов передачи параметров.

Унифицированный язык моделирования (Universal Model Language) − графический язык визуализации, специфицирования, конструирования и документирования программного обеспечения. С помощью UML можно разработать детальный план создаваемой системы, отображающий системные функции и бизнес-процессы, а также конкретные особенности реализации. А именно:

- классы, написанные на специальных языках программирования;
- схемы БД;
- программные компоненты многократного использования.

Поскольку UML интенсивно изменяется, то здесь не ставится цель дать все детали и аспекты UML. В настоящее время существуют две крайние точки зрения на применение UML и моделирвование как таковое:

1. Необходимо замоделировать всё.
2. Моделирование – пустая трата времени: заказчик, как правило, не оплачивает эту работу, поэтому надо сразу писать код.

Истина, как всегда, где-то рядом. Здравый смысл никто не отменял, поэтому– если есть необходимость что-то замоделировать, это следует сделать, если нет – не следует.

В текущей жизни человек всегда сталкивается с UML. Даже пытаясь набросать чертёж полочки для книг – это уже UML. Таким образом, графические наброски проекта всегда рядом с человеком. Для промышленного программирования, выполняемого большими и зачастую распределёнными интернациональными командами, наибольшая ценность UML – в возможности эффективного взаимодействия и в облегчении понимания. Хорошая диаграмма может помочь донести идею проекта, модуля, особенно если нужно избежать большого количества деталей, которые только усложнят понимание. Диаграммы более наглядно представляют систему или процесс, что облегчит понимание. Особенно важно отметить, что UML широко используется в пределах объектно-ориентированного (ОО) сообщества, хорошо стандартизован и является доминирующей графической нотацией. Так же UML популярен и вне пределов ИТ индустрии, поскольку он нейтрален к технологии, предметной области и т.д.

Не являясь заменой языков программирования, диаграммы – полезный помощник программиста. Хотя многие полагают, что в будущем графические методы будут доминировать на рынке программного обеспечения, следует скептиче-

ски относиться к этому. С трудом можно себе представить, хотя многие менеджеры мечтают об этом, что будет создан инструмент, в котором, нарисовав всё и нажав «волшебную кнопку», на выходе получишь готовое приложение.

Задача состоит в том, чтобы получить оценку того, в чём диаграммы могут быть полезны и в чём нет.

UML – семейство графических нотаций, базирующихся на единой метамодели, помогающий в описании и проектировании систем программного обеспечения, особенно систем программного обеспечения, использущих объектно-ориентированный стиль. Универсальность UML имеет побочный эффект – разные люди используют UML разными способами. Это ведет к долгим и бесполезным дискуссиям о том, как же он должен использоваться. Чтобы избежать этого, главное – добиться единого понимания использования UML в пределах группы разработчиков, где вы сейчас работаете. В другой группе может быть несколько другое понимание, но главное – чтобы все это понимали одинаково.

Основные пути использования UML

Эскизы

Главное преимущество рисования эскизов – их выборочность. Рисуя эскизы, можно набрасывать условно некоторые проблемы кода, который будет создан, обычно после обсуждения их с группой людей в команде. Главная цель состоит, с одной стороны в том, чтобы, используя эскизы, донести до своих коллег идеи и альтернативы того, что планируется сделать, с другой стороны – более полно понять проблемную область.

При этом нет смысла говорить о сотнях или тысячах строках кода, которые будут созданы или модифицированы. Концентрация осуществляется на ключевых, концептуальных проблемах. Визуализировав их, прежде чем начнется программирование, можно избежать многих часов бессмысленного кодирования.

Проектная модель

Идея заключается в том, что проект разработан проектировщиком или командой проектировщиков, которые должны сделать его настолько детально, что он будет понятен для программиста, который будет кодировать его. Такой проект должен быть настолько полон и детален, что гарантирует реализацию всех проектных решений программистами без дополнительного осмысления или обсуждения. Т.е. для программиста это выглядит как детальное руководство. В реальной ситуации правильный подход опытных дизайнеров – разработать на уровне модели проекта общие интерфейсы систем, оставив программистам возможность решать детали реализации этих систем.

UML как программный язык

Дизайнеры рисуют диаграммы UML, которые затем компилируются непосредственно в исполняемый код. Таким образом UML становится исходным кодом программы. Само создание инструментов, реализующих этот подход, требует немалых усилий.

Нотации и метамодель

Нотация – совокупность графических объектов, которые используются в моделях. В качестве примера на диаграмме показано, как в нотации диаграммы

класса определяются понятия и предметы типа «класс», «ассоциацция», «множественность» и т.д.

Нотация диаграммы классов определяет способ представления класса, ассоциации, множественности. Причем эти понятия должны быть точно определены.

Проектирование подразумевает всесторонний анализ всех ключевых вопросов разработки. И строгое определение всех понятий может не позволить описать реальные требования системы.

Большинство объектно-ориентированных методов является не слишком строгими. Их нотация прибегает в большей степени к интуиции, чем к формальному определению.

Метамодель – диаграмма, определяющая нотацию.

Метамодель помогает понять, что такое хорошо организованная, т.е. синтаксически правильная, модель.

Уровень владения и понимания языка моделирования зависит от задач, которые решаются с его помощью. В основном диаграммы используются как средства обмена информации между разработчиками.

Если не придерживаться согласованного понимания, то другие разработчики просто не поймут, что вы хотели выразить своей диаграммой.

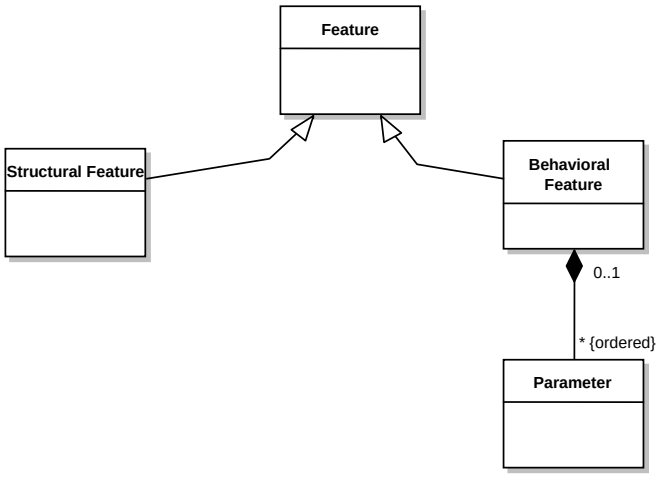


Рис. 1. Нотации и метамодель

- Activity – процедурное и параллельное поведение. Введено в UML 1;
- Class – классы, свойства и взаимоотношения. Введено в UML 1;
- Communication − взаимодействие между объектаими; акцент на связи. В UML 1 называлась Collaboration diagram;
- Component − структрура и связи компонентов. Введено в UML 1;
- Composite structure – декомпозиция класса во время выполнения. Новая в UML 2;
- Deployment − размещение артефактов. Введено в UML 1;
- Interaction overview – смешение Sequence и Activity. Новая в UML 2;
- Object – пример конфигурации экземпляров. Неофициальная в UML 1;

- Package − иерархическая структура во время компиляции. Неофициальная в UML 1;
- Sequence − взаимодействие между объектами. Акцент на последовательности. Введено в UML 1;
- State machine − способы изменения объекта различными событиями в течение его жизненного цикла. Введено в UML 1;
- Timing − взаимодействие между объектами. Акцент на распределении во времени. Новая в UML 2;
- Use case – способы взаимодействия пользователей с системой. Введено в UML 1.

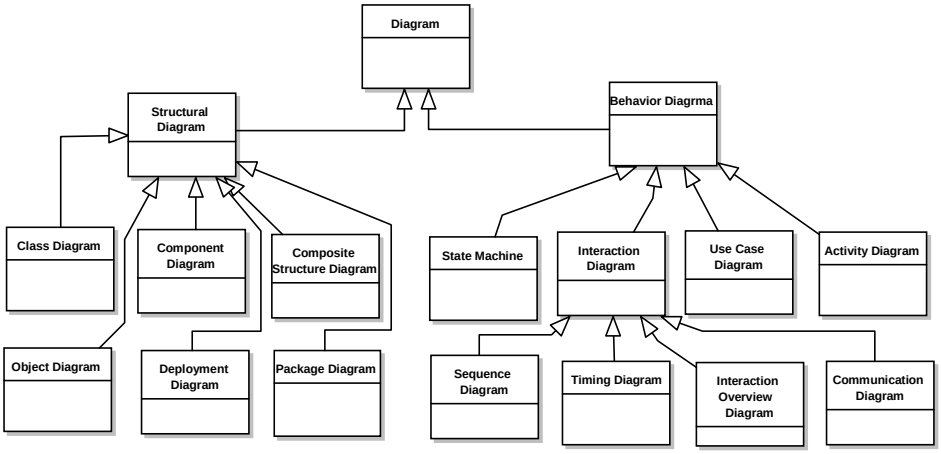


Рис. 2. Диаграммы UML

Основные понятия

К основным понятиям UML относятся:

- *Сущности* – абстракции, являющиеся основными элементами модели;
- *Отношения* – связывают различные сущности;
- *Диаграммы* – группируют представляющие интерес совокупности сущностей.

Сущности

- структурные – статические части модели, соответствующие концептуальным или физическим элементам модели;
- поведенческие – динамические составляющие, описывающие поведение модели во времени и в пространстве;
- группирующие;
- аннотационные.

Структурные сущности

Класс (Class) – описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Реализует несколько интерфейсов.

Интерфейс (Interface) – совокупность операций, которые определяют набор услуг, предоставляемых классом или компонентом. Описывает видимое извне поведение элементов.

Кооперация (Collaboration) – совокупность операций, которые производят некоторый общий эффект, не сводящийся к простой сумме слагаемых.

Вариант использования (Use case) – описание последовательности выполняемых системой действий, которая производит наблюдаемый результат, значимый для какого-либо определенного действующего лица (Actor).

Активный класс (Active class) – класс, объекты которого вовлечены в один или несколько процессов и могут инициировать управляющее воздействие.

Компонент (Component) – физическая заменяемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает его реализацию.

Узел (Node) – элемент реальной (физической) системы, который существует во время функционирования программного комплекса и представляет собой вычислительный ресурс.

Поведенческие сущности

Взаимодействие (Interaction) – поведение, суть которого заключается в обмене сообщениями между объектами в рамках конкретного контекста для достижения определенных целей.

Автомат (State machine) – поведение, определяющее последовательность состояний, через которые объект или взаимодействие проходят на протяжении своего жизненного цикла в ответ на различные события, а также реакция на эти события.

Правильный UML

Правильный UML – хорошо форматированный в соответствии со спецификацией UML. Однако всё находится в реальном мире и всё не так просто. Каким языком считать UML? Предписывающим, как языки программирования, или описательным? Сложившееся понимание в ИТ индустрии на данный момент – рассматривать UML как описательный язык. И в этом случае, всё, что считается правильным в разрабатываемом конкретной группой программистов проекте, и есть правильный UML. Формально можно написать персональную метамодель, и тогда мы получим новый, абсолютно формально правильный UML. Нужны ли такие усилия?

Другой важный момент, UML обеспечивает весьма значительное количество различных диаграмм, тем не менее этот список не должен рассматриваться как конечный набор, который можно использовать. Весьма часто есть необходимость изобразить нечто, для чего нет формального типа диаграммы. В этой ситуации используйте подходящую не-UML диаграмму.

Диаграммы, которые ниже будут рассмотрены с разной степенью детализации:

- диаграмма классов;
- диаграмма последовательности действий;
- диаграмма объектов;
- диаграмма пакетов;
- диаграмма Use cases;
- диаграмма активностей.

Диаграмма классов

Это главная диаграмма, с которой работает программист. Она описывает типы объектов в системе и различные виды статических отношений, которые существуют между ними. Также здесь показываются свойства и операторы класса

и ограничения, которые накладываются на способ, которым они связаны. UML использует термин «свойство» как общий термин для свойств и операторов (методов) класса. Следует различать свойства как поднабор операторов следующих контракту Java Beans – **get**<Имясвойства>, **set**<Имясвойства>.

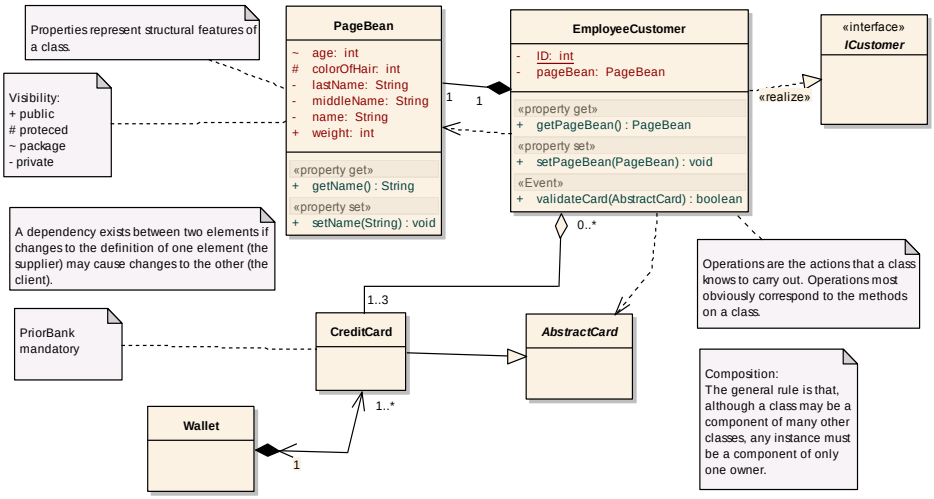


Рис. 3. «Свойства» классов

Свойства

Свойства представляют собой структурные особенности класса. В первом приближении можно думать о свойствах как о полях в классе. Действительность подправит понимание, но это хорошая стартовая точка для начала.

Свойства отображаются двумя существенно отличающимися нотациями: в виде атрибута и в виде ассоциации. Они выглядят отличными друг от друга на диаграмме, но в действительности они одно и то же.

Атрибут – нотация, описывающая свойство текстовой строкой внутри изображения класса.

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

Отношение – нотация, описывающая свойство линией, соединяющей классы между собой. В объектно-ориентированном проектировании особое значение имеют четыре типа отношений: зависимости, обобщения, реализации и ассоциации.

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.

Рис. 4. Отношение зависимости

Обобщение (Generalization) означает, что объекты подкласса могут использоваться всюду, где встречаются объекты суперкласса, но не наоборот. Подкласс наследует свойства родителя (атрибуты и методы). Идентификация суперклассов и подклассов осуществляется с использованием модели предметной области. В итоге улучшается понимание кода (особенно для систем с сотнями классов), уменьшается объем повторяемой информации.

Например, понятия **`CashPayment`**, **`CreditPayment`**, **`CheckPayment`** очень похожи, и разумно организовать их в иерархию обобщения, чтобы подчеркнуть специализацию классов. Класс **`Payment`** представляет более общее понятие, а его подклассы – специализированные свойства.

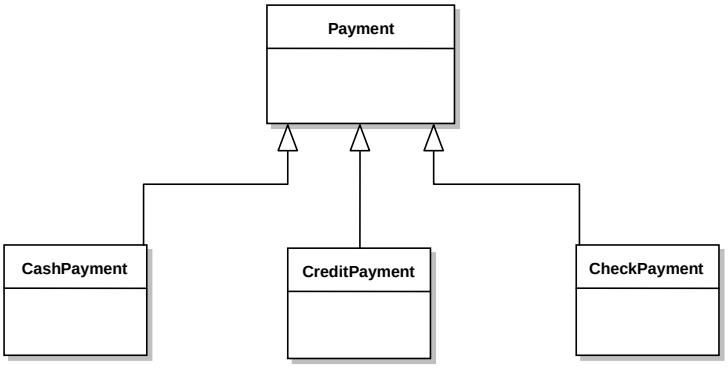


Рис. 5. Отношение обобщения

Подкласс создается в случаях, если:

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

Отношение обобщения реализуется при наследовании классов.

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

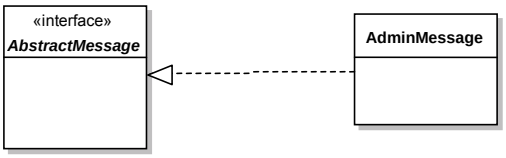


Рис. 6. Отношение реализации

Ассоциация (Association) показывает, что объект одного класса связан с объектом другого класса и отражает некоторое отношение между ними. В этом случае можно перемещаться (с помощью вызова методов) от объекта одного класса к объекту другого. Изображается сплошной линией между двумя классами, направленной от исходного класса к целевому классу.

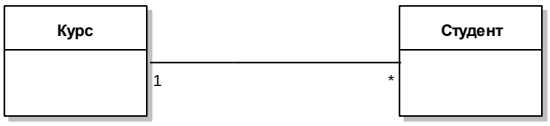


Рис. 7. Отношение ассоциации

Одним из вариантов отношения ассоциации является агрегация.

Агрегация – ассоциация, моделирующая взаимосвязь “часть/целое” между классами, которые в то же время могут быть равноправными. Оба класса при этом находятся на одном концептуальном уровне, и ни один не является более важным, чем другой.

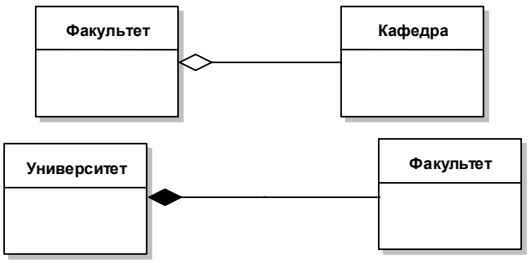


Рис. 8. Отношения коллективной агрегации и композиции

Множественность

В общем случае множественность определяет нижнюю и верхнюю границу количества объектов, которые могут соответсвововать свойствам.

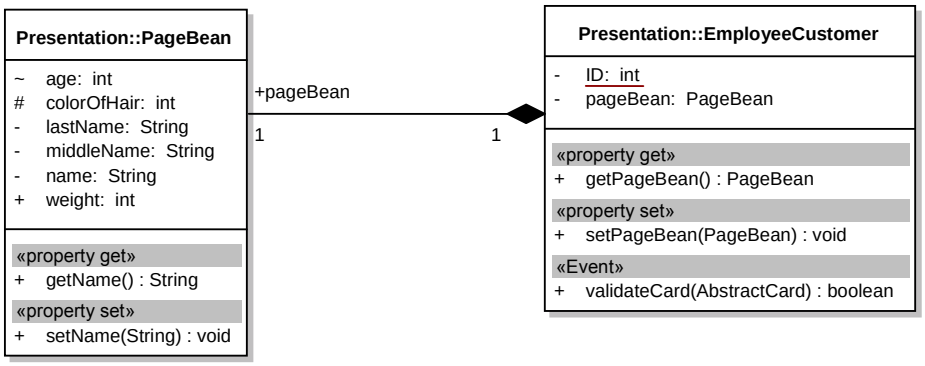


Рис. 9. Множественность

```
/* пример # 1: множественность отношений между классами :
EmployeeCustomer.java */
public class EmployeeCustomer {
      private int ID = 0;
      private PageBean pageBean = null;

      public int getID() {
            return this.ID;
      }
      public void setID(int newID) {
```

```
            this.ID = newID;
      }
      public PageBean getPageBean() {
            return this.pageBean;
      }
}
```

Двунаправленная ассоциация

Рис. 11. Двунаправленная ассоциация

```
/* пример # 2: взаимная видимость классов :  Wallet.java, CreditCard.java */
public class Wallet {
      private List<CreditCard> creditCards = null;
      public void addCreditCard(CreditCard newCreditCard) {
            creditCards.add(newCreditCard);
      }
}
public class CreditCard {
      private Wallet wallet = null;

      public void setWallet(Wallet newWallet) {
            this.wallet = newWallet;
      }
}
```

Операторы

Наиболее очевидный пример оператора – метод класса. То есть операторы можно рассматривать как действия, которые класс знает, как выполнить. Хотя getter и setter для свойства тоже являются методами, как правило, их не указывают на диаграммах, так как их наличие очевидно в соответствии с конкрактом Java Beans. Обычно их указывают на диаграммах в случае несимметричного использования, например, класс может иметь только getter-методы по каким-то специфическим причинам.

При наличии ассоциации между классами объекты одного класса могут «видеть» объекты другого и осуществлять навигацию к ним, если это не запрещено явным указанием односторонней навигации. Навигация осуществляется посредством вызова метода. Вызов метода объектом одного класса с помощью ссылки на другой класс определяет передачу сообщения от первого класса ко второму.

Интерфейсом называется набор операций, которые используются для спецификации услуг, предоставляемых классом или компонентом, причем один класс может реализовать несколько интерфейсов. Перечень всех реализуемых классом интерфейсов образует полную спецификацию поведения класса. Однако в контексте ассоциации с другим целевым классом исходный класс может не раскрывать все свои возможности.

Приложение 4

БАЗЫ ДАННЫХ И ЯЗЫК SQL

Каждая область человеческой деятельности нуждается в хранении и обработке сопутствующей информации. Например, библиотека хранит сведения о книжных фондах и параллельно ведет списки читателей. Бухгалтерия оперирует счетами и производит различные операции над ними. Склад ведет учет наличия товаров и отслеживает их движение.

Под базой данных (БД) понимается некий организованный набор информации. В качестве примера простейшей БД можно привести список товаров, каждый из которых обладает набором стандартных характеристик (наименование, единица измерения, количество, цена и т.д.):

№	Наименование	Ед. изм.	Цена	Кол-во
1	Кирпич	штука	255	10000
2	Краска	литр	580	670
3	Шифер	лист	130	500
…	…	…	…	…
10001	Гвоздь	штука	20	8000
10002	Кабель	метр	100	200

Способов организации баз данных великое множество. В недавнем прошлом бумажные листки со списками товаров держали подшитыми в отдельную папку либо раскладывали по ящикам стола в соответствии с некоторыми критериями. В наше время для подобных целей повсеместно используются компьютеры, что значительно облегчает процесс создания базы данных и дальнейшей работы с ней. Существуют специальные компьютерные программы, позволяющие полностью автоматизировать процесс хранения, получения и модификации данных любого типа и назначения. В общем случае они называются системами управления базами данных (СУБД) и состоят из языковых и программных средств, предназначенных для создания и эксплуатации баз данных. Базовые свойства любой СУБД включают в себя:

- скорость (время доступа к данным);
- разграничение доступа (отдельные категории пользователей имеют доступ только к определенным категориям данных);
- гибкость (возможность формировать и обрабатывать сложные запросы к данным);
- целостность (средства поддержки согласованности взаимосвязанных данных при их изменении);
- отказоустойчивость (средства архивирования и восстановления данных на случай выхода из строя оборудования либо ошибок в программном обеспечении).

Базовые функции СУБД:

- интерпретация запросов пользователя, сформированных на специальном языке (обычно – SQL);

- определение данных (создание и поддержка специальных объектов, хранящих поступающие от пользователя данные, ведение внутреннего реестра объектов и их характеристик – так называемого словаря данных);
- исполнение запросов по выбору, изменению или удалению существующих данных или добавлению новых данных;
- безопасность (контроль запросов пользователя на предмет попытки нарушения правил безопасности и целостности, задаваемых при определении данных);
- производительность (поддержка специальных структур для обеспечения максимально быстрого поиска нужных данных);
- архивирование и восстановление данных.

Реляционные СУБД

Модель данных в реляционных СУБД

Прежде чем сохранять какие-либо данные в СУБД, необходимо описать модель этих данных. По типу модели данных СУБД делятся на *сетевые, иерархические* и *реляционные*. СУБД реляционного типа являются наиболее распространенными и часто используемыми. В качестве примеров можно привести Oracle и Microsoft SQL Server.

Теория реляционных СУБД была разработана Коддом из IBM в 60-х годах XX века и базируется на математической теории отношений. Важнейшие понятия этой теории – таблица, строка, столбец, отношение, первичный и вторичный ключ.

Реляционная СУБД представляет собой совокупность именованных двумерных таблиц данных, логически связанных (находящихся в отношении) между собой. Таблицы состоят из строк и именованных столбцов, строки представляют собой экземпляры информационного объекта, столбцы – атрибуты объекта. В рассмотренном ранее примере таблица (назовем ее “Склад”) состоит из информационных объектов-строк, отдельная строка содержит сведения об отдельном товаре. Каждый товар характеризуется некими параметрами-атрибутами (“Наименование”, “Цена” и т.д.). Строки иногда называют записями, а столбцы – полями записи.

Таким образом, в реляционной модели все данные представлены для пользователя в виде таблиц значений данных, и все операции над базой сводятся к манипулированию таблицами.

Связи между отдельными таблицами в реляционной модели в явном виде могут не описываться. Они устанавливаются пользователем при написании запроса на выборку данных и представляют собой условия равенства значений соответствующих полей.

Пример логически взаимосвязанных таблиц:

Сотрудники

Табельный №	Фамилия	Должность	№ отдела
1	Иванов	Начальник	15
2	Петров	Инженер	15
3	Сидоров	Менеджер	10

Отделы

№ отдела	Наименование
15	Производственный отдел
10	Отдел продаж

В реляционной модели при логическом связывании таблиц применяется следующая терминология:

- Первичный ключ (или главный ключ, primary key, **PK**). Представляет собой столбец или совокупность столбцов, значения которых однозначно идентифицируют строки. В данном примере первичным ключом в таблице “Сотрудники” является столбец “Табельный №”, ибо в одной организации не бывает сотрудников с одинаковыми табельными номерами. Очевидно, что в таблице “Отделы” первичным ключом является столбец, содержащий номер отдела;
- Вторичный (или внешний ключ, foreign key, **FK**). Столбец или совокупность столбцов, которые в данной таблице не являются первичными ключами, но являются первичными ключами в другой таблице. В рассматриваемом примере столбец “№ отдела” таблицы “Сотрудники” содержит вторичный ключ, с помощью которого может быть установлена логическая взаимосвязь строк таблицы с соответствующими строками таблицы “Отделы”.

Если какая-либо таблица содержит вторичный ключ, то она считается логически взаимосвязанной с таблицей, содержащей соответствующий первичный ключ. В общем случае эта связь имеет характер “один ко многим” (одному значению первичного ключа может соответствовать несколько значений вторичного, пример – отдел № 15). Возможны варианты, когда вторичный ключ входит в состав первичного ключа. Всё зависит от предметной области, которую описывает модель. В общем случае СУБД ничего “не знает” о логической взаимосвязи таблиц модели. При обращении к СУБД с запросом пользователь должен в явном виде указать условия связывания двух таблиц. В нашем примере условие будет выглядеть примерно так: “Сотрудники”.”№ отдела” = “Отделы”.”№ отдела”. Следовательно, в процессе написания запроса возможно связать две таблицы по любым произвольным полям (не только по первичным и вторичным ключам), которые в принципе могут быть сравнимы друг с другом. В этом случае связь носит характер “многие ко многим”. Иногда это бывает необходимо делать при написании сложных и специфических запросов, но в общем случае не рекомендуется и свидетельствует об ошибках при проектировании логической модели БД.

В некоторых реляционных СУБД возможно создавать так называемые ограничения целостности, которые в том числе контролируют взаимосвязь между **PK** и **FK**. Так, СУБД заблокирует попытки удалить запись из таблицы, на первичный ключ которой “ссылаются” вторичные ключи в других таблицах. И наоборот – нельзя будет внести в поле вторичного ключа значение, отсутствующее в первичном ключе логически взаимосвязанной таблицы. Но это – только средство поддержания целостности данных и защиты от ошибок. Даже при наличии таких конструкций СУБД всё равно требует от пользователя логического связывания таблиц в явном виде при написании запросов к данным.

Нормализация модели данных

Основным критерием качества разработанной модели данных является ее соответствие так называемым нормальным формам (НФ). Основная цель нормализации – устранение избыточности данных. Коддом были определены три нормальные формы, которые включают одна другую. Другими словами, если модель данных соответствует 2НФ, то она одновременно соответствует и 1НФ. Соответствие 3НФ подразумевает соответствие 1НФ и 2НФ.

Первая нормальная форма гласит: информация в каждом поле таблицы является неделимой и не может быть разбита на подгруппы. Пример информации, не соответствующей 1НФ:

…	Иванов, 15 отдел, начальник	…

Правильно:

Фамилия	Должность	№ отдела
Иванов	Начальник	15

Вторая нормальная форма гласит: таблица соответствует 1НФ и в таблице нет неключевых атрибутов, зависящих от части сложного (состоящего из нескольких столбцов) первичного ключа. Пример информации, не соответствующей 2НФ:

№ отдела	Должность	Отдел	Количество сотрудников
15	Начальник	Производственный отдел	1
15	Инженер	Производственный отдел	5
10	Начальник	Отдел продаж	1
10	Менеджер	Отдел продаж	10

Предположим, что данная модель описывает структуру отделов по должностям. Первичный ключ (выделен серым цветом) является сложным и состоит из двух столбцов (номер отдела и наименование должности). В данном случае наименование отдела логически зависит только от номера отдела и не зависит от должности (одна и та же должность может существовать в разных отделах). Чтобы привести модель ко 2-й НФ, необходимо разбить эту таблицу на две логически взаимосвязанные:

Отделы

№ отдела	Наименование
15	Производственный отдел
10	Отдел продаж

Структура

№ отдела	Должность	Количество сотрудников
15	Начальник	1
15	Инженер	5
10	Начальник	1
10	Менеджер	10

Кстати, это пример случая, когда вторичный ключ (“№ отдела”) одновременно является частью первичного (“№ отдела”, “Должность”).

Следствие: если в таблице первичный ключ состоит из одного столбца, то эта таблица автоматически соответствует 2НФ (при условии соответствия и 1НФ).

Третья нормальная форма гласит: таблица соответствует первым двум НФ и все неключевые атрибуты зависят только от первичного ключа и не зависят друг от друга. Пример несоответствия 3НФ:

Сотрудники

Табельный №	Фамилия	Оклад	Наименование отдела	№ отдела
1	Иванов	500	Производственный отдел	15
2	Петров	400	Производственный отдел	15
3	Иванов	600	Отдел продаж	10

Очевидно, что неключевые атрибуты “Наименование отдела” и “№ отдела” логически взаимосвязаны друг с другом, в то время как комбинация фамилия-отдел-оклад имеет смысл только в сочетании с табельным номером сотрудника (предположим, что в организации работают два Иванова-однофамильца в разных отделах). Решение может быть следующим:

Сотрудники

Табельный №	Фамилия	Оклад	№ отдела
1	Иванов	500	15
2	Петров	400	15
3	Иванов	600	10

Отделы

№ отдела	Наименование
15	Производственный отдел
10	Отдел продаж

Язык SQL

Взаимодействие приложений и пользователей с реляционными СУБД осуществляется посредством специального языка структурированных запросов (Structured Query Language, сокращенно – SQL). SQL был разработан еще в начале 70-х годов XX века и представляет собой непроцедурный язык, состоящий из набора стандартных команд на английском языке. Термин “непроцедурный” означает, что изначально в языке отсутствуют алгоритмические конструкции (переменные, переходы по условию, циклы и т.д.) и возможность компоновать логически связанные команды в единые программные блоки (процедуры и функции).

Язык SQL в настоящий момент стандартизован, последний действующий стандарт носит название SQL2. Практически все известные СУБД поддерживают требования стандарта SQL2 плюс вводят собственные расширения языка SQL, учитывающие особенности конкретной СУБД (в том числе и процедурные расширения).

Общий принцип работы с СУБД посредством SQL можно кратко описать следующим образом: выдал команду – получил результат. Отдельные команды изначально никак логически не связаны друг с другом. Например, для извлечения данных из таблицы пользователь должен сформировать специальное предложение на языке SQL. СУБД обрабатывает запрос, извлекает нужные данные и возвращает их пользователю, после чего “забывает” об этом и переходит в состояние готовности выполнить любой очередной запрос SQL.

“Общение” пользователя с СУБД осуществляется с помощью специальных утилит, которые обычно входят в комплект поставки СУБД. В частности, у Oracle эта утилита называется SQL*Plus, а у MS SQL Server – Query Analyzer. Любая из них способна как минимум принять от пользователя SQL-команду, отправить ее на выполнение ядру СУБД и отобразить на экране результат операции. Вот как выглядит экран Oracle SQL*Plus:

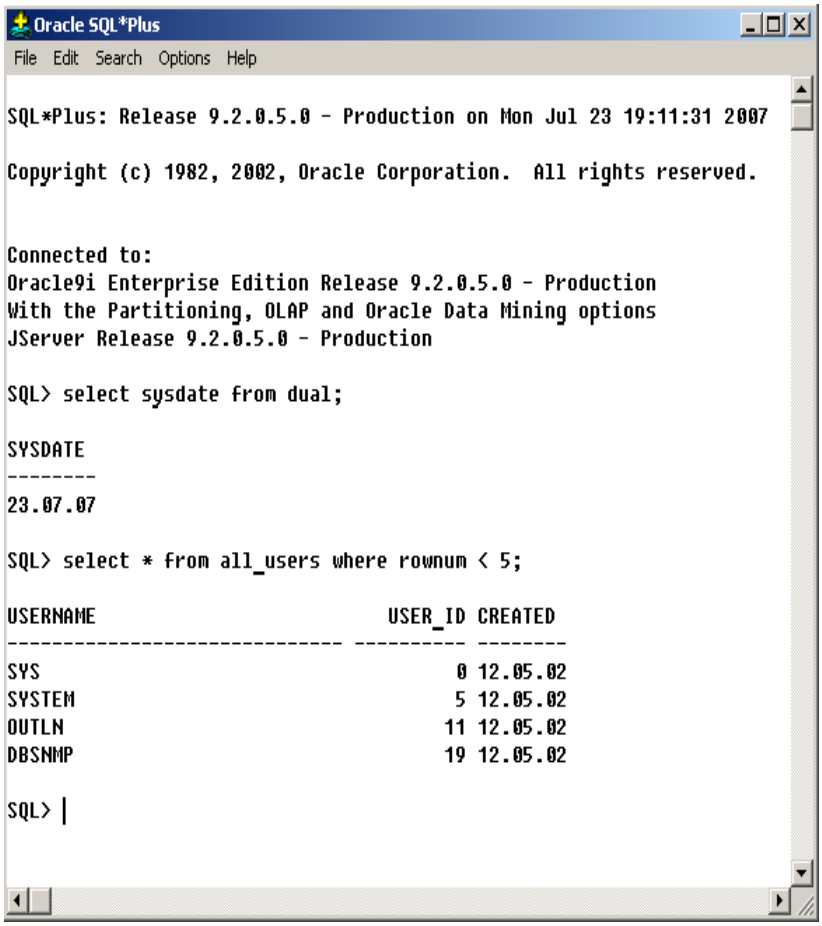


Рис. 1. Интерактивная утилита

В состав дистрибутива СУБД входят и API-библиотеки, позволяющие исполнять SQL-запросы из написанного пользователем программного кода.

Команды SQL

Как будет подробнее рассмотрено ниже, SQL позволяет не только извлекать данные, но и изменять их, добавлять новые данные, удалять данные, определять структуру данных, управлять пользователями, разграничивать доступ к данным и многое другое.

Базовый вариант SQL содержит порядка 40 команд (часто еще называемых запросами или операторами) для выполнения различных действий внутри СУБД.

Все SQL-команды начинаются с глагола (команды), определяющего, что именно нужно сделать. Далее с помощью внутренних ключевых слов задаются дополнительные условия выполнения. Например, команда на выборку табельных номеров сотрудников с зарплатой больше 500 у.е. из таблицы, содержащей список сотрудников некоей организации, выглядит следующим образом:

```
SELECT TabNum FROM Employees WHERE Salary > 500
```

где:

- **SELECT** – глагол (“выбрать”);
- **FROM**, **WHERE** – ключевые слова (“откуда”, “где”);
- **Employees** – имя таблицы;
- **TabNum**, **Salary** – имена столбцов таблицы.

В общем случае структура каждой команды зависит от ее типа.

В зависимости от вида производимых действий все команды разбиты на несколько групп.

Команды определения структуры данных (Data Definition Language – DDL)

В состав DDL-группы входят команды, позволяющие определять внутреннюю структуру базы данных. Перед тем как сохранять данные в БД, необходимо создать в ней таблицы и, возможно, некоторые другие сопутствующие объекты (увеличивающие скорость поиска индексы, ограничения целостности и др.).

Пример некоторых DDL-команд:

Команда	Описание
CREATE TABLE	Создать новую таблицу
DROP TABLE	Удалить существующую таблицу
ALTER TABLE	Изменить структуру существующей таблицы

Команды манипулирования данными (Data Manipulation Language – DML)

DML-группа содержит команды, позволяющие вносить, изменять, удалять и извлекать данные из таблиц.

Примеры DML-команд:

Команда	Описание
SELECT	Извлечь данные из таблицы
INSERT	Добавить новую строку данных в таблицу
DELETE	Удалить строки из таблицы
UPDATE	Изменить информацию в строках таблицы

Команды управления транзакциями (Transaction Control Language – TCL)

TCL-команды используются для управления изменениями данных, производимыми DML-командами. С их помощью несколько DML-команд могут быть объединены в единое логическое целое, называемое транзакцией. При этом все команды на изменение данных в рамках одной транзакции либо завершаются успешно, либо все могут быть отменены в случае возникновения каких-либо проблем с выполнением любой из них. Транзакции есть одно из средств поддержания целостности и непротиворечивости данных и являются одной из важнейших функций современных СУБД.

TCL-команды:

Команда	Описание
COMMIT	Завершить транзакцию и зафиксировать все изменения в БД
ROLLBACK	Отменить транзакцию и отменить все изменения в БД

SET TRANSACTION	Установить некоторые условия выполнения транзакции

Команды управления доступом (Data Control Language – DCL)

DCL-команды управляют доступом пользователей к БД и отдельным объектам:

Команда	Описание
GRANT	Разрешить доступ
REVOKE	Отменить доступ

Работа с командами SQL

Извлечение данных, команда SELECT

Быстрое извлечение данных, хранящихся в таблицах, одна из основных задач СУБД. Для выборки данных используется команда **SELECT**. В общем виде синтаксис этой команды выглядит следующим образом:

```
SELECT [DISTINCT] <список столбцов>
FROM <имя таблицы> [JOIN <имя таблицы> ON <условия связывания>]
[WHERE <условия выборки>]
[GROUPUP BY <список столбцов для группировки> [HAVING <условия выборки групп>] ]
[ORDER BY <список столбцов для сортировки>]
```

В квадратных скобках указаны необязательные элементы команды. Ключевые слова **SELECT** и **FROM** должны присутствовать всегда. Ниже рассмотрены возможные варианты написания этой команды подробнее.

Список столбцов содержит перечень имен столбцов таблицы, которые должны быть включены в результат. Имена, если их несколько, отделяются друг от друга запятой:

```
SELECT TabNum FROM Employees
SELECT TabNum, Name FROM Employees
```

Звездочка (*****) на месте списка столбцов обозначает все столбцы таблицы:

```
SELECT * FROM Employees
```

При выборке столбцов с одинаковыми именами из нескольких таблиц перед именем каждого столбца надо указать через точку имя таблицы:

```
SELECT Employees.Name, Departments.Name FROM …
```

Ключевое слово DISTINCT

Если в результирующем наборе данных встречаются одинаковые строки (значения всех полей совпадают), можно от них избавиться, указав ключевое слово **DISTINCT** перед списком столбцов. Приведенный ниже запрос вернет уникальный список должностей, существующих в организации:

```
SELECT DISTINCT Position FROM Employees
```

Секция FROM, логическое связывание таблиц

Перечень таблиц, из которых производится выборка данных, указывается в секции **FROM**. Выборка возможна как из одной таблицы, так и из нескольких логически взаимосвязанных. Логическая взаимосвязь осуществляется с помощью подсекции **JOIN**. На каждую логическую связь пишется отдельная подсекция. Внутри подсекции указывается условие связи двух таблиц (обычно по условию равенства первичных и вторичных ключей). Примеры для модели данных `Сотрудники-Отделы-Города`:

Employees

TabNum	Name	Position	DeptNum	Salary
1	Иванов	Начальник	15	1000
2	Петров	Инж.	15	500
3	Сидоров	Менеджер	10	700

Departments

DeptNum	City	Name
15	1	Производственный отдел
10	2	Отдел продаж

Cities

City	Name
1	Минск
2	Москва

```
SELECT Employees.TabNum, Employees.Name, Departments.Name
      FROM Employees
            JOIN Departments ON Employees.DeptNum =
                  Departments.DeptNum
```

Результат запроса будет выглядеть следующим образом:

1	Иванов	Производственный отдел
2	Петров	Производственный отдел
3	Сидоров	Отдел продаж

```
SELECT Employees.TabNum, Employees.Name, Departments.Name,
Cities.Name
      FROM Employees
            JOIN Departments ON Employees.DeptNum = Depart-
      ments.DeptNum
            JOIN Cities ON Departments.City = Cities.City
```

Результат запроса будет выглядеть следующим образом:

1	Иванов	Производственный отдел	Минск
2	Петров	Производственный отдел	Минск
3	Сидоров	Отдел продаж	Москва

Пример связывания таблиц по нескольким полям:

```
SELECT Table1.Field1, Table2.Field2
```

```
FROM Table1
JOIN Table2
      ON Table2.ID1 =Table1.ID1
      AND Table2.ID2 =Table1.ID2
      AND ….
```

Существует несколько типов связывания:

Тип	Результат
JOIN	Внутреннее соединение. В результирующем наборе присутствуют только записи, значения связанных полей в которых совпадают
LEFT JOIN	Левое внешнее соединение. В результирующем наборе присутствуют все записи из Table1 и соответствующие им записи из Table2. Если соответствия нет, поля из Table2 будут пустыми
RIGHT JOIN	Правое внешнее соединение. В результирующем наборе присутствуют все записи из Table2 и соответствующие им записи из Table1. Если соответствия нет, поля из Table1 будут пустыми
FULL JOIN	Полное внешнее соединение. Комбинация двух предыдущих. В результирующем наборе присутствуют все записи из Table1 и соответствующие им записи из Table2. Если соответствия нет – поля из Table2 будут пустыми. Записи из Table2, которым не нашлось пары в Table1, тоже будут присутствовать в результирующем наборе. В этом случае поля из Table1 будут пустыми.
CROSS JOIN	Cartesian product. Результирующий набор содержит все варианты комбинации строк из Table1 и Table2. Условие соединения при этом не указывается.

Проиллюстрируем каждый тип примерами. Модель данных:

Table1

Key1	Field1
1	A
2	B
3	C

Table2

Key2	Field2
1	AAA
2	BBB
2	CCC
4	DDD

```
SELECT Table1.Field1, Table2.Field2
      FROM Table1
      JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC

```
SELECT Table1.Field1, Table2.Field2
      FROM Table1
```

```
      LEFT JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC
C	

```
SELECT Table1.Field1, Table2.Field2
      FROM Table1
      RIGHT JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC
	DDD

```
SELECT Table1.Field1, Table2.Field2
      FROM Table1
      FULL JOIN Table2 ON Table1.Key1 = Table2.Key2
```

Результат:

A	AAA
B	BBB
B	CCC
	DDD
C	

```
SELECT Table1.Field1, Table2.Field2
      FROM Table1
      CROSS JOIN Table2
```

Результат:

A	AAA
A	BBB
A	CCC
A	DDD
B	AAA
B	BBB
B	CCC
B	DDD
C	AAA
C	BBB
C	CCC
C	DDD

Секция WHERE

Для фильтрации результатов выполнения запроса можно использовать условия выборки в секции **WHERE**. В общем виде синтаксис **WHERE** выглядит следующим образом:

```
WHERE [NOT] <условие1>  [ AND | OR <условие2>]
```

Условие представляет собой конструкцию вида:

```
<столбец таблицы, константа или  выражение>
      <оператор сравнения> <столбец таблицы, константа или выражение>
      или
      IS [NOT] NULL
      или
      [NOT] LIKE <шаблон>
      или
      [NOT] IN (<список значений>)
      или
      [NOT] BETWEEN <нижняя граница> AND <верхняя граница>
```

Операторы сравнения:

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

Примеры запросов с операторами сравнения:

```
SELECT * FROM Table WHERE Field > 100
SELECT * FROM Table WHERE Field1 <= (Field2 + 25)
```

Выражение **IS [NOT] NULL** проверяет данные на [не]пустые значения:

```
SELECT * FROM Table WHERE Field IS NOT NULL
SELECT * FROM Table WHERE Field IS NULL
```

Необходимо отметить, что язык SQL, в отличие от языков программирования, имеет встроенные средства поддержки факта отсутствия каких-либо данных. Осуществляется это с помощью **NULL**-концепции. **NULL** не является каким-то фиксированным значением, хранящимся в поле записи вместо реальных данных. Значение **NULL** не имеет определенного типа. **NULL** – это индикатор, говорящий пользователю (и SQL) о том, что данные в поле записи отсутствуют. Поэтому его нельзя использовать в операциях сравнения. Для проверки факта наличия-отсутствия данных в SQL введены специальные выражения.

Выражение [**NOT**] **LIKE** используется при проверке текстовых данных на [не]соответствие заданному шаблону. Символ ‘%’ (процент) в шаблоне заменяет собой любую последовательность символов, а символ ‘_’ (подчеркивание) – один любой символ.

```
SELECT * FROM Employees WHERE Name LIKE 'Иван%'
```

Попадающие под заданное условие фамилии: **Иван**ов

```
SELECT * FROM Employees WHERE Name LIKE '__д%'
```

Попадающие под заданное условие фамилии: Сидоров

Выражение [NOT] IN проверяет значения на [не]вхождение в определенный список:

```
SELECT * FROM Employees WHERE Position IN ('Начальник', 'Менеджер')
```

Выражение [**NOT**] **BETWEEN** проверяет значения на [не]попадание в некоторый диапазон:

```
SELECT * FROM Employees WHERE Salary BETWEEN 200 AND 500
```

Этот запрос вернет список работников, зарплата которых больше либо равна 200 у.е. и меньше либо равна 500 у.е.

Несколько условий поиска могут комбинироваться посредством логических операторов **AND**, **OR** или **NOT**:

```
SELECT *
      FROM Employees
      WHERE Position IN ('Начальник', 'Менеджер')
            AND Salary BETWEEN 200 AND 500
SELECT *
      FROM Employees
      WHERE (Position = 'Начальник' OR  Position =
                                                'Менеджер')
            AND Salary BETWEEN 200 AND 500
SELECT *
      FROM Employees
      WHERE NOT (Position = 'Начальник' OR  Position =
                                                'Менеджер')
```

Секция ORDER BY

Необязательная секция **ORDER BY** в команде **SELECT** предназначена для сортировки строк результирующего набора данных. Формат этой секции в общем виде выглядит так:

```
ORDER BY Field1 [ASC | DESC] [, Field2 [ASC | DESC] ] [, …]
```

Ключевое слово **ASC** предписывает производить сортировку по возрастанию, а **DESC** – по убыванию. Если **ASC** и **DESC** отсутствуют, по умолчанию подразумевается **ASC**. Например, выберем записи о начальниках и отсортируем результат в порядке убывания размера зарплат:

```
SELECT *
      FROM Employees
      WHERE Position = 'Начальник'
      ORDER BY Salary DESC
```

Следующий запрос отсортирует сотрудников по отделам (в порядке возрастания номера отдела) и по размеру зарплат внутри каждого отдела (в порядке убывания зарплаты):

```
SELECT *
```

```
      FROM Employees
      ORDER BY DeptNum ASC, Salary DESC
```

Ключевое слово **ASC** можно опустить, ибо оно действует по умолчанию:

```
SELECT *
      FROM Employees
      ORDER BY DeptNum, Salary DESC
```

Групповые функции

Если нас не интересуют строки таблицы как таковые, а интересуют некоторые итоги, мы можем использовать в процессе выборки колонок таблиц групповые функции. Основные групповые функции представлены ниже:

Функция	Описание
SUM(Field)	Вычисляет сумму по указанной колонке
MIN(Field)	Вычисляет минимальное значение по указанной колонке
MAX(Field)	Вычисляет максимальное значение по указанной колонке
AVG(Field)	Вычисляет среднее значение по указанной колонке
COUNT(*)	Вычисляет количество строк в результирующей выборке
COUNT(Field)	Вычисляет количество не пустых значений в колонке

Например, чтобы узнать максимальную зарплату, получаемую сотрудниками в организации, можно выполнить запрос вида:

```
SELECT MAX(SALARY)
      FROM Employees
```

Общее количество записей в таблице вернет запрос вида:

```
SELECT COUNT(*)
      FROM Employees
```

Секция GROUP BY

По умолчанию группой, на которой вычисляется групповая функция, является вся результирующая выборка. Если мы нуждаемся в вычислении промежуточных итогов, мы можем разбить итоговую выборку на подгруппы с помощью необязательной секции **GROUP BY**:

```
GROUP BY Field1 [, Field2] [, …]
```

Например, подсчитаем максимальную зарплату по отделам организации:

```
SELECT DeptNum, MAX(SALARY)
      FROM Employees
      GROUP BY DeptNum
```

В этом случае функция **MAX** будет считаться отдельно для всех записей с одинаковым значением поля **DeptNum**.

Секция HAVING

На промежуточные итоги может быть наложен дополнительный фильтр посредством секции **HAVING**. В нижеприведенном примере в результат попадут только отделы, максимальная зарплата в которых превышает 1000 у.е.:

```
SELECT DeptNum, MAX(SALARY)
      FROM Employees
      GROUP BY DeptNum
      HAVING MAX(SALARY) > 1000
```

Важно понимать, что секции **HAVING** и **WHERE** взаимно дополняют друг друга. Сначала с помощью ограничений **WHERE** формируется итоговая выборка, затем выполняется разбивка на группы по значениям полей, заданных в **GROUP** BY. Далее по каждой группе вычисляется групповая функция и в заключение накладывается условие **HAVING.**

Изменение данных

Под изменением данных понимаются следующие операции:

- вставка новых строк в таблицу;
- изменение существующих строк;
- удаление существующих строк.

Команда INSERT

Добавление новых записей в таблицу осуществляется посредством команды **INSERT**. Она имеет следующий синтаксис:

```
INSERT INTO <имя таблицы> [(<список имен колонок>)]
      VALUES(<список констант>)
```

Например, для внесения сведений о новом работнике необходимо выполнить следующую команду:

```
INSERT INTO Employees(TabNum, Name, Position, DeptNum,
                                                Salary)
      VALUES(45, 'Сергеев', 'Старший менеджер', 15, 850)
```

После выполнения команды таблица Employees будет выглядеть следующим образом:

Employees

TabNum	Name	Position	DeptNum	Salary
1	Иванов	Начальник	15	1000
2	Петров	Инженер	15	500
3	Сидоров	Менеджер	10	700
45	Сергеев	Старший менеджер	15	850

Если какая-либо колонка в списке будет опущена при вставке, в соответствующее поле записи автоматически будет занесено пустое значение (**NULL**):

```
INSERT INTO Employees(TabNum, Name, DeptNum, Salary)
      VALUES(45, 'Сергеев', 15, 850)
```

После выполнения команды таблица Employees будет выглядеть следующим образом:

Employees

TabNum	Name	Position	DeptNum	Salary
1	Иванов	Начальник	15	1000
2	Петров	Инженер	15	500
3	Сидоров	Менеджер	10	700
45	Сергеев		15	850

Количество констант в секции **VALUES** всегда должно соответствовать количеству колонок. Список колонок в команде **INSERT** может быть опущен целиком. В этом случае список констант в секции **VALUES** должен точно соответствовать описанию колонок таблицы в словаре данных СУБД, иначе команда будет отвергнута ядром БД. Пример правильной команды:

```
INSERT INTO Employees VALUES(45, 'Сергеев',
                             'Старший менеджер', 15, 850)
```

Команда вида:

```
INSERT INTO Employees VALUES(45, 'Сергеев', 15, 850)
```

завершится ошибкой, так как количество констант не соответствует реальному количеству колонок в таблице.

В колонку можно в явном виде внести пустое значение посредством ключевого слова **NULL**. Последний запрос можно переписать следующим образом:

```
INSERT INTO Employees VALUES(45, 'Сергеев', NULL, 15, 850)
```

В этом случае команда вставки отработает корректно, и в поле Position будет внесено пустое значение. Очевидно, что к аналогичному результату приведет и команда:

```
INSERT INTO Employees(TabNum, Name, Position, DeptNum,
                                                Salary)
      VALUES(45, 'Сергеев', NULL, 15, 850)
```

Кроме простого добавления новых записей, команда **INSERT** позволяет осуществлять пакетную перекачку данных из таблицы в таблицу. Синтаксис подобной команды следующий:

```
INSERT INTO <имя таблицы> [(<список имен колонок>)]
      <команда SELECT>
```

Например:

```
INSERT INTO Table1(Field1, Field2)
      SELECT Field3, (Field4 + 5) FROM Table2
```

Команда DELETE

Чтобы удалить ненужные записи из таблицы, следует использовать команду **DELETE**:

```
DELETE FROM <имя таблицы> [WHERE <условия поиска>]
```

Если опустить секцию условий поиска **WHERE**, из таблицы будут удалены все записи. Иначе – только записи, удовлетворяющие критериям поиска. Форматы секций **WHERE** команд **SELECT** и **DELETE** аналогичны.
Примеры команды **DELETE**:

```
DELETE FROM Employees
DELETE FROM Employees WHERE TabNum = 45
```

Команда UPDATE

Изменить ранее внесенные командой **INSERT** данные можно с помощью команды **UPDATE**:

```
UPDATE < имя таблицы>
SET <имя колонки> = <новое значение> , <имя колонки> =
                                    <новое значение>, …
WHERE <условия поиска>]
```

Как и в случае команды **DELETE**, при отсутствии секции **WHERE** обновлены будут все строки таблицы. Иначе – только подходящие под заданные условия. Примеры:

```
UPDATE Employees SET Salary = Salary + 100
UPDATE Employees
SET Position = 'Старший менеджер', Salary = 1000
WHERE TabNum = 45 AND Position IS NULL
```

Определение структуры данных

Команда CREATE TABLE

Для создания новых таблиц используется команда **CREATE TABLE**. В общем виде ее синтаксис следующий:

```
CREATE TABLE <имя таблицы>
(
 <имя колонки> <тип колонки>[(<размер колонки>)] [<ограни-
чение целостности уровня колонки>]
[, <имя колонки> <тип колонки>[(<размер колонки>)] [<огра-
ничение целостности уровня колонки>]]
[, …]
[<ограничение целостности уровня таблицы>]
[,<ограничение целостности уровня таблицы>]
[, …]
)
```

Примеры:

```
CREATE TABLE Departments
(
DeptNum int NOT NULL PRIMARY KEY,
Name varchar(80) NOT NULL
)
CREATE TABLE Employees
```

```
(
 TabNum int NOT NULL PRIMARY KEY,
 Name varchar(100) NOT NULL,
 Position varchar(200),
 DeptNum int,
Salary decimal(10, 2) DEFAULT 0,
CONSTRAINT FK_DEPARTMENT FOREIGN KEY (DeptNum)
      REFERENCES Departments(DeptNum)
)
```

Помимо команды **CREATE TABLE**, можно создать новую таблицу с помощью специальной формы команды **SELECT**:

```
SELECT [DISTINCT] <список колонок>
INTO <имя новой таблицы>
FROM <имя таблицы> [JOIN <имя таблицы> ON <условия связыва-
ния>]
[WHERE <условия выборки>]
[GROUP BY <список колонок для группировки> [HAVING <условия
выборки групп>] ]
[ORDER BY <список колонок для сортировки>]
```

При наличии ключевого слова **INTO** в команде **SELECT** ядро СУБД не вернет результирующую выборку пользователю, а автоматически создаст новую таблицу с указанным именем и заполнит ее данными из результирующей выборки. Имена колонок таблицы и типы будут определены автоматически при анализе команды **SELECT** и словаря базы данных.

Команда ALTER TABLE

Созданную таблицу можно впоследствии изменить с помощью команды **ALTER TABLE**. Команда **ALTER TABLE** позволяет добавлять новые колонки и ограничения целостности, удалять их, менять типы колонок, переименовывать колонки.

Примеры различных вариантов команды **ALTER TABLE**:

```
ALTER TABLE Departments ADD COLUMN City int
ALTER TABLE Departments DROP COLUMN City
ALTER TABLE Departments ADD
      CONSTRAINT FK_City
            FOREIGN KEY (City)
            REFERENCES Cities(City)
ALTER TABLE Departments DROP CONSTRAINT FK_City
```

Команда DROP TABLE

Удаление ранее созданной таблицы производится командой **DROP TABLE**:

```
DROP TABLE Departments
```

Приложение 5

HIBERNATE

Hibernate – инструмент объектно-реляционного отображения (Object-Relational Mapping, ORM) данных для Java-окружения. Целью Hibernate является освобождение разработчика от большинства общих работ, связанных с задачами получения, сохранения данных в СУБД. Эта технология помогает удалить или инкапсулировать зависящий от поставщика SQL-код, а также решает стандартную задачу преобразования типов Java-данных в типы данных SQL и наборов данных из табличного представления в объекты Java-классов.

Установка

1. **Загрузить и установить сервер баз данных MySQL с сервера** `http://dev.mysql.com/`
2. **Загрузить и подключить Hibernate с сервера** `http://hibernate.org/`
3. **Загрузить и подключить JDBC-драйвер, используемой базы данных в** `tomcat/common/lib` **(или, в случае использования Ant, в папку** `lib` **проекта), в данном случае** `mysql-connector-java-3.1.12.jar`**. Обычно JDBC-драйвер предоставляется на сайте разработчика сервера баз данных.**

Библиотека `hibernate3.1.3.jar` **является основной. Кроме нее, устанавливается еще целый ряд необходимых библиотек из дистрибутива Hibernate,**
а именно:

`cglib.jar`, `commons-collections.jar`, `commons-logging.jar`, `jta.jar`, `dom4j.jar`, `log4j.jar`, **а также библиотеки** `antlr.jar`, `asm.jar`, `asm-attrs.jar` **для запуска приложения из-под Apache Ant.**

`dom4j.jar` **– отвечает за разбор файлов XML-настроек и файла XML-mapping метаданных;**

`CGLIB (cglib.jar)` **– эта библиотека используется для генерации кода для расширения классов во время исполнения;**

`commons-collections.jar` **– вспомогательная библиотека из проекта Apache Jakarta Commons;**

`commons-logging.jar` **– вспомогательная библиотека для журналирования событий из проекта Apache Jakarta Commons;**

`ODMG4 (odmg.jar)` **– пакет, необходимый для mapping-коллекций;**

`EHCache (ehcache.jar)` **– кэш-провайдер второго уровня;**

ANother Tool for Language (antlr.jar) **– инструмент, позволяющий создавать трансляторы, компиляторы и распознаватели для текстов, содержащие Java, C#, C++ или Python код;**
ASM (asm.jar, asm-attrs.jar) **– библиотека, предназначенная для динамического создания классов, а также их динамической корректировки.**

Создание простейшего приложения

Hibernate использует JDBC-соединения для вызова SQL-запросов к базе данных, поэтому необходимо указать настроенный пул JDBC-соединений либо использовать один из поддерживаемых пулов, встроенных в Hibernate (C3P0, Proxool). Tomcat настраивает и дает доступ к пулу соединений, используя встроенный DBCP пул, а Hibernate запрашивает данные соединения через интерфейс JNDI. Tomcat связывает пул соединений с JNDI, объявляя ресурс в свой основной конфигурационный файл Tomcat 5.5/conf/server.xml, в виде:

```
<Context path="/test_db" docBase="test_db">
    <Resource  name="jdbc/test_db" scope="Shareable"
            type="javax.sql.DataSource"/>

    <ResourceParams name="jdbc/test_db">
        <parameter>
            <name>factory</name>
  <value>org.apache.commons.dbcp.BasicDataSourceFactory
</value>
        </parameter>
<Context/>
```

Далее следует настроить конфигурационный файл Hibernate на использование пула соединений. Этот файл должен располагаться в каталоге
WEB-INF/classes **и иметь имя** hibernate.cfg.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
      <session-factory>
        <property name="connection.datasource"> ja-
va:comp/env/jdbc/test_db</property>
        <property name="show_sql">true</property>
        <property name="dialect">
net.sf.hibernate.dialect.MySQLDialect</property>
        <property
name="hibernate.connection.password">pass</property>
        <property
name="hibernate.connection.username">root</property>

        <mapping resource="courses/hiber/Course.hbm.xml"/>
```

```
        <mapping resource="courses/hiber/Student.hbm.xml"/>
      </session-factory>
   </hibernate-configuration>
```

Здесь указан способ получения JDBC-соединения, включено логирование команд SQL, настроен диалект SQL.

Для настройки параметров работы Hibernate вместо конфигурационного XML-файла можно использовать файл **hibernate.properties** в **WEB-INF/classes** (в случае, если приложение разбито по пакетам, необходимо это учитывать, чтобы данный файл был доступен):

```
##Данная директива позволяет делать автозамену значений
##полей класса в другие значения для удобства хранения
в ##базе данных
hibernate.query.substitutions true 1, false 0, yes 'Y',
no 'N'

##Прописывается JDBC-драйвер для базы данных MySQL
hibernate.dialect net.sf.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver

##Прописывается адрес для подсоединения к серверу
##баз данных
##Формат подключения следующий:
##hibernate.connection.url
##jdbc:mysql://АДРЕС_МАШИНЫ:НОМЕР_ПОРТА/ИМЯ_БАЗЫ_ДАННЫХ?
##autoReconnect=Автоматически переподключатся к базе данных
##или нет, при потере соединения. Этот параметр должен быть
##выставлен в true, т.к. отключение от БД происходит в
##случае 30-ти минутного простоя.

##Параметр useUnicode=true отвечает за использование
##кодировки unicode при работе с БД.
##Параметр characterEncoding=Cp1251 отвечает за кодировку
##при передаче данных в базу данных
hibernate.connection.url
jdbc:mysql://localhost:3306/test_db?autoReconnect
=true&useUnicode=true&characterEncoding=Cp1251

##Имя пользователя при подключении к БД
hibernate.connection.username root

##Пароль при подключении к БД
hibernate.connection.password pass

##Размер пула соединений к БД
##Обычно используется значение в 50 соединений при
##создании реальных проектов
hibernate.connection.pool_size 50
```

```
##Кеширование запросов. Кеш повышает быстродействие при
##использовании большого числа однотипных запросов к базе
##данных
hibernate.statement_cache.size 20
##Директива, которая используется при debug. Результатом
##является то, что все запросы, которые осуществляются
к базе ##данных, будут показаны (или нет) разработчику.
hibernate.show_sql true
```

Как еще одну альтернативу (не очень удачную) можно рассматривать метод, генерирующий объект java.util.Properties. **Например, в виде:**

```
private Properties createProperties() {
      Properties properties = new Properties();
      properties.setProperty(
            "hibernate.dialect",
            "net.sf.hibernate.dialect.MySQLDialect");
      properties.setProperty(
            "hibernate.connection.driver_class",
            "com.mysql.jdbc.Driver");
      properties.setProperty(
            "hibernate.connection.url",
            "jdbc:mysql://localhost/test_db");
      properties.setProperty(
            "hibernate.connection.username", "root");
      properties.setProperty(
            "hibernate.connection.password", "pass");
      return properties;
}
```

Создание POJO-классов

В примере, рассматривающем учебные курсы и студентов, их посещающих, будут созданы следующие классы POJO (Plain Ordinary Java Objects):

Класс Course **– хранит информацию об учебном курсе (название курса**
и список студентов, записанных на курс);
Класс Student **– содержит информацию о студенте (имя, фамилия).**

Кроме указанных выше полей, оба эти класса имеют поле id. **Это свойство содержит значение столбца первичного ключа в таблице базы данных. Такое свойство может называться любым именем, и иметь любой примитивный тип,**
тип класса-оболочки базового типа, а также типы java.lang.String
и java.util.Date. **Свойство-идентификатор не является обязательным для класса, можно не создавать его и дать Hibernate самостоятельно следить за идентификацией объекта.**

```
/* пример # 1: POJO-класс сущности : Course.java */
package courses.hiber;
```

```
import java.util.Set;

public class Course {
	private Integer id;
	private String title;
	private Set students;

	public Integer getId() {
		return id;
	}
	protected void setId(Integer id) { /*в данном случае использовать protected как спецификатор доступа, поскольку данное поле не будет определяться вручную, а значение генерируется автоматически в соответствии с mapping-файлом. Сама технология Hibernate имеет доступ к полям и методам, имеющим любые спецификаторы доступа(friendly, public, protected, private)*/
		this.id = id;
	}
	public String getTitle() {
		return title;
	}
	public void setTitle(String title) {
		this.title = title;
	}
	public Set getStudents() {
		return students;
	}
	public void setStudents(Set students) {
		this.students = students;
	}
}
/* пример # 2: POJO-класс сущности : Student.java */
package courses.hiber;

public class Student {
	private Integer id;
	private String lastname;
	private String login;
	private String password;

	public Integer getId() {
		return id;
	}
	protected void setId(Integer id) {
		this.id = id;
	}
	public String getLogin() {
		return login;
	}
	public void setLogin(String login) {
		this.login = login;
	}
```

```
	public String getLastname() {
		return lastname;
	}
	public void setLastname(String lastname) {
		this.lastname = lastname;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}
}
```

После создания таблиц в базе данных задается соответствие между POJO-классами и таблицами. Объектно-реляционный mapping описывается в виде XML-документа **hbm.xml** в каталоге, содержащем ***.class** файл соответствующего класса: для Tomcat - это **WEB-INF\classes\'каталог_пакета'**, а для Ant - **bin\'каталог_пакета'**. Эти файлы создаются для того, чтобы обеспечить Hibernate данными о том, какие объекты по каким таблицам базы данных создавать и как их инициализировать. Язык описания mapping-файла ориентирован на Java, следовательно, mapping конструируется вокруг объявлений java-классов, а не таблиц БД.

/ пример # 3: Mapping-файл для класса courses.hiber.Course */*

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="courses.hiber.Course" table="course">
	<id name="id" column="id" type="java.lang.Integer">
		<generator class="increment"/>
	</id>
	<property name="title" type="java.lang.String"/>
	<set name="students" table="course_student"
cascade="all">
		<key column="course_id"/>
		<many-to-many column="student_id"
class="courses.hiber.Student"/>
	</set>
</class>
</hibernate-mapping>
```

/ пример # 4: Mapping-файл для класса courses.hiber.Student */*

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "- //Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">
<hibernate-mapping package="courses.hiber">
```

```
<class name="Student" table="student">
      <id name="id" column="id" type="java.lang.Integer">
            <generator class="native"/>
      </id>
      <property name="lastname" type="java.lang.String"/>
      <property name="login" type="java.lang.String"/>
      <property name="password" type="java.lang.String"/>
</class>
</hibernate-mapping>
```

Пояснения по приведенному коду:

<class name="courses.hiber.Course"> – необходимо указать класс, который будет использоваться при работе с указанной ниже таблицей базы данных. Причём есть две возможности указать пакет расположения класса: либо явно указать полное имя класса, либо указать атрибут **package** в теге <**hibernate-mapping>**;

table="course" **– указывается таблица на сервере баз данных, к которой будет вестись обращение. Если данный атрибут не указан, то за название таблицы берется название класса, т.е. в данном случае** COURSE**, поэтому в данном примере атрибут** table **можно было опустить;**

<id **name**="id" **column**="id" **type**="java.lang.Integer">
      <generator **class**="native"/>
</id> **– указывается соответствие поля класса и столбца в базе данных, которые являются основными идентификаторами, т.е. уникальны, не имеют значений** null**. Тег** <generator> **указывает способ генерация значений в данном столбце, возможные его значения:** increment**,** identity**,** sequence**,** hilo**,,** seqhilo**,** uuid**,** guid**,** native**,** assigned**,** select**,** foreign**;**

<property **name**="title" **column**="column" **type**="java.lang.String"/> **– указывается соответствие полей класса и столбцов базы данных. В mapping-файле несколько параметров может быть выделено как** id**. Это позволяет получать объекты из БД, удалять, создавать их без написания SQL-запросов. Процесс реализации будет продемонстрирован ниже.**

Кроме того, следует обратить внимание на следующие теги и их параметры:

<**many-to-one** name="**propertyName**" column="**column_name**" class="**ClassName**" lazy="**proxy|no-proxy|false**"> – данный тег используется для указания связи таблиц (классов). К примеру, объект одного класса содержит ссылку на объект другого класса, а последний, в свою очередь, содержит коллекцию объектов первого класса. Параметры **name** и **column** аналогичным параметрам в предыдущих тегах и несут такой же смысл. Атрибут **class** указывает, какой класс будет связан с данным. Параметр **lazy** в большей части случаев является существенным, т.е. его необходимо выставлять вручную, поскольку значение по умолчанию, **proxy**, означает, что объекты класса, с которым связан данный, не будут автоматически загружены в память.

```
<set name="propertyName" inverse="true">
   <key column="foreignId"/>
```

```
<one-to-many class="ClassName"/>
```

</set> - является интерпретацией коллекции «множество».

Необходимо строгое соответствие между столбцами таблиц базы данных и mapping-файлами.

При изменении базы данных необходимо следить за изменением mapping-файлов и соответствующих им классов, т.к. такие ошибки достаточно сложно выявить.

Более подробно о рассмотренных и прочих тегах и их атрибутах вы можете прочитать в документации по технологии Hibernate, которая доступна на сайте разработчика.

Configuration, Session и SessionFactory

Ниже будет приведено несколько вариантов конфигурирования Hibernate. Конфигурация или отображение (mapping) обычно осуществляется один раз в течение работы приложения. Конкретная конфигурация содержится в объекте класса **net.sf.hibernate.cfg.Configuration**.

```
private Configuration createConfiguration()
    throws ClassNotFoundException, MappingException {
    Configuration configuration =
    new Configuration()
.addClass(Class.forName("courses.hiber.Course"))
.addClass(Class.forName("courses.hiber.Student"));
    return configuration;
}
```

Этот метод создает новую конфигурацию и добавляет в нее классы **courses.hiber.Course** и **courses.hiber.Student** (данное добавление необходимо только в том случае, если mapping-файлы не были указаны в конфигурационном файле Hibernate). Hibernate ищет mapping по принципу, указанному выше. В конфигурацию могли быть также добавлены другие источники mapping, например поток, jar-файл, **org.dom4j.Document**. Метод **addClass()** возвращает объект **Configuration**, поэтому он вызывается несколько раз подряд. Далее добавляется в приложение метод, генерирующий по конфигурации таблицы в базе данных.

Hibernate содержит консольное приложение для генерации скриптов. В приведенном ниже коде объект класса **SchemaExport** генерирует таблицы базы данных по настроенному mapping и заданным свойствам, записывает текст в файл (**c:/temp/courses_script.sql**) и выполняет сгенерированный SQL-код.

```
private void generateAndExecuteCrea-
tionScript(Configuration configuration, Properties proper-
ties) throws HibernateException {
    SchemaExport export = new SchemaExport(configuration,
properties);
```

```
        export.setOutputFile("c:\\temp\\courses_script.sql")
.create(true, true);
    }
```

В методе **create(true, true)** генерируется script-текст для создания таблиц и посылается на консоль (определяется первым параметром – **true**), и на его основе создаются таблицы в базе данных (второй параметр – **true**).

Session – это интерфейс, используемый для сохранения в базу данных и восстановления из нее объектов классов **Course** и **Student**. **SessionFactory** – потокобезопасный, неизменяемый кэш откомпилированных mapping для одной базы данных, фабрика для создания объектов **Session**.

1. Создается объект **SessionFactory**
   ```
   SessionFactory factory = new Configuration()
   .configure().buildSessionFactory();
   ```
   Метод **configure()** класса **Configuration** заносит информацию в объект **Configuration** из конфигурационного файла Hibernate;
2. Создается сессия
   ```
   Session session = factory.openSession();
   ```
3. **Извлекаются все строки из таблиц** course **и** student. **Текст запросов и команд пишется на Hibernate-диалекте. Он похож на SQL, только в качестве сущностей-носителей данных выступают классы, а не таблицы.**
   ```
   List courses = session.find("from Course");
   List students = session.find("from Student");
   ```
4. **В конце обращения сессия закрывается.**
   ```
   session.close();
   ```

Интерфейс **net.sf.hibernate.SessionFactory** содержит ряд необходимых методов:

openSession() – создает соединение с базой данных и открывает сессию. В качестве параметра может быть передано и соединение, тогда будет создана сессия по существующему соединению;

close() – уничтожение **SessionFactory** и освобождение всех ресурсов, используемых объектом.

Интерфейс **net.sf.hibernate.Session** – однопоточный, короткоживущий объект, являющийся посредником между приложением и хранилищем долгоживущих объектов, используется для навигации по объектному графу или для поиска объектов по идентификатору. По сути, является классом-оболочкой вокруг JDBC-соединения. В то же время представляет собой фабрику для объектов **Transaction**.

load(Class theClass, Serializable id) – возвращает объект данного класса с указанным идентификатором;

load(Object object, Serializable id) – загружает постоянное состояние объекта с указанным идентификатором в объект, указатель которого был передан;

save(Object object [, Serializable id]) – сохраняет переданный объект. Если передан идентификатор, то использует его;

update(Object object [, Serializable id]) – обновляет постоянный объект по идентификатору объекта, а если передан идентификатор, то обновляет объект с указанным идентификатором;

saveOrUpdate(Object object) – в зависимости от значения идентификатора сохраняет или обновляет объект;

get(Class class, Serializable id) – возвращает объект данного класса, с указанным идентификатором или **null**, если такого объекта нет;

delete(Object object) – удаляет объект из базы данных;

delete(String query) – удаляет все объекты, полученные по запросу. Возможен и вызов запроса с параметрами **delete(String query, Object[] objects, Type[] types)**;

Transaction beginTransaction() – начинает единицу работы и возвращает ассоциированную транзакцию.

Для осуществления запросов используется экземпляр интерфейса **org.hibernate.Query**. Получить его можно с помощью объекта **Session**:

```
session.createQuery("from Company")
```

Интерфейс **Query** имеет следующие методы:

list() - выполняет запрос, результат возвращается в коллекции **List**;

```
session.createQuery("from Company").list();
```

executeUpdate() – для выполнения удалений, изменений, применяемых к многочисленным объектам;

```
session.createQuery("delete from Company where status ='closed'").executeUpdate();
```

setString(int index, String value) , setDate() и т. д. – устанавливает параметр в запрос по индексу;

```
session.createQuery("delete from Company where status =?").setString(0, 'closed').executeUpdate();
```

также можно установить параметр по имени:

```
session.createQuery("delete from Company where status =:stat").setString('stat', 'closed').executeUpdate();
```

iterate() - возвращает **Iterator** по результатам запроса

```
Iterator it = createQuery("from Company").iterate();
```

Долгоживущие Объекты и Коллекции (Persistent Objects and Collections) – это однопоточные, короткоживущие объекты, содержащие сохраняемое состояние и бизнес-функции. Это могут быть обычные JavaBean/POJO (Plain Old Java Objects) объекты, их отличительная особенность – это то, что они ассоциированы с одной сессией (**Session**). Как только их сессия закрыта, эти объекты становятся отсоединенными и свободными для использования на любом уровне приложения, например, непосредственно как объекты передачи данных на уровень представления и с него.

Временные Объекты и Коллекции (Transient Objects and Collections) – это экземпляры долгоживущих классов, которые в данный момент не ассоциированы

с сессией (**Session**). Это могут быть объекты, созданные приложением и в данный момент еще не переведенные в долгоживущее состояние.

Транзакция **net.sf.hibernate.Transaction** – однопоточный, короткоживущий объект, используемый приложением для указания атомарной единицы выполняемой работы. Он абстрагирует приложение от нижележащих JDBC, JTA или CORBA транзакций. В некоторых случаях одна сессия (**Session**) может породить несколько транзакций:

commit() – фиксирует транзакцию базы данных;

rollback() – принуждает транзакцию возвращаться обратно.

Интерфейс **net.sf.hibernate.connection.ConnectionProvider** – поставщик соединений, фабрика и пул для JDBC-соединений. Абстрагирует приложение от нижележащих объектов **Datasource** или **DriverManager**. Внутренний объект Hibernate недоступен для приложения, но может быть расширен или реализован разработчиком. Методы:

close() – освобождает все ресурсы, занимаемые поставщиком соединения;

closeConnection(Connection conn) – закрывает используемое соединение;

configure(Properties props) – инициализирует поставщика соединений из переданных свойств.

Фабрика транзакций **net.sf.hibernate.TransactionFactory** – фабрика для экземпляров класса **Transaction**. Внутренний объект Hibernate недоступен для приложения, но также может быть расширен или реализован разработчиком.

beginTransaction(SessionImplementor session) – начинает транзакцию и возвращает ее объект.

Простейшее применение объявленных выше классов при добавлении в сервлет реализаций методов **generateAndExecuteCreationScript()** и **createProperties()** выглядит следующим образом:

/ пример # 5: простейшее применение hibernate : MainServlet.java */*

```
package courses.servlet;
import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;
import net.sf.hibernate.MappingException;
import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.util.List;
import java.util.Properties;
```

```
public class MainServlet extends HttpServlet {
      public static SessionFactory factory;
      static{
            factory =
new Configuration().configure().buildSessionFactory();
      }
      private String ACTION_SHOW = "show";
      private String ACTION_GENERATE = "generate";

      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);
      }
      private void performTask(HttpServletRequest req,
                  HttpServletResponse resp)
                  throws ServletException, IOException {
            String action = req.getParameter("action");

            if (action.equals(ACTION_SHOW)) {
                  try {

            Session session = factory.openSession();

            List courses = session.createQuery(
                              "from Course").list();
                  req.setAttribute("courses", courses);
            List students = session.createQuery(
                              "from Student").list();
                  req.setAttribute("students", students);

                        resp.sendRedirect("show.jsp");
req.getRequestDispatcher("show.jsp").forward(resp, req);
                        session.close();
                  } catch (Exception e) {
                        throw new RuntimeException(e);
                  }
            } else if (action.equals(ACTION_GENERATE)) {
                  try {
Configuration configuration = new Configuration();
                              //или createConfiguration();

            Properties properties = createProperties();
```

```
            generateAndExecuteCreationScript(
                  configuration.configure(), properties);
            resp.sendRedirect("generated.jsp");
                  } catch (Exception e) {
                        throw new RuntimeException(e);
                  }
            }
      }
}
```

<!-- пример # 6: отображение информации, извлеченной из БД: show.jsp -->

```
<%@ page import="java.util.List,
                 courses.hiber.Course,
                 courses.hiber.Student"%>
<HTML><HEAD><TITLE>Data from database</TITLE></HEAD>
    <BODY>
        All Courses:
        <TABLE width="100%">
            <TR>
                <TD>Id</TD>
                <TD>Title</TD>
            </TR>
            <%
List courses = (List)request.getAttribute("courses");
                if (courses != null) {
                    for (int i=0; i<courses.size(); i++)
                    {%>
             <TR>
<TD><%=((Course)courses.get(i)).getId()%></TD>
<TD><%=((Course)courses.get(i)).getTitle()%></TD>
             </TR>
                    <%} }%>
        </TABLE>
        All Students:
        <TABLE width="100%">
            <TR>
                <TD>Id</TD>
                <TD>First Name</TD>
                <TD>Last Name</TD>
            </TR>
            <%
    List students = (List)request.getAttribute("students");
            if (students != null) {
              for (int i=0; i<students.size(); i++) {%>
             <TR>
<TD><%=((Student)students.get(i)).getId()%></TD>

<TD><%=((Student)students.get(i)).getFirstname()%></TD>
```

```
<TD><%=((Student)students.get(i)).getLastname()%></TD>
            </TR>
                    <%}}%>
        </TABLE></BODY></HTML>
<!-- пример # 7: сообщение о генерации скрипта : generated.jsp -->
<HTML><HEAD><TITLE>Script was generated</TITLE></HEAD>
  <BODY>Script was generated</BODY></HTML>
```

Очевидно, что если сервлет при каждом обращении к нему будет создавать заново объект SessionFactory, **то этот процесс будет слишком трудоемким и с точки зрения производительности системы, и с точки зрения разработчика при дальнейшем расширении системы. Поэтому следует создать класс, задачами которого будет создание и необходимая инициализация объектов** Configuration, SessionFacroty **и один из методов класса будет возвращать готовый объект** Session. **Следовательно, в методе сервлета для вывода содержимого таблиц базы данных на экран можно использовать готовый объект** Session **и из него загрузить интересующие параметры.**

Профессиональная реализация

Используя заданную выше последовательность, для инициализации mapping-файлов и для корректного манипулирования соединением с базой данных создается класс **ConnectionFactory**. Было замечено, что в некоторых версиях hibernate, несмотря на директиву **autoReconnect=true**, автоматическое переподключение к БД происходит со второго раза, то есть подключение происходит сразу же, но появляется исключение о невозможности подключиться к БД.

```
/* пример # 8: инициализация mapping и подключение к БД :
ConnectionFactory.java */
package com.hib;
import net.sf.hibernate.HibernateException;
import net.sf.hibernate.MappingException;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.cfg.Configuration;
import java.util.Date;

public class ConnectionFactory {
      public static long timer = 0;
      public static SessionFactory sessionFactory = null;

      public static void initConnectionFactory() {
            Date dt = new Date();
            timer = dt.getTime();
            try {
            //добавление mapping-файлов в конфигурацию подключения
```

```
      Configuration cfg = new Configuration()
.addClass(Student.class).addClass(Course.class);

                  //создание подключения к БД
            sessionFactory = cfg.buildSessionFactory();
            } catch (MappingException e) {
                  System.err.print(e);
            } catch (HibernateException e) {
                  System.err.print(e);
                  destroy();
            }
      }
      public static Session getOrInitSession() {
            try {
                  Date curDate = new Date();
                  long curTime = curDate.getTime();
                  long tenminutes = 10 * 60 * 1000;

                  if (curTime - timer > tenminutes){
                        destroy();
                  }
                  else {
                        curDate = new Date();
                        timer = curDate.getTime();
                  }
                  if (sessionFactory == null) {
                        initConnectionFactory();
                        }
            return sessionFactory.openSession();

            } catch (HibernateException e) {
                  System.err.print(e);
                  destroy();
                  return null;
            }
      }
      public static void destroy() {
            timer = 0;
            try {
                  //необходимо вызывать, т.к. иначе будут утечки памяти
                  sessionFactory.close();
            } catch (Exception e) {
                  e.printStackTrace();
            }
            sessionFactory = null;
      }
}
```

Данный класс проверяет наличие подключения к БД и сохраняет время проверки. Если на момент проверки класс «простаивает» больше 10 минут, то идет повторное подключение к базе данных. В случае удаления класса он предотвращает утечки памяти. Хотя в Hibernate есть средства для того, чтобы исключить необходимость подключать каждый `hbm.xml` **файл по отдельности, всё же лучше это делать вручную во избежание указанных выше ошибок.**
Этот класс обрабатывает ошибки, которые возникали на практике при работе с Hibernate, JDBC драйвером MySQL и сервером баз данных MySQL.

Взаимодействие с БД

При работе с базой данных удобно функции манипулирования с объектами собирать воедино и создавать менеджеры транзакций.

```
/* пример # 9: инициализация mapping и подключение к БД : StudentDAO.java */
package com.hib;
import net.sf.hibernate.*;
public class StudentDAO {
      //проверка на существование записи в базе данных
      public static Boolean studentExists(String login) {
            Session session =
                  ConnectionFactory.getOrInitSession();
            Student _student = null;
            try {
                  // создание запроса к БД
                  Query query =
                        session.createQuery(
                  "from Student a where a.login = :login");
                  query.setParameter("login", login);
/*этот метод позволяет получать уникальные результаты. Необходимо обеспечить уникальность результатов на уровне БД, если используется данная функция */
            _student = (Student) query.uniqueResult();
            } catch (HibernateException e) {
                  e.printStackTrace();
                  ConnectionFactory.destroy();
            } finally {
                  if (session != null) {
                        try {
                              session.close();
                        } catch (HibernateException e) {
                              e.printStackTrace();
                              ConnectionFactory.destroy();
                        }
                  }
            }
            if (_student != null) {
                  return new Boolean(true);
```

```
      } else {
            return null;
      }
}
public static Student getStudent(String login,
             String password) {
      Session session =
            ConnectionFactory.getOrInitSession();
      Student _student = null;
      try {
            Query query =
                  session.createQuery(
            "from Student a where a.login = :login");
            query.setParameter("login", login);
      _student = (Student) query.uniqueResult();
      } catch (HibernateException e) {
            e.printStackTrace();
            ConnectionFactory.destroy();
      } finally {
            if (session != null) {
                  try {
                        session.close();
                  } catch (HibernateException e) {
                        e.printStackTrace();
                        ConnectionFactory.destroy();
                  }
            }
      }
if (_student != null) {
      if (password.equals(_student.getPassword())) {
                  return _student;
      } else
                  return null;
} else
            return null;
}
//обновление учетной записи студента
//для возможности таких обновлений прописывался тег id в mapping
public static void updateStudent(Student _student) {
      Session session =
            ConnectionFactory.getOrInitSession();
      try {
            Transaction tx = null;
            tx = session.beginTransaction();
            session.update(_student);
            tx.commit();
      } catch (HibernateException e) {
            e.printStackTrace();
```

```
                tx.rollback();
                ConnectionFactory.destroy();
        } finally {
                if (session != null) {
                        try {
                                session.close();
                        } catch (Exception e) {
                                e.printStackTrace();
                                ConnectionFactory.destroy();
                        }
                }
        }
}
//добавление учетной записи студента
// в этом случае работает директива native из mapping
public static Boolean addStudent(Student _student) {
        Session session =
                ConnectionFactory.getOrInitSession();
        try {
                Transaction tx = null;
                tx = session.beginTransaction();
                session.save(_student);
                tx.commit();
        } catch (HibernateException e) {
                e.printStackTrace();
                tx.rollback();
                ConnectionFactory.destroy();
                return null;
        } finally {
                if (session != null) {
                        try {
                                session.close();
                        } catch (Exception e) {
                                e.printStackTrace();
                                ConnectionFactory.destroy();
                                return null;
                        }
                }
        }
        return new Boolean(true);
}
//удаление учетной записи студента
public static void deleteStudent(Student _student) {
        Session session =
                ConnectionFactory.getOrInitSession();
        try {
                Transaction tx = null;
                tx = session.beginTransaction();
```

```
                session.delete(_student);
                tx.commit();
            } catch (HibernateException e) {
                e.printStackTrace();
                tx.rollback();
                ConnectionFactory.destroy();
            } finally {
                if (session != null) {
                    try {
                        session.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                        ConnectionFactory.destroy();
                    }
                }
            }
        }
}
```

Класс **SessionFactory** относится к потокобезопасным классам, поэтому обычно его объект будет создаваться один раз и храниться в статической переменной. Если же возникли проблемы с базой данных, с пулом соединений либо с освобождением ресурсов, которые хранятся в объекте **SessionFactory**, то, возможно, придется поддержать вышеприведенный подход.

Для управления **Session** используются два похода:

Session для каждой операции с базой данных, либо **Session per request** в контексте транзакции.

Начиная с версии 3.1, в Hibernate появился новый параметр **hibernate.current_session_context_class**, который может принимать одно из трех коротких значений, **"jta"**, **"thread"** и "managed", либо классов, которые реализуют интерфейс **org.hibernate.context.CurrentSessionContext** и будут ответственны за открытие/закрытие сессии.

Если данный параметр установлен в конфигурационном файле, то в коде для получения сессии нужно только вызвать метод **SessionFactory.getCurrentSession()**, который либо вернет уже привязанную к данному контексту сессию, либо создаст новую.

Установив свойство в конфигурационном файле

<property name= "current_session_context_class">thread </property> метод **deleteStudent()** можно переписать следующим образом.

```
public static void deleteStudent(Student _student) {

            Session session =
                ConnectionFatory.sessionFactory
                .getCurrentSession();
```

```
        try {
              Transaction tx = null;
              tx = session.beginTransaction();
              session.delete(_student);
              tx.commit();
        } catch (HibernateException e) {
              e.printStackTrace();
              tx.rollback();
        }
  }
```

В данном случае управление соединением будет производиться классом **org.hibernate.context.ThreadLocalSessionContext**, который будет использовать соединение, пока метод **beginTransaction()** не был вызван, и соответственно отпустит соединение при вызове метода **commit()** либо **rollback()**.

Запуск из-под Apache Ant

Для того чтобы запустить проект из-под Apache Ant, необходимо, чтобы папки проекта были организованы следующим образом:

```
+lib
   <Hibernate and third-party libraries>
+src
   <All source files and packages including hbm files>
   hibernate.cfg.xml
build.xml
```

Файл **build.xml** служит руководством к действию для Ant. Изначально он должен выглядеть следующим образом:

```
<project name="hibernate-tutorial" default="compile">
   <property name="sourcedir" value="${basedir}/src"/>
   <property name="targetdir" value="${basedir}/bin"/>
   <property name="librarydir" value="${basedir}/lib"/>
   <path id="libraries">
         <fileset dir="${librarydir}">
               <include name="*.jar"/>
         </fileset>
   </path>
   <target name="clean">
         <delete dir="${targetdir}"/>
         <mkdir dir="${targetdir}"/>
   </target>
   <target name="compile" depends=
                     "clean, copy-resources">
         <javac srcdir="${sourcedir}"
               destdir="${targetdir}"
               classpathref="libraries"/>
   </target>
   <target name="copy-resources">
```

```
            <copy todir="${targetdir}">
                  <fileset dir="${sourcedir}">
                        <exclude name="**/*.java"/>
                  </fileset>
            </copy>
      </target>
</project>
```

Далее выполняется команда `ant`, находясь в папке проекта или явно указывая рабочую папку. Если компиляция прошла успешно, то в файл **`build.xml`** необходимо добавить следующие строки:

```
<target name="run" depends="compile">
   <java fork="true" classname="ClassName"
                  classpathref="libraries">
            <classpath path="${targetdir}"/>
   </java>
</target>
```

где параметр **`ClassName`** содержит имя класса, который можно запустить (есть метод **`main`**, правильно объявленный). В конце концов, для запуска используется команда **`ant run`**.

Приложение 6

STRUTS

Проект Struts был запущен в мае 2000 г. К. Макклэнаном (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.

Загрузить последнюю версию **`struts-2.0.9-bin.zip`** можно по адресу:

`http://struts.apache.org`

Три основных класса, задействованные при обработке запроса:

`org.apache.struts.action.ActionServlet` – класс, связывающий между собой все части MVC-шаблона;

`org.apache.struts.action.Action` – класс-обработчик запроса;

`org.apache.struts.action.ActionForm` – класс, предназначенный для получения данных, которые приходят с клиентской стороны? либо для их отображения в браузере пользователя.

Запрос к серверу обрабатывается классом-контроллером **`ActionServlet`** в соответствии с настройками в файле **`web.xml`**. Запрос на сервер выполняет пользователь нажатием кнопки Submit, введением URL в поле браузера, вызовом submit-формы на JavaScript и пр. Во время инициализации главный контроллер считывает (parse) конфигурационный файл **`struts-config.xml`**, который

однозначно определяет все соответствия и альтернативы для всех запросов данного приложения и упаковывает их в объект класса
org.apache.struts.action.ActionMapping.

Для запроса контроллер находит соответствующие ему **ActionForm**- и **Action**-классы. Первым создается ActionForm, если такой объект еще не создавался и не находится ни в одной из областей видимости (сессия, запрос). Далее поля объекта ActionForm заполняются данными, которые пришли с запросом, т.е. для каждого параметра, содержащегося в запросе, вызывается соответствующий set-метод. Например, если в запросе есть параметр **login=goch**, то в **ActionForm** будет вызван метод **setLogin()** с передачей значения параметра. В Struts существует возможность заполнения структуры объектов. Пусть в **ActionForm** с помощью методов **setCompany()** и **getCompany()** передается объект класса Company, полем которого является коллекция объектов типа **Employee**, для каждого из которых определено поле типа **Address**. Тогда для заполнения поля адреса одного из работников достаточно присутствия параметра **company.employees[n].address.phone**, что приведет к последовательному вызову следующих методов:
getCompany().getEmployees().getItem(n).getAddress().setPhone().

Заполнив форму данными, сервлет-контроллер Struts направляет HTTP-запрос к соответствующему запрашиваемому URL-наследнику класса **Action.**

Создается только один экземпляр **Action** для всех пользователей, поэтому глобальные переменные должны быть синхронизированы либо ограничены в использовании, передавая в метод **execute()** заполненный объект **ActionForm** и **ActionMapping**. Класс **Action** отвечает за вызов Model и перенаправление к необходимому элементу View в зависимости от результата действия.

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.

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.

Простое приложение

Приведенный в главе «Java Server Pages» пример распределенного приложения входа в систему с проверкой логина и пароля можно легко переписать с использованием технологии Struts. Следует обратить внимание на то, что использование технологии Struts упрощает организацию приложения, поэтому диаграмма классов будет выглядеть проще.

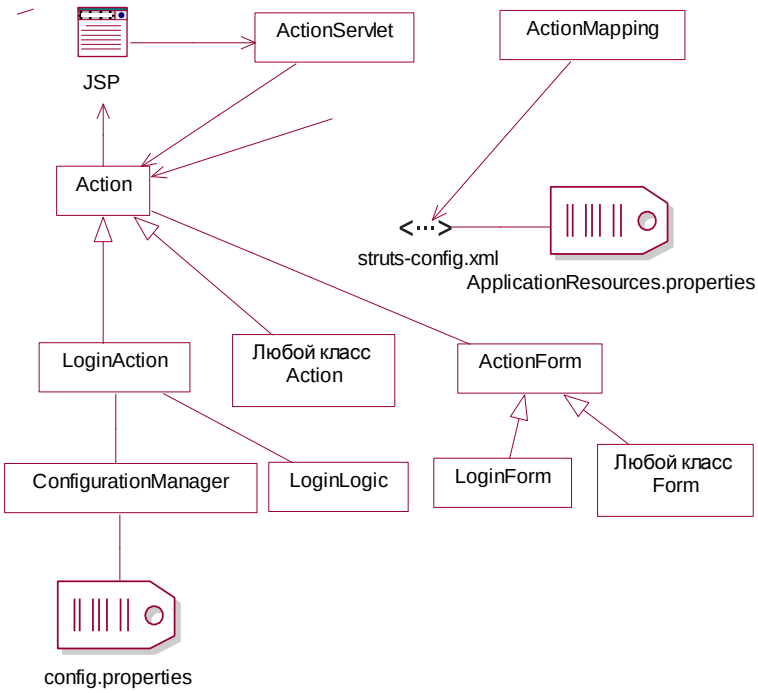


Рис. 1. Диаграмма взаимодействия классов и JSP в Struts-приложении.

```
/* пример # 1: Action класс : LoginAction.java */
package by.bsu.famcs.jspservlet;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

public class LoginAction extends Action {
      public ActionForward execute(ActionMapping mapping,
                                   ActionForm form, HttpS-
                              ervletRequest request, HttpS-
                              ervletResponse response)
                    throws IOException, ServletException {
            ActionMessages errors = new ActionMessages();

            LoginForm actionForm = (LoginForm)form;
            String login = actionForm.getLogin();
            String password = actionForm.getPassword();
```

```
      if (login != null && password != null) {
            if (LoginLogic.checkLogin(login, password)) {
                  return mapping.findForward("success");
            } else {
errors.add(ActionMessages.GLOBAL_MESSAGE,
new ActionMessage("error.login.incorrectLoginOrPassword"));
                  saveErrors(request, errors);
                              }
      }
      //загрузка формы для логина
      return mapping.findForward("loginAgain");
    }
}
```

Класс **LoginForm**, объект которого представляет форму, соответствующую странице ввода логина и пароля, выглядит следующим образом:

```
/* пример # 2: класс хранения информации, передаваемой из login.jsp :
LoginForm.java */
package by.bsu.famcs.jspservlet;
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;
            //очищение полей формы
      public void reset(ActionMapping mapping,
                        HttpServletRequest request) {
            super.reset(mapping, request);
            this.password = null;
      }
      public String getLogin() {
            return login;
      }
      public String getPassword() {
            return password;
      }
      public void setLogin(String login) {
            this.login = login;
      }
      public void setPassword(String password) {
            this.password = password;
      }
   }
```

Класс **LoginLogic**, представляющий бизнес-логику, не был изменен. Его необходимо перенести в соответствующую папку данного проекта Struts. Класс **ConfigurationManager** изменился в сторону уменьшения атрибутов, так как часть информации хранится теперь в **struts-config.xml**.

*/*пример # 3: служебный класс, извлекающий из файла config.properties информацию о драйвере и расположении БД : ConfigurationManager.java */*

```
package by.bsu.famcs.jspservlet.manager;
import java.util.ResourceBundle;

public class ConfigurationManager {
      private static ConfigurationManager instance;
      private ResourceBundle resourceBundle;
      private static final String BUNDLE_NAME = "config";
      public static final String DATABASE_DRIVER_NAME =
"DATABASE_DRIVER_NAME";
      public static final String DATABASE_URL =
"DATABASE_URL";

      public static ConfigurationManager getInstance() {
            if (instance == null) {
                  instance = new ConfigurationManager();
                  instance.resourceBundle =
                  ResourceBundle.getBundle(BUNDLE_NAME);
            }
            return instance;
      }
      public String getProperty(String key) {
            return (String)resourceBundle.getObject(key);
      }
}
```

Далее приведено содержимое файла *config.properties:*

```
################################
## Application configuration ##
################################
DATABASE_DRIVER_NAME=com.mysql.jdbc.Driver
DATABASE_URL=jdbc:mysql://localhost:3306/db1?user=
root&password=pass
```

JSP-страницы были изменены следующим образом:

<%--пример # 4: стартовая страница в оформлении Struts :index.jsp --%>

```
<%@ page language="java" contentType="text/html; char-
set=ISO-8859-1"  pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html><head>
<meta http-equiv="Content-Type" content="text/html; char-
set=ISO-8859-1">
<title>Index JSP</title>
</head>
<body>
<a href="login.do">Controller</a>
</body></html>
```

<%--пример # 5: страница запроса логина и пароля :login.jsp --%>

```
<%@ page language="java" contentType="text/html;
```

```
charset=ISO-8859-1"   pageEncoding="ISO-8859-1"%>
<%-- Подключение библиотек тегов Struts--%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html:html locale="true">
<head>
<meta http-equiv="Content-Type" content="text/html; char-
set=ISO-8859-1">
<%-- 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>
<h3><bean:message key="jsp.login.header"/></h3>
<hr/><%-- форма для логина --%>
<html:form  action="/login" method="POST">
      <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>
<%-- пользовательский тег для отображения ошибок списком, открываю-
щие/закрывающие значения которого берутся из файла ресурсов --%>
<html:errors/>
<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(ActionMessages.GLOBAL_MESSAGE,
new ActionMessage("error.login.incorrectLoginOrPassword"));
```

Для данного приложения файл ресурсов может быть создан в виде:

пример # 6: файл ресурсов : ApplicationResources.properties
header и footer, которые будут использоваться для
обрамления ошибок, выдаваемых тегом <errors/>.
```
errors.header=<ul>
errors.footer=</ul>
errors.prefix=<li>
errors.suffix=</li>
```
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
jsp.main.hello=Hello
```

текстовая информация на error.jsp
```
jsp.error.title=Error
jsp.error.header=Error
jsp.error.returnToLogin=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 # 7: 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 1.2//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
      <data-sources>
      <!--источники данных к БД. Как правило, это организация пула соединений. Пример приведен в конце раздела-->
      </data-sources>
<!-- ========== Form Bean Definitions -->
            <form-beans>
<form-bean  name="loginForm"
            type="by.bsu.famcs.jspservlet.LoginForm" />
    </form-beans>
```

<!-- ============== Global Exception Definitions (если есть)-->

```
    <global-exceptions>
            </global-exceptions>
<!-- ============== Global Forward Definitions(если есть) -->
      <global-forwards>
            </global-forwards>
<!-- ================ Action Mapping Definitions -->
      <action-mappings>
            <!-- Action для процесса логина -->
<action name="loginForm"
        path="/login"
        scope="request" <!—- Задается область видимости формы. Часто необходимо, чтобы форма лежала в сессии, а не в запросе. -->
        type="by.bsu.famcs.jspservlet.LoginAction"
        validate="false">
<!-- форварды на JSP, доступные из данного action -->
<forward name="againlogin" path="/WEB-INF/jsp/login.jsp" />
<forward name="success" path="/WEB-INF/jsp/main.jsp" />
            </action>
      </action-mappings>
<!-- ==================== Message Resources Definitions -->
<!-- имя файла ресурсов .properties, в котором будет храниться вся статическая текстовая информация для приложения. Путь берется относительно папки classes. -->
      <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 `login`, ассоциированные с путями вызова страниц **/jsp/login.jsp** и **/jsp /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 # 8: 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 контроллера, указывающее, что
контроллер будет вызываться, если адрес заканчивается на .do -->
      <servlet-mapping>
          <servlet-name>action</servlet-name>
          <url-pattern>*.do</url-pattern>
      </servlet-mapping>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
<!-- подключение стандартных struts-ских библиотек пользовательских 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>
```

Чтобы получить пул соединений, в приложении следует заполнить **data-source** в **struts-config.xml**, следующим образом:

пример # 9: организация пула соединений : data-sources

```
<data-sources>
   <data-source
type="org.apache.commons.dbcp.BasicDataSource">
      <set-property
        property="driverClassName"
        value="com.mysql.jdbc.Driver" />
      <set-property
        property="url"
        value="jdbc:mysql://localhost/имя_бд" />
      <set-property
        property="username"
        value="root" />
      <set-property
        property="password"
        value="pass" />
      <set-property
        property="maxActive"
        value="10" />
      <set-property
        property="maxWait"
        value="5000" />
      <set-property
        property="defaultAutoCommit"
        value="false" />
      <set-property
        property="defaultReadOnly"
        value="false" />
   </data-source>
   </data-sources>
```

Получить данный пул из класса, реализующего **org.apache.struts.action.Action**, можно с помощью метода **getDataSource(HttpServletRequest request)**.

Оставшиеся JSP-страницы были изменены несущественно:

<%-- пример # 10: 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>
<h3><bean:message key="jsp.main.header"/></h3>
```

```
<hr/>
<bean:message key="jsp.main.hello"/>,
            <bean:write name="loginForm" property="login"/>
<hr/>
<a href="login.do"><bean:message
                    key="jsp.error.returnToLogin"/></a>
<html:errors/>
</body>
</html:html>
```

Страница ошибок, к которой осуществляется переход в случае возникновения исключений, модернизирована следующим образом:

```
<%--пример # 11: 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.returnToLogin"/></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/StrutsProject
```

Наследники Action

Существуют альтернативы классу **`Action`**, наследуемые от **`Action`** и облегчающие задачу разработчика.

Класс **`Action`** объявляеет только метод **`execute()`** и ранее использовался в виде ответа на одно действие пользователя, что увеличивало количество классов приложения и усложняло его поддержку.

Класс **`DispatchAction`** позволяет объединить несколько методов работы с одним модулем в одном классе и вызывать их в зависимости от значения параметра запроса. Например, для работы с модулем **`User`** класс

UserDispatchAction мог бы содержать методы **save()**, **delete()**, **list()** и другие.

Параметр, значение которого будет использовано для вызова метода, описывается в **struts-config.xml** в описании **action**.

```
<action path="/ModuleAction"
      type="com.struts.ModuleDispatchAction"
      parameter="method"
      ...
</action>
```

В коде JSP теперь требуется расставить линки с параметром **method**, значение которого должно соответствовать методу, вызываемому в подклассе.

Например:

```
<a href="ModuleAction.do?method=edit&id=1>Edit</a>
<a href="ModuleAction.do?method=view&id=1>View</a>
```

Далее осталось реализовать методы в **ModuleDispatchAction**. Если параметр не содержит значения, то вызовется метод **unspecified()** данного класса. Так же можно реализовать метод **cancelled()**, который будет вызываться при использовании кнопки **<html:cancel/>**.

```
import org.apache.struts.actions.DispatchAction;

public class ModuleDispatchAction extends DispatchAction{
      public ActionForward edit(ActionMapping mapping,
                              ActionForm form,
                              HttpServletRequest request,
                              HttpServletResponse response)
                              throws IOException,
                              ServletException {
            // код для обработки
      }
      public ActionForward view(...)
            throws IOException, ServletException {
            // код для обработки
      }
      ...
      public ActionForward unspecified(...)
            throws IOException, ServletException {

            mapping.findForward("methodNotFound");
      }
      public ActionForward cancelled(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {

            mapping.findForward("cancelPage");
      }
}
```

Класс **LookupDispatchAction** является наследником от **DispatchAction**. В отличие от предыдущего класса, который в основном используется через использование линков на View, данный класс применяется если форма должна иметь несколько различных кнопок для выполнения запроса на сервер.

Пусть на странице JSP с помощью тега **<html:submit/>** расположены кнопки, свойство **property** которых указывает на название параметра, отправляемого на сервер со значением тега **<bean:message/>** после нажатия на эту кнопку.

```
<html:form action="ModuleAction.do">
      ...
      <html:submit property="method">
      <bean:message key=" submit.button.print" />
      </html:submit>
      <html:submit property="method">
      <bean:message key="button.save" />
      </html:submit>
</html:form>
```

Затем так же, как и для класса **DispatchAction**, должен быть задан параметр, который содержит вызываемый метод.

```
<action path="/ModuleAction"
      type="com. struts.ModuleLookupDispatchAction"
      parameter="method"
      ...
</action>
```

Реализуем методы аналогично **DispatchAction** классу, но также реализуем метод **getKeyMethodMap()**, который организует mapping между ключом названия кнопки и методом.

```
import org.apache.struts.actions.LookupDispatchAction;
...
public class ModuleLookupDispatchAction
                  extends LookupDispatchAction {
      private static Map m = new HashMap();
        static {
            m.put("myapp.submit.button.print","print");
            m.put("myapp.submit.button.save","save");
      }
      protected Map getKeyMethodMap(){
            return m;
      }
      public ActionForward print(...)
            throws IOException, ServletException {
            ...
      }
      public ActionForward save(...)
            throws IOException, ServletException {
```

```
            ...
      }
      ...
}
```

Класс **MappingDispatchAction** также является наследником **DispatchAction** класса, где вызываемый метод определяется не по значению параметра, а по запрашиваемому URL-адресу маппинг которого хранится в **struts-config.xml**.

Для одного класса **MappingDispatchAction** может существовать несколько возможных элементов **<action>**:

```
<action path="/userUpdate"
      type="com.struts.UserMappingDispatchAction"
      parameter="update"
...
<action path="/userAdd"
      type="com.struts.UserMappingDispatchAction"
      parameter="create"
...
```

Значение атрибута **parameter** содержит название метода, который должен отреагировать на данный запрос.

И, как в предыдущих случаях, осталось реализовать необходимые методы.

```
import org.apache.struts.actions.MappingDispatchAction;
...
public class UserMappingDispatchAction
                  extends MappingDispatchAction{

      public ActionForward update(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws IOException, ServletException {
            ...
      }
      public ActionForward create(...)
            throws IOException, ServletException {
            ...
      }
      ...
      public ActionForward unspecified(...)
            throws IOException, ServletException {
            mapping.findForward("methodNotFound");
      }
}
```

Библиотеки тегов Struts

Struts Bean Tags

- **cookie**

  Получает значение для указанного **cookie**.

```
<p>Display the properties of our current session ID
cookie (if there is one):</p>

<bean:cookie id="sess" name="JSESSIONID" val-
ue="JSESSIONID-IS-UNDEFINED"/>
```

- **define**

  Создает переменную доступную по значению **id** в одной из областей видимости согласно атрибуту **toScope** (по умолчанию page scope) и назначает ей значение атрибута **value,** либо Java bean по ключу из атрибута **name** или его свойство по атрибуту **property**.

```
 <bean:define id="test1_boolean" name="test1"
     property="booleanProperty" toScope=”session”/>
 <bean:define id="test1_double" name="test1"
     property="doubleProperty"/>
 <bean:define id="test1_float" name="test1"
     property="floatProperty"/>
 <bean:define id="test1_int" name="test1"
     property="intProperty"/>
 <bean:define id="test1_long" name="test1"
     property="longProperty"/>
 <bean:define id="test1_short" name="test1"
     property="shortProperty"/>
 <bean:define id="test1_string" name="test1"
     property="stringProperty"/>
 <bean:define id="test1_value" value="ABCDE"/>
```

- **header**

  Получает значение соответствующего заголовка запроса.

```
 Display the values of the headers included in this re-
 quest.<br><br>
 <%
    java.util.Enumeration names =
      ((HttpServletRequest) request).getHeaderNames();
 %>

 <table border="1">
   <tr>
     <th>Header Name</th>
     <th>Header Value</th>
   </tr>
 <%
   while (names.hasMoreElements()) {
     String name = (String) names.nextElement();
 %>
```

```
    <bean:header id="head" name="<%= name %>"/>
    <tr>
      <td><%= name %></td>
      <td><%= head %></td>
    </tr>
<%
  }
%>
```

- **include**

Включает в ответ данные с указанного источника. Данные хранятся в области видимости **page** под значением атрибута **id**.

```
<bean:include id="index" page="/index.jsp"/>
```

- **message**

Получает локализированное сообщение для указанного ключа из файла ресурсов.

- **page**

Получает значение из контекста страницы.

- **parameter**

Получает значение указанного параметра из запроса.

```
<bean:parameter id="param1" name="param1"/>
```

- **resource**

Получает значение указанного ресурса Web-приложения.

```
<bean:resource id="webxml" name="/WEB-INF/web.xml"/>
```

- **size**

Получает ссылку на массив или коллекцию и создает новый объект типа **java.lang.Integer** со значением, равным размеру массива или коллекции.

```
<bean:size id="dataSize" collection="<%= data %>"/>
```

- **struts**

Получает значение указанного внутреннего конфигурационного объекта Struts.

- **write**

Показывает значение указанного атрибута указанного бина.

Struts Html Tags

- **button, cancel, checkbox, file, form, frame, hidden, image, img, link, multibox, password, radio, reset, submit, text, textarea**

Представляют стандартные html-элементы.

- **errors**

Отображает множество подготовленных сообщений об ошибках.

- **javascript**

Представляет JavaScript-валидацию, основанную на наборе правил для данной формы, описанных в конфигурационном файле на сервере.

- **messages**

Отображает множество подготовленных сообщений.

```
<%
```

```
  ActionMessages messages = new ActionMessages();
  messages.add("property1",
     new ActionMessage("property1message1"));
  messages.add("property2",
     new ActionMessage("property2message1"));
  messages.add("property2",
     new ActionMessage("property2message2"));
  messages.add(ActionMessages.GLOBAL_MESSAGE,
new ActionMessage("globalMessage"));
  request.setAttribute(Action.MESSAGE_KEY, messages);
%>
...
<html:messages property="property1" message="true"
id="msg"
          header="messages.header" foot-
er="messages.footer">
        <tr><td><%= pageContext.getAttribute("msg")
%></td></tr>
</html:messages>
```

- **select, option, optionsCollection**

Представляет собой html-элемент **select** с указанным в **option** элементом списка. С помощью **optionsCollection** можно задать массив или коллекцию, на основе которой строятся элементы списка.

```
<html:select property="singleSelect" size="10">
        <html:option value="Single 0">Single 0
</html:option>
        <html:option value="Single 1">Single 1
</html:option>
        <html:option value="Single 2">Single 2
</html:option>
        <html:option value="Single 3">Single 3
</html:option>
        <html:option value="Single 4">Single 4
</html:option>
        <html:option value="Single 5">Single 5
</html:option>
        <html:option value="Single 6">Single 6
</html:option>
        <html:option value="Single 7">Single 7
</html:option>
        <html:option value="Single 8">Single 8
</html:option>
        <html:option value="Single 9">Single 9
</html:option>
    </html:select>
```

Struts Logic Tags

- **empty, notEmpty**

Тело данного тега выполняется, только если указанное значение равно (не равно) **null** или пусто.

```
<logic:empty name="bean" property="nullProperty">
     empty
</logic:empty>
<logic:notEmpty name="bean" property="nullProperty">
     notEmpty
</logic:notEmpty>
```

- **equal, notEqual**

Тело тега выполняется, если совпадают/не совпадают указанные в теге значения.

- **forward**

Выполняет переход по указанному адресу URL.

- **greaterEqual, lessEqual**

Тело тега выполняется, если первое указанное значение меньше (больше), чем второе значение, или равно ему.

- **greaterThan, lessThan**

Тело тега выполняется, если первое указанное значение меньше (больше), чем второе значение.

- **iterate**

Повторяет тело тега для каждого элемента из указанной коллекции.

```
<%  {
    java.util.ArrayList list =
new java.util.ArrayList();
    list.add("First");
    list.add("Second");
    list.add("Third");
    list.add("Fourth");
    list.add("Fifth");
    pageContext.setAttribute("list", list, PageCon-
text.PAGE_SCOPE);

    int intArray[] = new int[]
     { 0, 10, 20, 30, 40 };
    pageContext.setAttribute("intArray", intArray, Page-
Context.PAGE_SCOPE);
  }%>
…
<h3>Test 1 - Iterate Over A String Array [0..4]</h3>

<ol>
<logic:iterate id="element" name="bean"
property="stringArray" indexId="index">
  <li><em><bean:write
name="element"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 2 - Iterate Over A String Array [0..2]</h3>
```

```
<ol>
<logic:iterate id="element" name="bean" proper-
ty="stringArray" indexId="index"
        length="3">
  <li><em><bean:write
name="element"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 3 - Iterate Over A String Array [3..4]</h3>

<ol>
<logic:iterate id="element" name="bean" proper-
ty="stringArray" indexId="index"
        offset="3">
  <li><em><bean:write
name="element"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 4 - Iterate Over A String Array [1..3]</h3>

<ol>
<logic:iterate id="element" name="bean"
               property="stringArray" indexId="index"
               offset="1" length="3">
  <li><em><bean:write
name="element"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 5 - Iterate Over an Array List</h3>

<ol>
<logic:iterate id="item" name="list" indexId="index">
  <li><em><bean:write
name="item"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 6 - Iterate Over an Array List [0..2]</h3>

<ol>
<logic:iterate id="item" name="list" indexId="index"
       offset="0" length="3">
```

```
  <li><em><bean:write
name="item"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 7 - Iterate Over an Array List [2..4]</h3>

<ol>
<logic:iterate id="item" name="list" indexId="index"
       offset="2" length="3">
  <li><em><bean:write
name="item"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 8 - Iterate Over an int array</h3>

<ol>
<logic:iterate id="item" name="intArray" index-
Id="index">
  <li><em><bean:write
name="item"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 9 - Iterate Over an int array [0..2]</h3>

<ol>
<logic:iterate id="item" name="intArray" indexId="index"
           length="3">
  <li><em><bean:write
name="item"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>

<h3>Test 10 - Iterate Over an int array [2..4]</h3>

<ol>
<logic:iterate id="item" name="intArray" indexId="index"
           offset="2" length="3">
  <li><em><bean:write
name="item"/></em> [<bean:write
name="index"/>]</li>
</logic:iterate>
</ol>
```

- **messagesNotPresent, messagesPresent**

  Выполняет тело тега, если объект типа **ActionMessages** или **ActionErrors** находится (не находится) в запросе.

- **notPresent**

  Тело данного тега выполняется, только если указанное значение присутствует.

```
<logic:present name="bean" property="stringProperty">
     present
</logic:present>
<logic:notPresent name="bean" property ="stringProperty">
     notPresent
</logic:notPresent>
```

Struts Nested Tags

Данная библиотека тегов расширяет возможности тегов всех предыдущих библиотек так, что позволяет определить, в контексте какого nested-тега находится данный тег, тем самым позволяя избежать вызова лишних методов.

Пусть, например, дана страница редактирования объекта класса **Company**, который содержит как поле объект класса **Address** со своими свойствами.

```
<%@ page contentType="text/html;charset=UTF-8"
language="java" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html>
      <body>
            <html:form action="company.do">
                  Company name:
                  <html:text property="company.name"
                  size="60" />
                  <br>
                  Company address:
                  <br>
                  City:
                  <html:text property=
                  "company.address.city" size="60" />
                  <br>
                  Street:
                  <html:text property=
                  "company.address.street" size="60" />
                  <br>
                  …
                  <html:submit/>
            </html:form>
      </body>
</html:html>
```

Здесь можно заметить, что обращение к методу **getCompany()** формы происходит каждый раз. Также неизбежен и повторный вызов метода **getAddress()** класса **Company**.

С использованием nested-тегов данная JSP будет выглядеть как:

```
<%@ page contentType="text/html;charset=UTF-8" lan-
guage="java" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-nested" prefix="nested" %>
<html:html>
      <body>
            <html:form action="company.do">
                  <nested:nest property="company"/>
                        Company name:
                        <html:text property="name"
                        size="60" />
                        <br>
                        Company address:
                        <br>
                        <nested:nest property="address">
                              City:
                              <html:text property="city"
                              size="60" />
                              <br>
                              Street:
                              <html:text property="street"
                              size="60" />
                              <br>
                              …
                        </nested:nest>
                  </nested:nest>
                  <html:submit/>
            </html:form>
      </body>
</html:html>
```

С помощью тега **nest** устанавливается контекст одного из объектов и далее, используя nested-теги, работа ведется именно с объектом, в котором находится текущий nested-тег. Таким образом, с помощью nested-тегов исчезает необходимость повторного вызова методов для получения доступа к объектам, которые уже были доступны ранее.

Приложение 7

ЖУРНАЛ СООБЩЕНИЙ (LOGGER)

В процессе функционирования сложных приложений необходимо вести журнал сообщений и ошибок, чтобы была возможность отследить время входа и выхода пользователя из системы, возникновение исключительных ситуаций и т.д. Существуют различные API регистрации сообщений и ошибок. В практическом программировании для этих целей применяется API Log4j, разработанный в рамках проекта Jakarta Apache.

API Log4j – это инструмент для формирования журнала сообщений (отладочных, информационных, сообщений об ошибках). API Log4j можно загрузить по адресу: `http://logging.apache.org/log4j/`. Перед использованием необходимо зарегистрировать загруженную библиотеку `log4j-1.2.13.jar` в приложении.

Log4j состоит из трех элементов:

- регистрирующего (logger);
- направляющего вывод (appender);
- форматирующего (layout).

Таким образом logger регистрирует и направляет вывод события в пункт назначения, определяемый элементом appender, в формате, заданном элементом layout.

В стандартной библиотеке **`java.util.logging`** также существует возможность журналирования событий. Однако функциональность классов этого пакета несколько уже, чем у классов проекта Log4j, поэтому профессиональные программисты предпочитают использовать последний.

Logger

Основным элементом API регистрации событий и ошибок является регистратор **`Logger`**, который управляет регистрацией сообщений. Вывод регистратора может быть направлен на консоль, в файл, базу данных, GUI-компонент или сокет. Это компонент приложения, принимающий и выполняющий запросы на запись в регистрационный журнал.

Каждый класс приложения может иметь свой собственный logger или быть прикреплен к общему для всего приложения. Регистраторы образуют иерархию, как и пакеты Java. Регистратор может быть создан или получен с помощью статического метода **`getLogger(String name)`**, где **`name`** – имя пакета. В вершине иерархии находится корневой регистратор. Он всегда существует и у него нет имени. Он может быть получен статическим методом **`getRootLogger()`**.

У каждого регистратора есть уровень сообщения по возрастанию (**`TRACE`**, **`DEBUG`**, **`INFO`**, **`WARN`**, **`ERROR`**, **`FATAL`**), который управляет выводом сообщений. Для вывода сообщений конкретного уровня используются методы **`debug()`**, **`info()`**, **`warn()`**, **`error()`**, **`fatal()`**. Чтобы вывести информацию о возникшем исключении в качестве второго параметра, в вышеперечисленные методы нужно передать объект класса, производного от **`Throwable`**. Для вывода сообщения необходимо, чтобы уровень выводимого сообщения был не ниже, чем уровень

регистратора (**TRACE** < **DEBUG** < **INFO** < **WARN** < **ERROR** < **FATAL**) , т. е. если уровень регистратора **INFO**, то вызов **logger.debug("message")** не даст никакого эффекта, т. к. **DEBUG** < **INFO**. Уровень регистратора можно указать с помощью метода **setLevel(Level level)**, который принимает объект класса **Level**, содержащий одноименные константы для каждого уровня. Если уровень регистратора не указывается, то наследуется уровень от его родителя. Уровень корневого регистратора **DEBUG.**

Существуют следующие методы для вывода сообщений:

log(Priority priority, Object message, Throwable t) – выводит сообщения указанного уровня с информацией об исключительной ситуации **t.**

log(Priority priority, Object message) – выводит сообщения указанного уровня.

Appender и Layout

Вывод регистратора может быть направлен в различные места назначения. Каждому из них соответствует класс, реализующий интерфейс **Appender**. С помощью метода **addAppender(Appender newAppender)** класса **Logger** можно добавить **Apeender** к регистратору. Один регистратор может иметь несколько элементов **Appender**. Вывод на консоль осуществляется с помощью класса **ConsoleAppender**. Класс **FileAppender** используется для вывода сообщений в файл. Для установки файла, в который будет выполняться вывод, нужно передать имя файла в конструктор **FileAppender(Layout layout, String filename)** или метод **setFile(String file)**. По умолчанию любые сообщения, записанные в файл, будут добавляться к уже имеющимся. Изменить это можно с помощью конструктора **FileAppender(Layout layout, String filename, boolean append)** сбросив флаг **append** или с помощью метода **setAppend(boolean append).**

Кроме того, вывод в базу данных можно произвести с помощью класса **JDBCAppender**, в журнал событий ОС – **NTEventLogAppender**, на SMTP-сервер – **SMTPAppender**, на удаленный сервер – **SocketAppender.**

Любой вывод, сделанный в регистраторе, будет направлен всем его предкам. Чтобы этого избежать, в регистраторе следует установить флаг аддитивности с помощью метода **setAdditivity(boolean additive)**. В этом случае вывод будет направлен всем его предкам вплоть до регистратора с установленным флагом аддитивности.

Вывод регистратора может иметь различный формат. Каждый формат представлен классом, производным от **Layout**. Все методы класса **Layout** предназначены только для создания подклассов. В библиотеке определены следующие:

HTMLLayout – вывод в HTML-формате;

XMLLayout – вывод в XML-формате;

SimpleLayout – вывод в простом текстовом формате.

Более информативен вывод в XML-формате.

Установить **Layout** для **FileAppender** или **ConsoleAppender** можно с помощью метода **setLayout(Layout layout)** или передать его в вышеперечисленные конструкторы этих классов.

В приведенном ниже примере производятся регистрация и вывод как обычных информационных сообщений о выполненных действиях, так и сообщений о возникающих ошибках (попытке вычисления факториала отрицательного числа).

```
/* пример # 1: регистратор ошибок : DemoLog.java */
package app6;
import org.apache.log4j.Logger;
import org.apache.log4j.FileAppender;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.Level;
import java.io.IOException;

public class DemoLog {
    static Logger logger = Logger.getLogger(DemoLog.class);

      public static void main(String[] args) {
            try {
                  //возможна и программная настройка
                  factorial(9);
                  factorial(-3);
            } catch (IllegalArgumentException e) {
                  //вывод сообщения уровня ERROR
                  logger.error("negative argument", e);
            }
      }
      public static int factorial(int n) {
            if (n < 0)
                  throw new IllegalArgumentException(
                      "argument " + n +" less then zero");
            //вывод сообщения уровня DEBUG
            logger.debug("Argument n is " + n);
            int result = 1;
            for (int i = n; i >= 1; i--)
                  result *= i;
            //вывод сообщения уровня INFO
            logger.info("Result is " + result);
            return result;
      }
}
```

При этом в корне проекта должен находиться конфигурационный файл **"log4j.xml"** со следующим содержимым:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration
xmlns:log4j='http://jakarta.apache.org/log4j/'>
      <appender name="TxtAppender"
class="org.apache.log4j.FileAppender">
           <param name="File" value="log.txt" />
           <layout class="org.apache.log4j.SimpleLayout"/>
      </appender>
```

```
      <logger name="app6">
            <level value="debug" />
      </logger>
      <root>
            <appender-ref ref="TxtAppender" />
      </root>
</log4j:configuration>
```

Вывод регистратора "**app6.DemoLog**", в файл **log.txt** будет следующим:

```
DEBUG - Argument n is 9
INFO - Result is 362880
ERROR - negative argument ja-
va.lang.IllegalArgumentException: argument -3 less then ze-
ro
      at app6.DemoLog.factorial(DemoLog.java:35)
      at app6.DemoLog.main(DemoLog.java:27)
```

Возможна также и программная настройка логгирования. Тогда в код программы необходимо добавить следующее:

```
      FileAppender appender =
                     new FileAppender(
                        new SimpleLayout(), "log.txt");
      logger.addAppender(appender);
      logger.setLevel(Level.DEBUG);
```

Для вывода на консоль и в XML необходимо добавить следующее в конфигурационный файл:

```
      <appender name="ConsAppender"
class="org.apache.log4j.ConsoleAppender">
           <layout class="org.apache.log4j.SimpleLayout"/>
      </appender>
      <appender name="XMLAppender"
class="org.apache.log4j.FileAppender">
           <param name="File" value="log.xml" />
           <layout class="org.apache.log4j.xml.XMLLayout"/>
      </appender>
      <root>
           <appender-ref ref="ConsAppender" />
           <appender-ref ref="XMLAppender" />
      </root>
```

Или программно:

```
      ConsoleAppender appender2 =
       new ConsoleAppender(new SimpleLayout());
      FileAppender appender3 =
       new FileAppender(new XMLLayout(), "log.xml");
      logger.addAppender(appender2);
      logger.addAppender(appender3);
```

В классе **Logger** объявлены методы, реагирующие на соответствующие события, а именно: **fine()**, **info()**, **warning()**, **log()**, **throwing()** и др.

В приведенном примере запись в файл **log.xml** производится в зависимости от значения остатка после деления.

```
/* пример # 2: стандартный регистратор ошибок : StandartDemoLog.java */
package app6;
import java.io.IOException;
import java.util.logging.FileHandler;
import java.util.logging.Level
import java.util.logging.Logger;

public class StandartDemoLog {
      static Logger log = Logger.getLogger("app6");

      public static void main(String[] args)
                  throws SecurityException, IOException {
            /*инициализация и назначение для вывода логов простого файла,
            который по умолчанию использует XMLFormatter, то есть
            в файле информация будет сохраняться в виде XML */
            FileHandler fh = new FileHandler("log.xml");
            log.addHandler(fh);
      log.setLevel(Level.WARNING);//установка уровня сообщений
            int arr[] = { 5, 6, 1, -4 };
            for (int i = 0; i < arr.length; i++) {
                  int j = arr[i] % 3;
                  switch (j) {
                  case 0:
                        log.fine(arr[i] + "%3 = 0");
                        break;
                  case 1:
                        log.info(arr[i] + "%3 = 1");
                        break;
                  case 2:
                        log.warning(arr[i] + "%3 = 2");
                        break;
                  default:
                        log.severe(arr[i] + "%3 < 0");
                  }
            }
      }
}
```

В результате на консоль будет выведено:

```
17.03.2006 15:39:03 app6.DemoLog main
WARNING: 5%3 = 2
17.03.2006 15:39:03 app6.DemoLog main
INFO: 1%3 = 1
17.03.2006 15:39:03 app6.DemoLog main
SEVERE: -4%3 < 0
```

В файле **log.xml** та же информация будет сохранена в виде

```
<?xml version="1.0" encoding="windows-1251"
standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
```

```
<record>
  <date>2006-03-17T15:39:03</date>
  <millis>1142602743635</millis>
  <sequence>0</sequence>
  <logger>app6</logger>
  <level>WARNING</level>
  <class>app6.DemoLog</class>
  <method>main</method>
  <thread>10</thread>
  <message>5%3 = 2</message>
</record>
<record>
  <date>2006-03-17T15:39:03</date>
  <millis>1142602743736</millis>
  <sequence>1</sequence>
  <logger>app6</logger>
  <level>INFO</level>
  <class>app6.DemoLog</class>
  <method>main</method>
  <thread>10</thread>
  <message>1%3 = 1</message>
</record>
<record>
  <date>2006-03-17T15:39:03</date>
  <millis>1142602743736</millis>
  <sequence>2</sequence>
  <logger>app6</logger>
  <level>SEVERE</level>
  <class>app6.DemoLog</class>
  <method>main</method>
  <thread>10</thread>
  <message>-4%3 < 0</message>
</record>
</log>
```

В несложных учебных проектах проще использовать стандартную библиотеку.

JUnit

Технология JUnit предлагает сделать более тесной связь между разработкой кода и его тестированием. В Java существует возможность проверить корректность работы класса, не прибегая к пробному выводу при отладке программы.

Технология JUnit позволяет вне класса создавать тесты, при выполнении которых произойдет корректное завершение программы в результате неправильной ее работы. Кроме того, будет создано сообщение о возникшей ошибке. Если же результат работы теста не выявит ошибок, то программа продолжит её выполнение.

Тесты создаются в процессе разработки в случае, когда требуется расширить функциональность системы. Разработка завершается, когда тест пройдет успешно. В процессе отладки тесты создаются, если в коде обнаруживается ошибка. Затем отладку следует продолжить до момента корректной отработки теста.

Компиляция и запуск

1. Загрузить JUnit с сервера `www.junit.org`
2. Распаковать загруженный архив
3. Добавить в переменную окружения `CLASSPATH`
   `CLASSPATH=%CLASSPATH%;%JUNIT_HOME%\junit.jar`
4. Для консольного режима запуск выполнить:
   `java junit.textui.TestRunner junit.samples.AllTests`

Пусть класс **ChangedName** манипулирует именами файлов и генерирует имя файла с определенным расширением на основании заданного имени. Тестируемый класс будет содержать метод **String rename(String ext)**, где параметр **ext** – новое расширение файла. Также класс будет иметь конструктор, принимающий имя изменяемого файла.

*/*пример # 3: генерация имени файла с заданным расширением: ChangedName.java */*

```
package app6;

public class ChangedName {
      private String name;
      // реализация
      public ChangedName(String name){
            this.name = name;
      }
      public String rename(String ext) {
            String old = name;
            int dot_pos = old.indexOf('.');
            if (dot_pos > 0)
                  old = old.substring(0, dot_pos);
        return old + "." + ext;
      }
}
```

Далее создан тест для метода **rename()** с учетом того, что он должен задавать имени файла новое расширение.

*/*пример # 4: тестирование метода rename() класса ChangedName: ChangedNameTest.java */*

```
package app6;

public class ChangedNameTest extends TestCase {
   // метод-тестировщик должен называться testИмя, где Имя – это имя тестируемого метода
   public void testRename(){
      ChangedName changed = new ChangedName("report");
      /*метод проверяет, равны ли ожидаемая и полученная строки, и если результатом будет false, то тест завершит работу приложения*/
      assertEquals("report.txt", changed.rename("txt"));
   }
}
```

/ пример # 5: класс, который проверяет, является ли одно число делителем другого: Devisor.java*/*

```
package app6;
```

```
public class Devisor {
      public boolean isDevisor(int num1,int num2){
            if((num2!=0)&&(num1%num2==0))return true; //1
            // if(num1%num2==0) return true; // 2
            else return false;
      }
}
```

Ниже реализован тест, который проверят корректность работы метода **isDevisor()**.

```
/* пример # 6: тестирование метода isDevisor() класса Devisor:
DevisorTest.java*/
package app6;
import junit.framework.TestCase;

public class DevisorTest extends TestCase {
      public void testIsDevisor() {
            Devisor obj = new Devisor();
            boolean result1 = obj.IsDevisor(2,1);
            boolean result2 = obj.IsDevisor(1,0);
            assertEquals(true, result1); // test1

            /*test1 проверяет, действительно ли результат работы метода
            для чисел 2 и 1 равен true. Если это верно, то выполнится следу-
            ющий тест, если нет, то приложение завершит работу, а те-
            стировщик сообщит об ошибке.*/

            assertEquals(false, result2);// test2
      }
}
```

В данном случае программа верна. Если же закомментировать строку 1 и убрать комментарий со строки 2 в классе **Devisor**, то **test1** выполнится корректно, а **test2** завершит работу приложения и выдаст сообщение об ошибке из-за генерации при работе метода необработанного исключения **ArithmeticException**.

```
/*пример # 7: класс позволяет считывать информацию из заданного файла
и преобразовывать её в строку: ReadFile.java*/
package app6;
import java.io.FileReader;
import java.io.IOException;

public class ReadFile {
      public String fileIntoString(String st) {
            String str = "";
            try {
                  FileReader stream = new FileReader(st);
                  int s;
                  while ((s = stream.read()) != -1) {
                        str = str + s;
                  }
```

```
                System.out.println(str);
                stream.close();
            } catch (IOException e) {
                System.err.println(e);
            }
            return str;
      }
}
```

Тест, проверяющий корректную работу этого класса:

```
/* пример # 8: тестирование класса ReadFile: ReadFileTest.java*/
package app609;
import junit.framework.TestCase;

public class ReadFileTest extends TestCase {
      public void testFileIntoString() {
            ReadFile obj = new ReadFile();
            String st =
                  obj.fileIntoString("D:\\temp\\test.txt");
            assertFalse("".equals(st));
/*если файл не существует или не пуст, то программа завершит работу с сообщением об ошибке*/
      }
}
```

В простом пользовательском классе **StringConCat** реализован метод **concat(String s1, String s2)**, который объединяет две строки.

```
/*пример # 9:  тестируемый класс: StringConCat.java*/
package app609;

public class StringConCat {
      // реализация

      public String concat(String st1, String st2) {
            String str = st1 + st2;
            return str;
      }
}
```

Тест, проверяющий корректность работы этого метода:

```
/* пример # 10: тестирование метода concat() класса StringConCat:
StringConCatTest.java*/
package app6;
import junit.framework.TestCase;

public class StringConCatTest extends TestCase {
      public void testConcat(){
            StringConCat obj = new StringConCat();
            String st = obj.concat("Java", "2");
            assertEquals("Java2", st);
      }
}
```

Последняя версия JUnit полностью основана на аннотациях и в явном виде не использует тип **TestCase**.

Приложение 8

APACHE ANT

Apache Ant – это основанный на Java набор инструментов для сборки приложений.

Ant – теговый язык. Он обрабатывает XML-файлы, организованные особым образом. Каждый тег по сути является Java-классом, и есть возможность создавать свои теги или расширять возможности уже имеющихся.

Ant – многоплатформенный, основанный на использовании командной строки продукт, следовательно, возможна интеграция с другими операционными системами.

Вместо того чтобы наследовать функции командной строки, Ant основан на Java-классах. Конфигурационный файл устроен в виде XML, из которого можно вызвать разветвлённую систему целей, состоящую из множества мелких задач. Каждая задача является объектом, который наследует соответствующий интерфейс класса **`Task`**. Всё это даёт возможность переносить программу с платформы на платформу. И если действительно необходимо вызвать какой-либо процесс у Ant есть задача **`<exec>`**, которая позволяет это сделать в зависимости от платформы.

Требования к системе

Ant может быть успешно использован на многих платформах, включая Linux, коммерческие версии Unix, такие как Solaris и HP-UX, Windows 9x и NT, OS/2 Warp, Novell Netware 6 и MacOS X.

Чтобы обрабатывать и использовать Ant, необходимо иметь JAXP-compliant XML-parser (он есть в любой версии JDK) установленным и включённым в **`classpath`** или лежащим в папке с библиотеками Ant. Для работы необходимо также иметь установленный JDK версии 1.2 или выше.

Установка Ant

Для начала работы достаточно скопировать Ant на компьютер и установить необходимые системные переменные.

Пусть Ant установлен на **`c:\ant\.`** Переменные устанавливаются следующим образом:

```
set ANT_HOME=c:\ant
set JAVA_HOME=c:\jdk1.2.2
set PATH=%ANT_HOME%\bin;%PATH%
```

Создание простейшего build-файла

Build-файлы Ant пишутся на языке XML. Каждый **`build`**-файл содержит один проект (**`project`**) и хотя бы одну цель (**`target`**). Цель содержит задачи (**`tasks`**). Каждая задача, встречающаяся в **`build`**-файле, может иметь **`id`** атрибут и может быть позже вызвана по нему. Идентификаторы должны быть уникальными.

Используемые теги:

Project

Тег **`Project`** имеет три атрибута:

Атрибут	Описание	Обязательность
`name`	Имя проекта	Нет
`default`	Цель по умолчанию, которая будет использоваться, если явно не указанно, какую цель выполнять	Да
`basedir`	Основная директория, из которой будут выходить все пути, использующиеся при работе (если она не указана, то будет использоваться текущая директория, в которой находится build-файл)	Нет

Каждый проект содержит одну или несколько целей. Цель представляет собой набор задач, которые необходимо выполнить. При запуске Ant можно выбрать цель, которую(ые) следует выполнить. Если цель не указывать, будет выполнена установленная по умолчанию.

Targets

Цель может зависеть от других целей. Например, имеются две цели: для компиляции и для изъятия файлов с базы данных. Соответственно скомпилировать файлы можно только после того, как они будут извлечены. Ant учитывает такие зависимости.

Следует отметить, что **`depends`**-атрибут Ant только обозначает порядок, в котором цели должны быть выполнены. Ant пробует выполнить цели в порядке, соответствующем порядку их появления в атрибуте **`depends`** (слева направо).

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

Пусть нужно выполнить цель D. Из её атрибута **`depends`** можно узнать, что первой выполнится цель C, затем B и, наконец, A. Неверно: C зависит от B, а B зависит от A, таким образом, первой выполнится цель A, затем B, потом C, а после D.

Цель будет исполнена только один раз, даже если более чем одна цель зависит от неё.

Цель также имеет возможность быть исполненной только в случае, если определённый параметр (**`property`**) был (или не был) установлен. Это позволяет лучше контролировать процесс сборки (например, в зависимости от операционной системы, версии Java и т.д.). Ant только проверяет, установлено ли то либо иное свойство, значение его не важно. Свойство, значением которого является пустая строка, считается заполненным. Например:

```
<target name="build-module-A" if="module-A-present"/>
<target name="build-own-fake-module-A" unless=
            "module-A-present"/>
```

Если не установлены **`if`** и **`unless`** атрибуты, цель будет выполняться всегда.

Опциональный атрибут **description** может быть использован как описание цели и будет выводиться при команде– **projecthelp**.

Target имеет следующие атрибуты:

Атрибут	Описание	Обязательность
name	Имя цели	Да
depends	Разделённый запятыми список имён целей, от которых эта цель зависит	Нет
if	Имя параметра, который должен быть установлен, чтобы эта цель выполнялась	Нет
unless	Имя параметра, который не должен быть установлен, чтобы эта цель выполнялась	Нет
description	Небольшое описание функции **function** цели	Нет

Имя цели должно состоять только из букв и цифр, включая пустую строку "", "," и пробел.

Пример **build-**файла:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<project name="MyProject" default="dist" basedir=".">
    <description>
        Простой пример build файла
    </description>
  <!-- установка глобальных параметров -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>
  <target name="init">
    <!-- Создать марку времени -->
    <tstamp/>
    <!-- Создать структуру build директории, которая будет использоваться при компиляции-->
    <mkdir dir="${build}"/>
  </target>
  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Компиляция java кода из ${src} в ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
  <target name="dist" depends="compile"
        description="генерация дистрибутива" >
    <!-- создание директории для дистрибутива -->
    <mkdir dir="${dist}/lib"/>

    <!-- Положить всё из ${build} в MyProject-${DSTAMP}.jar файл -->
```

```
       <jar   jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
basedir="${build}"/>
     </target>
     <target name="clean"
           description="очищает рабочие каталоги" >
   <!-- Delete the ${build} and ${dist} directory trees -->
       <delete dir="${build}"/>
       <delete dir="${dist}"/>
     </target>
   </project>
```

Некоторым целям было дано описание. Это значит, что командой **projecthelp** будет получен список этих целей с описанием; остальные цели считаются внутренними и не выводятся.

Чтобы всё работало, исходные коды в **src** поддиректории должны располагаться в соответствии с именами их **package.**

Пример результата выполнения:

```
D:\tmp\1>ant
Buildfile: build.xml
init:
    [mkdir] Created dir: D:\tmp\1\build
compile:
    [javac] Compiling 1 source file to D:\tmp\1\build
dist:
    [mkdir] Created dir: D:\tmp\1\dist\lib
      [jar]  Building  jar:  D:\tmp\1\dist\lib\MyProject-
20070815.jar
BUILD SUCCESSFUL
Total time: 3 seconds
```

Property

Свойства в Ant аналогичны переменным в языках программирования тем, что имеют имя и значение. Однако, в отличие от обычных переменных, свойства в Ant не могут быть изменены после их установки: они постоянны.

```
<property name="path"  value="./project"/>
```

Для обращения к этому свойству в остальных местах нашего файла компоновки можно было бы использовать следующий синтаксис:

```
${path}
```

Например:

```
<property name="libpath" value="${path}/lib"/>
```

Ant также позволяет установить переменные в отдельном property-файле. Пример property-файла:

```
#
# A sample "ant.properties" file
#
month=30 days
year=2004
```

и его использования

```
<?xml version="1.0"?>
```

```
<project name="test.properties" default="all" >
   <property file="ant.properties"/>
   <target name="all" description="Uses properties">
         <echo>This month is ${month}</echo>
         <echo>This year is ${year}</echo>
   </target>
</project>
```

Шаблоны

Часто является полезным выполнить эти операции с группой файлов сразу, например, со всеми файлами в указанном каталоге с названиями, заканчивающимися на .java, но не начинающимися с EJB. Пример копирования таких файлов:

```
<copy todir="archive">
   <fileset dir="src">
         <include name="*.java"/>
         <exclude name= "EJB*.java"/>
   </fileset>
</copy>
```

Filter

При работе с текстовыми файлами можно использовать фильтр для вставки любого текста в определенные места.

```
<filterset id="copy.filterset">
    <filter token="version" value="1.1"/>
</filterset>
<target name=”copy”>
    <copy file="file1.txt" tofile="file2.txt" filter-
ing="true">
        <filterset refid="copy.filterset" />
    </copy>
</target>
```

Содержимое исходного текстового файла:

```
# file.txt
Version is @version@
```

В результате получаем:

```
# file.txt
Version is 1.1
```

Path-like структуры

Можно определить типы ссылок **`path`** и **`classpath`**, используя как `":"` (unix-style) так и `";"` (windows-style) как разделитель символов. Ant скорректирует их в требуемые текущей операционной системой.

В случае, когда **`path-like`** значение надо определить, могут использоваться подсоединяемые элементы (nested elements). Это выглядит примерно так:

```
<classpath>
  <pathelement path="${classpath}"/>
  <pathelement location="lib/helper.jar"/>
</classpath>
```

Атрибут **location** определяет отдельный файл или директорию, в то время как атрибут **path** принимает список из **locations**. Атрибут **path** должен использоваться с только с определённым ранее путём.

В качестве сокращения <**classpath**> поддерживает **path** и **location** атрибуты так:

```
<classpath>
  <pathelement path="${classpath}"/>
</classpath>
```

Может быть сокращено до:

```
<classpath path="${classpath}"/>
```

В дополнение **DirSets**, **FileSets** и **FileLists** могут быть использованы как внутренние:

```
<classpath>
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="classes"/>
  <dirset dir="${build.dir}">
    <include name="apps/**/classes"/>
    <exclude name="apps/**/*Test*"/>
  </dirset>
  <filelist refid="third-party_jars"/>
</classpath>
```

Path-like структуры могут содержать ссылки на другие **path-like** структуры с помощью **<path>** элемента:

```
<path id="base.path">
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="classes"/>
</path>

<path id="tests.path">
  <path refid="base.path"/>
  <pathelement location="testclasses"/>
</path>
```

Опции командной строки

Запускать Ant на исполнение той или иной задачи очень просто:

ant [options] [target [target2 [target3] ...]]

Options

-help, -h	Выводит текущее сообщение
-projecthelp, -p	Выводит помощь по проекту
-version	Выводит версию и выходит

-diagnostics	Выводит полезную информацию для диагностики отлова ошибок
-quiet, -q	Очень тихий режим работы
-verbose, -v	Режим, выводящий максимум возможной информации
-debug, -d	Выводить информацию отладки
-lib <path>	Определяет путь, по которому происходит поиск **jar** файлов и **class** файлов
-logfile <file>	Использовать файл для лога
-l <file>	"
-logger <classname>	Класс, который обрабатывает логинг
-noinput	Запрещает интерактивный ввод информации
-buildfile <file>	Использовать данный **build**-файл (по умолчанию берётся **build.xml**)
-file <file>	"
-f <file>	"
-D<property>=<value>	Использовать значение для данного параметра
-keep-going, -k	Выполнять все цели, не имеющие зависимостей при ошибке в одной из них
-propertyfile <name>	Загрузить все параметры из файла с **-D**

Краткий экскурс по задачам Ant

Ant предоставляет слишком много задач, чтобы дать полное описание того, что каждая из них делает. Следующий список дает представление о категориях, на которые можно разделить все задачи.

Archive Tasks
Audit/Coverage Tasks
Compile Tasks
Deployment Tasks
Documentation Tasks
EJB Tasks
Execution Tasks
File Tasks
Java2 Extensions Tasks
Logging Tasks
Mail Tasks
Miscellaneous Tasks
.NET Tasks
Pre-process Tasks
Property Tasks
Remote Tasks
SCM Tasks
Testing Tasks
Visual Age for Java Tasks

Краткое описание основных:

Archive Tasks

Имя задачи	Описание
Jar	Упаковывает в Jar набор файлов
Unzip	Распаковывает **zip** архивы
Zip	Создаёт **zip** архивы

Compile Tasks

Имя задачи	Описание
Javac	Компилирует определённые исходные файлы внутри запущенной Ant'ом VM, или с помощью новой VM, если **fork** атрибут определён
JspC	Запускает JSP-компилятор. Используется для предварительной компиляции JSP-страниц для более быстрого запуска их с сервера, или при отсутствии JDK на нём, или просто для проверки синтаксиса, без установки их на сервер
Wljspc	Компилирует JSP-страницы, используя Weblogic JSP компилятор

Execution Tasks

Имя задачи	Описание
Ant	Запускает Ant для выбранного **build** файла, возможна передача параметров (или их новых значений). Эта задача может быть использована для запуска подпроектов
AntCall	Запускает другую цель внутри того же **build-**файла, по желанию передавая параметры
Exec	Исполняет системную команду. Когда атрибут **os** определён, команда исполняется, только если Ant запущен под определённую систему
Java	Исполняет Java класс внутри запущенной (Ant) VM или с помощью другой, если **fork** атрибут определён

File Tasks

Имя задачи	Описание
Copy	Копирует файл или **Fileset** в новый файл или директорию
Delete	Удаляет как один файл, так и все файлы и поддиректории в определённом каталоге, или набор файлов, определённых одним или несколькими **FileSet'**ами
Mkdir	Создаёт директорию. Не существующие внутренние директории создадутся, если будет необходимость
Move	Переносит файл в новый файл или каталог, или набор(ы) файлов в новую директорию

Miscellaneous Tasks

Имя задачи	Описание
Echo	Выводит текст в **System.out** или в файл
Fail	Выходит из текущей сборки, генерируя **BuildException**, по желанию печатая сообщение
Input	Позволяет пользователю интерактивно вмешиваться в процесс сборки путём вывода сообщений и считывания строки с консоли
Taskdef	Добавляет задачу в проект, после чего она может быть использована в текущем проекте

Property Tasks

Имя задачи	Описание
Available	Устанавливает параметр, если определенный файл, каталог, **class** в **classpath**, или JVM системный ресурс доступен во время выполнения
Condition	Устанавливает параметр, если определённое условие выполняется
LoadFile	Загружает файл в параметр
Property	Устанавливает параметр (по имени и значению), или набор параметров (из файла или ресурса) в проект

Типы

Краткий список основных типов (на самом деле их больше):

DirSet
FileSet
PatternSet

DirSet представляет собой набор каталогов. Эти каталоги могут находиться в базовой директории, и поиск осуществляется по шаблону. **DirSet** может находиться внутри некоторых задач или выноситься в проект с целью дальнейшего к нему обращения по ссылке.

PatternSet (набор шаблонов) может быть использован как внутренняя задача. В дополнение **DirSet** поддерживает атрибуты **PatternSet** и внутренние **<include>**, **<includesfile>**, **<exclude>** и **<excludesfile>** элементы **<patternset>**.

Атрибут	Описание	Обязательность
dir	Корневая директория этого **DirSet**	Да
includes	Список шаблонов (через запятую или пробел) для каталогов, которые должны быть включены, если атрибут пропущен, все каталоги включаются	Нет
includesfile	Имя файла; каждая строчка этого файла понимается как шаблон для включения в поиск	Нет

excludes	Список шаблонов (через запятую или пробел) для каталогов, которые должны быть исключены, если атрибут пропущен, все каталоги включаются	Нет
excludesfile	Имя файла; каждая строчка этого файла понимается как шаблон для исключения из поиска	Нет
casesensitive	Определяет влияние регистров для шаблонов (**true|yes|on** или **false|no|off**)	Нет; по умолчанию **true**

Примеры:

```
<dirset dir="${build.dir}">
  <include name="apps/**/classes"/>
  <exclude name="apps/**/*Test*"/>
</dirset>
```

Группирует все каталоги с именем **classes**, найденные под **apps** поддиректорией **${build.dir}** директории, пропуская те, что имеют текст **Test** в своём имени.

```
<dirset dir="${build.dir}">
  <patternset id="non.test.classes">
    <include name="apps/**/classes"/>
    <exclude name="apps/**/*Test*"/>
  </patternset>
</dirset>
```

Делает то же самое, но была установлена ссылка на **<patternset>**.

```
<dirset dir="${debug_build.dir}">
  <patternset refid="non.test.classes"/>
</dirset>
```

Таким образом можно к ней обратиться.

FileSet

FileSet есть набор файлов. Эти файлы могут быть найдены в дереве каталогов, начиная с базовой директории и удовлетворяющие шаблонам. **FileSet** может находиться внутри некоторых задач или выноситься в проект с целью дальнейшего к нему обращения по ссылке.

Атрибут	Описание	Обязательность
dir	Корень каталогов этого **FileSet**	Один должен быть обязательно
file	Сокращение для определения **Fileset** из одного файла	
includes	Список шаблонов (через запятую или пробел) для каталогов, которые должны быть включены, если атрибут пропущен, все каталоги включаются	Нет
includesfile	Имя файла; каждая строчка этого файла понимается как шаблон для включения в поиск	Нет

excludes	Список шаблонов (через запятую или пробел) для каталогов, которые должны быть исключены, если атрибут пропущен, все каталоги включаются	Нет
excludesfile	Имя файла: каждая строчка этого файла понимается как шаблон для исключения из поиска	Нет
casesensitive	Определяет влияние регистров для шаблонов (**true|yes|on** или **false|no|off**)	Нет; по умолчанию **true**

Примеры:

```
<fileset dir="${server.src}" casesensitive="yes">
  <include name="**/*.java"/>
  <exclude name="**/*Test*"/>
</fileset>
```

Группирует все файлы в каталоге ${**server.src**}, являющимися Java кодами и не содержащими текста **Test** в своём имени.

PatternSet

Шаблоны могут быть сгруппированы в наборы и позже использованы путём обращения по ссылке. **PatternSet** может находиться внутри некоторых задач или выноситься в проект с целью дальнейшего к нему обращения по ссылке.

Шаблоны могут определяться с помощью внутренних **<include>**, или **<exclude>** элементов или с помощью следующих атрибутов:

Атрибут	Описание
includes	Список шаблонов (через запятую или пробел) для каталогов, которые должны быть включены, если атрибут пропущен, все каталоги включаются
includesfile	Имя файла; каждая строчка этого файла понимается как шаблон для включения в поиск. Можно задавать несколько
excludes	Список шаблонов (через запятую или пробел) для каталогов, которые должны быть исключены, если атрибут пропущен, все каталоги включаются
excludesfile	Имя файла; каждая строчка этого файла есть шаблон для исключения из поиска. Можно задавать несколько

Параметры определённые как внутренние элементы **include** и **exclude**

Эти элементы определяют единичный шаблон включений или исключений.

Атрибут	Описание	Обязательность
name	Шаблон, который или включается, или исключается	Нет
if	Использовать этот шаблон, если параметр установлен	Нет
unless	Использовать этот шаблон, если параметр не установлен	Нет

Если брать шаблоны извне, то нужно использовать **includesfile/excludesfile** атрибуты или элементы.

Атрибут	Описание	Обязательность
name	Имя файла, который содержит шаблоны	Нет
if	Читать этот файл, только если параметр установлен	Нет
unless	Читать этот файл, только если параметр не установлен	Нет

Атрибут **patternset** может содержать внутри другой **patternset**.

Примеры:

```
<patternset id="sources">
  <include name="std/**/*.java"/>
  <include name="prof/**/*.java" if="professional"/>
  <exclude name="**/*Test*"/>
</patternset>
```

Включает файлы в подкаталоге **prof**, если параметру **professional** установлено некоторое значение.

Следующих два набора:

```
<patternset includesfile="some-file"/>
```

и

```
<patternset>
  <includesfile name="some-file"/>
<patternset/>
```

одинаковы.

```
<patternset>
  <includesfile name="some-file"/>
  <includesfile name="${some-other-file}"
                if="some-other-file"
  />
<patternset/>
```

Будет читать шаблоны из файлов, один из них только тогда, когда параметр **some-other-file** установлен.

Создание собственной задачи

Создаётся Java-класс, наследуемый от **org.apache.tools.ant.Task** или другого сходного с ним класса.

Для каждого атрибута пишется установочный метод. Он должен быть **public void** и принимать один-единственный параметр. Имя метода должно начинаться с **set**, предшествующего имени атрибута, с первым символом имени, написанным в верхнем регистре, а остальное в нижнем. Так, чтобы подключить атрибут с именем **file**, следует создать метод **setFile()**.

Если задача будет содержать другие задачи в качестве внутренних, класс должен наследоваться от **org.apache.tools.ant.TaskContainer**.

Для каждого внутреннего элемента указывается **create()**, **add()** или **addConfigured()** метод. Метод **create()** должен быть **public** методом,

который не принимает аргументов и возвращает **Object** тип. Метод **add()** (или **addConfigured()**) метод должен быть **public void** методом, который принимает единственный аргумент **Object** типа с конструктором без аргументов.

public void execute() – метод без аргументов, является главным методом в задаче, который генерирует **BuildException.**

В качестве примера можно создать собственную задачу, в результате выполнения которой будет выводиться сообщение в поток **System.out**. У задачи будет один атрибут с именем **message**.

```
package com.mydomain;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

public class MyVeryOwnTask extends Task {
      private String msg;

      // метод, выполняющий задачу
      public void execute() throws BuildException {
            System.out.println(msg);
      }
      // метод установки атрибута
      public void setMessage(String msg) {
            this.msg = msg;
      }
}
```

Чтобы добавить задачу в проект, следует удостовериться, что класс, который подключается, находится в **classpath**.

После добавления **<taskdef>**-элемента в проект элемент добавится в систему, используя предыдущую задачу в оставшемся **build**-файле.

```
<?xml version="1.0"?>

   <project name="OwnTaskExample" default="main"
basedir=".">
     <taskdef name="mytask" class-
name="com.mydomain.MyVeryOwnTask"/>

     <target name="main">
       <mytask message="Hello World! MyVeryOwnTask
works!"/>
     </target>
   </project>
```

Чтобы использовать задачу повсеместно в проекте, нужно поместить **<taskdef>** внутри проекта, а не задачи. Следует использовать **classpath** атрибут **<taskdef>**, чтобы указать, откуда брать код задачи.

```
   <?xml version="1.0"?>
```

```
   <project name="OwnTaskExample2" default="main"
basedir=".">

     <target name="build" >
       <mkdir dir="build"/>
       <javac srcdir="source" destdir="build"/>
     </target>

     <target name="declare" depends="build">
       <taskdef name="mytask"
           classname="com.mydomain.MyVeryOwnTask"
           classpath="build"/>
     </target>

     <target name="main" depends="declare">
      <mytask message="Hello World! MyVeryOwnTask works!"/>
     </target>
   </project>
```

В этой книге были рассмотрены основные возможности языка Java применительно к созданию приложений и распределенных систем. 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
```

Приложение 9

ПОРТЛЕТЫ

Порталы – это следующее поколение приложений, предоставляющих доступ к бизнес-приложениям через Web-интерфейс всем видам клиентов. Порталы предоставляют пользователям сайта единую точку доступа к различным приложениям и ресурсам. Независимо от того, где информация находится и какой тип имеет, портал представляет ее в виде, дружественном для конечного пользователя. Полноценное портал-решение должно обеспечивать пользователей удобным доступом ко всему, что необходимо для успешного выполнения их работы. Портал – web-сервер, который объединяет, настраивает и персонализирует содержимое, чтобы дать представление о корпоративной информационной системе.

Возможности порталов

Портал предоставляет всесторонний подход и доступ к инструментам и сервисам, доступным через Web-интерфейс.

Портлеты – основные блоки, из которых строится портал, в то же время это полноценные приложения, соответствующие шаблону проектирования Model-View-Controller. Каждый портлет разрабатывается, развертывается, управляется и отображается независимо от других портлетов.

У портлетов может быть несколько состояний и видов в соответствии с событийными возможностями. Базируясь на модели контейнера, портлеты выполняются внутри контейнера портлетов. Они могут перенаправить запрос или ошибки браузеру.

Портальный сервер

Портальный сервер обрабатывает запросы клиентов. Как сервер Web-приложений имеет Web-контейнер для управления выполняющимися компонентами, так и портальный сервер имеет контейнер для управления выполнением портлетов. Portlet API **– это расширение над спецификацией для сервлетов, т.е. контейнер портлетов по определению является и Web-контейнером.**

Портлет

Портлет – это подключаемый Web-компонент (вполне самостоятельное приложение, средой выполнения которого служит портальный сервер), обеспечивающий динамическое содержимое как часть визуального пользовательского интерфейса. Portlet API наследует возможности Servlet API (где-то расширяя эти возможности, а где-то ограничивая, например, сессия портала не повторяет до конца концепцию сервлетной сессии), и вследствие этого портлеты обладают возможностями сервлетов, а также дополнительными свойствами.

В отличие от сервлетов, портлеты обрабатывают запросы **doView()**, **doConfigure**() и **doEdit()**, приходящие не от браузера, а от портального сервера.

Возможности портлетов:

- **встроенная поддержка автоматического использования различных JSP-страниц для различных пользовательских устройств, таких как настольные компьютеры, Palm-компьютеры с ограниченными Web-браузерами, PDA и мобильные телефоны;**
- **назначать права пользователям групп на использование портлетов. В случае отсутствия оных они даже не будут видеть портлеты;**
- **создание сохраняемых между сессиями пользовательских настроек;**
- **публикация в виде Web-сервиса;**
- **разделение сложных приложений на задачи, когда группа тесно связанных задач равняется одному портлету;**
- **добавление новых функций к приложению;**
- **хорошая совместимость с брандмауэрами (firewalls), так как они (портлеты) используют стандартные Web-протоколы для получения и отображения информации;**
- **одноразовая установка и настройка портлета для пользователей.**

Сходства и различия сервлетов и портлетов

Из-за того, что Portlet API – это расширение Servlet API, у них есть некоторые сходства и различия.

Сходства между сервлетами и портлетами:

- **относятся к J2EE Web-компонентам;**
- **управляются контейнерами;**
- **генерируют динамическое Web-содержимое при помощи запросов и ответов.**

Различия между сервлетами и портлетами:

- **портлеты генерируют часть документа, в то время как сервлеты генерируют его полностью;**
- **за счёт того, что операции кодирования URL выполняются на стороне сервера, пользователь не может обратиться к нему напрямую, зная имя портлета: портлет ведь часть страницы, поэтому знания одного URL мало;**
- **портлеты имеют несколько иную схему управления запросами, которые делятся на запросы выполнения действий и запросы генерирования содержимого;**

- портлеты придерживаются стандартного набора состояний, которые определяют их контекст работы и правила генерации содержимого.

Портлеты превосходят сервлеты по следующим направлениям:

- имеют расширенный механизм для управления и сохранения своей конфигурационной информации;
- есть доступ к информации о пользовательском профиле, не таком тривиальном, как предоставляют сервлеты.

Приложения с использованием портлетов – это расширенные Web-приложения. Таким образом, оба типа приложений развертываются из WAR-файла и содержат дескриптор Web-приложения (файл `web.xml`). Портлет-приложения также содержат и дескриптор портлета (файл `portlet.xml`).

Жизненный цикл

Как и у сервлетов, жизненный цикл портлетов управляется контейнером,
и у него есть метод `init()`, который используется для инициализации всех данных, необходимых для корректной работы портлета (создание ресурсов, конфигурирование и т.д.).

Метод `init()` в качестве параметра принимает объект, который реализует интерфейс `PortletConfig`, и этот объект предоставляет необходимые для инициализации параметры. Он может быть использован для получения ссылки на объект, реализующий интерфейс `PortletContext`.

Разработчики портлетов, как правило, не тратят много времени на беспокойство о сложности обработки исключений инициализации портал-контейнера из-за того, что при происхождении оного разработчик реагирует на это должным образом (выясняя обстоятельство, при котором оно произошло). Как правило, нет ничего хуже, чем `UnavailableException`, которое обозначает, что портлет временно или постоянно недоступен. При создании портлета доступа к окружающему коду, например к контейнеру, нет и не может быть, поэтому код внутри портлета не может оценить, насколько портлет доступен извне.

Метод `destroy()` предоставляет возможность для произведения очистки ресурсов, которые были востребованы и инициализированы методом `init()`. Этот метод аналогичен методу `destroy()` в сервлетах и вызывается один раз: когда контейнер выгружает портлет.

Состояния

Состояния портлетов – это часть портальной модели отображения. Состояния позволяют портлету отображать различные «виды» в зависимости от ситуации.

Есть четыре состояния:

- `View` – основное состояние портлета;
- `Help` – если портлет обеспечивает состояние помощи;

- Edit – редактирование параметров портлета с сохранением результатов для этого конкретного пользователя;
- Configure – конфигурирование портлета с охранением результатов для всех пользователей, права к состояниям никак не относятся.

Портлет может быть минимизирован или максимизирован.

Портлет-контейнер

Портлет-контейнер – среда времени выполнения портлета. Она управляет жизненным циклом портлета и запросами от портала путем вызова портлетов внутри контейнера.

Компиляция и запуск

5. Установить переменную окружения JAVA_HOME=C:\jdk1.5.0
6. Добавить в переменную окружения PATH значение %JAVA_HOME%\bin
7. Загрузить и установить Apache Ant с http://ant.apache.org/
8. Распаковать скачанный архив, например, в папку C:\ant
9. Установить переменную окружения ANT_HOME=C:\ant
10. Добавить в переменную окружения PATH значение %ANT_HOME%\bin
11. Загрузить и установить Jetspeed2 с сервера http://portals.apache.org/jetspeed-2/
12. Распаковать загруженный архив, например, в папку C:\jetspeed2
13. Прописать в файле build.properties примеров jetspeed.deploy.dir=C:/jetspeed2/jakarta-tomcat-5.5.9 /webapps/jetspeed/WEB-INF/deploy
14. Перезагрузить Windows
15. Выполнить C:/jetspeed2/jetspeed-database/start-database.bat
16. Выполнить C:/jetspeed2/jakarta-tomcat-5.5.9/bin /startup.bat
17. Зайти в корневой каталог приложения и выполнить команду "ant deploy"
18. Ввести в браузер http://localhost:8080/jetspeed
19. Ввести пароль администратора (по умолчанию логин установлен в "admin", как и пароль)
20. Настроить отображаемые портлеты.

Простейший портлет

Для демонстрации возможностей Portlet API ниже рассмотрен пример, отображающий данные, введенные пользователем. На этой основе можно разрабатывать собственные portlet-приложения. Все необходимые для написания примеров классы находятся в пакете javax.portlet.

Вначале создается класс SamplePorltet. Этот класс, собственно, и является основным классом портлета. Все, что здесь происходит – это инициали-

зация портлета в методе init(PortletConfig config), **за каждое из представлений портлета отвечают методы** doEdit(), doView(), doHelp(). **В методе** processAction() **производится обработка событий портлета (в данном случае этот метод будет вызван при подтверждении пользователем желания отредактировать настройки портлета).**

```
/*пример # 1: простой портлет : SamplePorltet.java*/
package com.learning.portlet;
import java.io.IOException;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletConfig;
import javax.portlet.PortletContext;
import javax.portlet.PortletException;
import javax.portlet.PortletMode;
import javax.portlet.PortletPreferences;
import javax.portlet.PortletRequest;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

public class SamplePortlet extends GenericPortlet {
        private static final String EDIT_PAGE_PARAM
                                = "Edit-Page";
        private static final String HELP_PAGE_PARAM
                                = "Help-Page";
        private static final String VIEW_PAGE_PARAM
                                = "View-Page";

        public void init(PortletConfig config)
                                throws PortletException {
                super.init(config); // инициализация
        }
// метод, отвечающий за представление страницы редактирования
        public void doEdit(RenderRequest request,
                                RenderResponse response)
                throws PortletException,IOException {
                PortletContext context = getPortletContext();
                // получение контекста портлета
setRequestAttributes(request); // установка атрибутов
                // получение диспатчера для включения ресурсов в responce
                PortletRequestDispatcher rd = context.
getRequestDispatcher(getInitParameter(EDIT_PAGE_PARAM));
                rd.include(request, response); /* включение
                                                содержимого ресурса*/
        }
        // метод, отвечающий за представление страницы помощи
        public void doHelp(RenderRequest request,
                        RenderResponse response)
                        throws PortletException, IOException {
                PortletContext context = getPortletContext();
                // получение контекста портлета
        setRequestAttributes(request); // установка атрибута
        // получение диспатчера для включения ресурсов в responce
                PortletRequestDispatcher rd = context.
getRequestDispatcher(getInitParameter(HELP_PAGE_PARAM));
rd.include(request, response); // включение содержимого ресурса
        }
        // метод, отвечающий за представление страницы просмотра
        public void doView(RenderRequest request,
```

```
                                RenderResponse response)
                throws PortletException, IOException {
                PortletContext context = getPortletContext();
                // получение контекста портлета
        setRequestAttributes(request); // устанавливаем атрибуты
                // получение диспатчера для включения ресурсов в response
                PortletRequestDispatcher rd = context.
getRequestDispatcher(getInitParameter(VIEW PAGE PARAM));
rd.include(request, response); // включение содержимого ресурса
        }
        // вызывается контейнером для обработки событий
        public void processAction(ActionRequest request,
                 ActionResponse response)
                throws PortletException, IOException {
PortletMode mode = request.getPortletMode(); /* получение
                                                              состояния*/
                PortletPreferences preferences =
                        request.getPreferences(); // настройки
                if (mode.equals(PortletMode.VIEW)) {
                        // сохранение настроек
                        preferences.setValue("firstName",
                                request.getParameter("firstName"));

                        preferences.setValue("lastName",
                                request.getParameter("lastName"));

                        preferences.setValue("address",
                                request.getParameter("address"));

                        preferences.setValue("telephone",
                                request.getParameter("telephone"));
                        preferences.store();
                }
        }
        // для установки атрибутов
        private void setRequestAttributes(PortletRequest
                                request) {
                PortletPreferences preferences =
                        request.getPreferences();
                request.setAttribute("firstName",
        preferences.getValue("firstName", "undefined"));

                request.setAttribute("lastName",
        preferences.getValue("lastName", "undefined"));

                request.setAttribute("address",
        preferences.getValue("address", "undefined"));

                request.setAttribute("telephone",
        preferences.getValue("telephone", "undefined"));

                request.setAttribute("portletName",
        getPortletName());
        }
}
```

Ниже приведен файл portlet.xml, **который является дескриптором портлета и необходим для его развертывания.**

<!--пример # 2: дескриптор портлета : portlet.xml*- ->*

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app
    xmlns=
"http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
    version="1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/
portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">

    <portlet id="sample-application">
            <!-- описание -->
            <description>Portlet Application</description>
            <!-- название -->
            <portlet-name>SamplePortlet</portlet-name>
            <!-- отображаемое имя -->
            <display-name>Sample Portlet</display-name>
            <!-- класс портлета -->
            <portlet-class>
com.learning.portlet.SamplePortlet</portlet-class>
            <!-- параметры инициализации -->
            <init-param>
             <name>ViewPage</name>
             <value>/WEB-INF/jsp/sample-view.jsp</value>
            </init-param>
            <init-param>
             <name>HelpPage</name>
             <value>/WEB-INF/jsp/sample-help.jsp</value>
            </init-param>
            <init-param>
             <name>EditPage</name>
             <value>/WEB-INF/jsp/sample-edit.jsp</value>
            </init-param>
    <!-- количество секунд, на которое кешируется портлет. -1 значит cache
не истекает -->
            <expiration-cache>-1</expiration-cache>
            <!-- поддерживаемые режимы -->
            <supports>
                    <mime-type>text/html</mime-type>
                    <portlet-mode>view</portlet-mode>
                    <portlet-mode>help</portlet-mode>
                    <portlet-mode>edit</portlet-mode>
            </supports>
            <!-- локализация -->
            <supported-locale>en</supported-locale>
            <!-- текстовые ресурсы -->
            <resource-bundle>
com.learning.portlet.SamplePortlet</resource-bundle>
            <!-- информация -->
            <portlet-info>
                    <!-- заголовок -->
                    <title>Portlet Application</title>
                    <short-title>Portlet</short-title>
                    <!-- ключевые слова -->
                    <keywords>portlet</keywords>
            </portlet-info>
        </portlet>
    </portlet-app>
```

Файл web.xml **является дескриптором web-приложения, поскольку портлет-приложение является и web-приложением, то понадобится и этот файл.**

/ пример # 3: дескриптор приложения: web.xml*/*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
            version="2.4">
    <display-name>Sample</display-name>
</web-app>
```

Далее приведены JSP-страницы, отвечающие за представления портлета. Единственное, что следует отметить: библиотеку тегов для работы с портлетами, которая необходима для непосредственного доступа к специфическим классам портлетов (классам, используемым портлетом): `RenderRequest`, `RenderResponse`, `PortletConfig`. **Также эта библиотека предлагает функциональность по созданию ссылок, пригодных для работы с портлетами.**

<!-- пример # 4: представление портлета : sample-edit.jsp - ->

```
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
    <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
    <%@ taglib prefix="portlet" uri="http://java.sun.com/portlet"%>
    <portlet:defineObjects/>
    <portlet:actionURL var="editAction" portletMode="view"/>
    <c:set var="inputTextStyle" value="border: none; width: 150px; color:
#000000;"/>
    <c:set var="inputSubmitStyle" value="border: none; width: 75px;"/>
    <c:set  var="labelStyle"  value="text-align:  right;  color:  #000000;
white-space: nowrap;"/>
    <fmt:setBundle basename="com.learning.portlet.SamplePortlet"/>
    <form action="${editAction}" method="post">
        <table style="border: none;">
                <tbody>
                        <tr><td style="${labelStyle}">
                <fmt:message key="portlet.label.firstname"/>:
                                </td>
                                <td nowrap="nowrap">
    <input type="text" name="firstName" value="${firstName}"
                                        style="${inputTextStyle}"/>
                                </td>
                        </tr>
                        <tr><td style="${labelStyle}">
                <fmt:message key="portlet.label.lastname"/>:
                                </td>
                                <td nowrap="nowrap">
    <input type="text" name="lastName" value="${lastName}"
                                        style="${inputTextStyle}"/>
                                </td>
                        </tr>
                        <tr><td style="${labelStyle}">
                <fmt:message key="portlet.label.address"/>:
                                </td>
                                <td nowrap="nowrap">
    <input type="text" name="address" value="${address}"
                                        style="${inputTextStyle}"/>
                                </td>
                        </tr>
                        <tr><td style="${labelStyle}">
                <fmt:message key="portlet.label.telephone"/>:
                                </td>
                                <td nowrap="nowrap">
    <input type="text" name="telephone" value="${telephone}"
                                        style="${inputTextStyle}"/>
                                </td>
                        </tr>
                        <tr><td colspan="2" align="right">
            <button type="submit" style="${inputSubmitStyle}">
                <fmt:message key="portlet.button.submit"/>
                                        </button>
                                </td>
                        </tr>
                </tbody>
        </table>
    </form>
```

<!-- пример # 5:: sample-help.jsp- ->

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet"%>
<portlet:defineObjects/>
<fmt:setBundle basename="com.learning.portlet.SamplePortlet"/>
<table><tbody>
            <tr>
                   <td colspan="2">
                            
                   <fmt:message key="portlet.name.message">

    <fmt:param>${portletName}</fmt:param>
                          </fmt:message><br>
                            
            <fmt:message key="portlet.storeduser.message">
                                 <fmt:param>${firstName}</fmt:param>
                                 <fmt:param>${lastName}</fmt:param>
                          </fmt:message>
                   </td></tr>
            <tr>
                   <td align="center">
    <portlet:renderURL var="viewUrl" portletMode="view"/>
    <a href="${viewUrl}" style="text-decoration: none;">
            <fmt:message key="portlet.viewpage.link"/>.
                          </a>
                   </td>
            <td align="center">
    <portlet:renderURL var="editUrl" portletMode="edit"/>
    <a href="${editUrl}" style="text-decoration: none;">
    <fmt:message key="portlet.editpage.link"/>.
                          </a>
            </td></tr>
    </tbody>
</table>
```

<!-- *пример # 6:: sample-view.jsp* - ->

```
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
    <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
    <%@ taglib prefix="portlet" uri="http://java.sun.com/portlet"%>
    <portlet:defineObjects/>
    <c:set  var="labelStyle"  value="text-align:  right;  color:  #000000;
white-space: nowrap;"/>
    <c:set var="textStyle" value="text-align: left; color: #000000; white-
space: nowrap;"/>
    <fmt:setBundle basename="com.learning.portlet.SamplePortlet"/>
    <table>
        <tbody style="background-color: ">
                <tr>
                        <td style="${labelStyle}">
                <fmt:message key="portlet.label.firstname"/>:
                        </td>
                        <td style="${textStyle}">
                                ${firstName}
                        </td>
                </tr>
                <tr>
                        <td style="${labelStyle}">
                <fmt:message key="portlet.label.lastname"/>:
                        </td>
                        <td style="${textStyle}">
                                ${lastName}
                        </td>
                </tr>
                <tr>
                        <td style="${labelStyle}">
                <fmt:message key="portlet.label.address"/>:
                        </td>
                        <td style="${textStyle}">
                                ${address}
                        </td>
                </tr>
                <tr>
                        <td style="${labelStyle}">
                <fmt:message key="portlet.label.telephone"/>:
                        </td>
                        <td style="${textStyle}">
                                ${telephone}
                        </td>
                </tr>
        </tbody>
    </table>
```

пример # 7: файл свойств для русского языка : SamplePortlet_en.txt

```
    portlet.label.firstname = First name
    portlet.label.lastname = Last name
    portlet.label.address = Address
    portlet.label.telephone = Telephone number
    portlet.button.submit = Store
    portlet.name.message = This is the simple help page of the portlet
<span style="font-weight: bolder">{0}</span>.
    portlet.storeduser.message = Stored user is <span style="font-weight:
bolder"> {0} {1}</span>.
    portlet.viewpage.link = View page
    portlet.editpage.link = Edit page
```

пример # 8:: файл свойств для английского языка: SamplePortlet_ru.txt

```
    portlet.label.firstname = Имя
    portlet.label.lastname = Фамилия
    portlet.label.address = Адрес
    portlet.label.telephone = Номер телефона
    portlet.button.submit = Сохранить
    portlet.name.message = Страница помощи портлета <span style="font-
weight: bolder">{0}</span>.
    portlet.storeduser.message = Сохраненный пользователь <span
style="font-weight: bolder"> {0} {1}</span>.
    portlet.viewpage.link = Смотреть
    portlet.editpage.link = Редактировать
```

/ пример # 9: файл развертывания : build.xml*/*

```
    jetspeed.deploy.dir = E:/web/jetspeed-2.0-M3-Tomcat-5.5.9 /jakarta-
tomcat-5.5.9/webapps/jetspeed/WEB-INF/deploy
```

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`**. Методы создаются только в классах. Операторы при программировании перегружать нельзя. Множественное наследование, как и циклическое, 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 2.3.

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 2.4.

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.

Ответы:

2.1. 5), 6)
2.2. 1), 5)
2.3. 3), 4), 5)
2.4. 1), 2), 4)

Глава 3

Вопрос 3.1.

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 3.2.

Метод или логический блок могут вызывать конструктор только с помощью оператора **new**.

Ответ: 3.

Вопрос 3.3.

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 3.4.

При запуске приложения выполняются только статические логические блоки класса приложения и тело метода **main()**.

Ответ: 4.

Ответы:

3.1. 1), 5)
3.2. 3)
3.3. 1)
3.4. 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

Ответы:

5.1. 2)
5.2. 1), 4)
5.3. 2)
5.4. 3)
5.5. 3)

Глава 6

Вопрос 6.1.

Объявление пакета должно предшествовать любому другому коду, причем оно должно быть единственным. Комментарии могут находиться везде.

Ответ: 2, 3.

Вопрос 6.2.

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 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. 2), 3)
6.2. 2), 4)
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.

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 8.2.

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 8.3.

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 8.4.

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 8.5.

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 исключению типа **`Exception`** не предоставлен соответствующий блок **`catch`**. Вариант 2 содержит синтаксическую ошибку.

Ответ: 1.

Ответы:

8.1. 2), 5)
8.2. 1)
8.3. 4)
8.4. 2), 3)
8.5. 1)

Глава 9

Вопрос 9.1.

Методы класса **`File`** могут создавать, удалять, изменять имя каталога, но изменять корневой каталог можно только через переменные окружения.

Вопрос 9.3.

Методы класса **`File`** могут создавать файл, удалять его, изменять его имя, но к информации, содержащейся в файле, доступа не имеют.

Вопрос 9.5.

Использование **`transient`** указывает на отказ от сохранения значения помеченного поля объекта при записи объекта в поток.

Ответ: 3.

Ответы:

9.1. 4)
9.2. 2)
9.3. 2), 3)

9.4. 4)
9.5. 3)

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 объекта, 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**.

Ответ: 4.

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.

Для описания структуры данных используются только XSD и DTD. XSL используется для преобразований документа XML. CSS – каскадная таблица стилей для HTML.

Ответ: 1, 3.

Вопрос 16.2.

В строке 5 для тега </name> отсутствует открывающий тег. В строке 6 для тега <name> отсутствует закрывающий тег.

Ответ: 5, 6.

Вопрос 16.3.

Имя тега не может содержать пробельные символы или начинаться с цифры.

Ответ: 2, 4.

Вопрос 16.4.

Значения атрибутов XML могут помещаться только в двойные кавычки или апострофы.

Ответ: 1, 2.

Ответы:

16.1. 1), 3)
16.2. 5), 6)
16.3. 2), 4)
16.4. 1), 2)
16.5. 5)

Глава 17

Ответы:

17.1. 6)
17.2. 3)
17.3. 1), 4)
17.4. 1)
17.5. 3)

17.6. 2)

Глава 18

Вопрос 18.1.

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 18.2.

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 18.3.

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 18.5.

`ServletOutputStream` и **`ServletContextEvent`** – классы пакета **`javax.servlet`**. **`PageContext`** – класс пакета **`javax.servlet.jsp`**. Интерфейсами указанного пакета являются **`ServletRequest`** и **`Servlet`**.

Ответ: 1, 4.

Вопрос 18.6.

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:

18.1. 2), 3)
18.2. 1)
18.3. 4), 5)
18.4. 3)
18.5. 1), 4)
18.6. 3), 4)

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 20.1.

Объект **DriverManager** для установки соединения с БД использует драйвер БД и ее **URL**. Объект **DataSource** использует имя для поиска объекта.

Ответ: 1, 2.

Вопрос 20.2.

Производитель СУБД должен создать и предоставить драйвер для соединения с БД. Все драйвера должны реализовывать интерфейс **java.sql.Driver**.

Ответ: 1.

Вопрос 20.3.

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 20.5.

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:

20.1. 1), 2)
20.2. 1)
20.3. 1)
20.4. 4)
20.5. 2)

Глава 21

Вопрос 21.3.

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 21.4.

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 21.5.

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 21.6.

Объекты, представляющие cookies, присоединяются к объекту-ответу **`HttpServletResponse req`** только при помощи метода **`addCookie()`**.

Ответ: 2.

Ответы:

21.1. 2)
21.2. 1), 2), 5)
21.3. 1), 7)
21.4. 1), 3), 4)
21.5. 5), 6)
21.6. 2)

Глава 22

Вопрос 22.2.

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 22.3.

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 22.6.

Метод имеет сигнатуру

```
public void doInitBody() throws JSPException,
```

поэтому он не возвращает конкретных значений и может быть переопределен.

Ответ: 2, 4.

Ответы:

22.1. 2)
22.2. 5)
22.3. 4)
22.4. 5)
22.5. 2), 3)
22.6. 2), 4)
22.7. 1)

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ И 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. Д. Блох. Java. Эффективное программирование. // М.: Лори, 2002. – 224 с.
4. С. Макконнелл. Совершенный код.// СПб: Питер, 2005. – 868 с.
5. К.С. Хорстманн, Г. Корнелл. Библиотека профессионала. Java 2. Том 1. Основы. // М.: Вильямс, 2004. – 848 с.
6. К.С. Хорстманн, Г. Корнелл. Библиотека профессионала. Java 2. Том 2. Тонкости программирования. // М.: Вильямс, 2002. – 1120 с.
7. И.Н. Блинов, В.С. Романчик. Java 2. Практическое руководство. // Мн.: УниверсалПресс, 2005. – 400 с.
8. Г. Шилдт. Java 2, v5.0 (Tiger). Новые возможности. // СПб.: БХВ-Петербург, 2005. – 208 с.
9. Б.У. Перри. Java сервлеты и JSP : сборник рецептов. // М: Кудиц-пресс, 2006. – 768 с.
10. М. Холл. Сервлеты и JavaServer Pages. Библиотека программиста. // СПб.: Питер, 2001. – 496 с.
11. Б. Тейт. Горький вкус Java. // СПб: Питер, 2003. – 334 с.
12. Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML. Руководство пользователя. // М.: ДМК, 2000.– 432 с.
13. К. Ларман. Применение UML и шаблонов пректирования // М., СПб., К.: Вильямс, 2001. – 495 с.
14. С.Э. Эдди. XML. Справочник. // СПб., М., Х., Мн.: Питер, 2000. – 480 с.
15. Sun Developer Network Site. http://java.sun.com/
16. The World Wide Web Consortium. http://w3c.org

Производственно-практическое издание

БЛИНОВ Игорь Николаевич
РОМАНЧИК Валерий Станиславович

JAVA
ПРОМЫШЛЕННОЕ ПРОГРАММИРОВАНИЕ

Практическое пособие

Ответственный за выпуск *В.М. Стрельченя*
Редактор *А.С. Бржозовский*
Компьютерный набор *И.Н. Блинов*
Компьютерная верстка *И.Л. Богданова*
Дизайн обложки *Р.В. Щуцкий*

В оформлении обложки
использован стилизованный фрагмент полинезийского орнамента с острова Java.

Подписано в печать с готовых диапозитивов 20.09.2007 г.
Формат 70 x 100 1/16. Бумага офсетная. Гарнитура Times.
Печать офсетная. Усл. печ. л. 57,2. Уч. изд. л. 29,8.
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»,
ЛП № 02330/0148704 от 30.04.2004 г.
Республика Беларусь, г. Минск, ул. Я. Колоса, 73.

```
    <?xml version="1.0" encoding="UTF-8"?>
    <project name="SamplePortlet" default="war" basedir=".">
        <property file="build.properties"/>
        <property name="src.dir" value="src"/>
        <property name="jsp.dir" value="jsp"/>
        <property name="java.dir" value="java"/>
        <property name="message.dir" value="message"/>
        <property name="config.dir" value="config"/>
        <property name="lib.dir" value="lib"/>
        <property name="src.lib.dir" value="${src.dir}/${lib.dir}"/>
        <property name="src.java.dir" value="${src.dir}/${java.dir}"/>
        <property name="src.jsp.dir" value="${src.dir}/${jsp.dir}"/>
        <property                name="src.message.dir"                val-
ue="${src.dir}/${message.dir}"/>
        <property name="src.config.dir" value="${src.dir}/${config.dir}"/>
        <property name="webapp.dir" value="WEB-INF"/>
        <property name="result.dir" value="result"/>
        <property name="classes.dir" value="classes"/>
        <property name="project.name" value="SamplePortlet"/>
        <property name="localization.encoding" value="Cp1251"/>
        <property             name="localization.src.dir"             val-
ue="${src.message.dir}/com/learning/portlet"/>
        <property           name="localization.result.dir"            val-
ue="${result.dir}/${classes.dir}/com/learning/portlet"/>
        <property name="localization.ext" value=".properties"/>
        <target name="compile" description="Compiles all source files of
the portlet.">
                <mkdir dir="${result.dir}/${classes.dir}"/>
    <javac      srcdir="${src.java.dir}"      destdir=      "${result.dir}
/${classes.dir}" optimize="yes" debug="no">
                        <classpath>
                                <fileset dir="${src.lib.dir}">
                                        <include name="**/*.jar"/>
                                </fileset>
                        </classpath>
                </javac>
                <copy todir="${result.dir}/${classes.dir}">
    <fileset dir="${src.message.dir}" excludes="**/*.txt"/>
                </copy>
    <native2ascii src="${localization.src.dir}" dest=
    "${localization.result.dir}" includes="**/*.txt"
    encoding="${localization.encoding}" ext="${localization.ext}"/>
        </target>
        <target  name="war"  depends="compile"  description=  "Creates  .war
file of the portlet.">
        <war destfile="${result.dir}/${project.name}.war"
         webxml="${src.config.dir}/web.xml" compress="on">
                        <lib dir="${src.lib.dir}">
                        <exclude name="**/portlet-api-1.0.jar"/>
                        </lib>
                <classes dir="${result.dir}/${classes.dir}"/>
    <zipfileset dir="${src.config.dir}" prefix="${webapp.dir}">
                                <include name="**/*.*"/>
                                <exclude name="**/web.xml"/>
                        </zipfileset>
                <zipfileset dir="${src.jsp.dir}" prefix=
                                        "${webapp.dir}/${jsp.dir}"/>
                </war>
        </target>
```

```
        <target  name="deploy"  depends="war"  description  ="Deploys  .war
file of the portlet.">
                <copy todir="${jetspeed.deploy.dir}">
        <fileset file="${result.dir}/${project.name}.war"/>
                </copy>
        </target>
        <target name="clean" description=
    "Deletes all result files of the portlet.">
                <delete dir="${result.dir}"/>
        </target>
    </project>
```

пример # 10: файл свойств для ant : build.properties

```
    jetspeed.deploy.dir  =  E:/web/jetspeed-2.0-M3-Tomcat-5.5.9  /jakarta-
tomcat-5.5.9/webapps/jetspeed/WEB-INF/deploy
```