И.Н.БЛИНОВ
В.С.РОМАНЧИК

Java
Методы программирования

Java SE
Сервлеты и JSP
Паттерны GoF
JUnit
Log4J
Ant
UML
SQL
JPA
Hibernate

EPAM training center

И. Н. Блинов
В. С. Романчик

Java

Методы программирования

Учебно-методическое пособие

МИНСК
ИЗДАТЕЛЬСТВО «ЧЕТЫРЕ ЧЕТВЕРТИ»
2013

УДК 004.434
ББК 32.973.26-018.2
Б69

Рецензенты:
кандидат физико-математических наук *С. В. Сахоненко*,
кандидат физико-математических наук, доцент *П. В. Гляков*

Рекомендовано
Учебно-методическим объединением по естественнонаучному образованию
в качестве пособия для студентов высших учебных заведений,
обучающихся по специальности 1-31 03 01 «Математика (по направлениям)»,
направление специальности 1-31 03 01-05 «Математика (информационные технологии)»

Блинов, И.Н., Романчик, В. С.

Б69 Java. Методы программирования : уч.-мет. пособие / И. Н. Блинов, В. С. Романчик. — Минск : издательство «Четыре четверти», 2013. — 896 с.
ISBN 978-985-7058-30-3.

Пособие предназначено для программистов, начинающих и продолжающих изучение технологий Java SE, JEE и других. В его первой части рассматриваются основы языка Java и концепции объектно-ориентированного программирования. Во второй части изложены 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 и собственных тегов разработчика. В четвертой части даны основы практического применения шаблонов проектирования.

В конце каждой главы даются тестовые вопросы по материалу главы и задания для выполнения. В приложениях приведены дополнительные материалы, относящиеся к использованию UML, SQL, Ant, XML, а также краткое описание популярных технологий Log4J, JUnit, JPA и Hibernate.

УДК 004.434
ББК 32.973.26-018.2

ISBN 978-985-7058-30-3

ОГЛАВЛЕНИЕ

СОДЕРЖАНИЕ

Часть 2. Использование классов и библиотек

Часть 4. Шаблоны проектирования

ПРЕДИСЛОВИЕ

Пособие «Java. Методы программирования» расширяет и включает переработанную и обновленную версию предыдущих книг авторов «Java 2. Практическое руководство» 2005 года и «Java. Промышленное программирование» 2007 года.

Книга возникла в процессе обучения языку Java и технологиям студентов механико-математического факультета и факультета прикладной математики и информатики Белорусского государственного университета, а также слушателей курсов повышения квалификации и преподавательских тренингов и семинарах EPAM Systems, Oracle и других образовательных программ 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. Книга предназначена как для начинающих изучение Java-технологий, так и для студентов и программистов, переходящих на Java с другого языка программирования. Авторы считают, что «профессионала под ключ» обучить нельзя, им 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 четыре логические 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 и базы данных, многопоточность и взаимодействие с XML. В третьей части приведены основы программировании распределенных информационных систем с применением сервлетов, JSP. В четвертой даны основы применения шаблонов проектирования.

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 в информационных системах, основанных на применении Java-технологий, популярных технологий Log4J и JUnit, Apache Ant для сборки приложений, а также основы языков UML и SQL.

Авторы выражают благодарность компании EPAM Systems и ее сотрудникам, принимавшим участие в подготовке материалов этой книги и в ее издании.

Часть 1

Основы Java

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

Глава 1

ВВЕДЕНИЕ В ООП И КЛАССЫ

Каждый дурак может написать программу, которую может понять компьютер. Хороший программист пишет программу, которую может понять человек.

Мартин Фаулер

Основные понятия ООП

Java является объектно-ориентированным языком программирования, вследствие чего предварительно будут приведены основные парадигмы ООП.

В связи с проникновением компьютеров во все сферы социума программные системы становятся более простыми для пользователя и сложными по внутренней архитектуре. Программирование стало делом команды, где маленьким проектом считается тот, который выполняет команда из 5–10 специалистов за время от полугода до года.

Основным способом борьбы со сложностью программных продуктов стало объектно-ориентированное программирование (ООП), являющееся в настоящее время наиболее популярной парадигмой.

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 взаимодействие объектов.

Объект — именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение.

В применении к объектно-ориентированным языкам программирования понятия объекта и класса конкретизируются.

Объект — обладающий именем набор данных (полей и свойств объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним. Имя используется для работы с полями и методами объекта.

Любой объект относится к определенному классу. В классе дается обобщенное описание некоторого набора родственных объектов.

Объект — конкретный экземпляр класса.

В качестве примера можно привести абстракцию дома или его описание (класс) и реальный дом (экземпляр класса или объект). Объект соответствует логической модели дома, представляющей совокупное описание всех физических объектов.

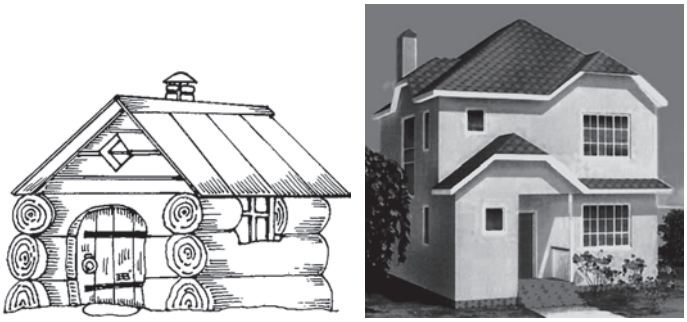

Рис. 2.1. *Описание класса (абстракция) и реальный объект*

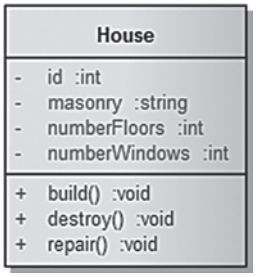

Рис. 2.2. *Графическое изображение класса*

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

В верхний прямоугольник помещается имя класса, в средний — набор полей с именами, типами, свойствами класса, и в нижний — список методов, их параметров и возвращаемых значений.

Реальный объект должен иметь конкретные значения всех полей, например:

id=35, masonry="brick", numberFloors=2, numberWindows=7.

Объектно-ориентированное программирование основано на принципах:

— инкапсуляции;

— наследования;

— полиморфизма, в частности, «позднего связывания».

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

Наследование (inheritance) — процесс, посредством которого один класс может наследовать свойства другого класса и добавлять к ним свойства и методы, характерные только для него.

Наследование бывает двух видов:

одиночное наследование — подкласс (производный класс) имеет один и только один суперкласс (предок);

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

Полиморфизм (polymorphism) — механизм, использующий одно и то же имя метода для решения похожих, но несколько отличающихся задач в различных объектах при наследовании из одного суперкласса. Целью полиморфизма является использование одного имени при выполнении общих для суперкласса и подклассов действий.

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 (виртуального) метода во время выполнения программы. Другими словами, иногда невозможно на этапе компиляции определить, какая версия переопределенного метода будет вызвана на этапе выполнения программы.

Краеугольным камнем наследования и полиморфизма предстает следующая парадигма: «***объект подкласса может использоваться всюду, где используется объект суперкласса***». То есть при добавлении к иерархии классов нового подкласса существующий код с экземпляром нового подкласса будет работать точно так же, как и со всеми другими экземплярами классов в иерархии.

При вызове метода сначала 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 в этом классе. Если поиск неудачен, то он продолжается вверх по иерархическому дереву вплоть до корня (верхнего класса **Object**) иерархии.

Язык Java

Объектно-ориентированный язык Java, разработанный в компании Sun Microsystems в 1995 году для оживления графики на стороне клиента с помощью апплетов, в настоящее время используется для создания переносимых на различные платформы и операционные системы программ. Язык Java нашел широкое применение в Интернет-приложениях, добавив на статические и клиентские веб-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 создания язык практически покинул клиентские страницы и перебрался на серверы. На стороне клиента его место заняли языки JavaScript, Adobe Flash и проч.

При создании язык Java предполагался более простым, чем его синтаксический предок C++. Сегодня с появлением новых версий возможности языка Java существенно расширились и во многом перекрывают функциональность C++. Java уже не уступает по сложности предшественникам и называть его простым нельзя.

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 безтиповые указатели позволяет игнорировать защиту памяти. Отсутствие в Java множественного наследования легко заменяется на более понятные конструкции с применением интерфейсов.

Системная библиотека классов языка Java содержит классы и пакеты, реализующие и расширяющие базовые возможности языка, а также сетевые

средства, взаимодействие с базами данных, графические интерфейсы и многое другое. Методы классов, включенных в эти библиотеки, вызываются JVM (Java Virtual Machine) во время интерпретации программы.

В Java все объекты программы расположены в динамической памяти — куче данных (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. Необходимо отметить, что объектная ссылка языка Java содержат информацию о классе объекта, на который она ссылается, так что объектная ссылка — это не указатель, а дескриптор (описание) объекта. 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**. Программист может только рекомендовать системе освободить выделенную динамическую память.

В отличие от C++, Java не поддерживает множественное наследование, перегрузку операторов, беззнаковые целые, прямое индексирование памяти и, как следствие, указатели. 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.

Простое приложение

Изучение любого языка программирования удобно начинать с программы передачи символьного сообщения на консоль.

// # 1 # простое линейное приложение # OracleSlogan.java

```
public class OracleSlogan {
        public static void main(String[ ] args) {
           // вывод строки
           System.out.println("Enabling the Information Age");
        }
}
```

Но уже в этом коде заложен фундамент будущих архитектурных ошибок. Пусть представленный выше класс является первым из множества классов системы, которые будут созданы в процессе разработки небольшой системы передачи некоторых сообщений, состоящей, например, из двух–трех десятков классов. Строка

```
System.out.println("Enabling the Information Age");
```

может встречаться в коде этих классов многократно. Пусть в процессе тестирования или внедрения системы окажется, что фразу необходимо заменить на другую, например, в конце поставить восклицательный знак. Для этого программисту придется обыскивать весь код, искать места, где встречается указанная фраза, и заменять ее новой. Это по меньшей мере, неудобно. Что делать если таких классов — не пара десятков, а несколько сотен? Во избежание подобных проблем сообщение лучше хранить в отдельном методе или константе (а еще лучше — в файле) и при необходимости вызывать его. Тогда изменение текста сообщения приведет к локальному изменению одной-единственной строки кода. В следующем примере этот код будет переписан с использованием двух классов, реализованных на основе простейшего применения объектно-ориентированного программирования:

/ # 2 # простое обьектно-ориентированное приложение # FirstProgram.java */*

```
package by.bsu.simple;
public class FirstProgram {
          public static void main(String [ ] args) {
                   // объявление и создание объекта firstObject
                   SloganAction firstObject = new SloganAction();
                   // вызов метода, содержащего вывод строки
                   firstObject.printSlogan();
          }
}
```

// # 3 # простой класс # SloganAction

```
class SloganAction {
          void printSlogan() { // определение метода
                    // вывод строки
                    System.out.println("Enabling the Information Age");
          }
}
```

Здесь класс **FirstProgram** используется для того, чтобы определить метод **main(),** который вызывается автоматически интерпретатором Java и может называться контроллером этого примитивного приложения. Метод **main()** получает в качестве параметра аргументы командной строки **String[]args**, представляющие массив строк, и является открытым (**public**) членом класса. Это означает, что метод **main()** может быть 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. Тело метода **main()** содержит объявление объекта

```
SloganAction firstObject = new SloganAction();
```

и вызов его метода

```
firstObject.printSlogan();
```

Вывод строки «Enabling the Information Age» в примере осуществляет метод **println()** (**ln** — переход к новой строке после вывода) статического поля **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 должно совпадать с именем **public**-класса.

Объявление классов предваряет строка

```
package by.bsu.simple;
```

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

Классы из примеров 2 и 3 могут сохраняться как в одном файле, так и в двух файлах **FirstProgram.java** и **SloganAction.java**. На практике следует хранить классы в отдельных файлах, что позволяет всем разработчикам проекта быстрее воспринимать концепцию приложения в целом.

/ # 4 # простое обьектно-ориентированное приложение # FirstProgram.java */*

```
package by.bsu.simple.run;
import by.bsu.simple.action.SloganAction; // подключение класса из пакета
public class FirstProgram {
          public static void main(String[ ] args) {
                    SloganAction firstObject = new SloganAction ();
                    firstObject.printSlogan();
          }
}
```

// # 5 # простой класс # SloganAction.java

```
package by.bsu.simple.action;
public class SloganAction {
          public void printSlogan() {
                    // вывод строки
                    System.out.println("Enabling the Information Age");
          }
}
```

Простейший способ компиляции — вызов строчного компилятора из корневого каталога, в котором находится каталог **by**, каталог **bsu** и так далее:

```
javac by/bsu/simple/action/SloganAction.java
javac by/bsu/simple/run/FirstProgram.java
```

При успешной компиляции создаются файлы **FirstProgram.class** и **SloganAction.class,** имена которых совпадают с именами классов. Запустить этот байткод можно с помощью интерпретатора Java:

```
java by.bsu.simple.run.FirstProgram
```

Здесь к имени приложения **FirstProgram.class** добавляется путь к пакету от корня проекта **by.bsu.simple.run**, в котором он расположен.

Чтобы компилировать и выполнить приложение, необходимо загрузить и установить последнюю версию пакета, например по адресу:

```
http://www.oracle.com/technetwork/java/javase/downloads/
```

При инсталляции рекомендуется указывать для размещения корневой каталог. Если JDK установлена в директории (для Windows) **c:\Java\jdk7**, то каталог, который компилятор Java будет рассматривать как корневой для иерархии пакетов, можно вручную задавать с помощью переменной среды окружения в виде: **CLASSPATH=.;c:\Java\jdk7**.

Переменной задано еще одно значение «**.**» для использования текущей директории, например, **c:\workspace** 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:\Java\jdk7\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 начинающим программистам лучше не определять вовсе.

Обработка аргументов командной строки, передаваемых в метод **main(),** относится к необходимым в самом начале обучения языку программам. Аргументы представляют последовательность строк, разделенных пробелами, значения которых присваиваются объектам массива **String[]args**. Элементу **args[0]** присваивается значение первой строки после имен компилятора и приложения. Количество аргументов определяется значением **args.length**. Поле **args.length** является константным полем класса, представляющего массив, значение которого не может быть изменено на протяжении всего жизненного цикла объекта-массива.

/ # 6 # вывод аргументов командной строки в консоль # PrintArguments.java */*

```
package by.bsu.arg;
public class PrintArguments {
          public static void main(String[ ] args) {
                    for (String str : args) {
                              System.out.printf("Аргумент-> %s%n", str);
                    }
          }
}
```

В данном примере используется цикл **for** для неиндексируемого перебора всех элементов и метод форматированного вывода **printf().** Тот же результат был бы получен при использовании традиционного вида цикла **for**

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

Запуск этого приложения осуществляется с помощью следующей командной строки вида:

```
java by.bsu.arg.PrintArguments 2012 Oracle "Java SE 7"
```

что приведет к выводу на консоль следующей информации:

Аргумент-> 2012
Аргумент-> Oracle
Аргумент-> Java SE 7

Приложение, запускаемое с аргументами командной строки, может быть использовано как один из примитивных способов ввода в приложение внешних данных.

Основы классов и объектов Java

Классы в языке Java объединяют поля класса, методы, конструкторы, логические блоки и внутренние классы. 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; 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 пакете и вне пакета), **final** (класс не может иметь подклассов), **abstract** (класс может содержать абстрактные методы, объект такого класса создать нельзя). По умолчанию, если спецификатор класса не задан, он 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 в сленге программистов, чтобы как-то коротко обозначить значение по умолчанию.

Класс наследует все свойства и методы суперкласса, указанного после ключевого слова **extends**, и может включать множество интерфейсов, перечисленных через запятую после ключевого слова **implements**. Интерфейсы очень похожи на абстрактные классы, содержащие только константы и сигнатуры методов без реализации.

Все классы любого приложения условно разделяются на две группы: классы — носители информации и классы, работающие с информацией. Классы, обладающие информацией, содержат данные о предметной области приложения. Например, если приложение предназначено для управления воздушным движением, то предметной областью будут самолеты, пассажиры и пр. При проектировании классов информационных экспертов важна инкапсуляция, обеспечивающая значениям полей классов корректность информации. У рассмотренного выше класса **House** есть поле **numberWindows**, значение которого на может быть отрицательным. С помощью инкапсуляции ключевым словом **private** закрывается прямой доступ к значению поля через объект, а метод, отвечающий за инициализацию значения поля, будет выполнять проверку входящего значения на корректность.

В качестве примера с нарушением инкапсуляции можно рассмотреть класс **Coin** в приложении по обработке монет.

// # 7 # простой пример класса носителя информации # Coin.java

```
package by.bsu.fund.bean;
public class Coin {
        public double diameter; // нарушение инкапсуляции
        private double weight; // правильная инкапсуляция
        public double getDiameter() {
                return diameter;
        }
        public void setDiameter(double value) {
                if (value > 0) {
                        diameter = value;
                } else {
                        diameter = 0.01; // значение по умолчанию
                }
        }
        public double takeWeight() { // некорректно: неправильное имя метода
                return weight;
        }
        public void setWeight(double value) {
                weight = value;
        }
}
```

Класс **Coin** содержит два поля **diameter** и **weight**, помеченные как **public** и **private**. Значение поля **weight** можно изменять только при помощи методов, например, **setWeight** (**double value**). В некоторых ситуациях замена некорректного значения на значение по умолчанию может привести к более грубым ошибкам в дальнейшем, поэтому часто вместо замены производится генерация исключения. Поле **diameter** доступно непосредственно через объект класса **Coin**. Поле, объявленное таким способом, считается объявленным с нарушением «тугой» инкапсуляции, следствием чего может быть нарушение корректности информации, как это показано ниже:

// # 8 # демонстрация последствий нарушения инкапсуляции # Runner.java

```
package by.bsu.fund.run;
import by.bsu.fund.bean.Coin;
public class Runner {
        public static void main(String[ ] args) {
                Coin ob = new Coin();
                ob.diameter = -0.12; // некорректно: прямой доступ
                ob.setWeight(100);
               // ob.weight = -150; // поле недоступно: compile error
        }
}
```

Чтобы компиляция кода вида

```
ob.diameter = -0.12;
```

стала невозможной, следует поле **diameter** класса **Coin** объявить в виде

```
private double diameter;
```

тогда строка с попыткой прямого присваивания значения поля с помощью ссылки на объект приведет к ошибке компиляции.

// # 9 # «туго» инкапсулированный класс (Java Bean) # Coin.java

```
package by.bsu.fund.bean;
public class Coin {
        private double diameter; // правильная инкапсуляция
        private double weight; // правильная инкапсуляция
        public double getDiameter() {
                return diameter;
        }
        public void setDiameter(double value) {
                if(value > 0) {
                   diameter = value;
                }  else {
                   System.out.println("Отрицательный диаметр!");
                }
        }
        public double getWeight() { // правильное имя метода
                return weight;
        }
        public void setWeight(double value) {
                weight = value;
        }
}
```

Проверка корректности входящей извне информации осуществляется в методе **setDiameter(double value)** и позволяет уведомить о нарушении инициализации объекта. Доступ к **public**-методам объекта класса осуществляется только после создания объекта данного класса.

/ # 10 # создание объекта, доступ к полям и методам объекта # CompareCoin.java # Runner.java */*

```
package by.bsu.fund.action;
import by.bsu.fund.bean.Coin;
public class CompareCoin {
        public void compareDiameter(Coin first, Coin second) {
                double delta = first.getDiameter() - second.getDiameter();
                if (delta > 0) {
                        System.out.println("Первая монета больше второй на " + delta);
                } else if (delta == 0) {
```

```
                        System.out.println("Монеты имеют одинаковый диаметр");
                } else {
                        System.out.println("Вторая монета больше первой на " + -delta);
                }
        }
}
package by.bsu.fund.run;
import by.bsu.fund.bean.Coin;
import by.bsu.fund.action.CompareCoin;
public class Runner {
        public static void main(String[ ] args) {
                Coin ob1 = new Coin();
                ob1.setDiameter(-0.11); // сообщение о неправильных данных
                ob1.setDiameter(0.12); // корректно
                ob1.setWeight(150);
                Coin ob2 = new Coin();
                ob2.setDiameter(0.21);
                ob2.setWeight(170);
                CompareCoin ca = new CompareCoin();
                ca.compareDiameter(ob1, ob2);
        }
}
```

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

Отрицательный диаметр!
Вторая монета больше первой на 0.09.

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

```
Coin ob1; // объявление ссылки
ob1 = new Coin(); // создание объекта
```

Однако эти два действия обычно объединяют в одно:

```
Coin ob1 = new Coin(); /* объявление ссылки и создание объекта */
```

Оператор **new** вызывает конструктор, в данном примере конструктор по умолчанию без параметров, но в круглых скобках могут размещаться аргументы, передаваемые конструктору, если у класса объявлен конструктор с параметрами. Операция присваивания для объектов означает, что две ссылки будут указывать на один и тот же участок памяти.

Метод **compareDiameter(Coin first, Coin second)** выполняет два действия, которые следует разделять: выполняет сравнение и печатает отчет. Действия слишком различны по природе, чтобы быть совмещенными. Естественным решением будет изменить возвращаемое значение метода на **int** и оставить в нем только вычисления.

/ # 11 # метод сравнения экземпляров по одному полю # */*

```
public int compareDiameter(Coin first, Coin second) {
        int result = 0;
        double delta = first.getDiameter() - second.getDiameter();
        if (delta > 0) {
                result = 1;
        } else if (delta < 0) {
                result = -1;
        }
        return result;
}
```

Формирование отчета следует поместить в другой метод другого класса.

Объектные ссылки

Java работает не с объектами, а с ссылками на объекты. Это объясняет то, что операции сравнения ссылок на объекты не имеют смысла, так как при этом сравниваются адреса. Для сравнения объектов на эквивалентность по значению необходимо использовать специальные методы, например, **equals(Object ob).** Этот метод наследуется в каждый класс из суперкласса **Object**, который лежит в корне дерева иерархии всех классов и должен переопределяться в подклассе для определения эквивалентности содержимого двух объектов этого класса.

/ # 12 # сравнение ссылок и объектов # ComparisonStrings.java */*

```
package by.bsu.strings;
public class ComparisonStrings {
        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
                s2 = new String("Java"); // эквивалентно s2 = new String(s1);
                System.out.println("сравнение ссылок "+ (s1 == s2)); // результат false
                System.out.println("сравнение значений " + s1.equals(s2)); // результат true
        }
}
```

В результате выполнения действия **s2 = s1** получается, что обе ссылки ссылаются на один и тот же объект. Оператор «==» возвращает **true** при сравнении ссылок только в том случае, если они ссылаются на один и тот же объект.

Если же ссылку инициализировать при помощи конструктора **s2 = new String(s1),** то создается новый объект в другом участке памяти, который инициализируется

значением, взятым у объекта **s1**. В итоге существуют две ссылки, каждая из которых независимо ссылается на объект, который никак физически не связан другим объектом. Поэтому оператор сравнения ссылок возвращает результат **false**, так как ссылки ссылаются на различные участки памяти. Объекты обладают одинаковыми значениями, что легко определяется вызовом метода **equals(Object o).**

Если в процессе разработки возникает необходимость в сравнении по значению объектов классов, созданных программистом, для этого следует переопределить в данном классе метод **equals(Object o)** в соответствии с теми критериями сравнения, которые существуют для объектов данного типа или по стандартным правилам, заданным в документации.

Консоль

Консоль определяется программой, предоставляющей интерфейс командной строки для интерактивного обмена текстовыми командами и сообщениями с операционной системой или программным обеспечением.

Взаимодействие с консолью с помощью потока (объекта класса) **System.in** представляет собой один из простейших способов передачи информации в приложение. При создании первых приложений такого рода передача в них информации является единственно доступной для начинающего программиста. 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.

// # 13 # чтение символа из потока System.in # ReadCharRunner.java

```
package by.bsu.console;
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) {
                 System.err.println("ошибка ввода " + e);
        }
    }
}
```

Обработка исключительной ситуации **IOException**, которая может возникнуть в операциях ввода/вывода и в любых других взаимодействиях с внешними устройствами, осуществляется в методе **main()** с помощью реализации

блока **try-catch**. Если ошибок при выполнении не возникает, выполняется блок **try {},** в противном случае генерируется исключительная ситуация, и выполнение программы перехватывает блок **catch {}.**

Ввод блока информации осуществляется посредством чтения строки из консоли с помощью возможностей объекта класса **Scanner**, имеющего возможность соединяться практически с любым источником информации: строкой, файлом, сокетом, адресом в Интернете, с любым объектом, из которого можно получить ссылку на поток ввода.

// # 14 # чтение строки из консоли # RunScanner.java

```
package by.bsu.console;
import java.util.Scanner;
public class RunScanner {
        public static void main(String[ ] args) {
                System.out.println("Введите Ваше имя и нажмите <Enter>:");
                Scanner scan = new Scanner(System.in);
                String name  = scan.next();
                System.out.println("Привет, " + name);
                scan.close();
        }
}
```

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 с помощью класса **Scanner**, в качестве которого может фигурировать не только консоль, но и дисковый файл, строка, сокетное соединение и пр.

Base code conventions

При выборе имени класса, поля, метода использовать цельные слова, полностью исключить сокращения. По возможности опускать предлоги и очевидные связующие слова. Аббревиатуры использовать только в том случае, когда они очевидны.

Имя класса всегда пишется с большой буквы: **Coin**, **Developer**.

Если имя класса состоит из двух и более слов, то второе и следующие слова пишутся слитно с предыдущим и начинаются с большой буквы: **AncientCoin**, **FrontendDeveloper**.

Имя метода всегда пишется с маленькой буквы: **perform(), execute().**

Если имя метода состоит из двух и более слов, то второе и следующие слова пишутся слитно с предыдущим и начинаются с большой буквы: **performTask(), executeBaseAction().**

Имя поля класса, локальной переменной и параметра метода всегда пишутся с маленькой буквы: **weight**, **price**.

Если имя поля класса, локальной переменной и параметра метода состоит из двух и более слов, то второе и следующие слова пишутся слитно с предыдущим и начинаются с большой буквы: **priceTicket**, **typeProject**.

Константы и перечисления пишутся в верхнем регистре: **DISCOUNT**, **MAX_RANGE**.

Все имена пакетов пишутся с маленькой буквы. Сокращения допустимы только в случае, если имя пакета слишком длинное: 10 или более символов. Использование цифр и других символов нежелательно.

Задания к главе 1

Вариант А

1. Приветствовать любого пользователя при вводе его имени через командную строку.
2. Отобразить в окне консоли аргументы командной строки в обратном порядке.
3. Вывести заданное количество случайных чисел с переходом и без перехода на новую строку.
4. Ввести пароль из командной строки и сравнить его со строкой-образцом.
5. Ввести целые числа как аргументы командной строки, подсчитать их суммы (произведения) и вывести результат на консоль.
6. Вывести фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания.

Вариант В

Ввести с консоли *n* целых чисел. 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 программ:

A)
```
public class Quest21 {
public static void main (String [] args) {
System.out.println ("Hello, java 7");
}
}
```

B)
```
public class Quest22 {
String java = "Java 7";
public static void main (String [] args) {
System.out.println (java);
}
}
```

C)
```
public class Quest23 {
{
System.out.println ("Java 7");
}
}
```

Укажите, что скомпилируется без ошибок (выберите 1).

1) AB
2) BC
3) ABC
4) A
5) AC

Вопрос 1.2.

Выберите правильные утверждения (2):

1) класс — это тип данных;
2) объект класса может использоваться всюду, где используется объект подкласса;
3) объект класса можно создать только один раз;
4) на объект класса может не ссылаться объектная переменная.

Вопрос 1.3.

Вставьте на место прочерка название одного из принципов ООП так, чтобы получилось верное определение (1):

1) наследование
2) полиморфизм
3) позднее связывание
4) инкапсуляция

_____ *— это объединение данных и методов, предназначенных для манипулирования этими данными в новом типе — классе.*

Вопрос 1.4.

Дан код:

```
String s; // 1
if ((s = "java") == "java") {// 2
        System.out.println (s+ " true");
} else {
        System.out.println (s+ " false");
}
```

Что будет результатом компиляции и запуска этого кода (1)?

1) ошибка компиляции в строке 1, переменная не проинициализирована
2) ошибка компиляции в строке 2, неправильное выражение для оператора if
3) на консоль выведется **java true**
4) на консоль выведется **java false**

Вопрос 1.5.

Дан код:

```
public class Quest5 {
        private static void main (String [] args) {
                System.out.println (args [2]);
        }
}
```

Что произойдет, если этот код выполняется следующей командной строкой: *java Quest5 Java 7""* (1)?

1) выведется: Java 7
2) ошибка времени выполнения NullPointerException
3) ошибка времени выполнения ArraylndexOutOfBoundsException
4) выведется: Java 7 <пустая строка>
5) выведется: <пустая строка>
6) приложение не запустится
7) код не скомпилируется

Вопрос 1.6.

Дан код:

```
public class Quest6 {
        public static void main (String [] args) {
                System.out.print ("A");
                main ("java7");
        }
        private static void main (String args) {
                System.out.print ("B");
        }
}
```

Что будет выведено в результате запуска и компиляции (1)?

1) ошибка компиляции
2) BA
3) AB
4) AA
5) компиляция пройдет успешно, а при выполнении программа зациклится

Вопрос 1.7.

Дан код:

```
class Book {
        private String book;
        public void setBook (String b) {book = b;}
}
public class Quest7 {
        public static void main (String [] args) {
                Book book1 = new Book (); book1.setBook ("Java 7");
                Book book2 = new Book (); book2.setBook ("Java 7");
                if (book1.equals (book2)) {
                        System.out.println ("True");
                } else {
                        System.out.println ("False");
                }
        }
}
```

Результатом компиляции и запуска кода будет (1)?

1) True
2) ошибка компиляции
3) False
4) код скомпилируется, но при выполнении оператор if будет пропущен.

Глава 2

ТИПЫ ДАННЫХ И ОПЕРАТОРЫ

Программирование всегда осуждалось
светскими и духовными властями.

«Дозор»

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

Базовые типы данных и литералы

Java — язык объектно-ориентированного программирования, однако не все данные в языке есть объекты. Для повышения производительности в нем кроме объектов используются базовые типы данных, значения которых размещаются в стековой памяти при компиляции программы. Для каждого базового типа имеются также классы-оболочки, которые инкапсулируют данные базовых типов в объекты, располагаемые в динамической памяти (heap). Базовые типы обеспечивают более высокую производительность вычислений по сравнению с объектами классов-оболочек и другими объектами.

Определено восемь базовых типов данных, размер каждого из которых остается неизменным независимо от платформы (рис. 2.1.).

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.

В Java используются целочисленные литералы, например: **35** — целое десятичное число, **071** — восьмеричное число, **0x51b** — шестнадцатеричное число, **0b1010** — двоичное число (введено в Java 7). 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 полагает, что оно типа **long**.

В Java 7 для удобства восприятия литералов стало возможно использовать знак «_» при объявлении больших чисел, то есть вместо **int m = 7000000** можно записать **int m = 7_000_000**. Эта форма применима и для чисел с плавающей запятой. Однако некорректно: _12 или 21_.

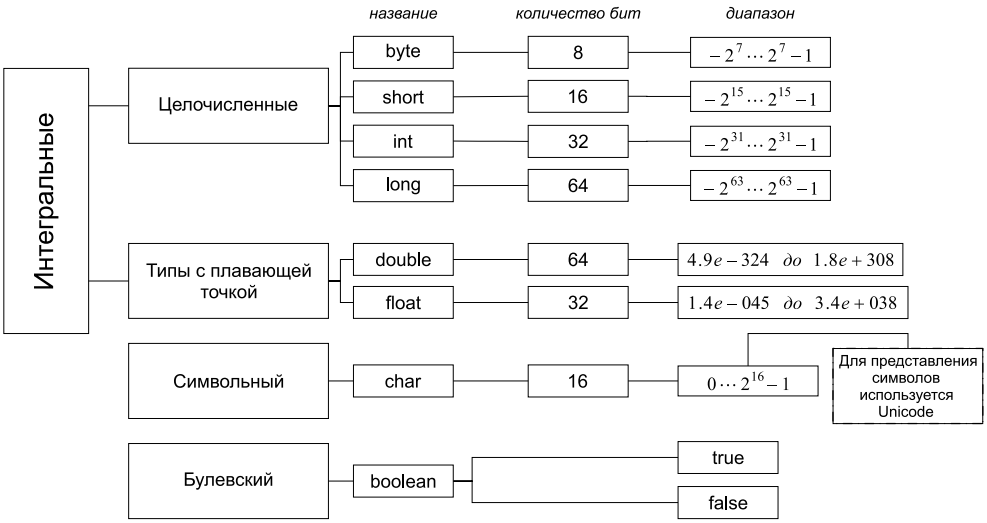

Рис. 2.1. *Базовые типы данных и их свойства*

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** или **f**. 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.

К булевским литералам относятся значения **true** и **false**. Литерал **null** — значение по умолчанию для объектной ссылки.

Символьные литералы 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. Java 7 обеспечивает поддержку стандарта Unicode 6.0.0 наличием специальных методов в классе-оболочке **Character**.

Строки, заключенные в двойные апострофы, считаются литералами и размещаются в пуле литералов, но в то же время такие строки представляют собой объекты класса **String**. При инициализации строки создается объект класса

String. При работе со строками кроме методов класса **String** можно использовать единственный в языке перегруженный оператор «+» конкатенации (объединения) строк. Конкатенация строки с объектом любого другого типа добавляет к исходному объекту-строке строковое представление объекта другого типа. Строковая константа заключается в двойные кавычки и не заканчивается символом **'\0'**, это не ASCII-строка, а объект из набора (массива) символов.

```
String s = "clown";
// создание нового объекта добавлением символа и значения базового типа
s += '2';
s = s + 4;
s += new Double(3.14159D);
// перегружен только оператор "+", то есть
// s-="c"; // ошибка, вычитать строки нельзя. Оператор "-" для строки не перегружен
```

В арифметических выражениях автоматически выполняются расширяющие преобразования типа

byte → short → int → long → float → double.

Это значит, что любая операция с участием различных типов даст результат, тип которого будет соответствовать большему из типов операндов. Например, результатом сложения значений типа **double** и **long** будет значение типа **double**.

Java автоматически расширяет тип каждого **byte** или **short** операнда до **int** в арифметических выражениях. Для сужающих диапазон значений преобразований необходимо производить явное преобразование вида **(тип)значение**. Например:

```
int i = 5;
byte b = (byte)i; // преобразование int в byte
```

При инициализации полей класса и локальных переменных методов с использованием арифметических операторов автоматически выполняется неявное приведение литералов к объявленному типу без необходимости его явного указания, если только их значения находятся в допустимых пределах. В операциях присваивания нельзя 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.

/ # 1 # типы данных, литералы и операции над ними */*

```
byte b = 1, b1 = 1 + 2;
final byte B = 1 + 2;
//b = b1 + 1; // ошибка приведения типов int в byte
/* переменная b1 на момент выполнения кода b = b1 + 1; может измениться, и выражение b1 + 1
может превысить допустимый размер byte- типа */
```

```
b = (byte)(b1 + 1);
b = B + 1; // работает
/* B - константа, ее значение определено, компилятор вычисляет 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 D = 3;
b = D; // работает
/* D –константа. Компилятор проверяет, не превышает ли ее значение допустимый размер для
типа byte, если не превышает, то ошибка не возникает */
final int D2 = 129;
//b=D2; // ошибка приведения типов, т.к. 129 больше, чем допустимое 127
b = (byte) D2;

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

Переменные в Java могут быть либо членами класса, либо переменными метода. По стандартным соглашениям 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. Каждая переменная должна быть объявлена с одним из указанных выше типов.

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 описание документа с помощью дескрипторов вида:

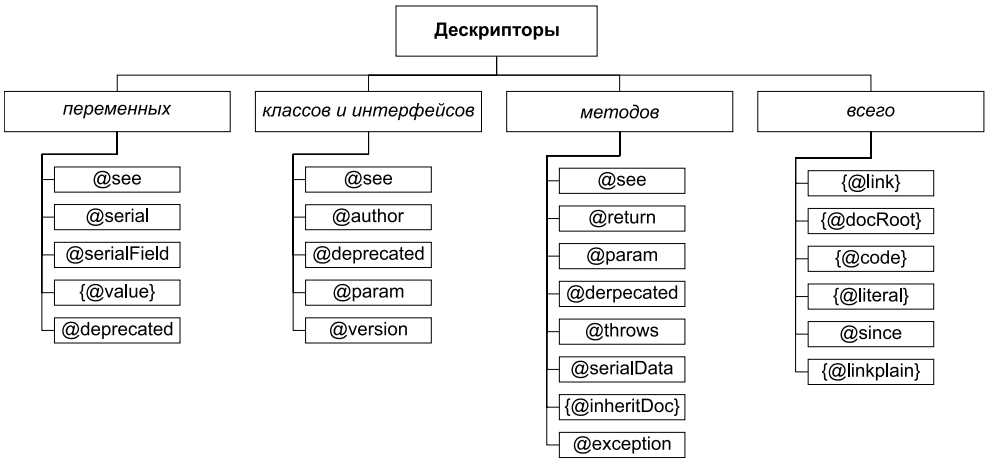

Рис. 2.2. *Дескрипторы документирования кода*

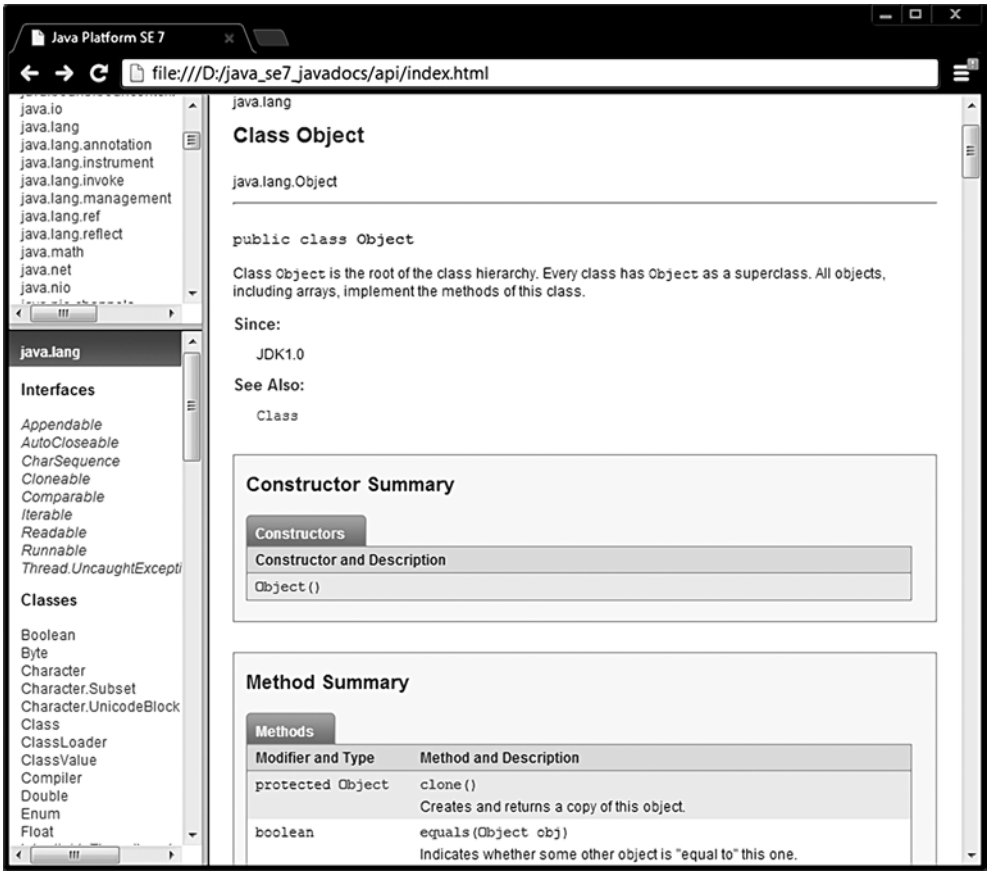

Рис. 2.3. *Сгенерированная документация для класса Object*

@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 — определяет версию, с которой метод (член класса, класс) 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 и образца для подражания следует рассматривать исходный код языка Java и документацию, сгенерированную на его основе (рис. 2.3.).

/ # 2 # фрагмент класса Object с дескрипторами документирования # Object.java */*

```
package java.lang;
/**
 * Class {@code Object} is the root of the class hierarchy.
 * Every class has {@code Object} as a superclass. All objects,
 * including arrays, implement the methods of this class.
 *
 * @author  unascribed
 * @see     java.lang.Class
 * @since   JDK1.0
 */
public class Object {
/**
     * Indicates whether some other object is "equal to" this one.
     * <p>
     *   MORE COMMENTS HERE
     * @param   obj   the reference object with which to compare.
     * @return  {@code true} if this object is the same as the obj
     *          argument; {@code false} otherwise.
     * @see     #hashCode()
     * @see     java.util.HashMap
     */
    public boolean equals(Object obj) {
        return (this == obj);
    }

    /**
     * Creates and returns a copy of this object.
     *   MORE COMMENTS HERE
     * @return     a clone of this instance.
```

```
     * @exception  CloneNotSupportedException if the object's class does not
     *               support the {@code Cloneable} interface. Subclasses
     *               that override the {@code clone} method can also
     *               throw this exception to indicate that an instance cannot
     *               be cloned.
     * @see java.lang.Cloneable
     */
    protected native Object clone() throws CloneNotSupportedException;
// more code here
}
```

Всегда следует помнить, что точные названия классов, их полей и методов улучшают восприятие кода и уменьшают размер комментариев. Наличие комментария должно еще больше облегчить скорость восприятия разработанного кода. Код системы будет читаться чаще и больше по времени, чем требуется на его создание. Комментарии помогут программисту, сопровождающему код, быстрее разобраться в нем и грамотнее использовать или изменять его.

Операторы

Операторы Java практически совпадают с операторами C++ и имеют такой же приоритет, как приведенный на рисунке 2.4. Поскольку указатели в явном виде в Java отсутствуют, то отсутствуют операторы языка C: * (унарный); &; -->. Операторы работают с базовыми типами, для которых они определены, и объектами классов-оболочек над базовыми типами. Кроме этого операторы «+» и «+=» производят также действия по конкатенации операндов 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. Деление на ноль целочисленного типа вызывает исключительную ситуацию, переполнение не контролируется.

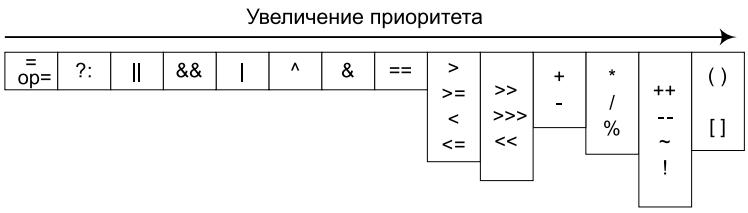

Рис. 2.4. *Таблица приоритетов операций*

Операции выполняются в определенном порядке:

Например:

```
int a = 2, b = 3, c = 4, d = 5, r;
r = a + b * c - d;
```

Первой будет выполнена операция умножения, затем в порядке очереди равноправные операции сложения и вычитания, последним выполняется присваивание как обладающее самым низким приоритетом.

Над числами с плавающей запятой выполняются арифметические операции и операции отношения, как и в других алгоритмических языках.

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		

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

// # 3 # битовые операторы и %

```
    System.out.println("5%1=" + 5%1 + " 5%2=" + 5%2);
    int b1 = 0b1110; //14
    int b2 = 0b1001; // 9
    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

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

Тернарный оператор «**?:**» используется в выражениях вида:

```
boolean_значение ? выражение_первое : выражение_второе
```

Если *boolean_значение* равно **true**, вычисляется значение выражения *выражение_первое*, и оно становится результатом всего оператора, иначе результатом является значение выражения *выражение_второе*. Например,

```
int defineBonus(int purchaseItem) {
     int bonus;
     bonus = purchaseItem > 3 ? 10 : 0 ;
     return bonus;
}
```

если число купленных предметов более трех, то клиент получает **bonus** в размере десятипроцентной скидки, в противном случае скидка не вычисляется.

Такое применение делает оператор простым для понимания, так же, как и следующий вариант:

```
experience > requirements ? acceptToProject() : learnMore();
```

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

/ # 4 # учебные курсы в учебном заведении: иерархия */*

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

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

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

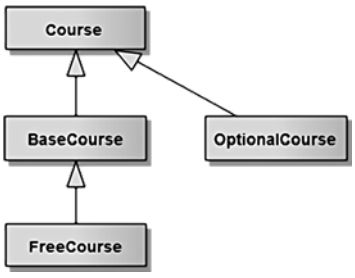

Рис. 2.5. *Иерархия наследования*

Результатом действия оператора **instanceof** будет истина, если объект является объектом данного класса 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 классу. 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.

Объект любого из этих классов представляет собой полноценный экземпляр в динамической памяти, в котором хранится его неизменяемое значение.

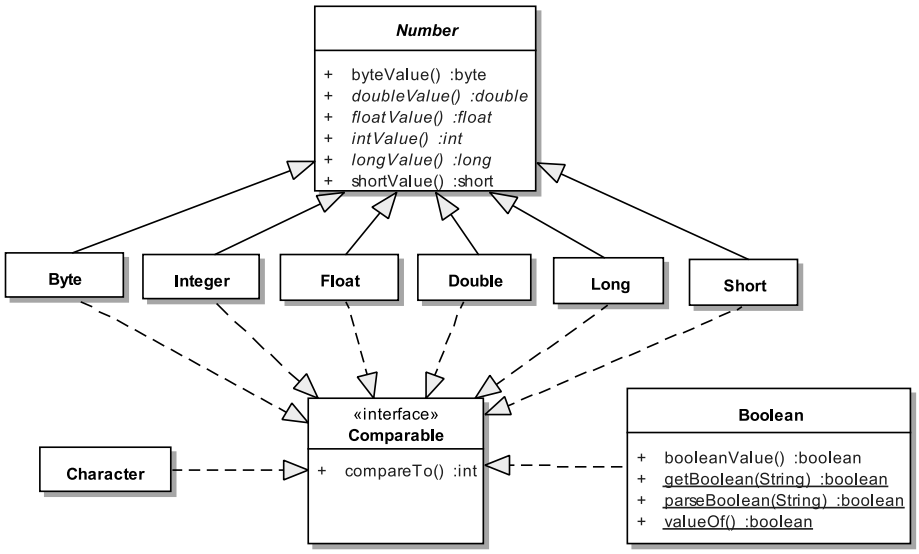

Рис. 2.6. *Иерархия классов-оболочек*

Значения базовых типов хранятся в стеке и не являются объектами. Классы, соответствующие числовым базовым типам, находятся в библиотеке **java.lang**, являются наследниками абстрактного класса **Number** и реализуют интерфейс **Comparable<T>**. Этот интерфейс определяет возможность сравнения объектов одного типа между собой с помощью метода **int compareTo(T ob).** Объекты классов-оболочек по умолчанию получают значение **null**.

Создаются экземпляры интегральных или числовых классов с помощью одного из двух конструкторов с параметрами типа **String** и соответствующего базового типа.

Объект класса-оболочки может быть преобразован к базовому типу методом ***mun*Value()** или обычным присваиванием.

Класс **Character** не является подклассом **Number**, этому классу нет необходимости поддерживать интерфейс классов, предназначенных для хранения результатов арифметических операций. Вместо этого 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**.

/ # 5 # простые преобразования типов данных */*

```
Float ft = new Float(1.7); // double в Float
Short s = new Short((short)5); // int в Short
Short sh = new Short("5"); // String в Short
double d = s.doubleValue(); // Short в double
```

```
byte b = (byte)(float)ft; // Float в byte
Character ch = new Character('3');
int i = Character.digit(ch.charValue(), 10); /* Character в int */
```

Конструкторы классов-оболочек с параметром типа **String** и их методы **valueOf(String str), decode(String str)** и **parse*Tun*(String str)** выполняют действия по преобразованию значения, заданного в виде строки, к значению соответствующего объектного типа данных. Исключение составляет класс **Character**. При преобразовании строки к конкретному типу может возникнуть ошибка формата данных, если строка не соответствует этому типу данных. Для устойчивой работы приложения все операции по преобразованию строки в типизированные значения желательно заключать в блок **try-catch** для перехвата и обработки возможного исключения.

Четыре стандартных способа преобразования строки в число:

/ # 6 # преобразование строки в целое число # StringToInt.java */*

```
package by.bsu.transformation;
public class StringToInt {
        public static void main(String[ ] args) {
                String arg = "71"; // 071 или 0x71или 0b1000111
                try {
                        int value1 = Integer.parseInt(arg); // возвращает int
                        int value2 = Integer.valueOf(arg); // возвращает Integer
                        int value3 = Integer.decode(arg); // возвращает Integer
                        int value4 = new Integer(arg); /* создает Integer,
                                        для преобразования применяется редко */
                } catch (NumberFormatException e) {
                        System.err.println("Неверный формат числа " + e);
                }
        }
}
```

У приведенных способов есть определенные различия при использовании разных систем счисления и представления чисел.

Обратное преобразование из типизированного значения (в частности **int**) в строку можно выполнить следующими способами:

```
int value = 71;
String arg1 = Integer.toString(value); // хороший способ
String arg2 = String.valueOf(value); // хороший способ
String arg3 = "" + value; // плохой способ
```

Существует 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.

Начиная с версии 5.0, введен процесс автоматической инкапсуляции данных базовых типов в соответствующие объекты оболочки и обратно (автоупаковка/

автораспаковка). При этом нет необходимости в явном создании соответствующего объекта с использованием оператора **new**:

```
Integer iob = 71; // эквивалентно Integer iob = new Integer(71);
```

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

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

вызывает ошибку компиляции.

С другой стороны, справедливо:

```
Float f = new Float("7");
```

Автораспаковка — процесс извлечения из объекта-оболочки значения базового типа. Вызовы методов **intValue(), doubleValue()** и им подобных для преобразования объектов в значения базовых типов становятся излишними.

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

```
// autoboxing & unboxing:
Integer i = 71; // создание объекта+упаковка
++i; // распаковка+операция+создание объекта+упаковка
int j = i; // распаковка
```

Однако следующий код генерирует исключительную ситуацию **NullPointerException** при попытке присвоить базовому типу значение **null** объекта класса **Integer**, литерал **null** — не объект и не может быть преобразован к значению «ноль»:

```
Integer j = null; // объект не создан! Это не ноль!
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. Для сравнения значений объектов следует использовать метод **equals().**

```
int i = 128; // заменить на 127 !!!
Integer a = i; // создание объекта+упаковка
Integer b = i;
System.out.println("a==i " + (a == i)); // true – распаковка и сравнение значений
System.out.println("b==i " + (b == i)); // true
System.out.println("a==b " + (a == b)); /* false(ссылки на разные объекты) */
System.out.println("equals ->" + a.equals(i)
                               + b.equals(i)
                               + a.equals(b)); // true, true, true
```

Метод **equals()** сравнивает не значения объектных ссылок, а значения объектов, на которые установлены эти ссылки. Поэтому вызов **a.equals(b)** возвращает значение **true**.

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

```
i.equals(a); // ошибка компиляции
```

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

```
Number n1 = 1; // идентично new Integer(1)
Number n2 = 7.1; // идентично new Double(7.1)
```

Практическое применение таких объектов крайне ограничено.

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 условного перехода **if** имеет следующий синтаксис:

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

Если выражение *boolean_значение* принимает значение **true**, то выполняется группа операторов **1**, иначе — группа операторов **2**. Оператор **else** может отсутствовать, в этом случае операторы, расположенные после окончания оператора **if** (строка 2), выполняются вне зависимости от значения булевского выражения оператора **if**.

```
if (counter > 1) {
          System.out.println("Value is Valid");
} else {
          System.out.println("Value is Broken");
}
```

В отличие от приведенного варианта

```
if (counter > 1) {
          System.out.println("Value is Valid");
}
System.out.println("More Opportunities"); // выполнится всегда
```

Если после оператора **if** следует только одна инструкция для выполнения, то фигурные скобки для выделения блока кода можно опустить

```
if (condition)
          baseCode(); // выполнение зависит от условия
          restOfCode();
```

Но при корректировке кода может понадобиться добавить строку кода, например, вызов метода **checkRole()** перед вызовом **baseCode().** Поведение программы резко изменится и не будет соответствовать идее, что метод **baseCode()** вызывается только при истинности условия **condition**.

```
if (condition)
          checkRole(); // выполнение зависит от условия
          baseCode(); // будет выполняться всегда!
          restOfCode();
```

Теперь указанный метод будет вызываться независимо от выполнения условного оператора. Во избежание таких нелепых ошибок следует использовать фигурные скобки даже при одной исполняемой строке кода. К тому же, код будет выглядеть более понятным, что немаловажно.

```
if (condition) {
          checkRole();
          baseCode();
}
          restOfCode();
```

Этих же правил следует придерживаться и для оформления циклов.

Допустимо также использование конструкции-лесенки **if-else-if**.

Оператор множественного выбора **switch:**

```
switch(value) {
       case val1: /* операторы */
              break; /* не обязателен */
       ...
       case valN: /* операторы */
              break;
       default: /* операторы */
}
```

При совпадении значения **value** со значением **val1** выполняется следующий за ним вариант. Затем, если отсутствует оператор **break**, выполняются подряд все блоки операторов до тех пор, пока не встретится оператор **break**. Если значение **value** не совпадает ни с одним из значений в **case**, то выполняется блок **default**. Значения **val1,…, valN** должны быть константами и могут иметь значения типа **int**, **byte**, **short**, **char** или **enum**. В Java7 в этот список включен и тип **String**:

/ # 7 # тип String в операторе switch */*

```
public int defineLevel(String role) {
        int level = 0;
        switch (role) { // или role.toLowerCase()
            case "guest":        level = 1;
                break;
```

```
            case "client":      level = 2;
                break;
            case "moderator":   level = 3;
                break;
            case "admin":       level = 4;
                break;
             default: throw new IllegalArgumentException(); // или собственное исключение
        }
        return level;
    }
```

Параметр типа **String** разумно применять в случае одноразового использования набора литералов из списка **case**. При многократном использовании этого набора литералов следует задуматься об организации класса-перечисления, что позволит избежать ошибок «по невнимательности» при частом включении в код обработки набора информации, содержащего строковые литералы.

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. Используется и обратный порядок в случае **if** без **else**, например, при проверке значения ссылки на **null**.

```
if (obj == null) {
          throw new IllegalArgumentException(); // один из вариантов реакции
}
```

Аналогично для оператора **switch** нормальное исполнение алгоритма следует располагать в инструкциях **case**, а именно, наиболее вероятные варианты размещаются раньше остальных, альтернативные или для значений по умолчанию — в инструкции **default**. Цепочки вызовов **if-else-if** и **switch-case** иногда при грамотном проектировании можно заменить вызовом полиморфного метода.

В Java существует четыре вида циклов. Приведем сначала первые три:

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

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

for (выражение_1; boolean_значение; выражение_3) { /* операторы */ } // цикл с параметрами
```

Циклы выполняются, пока булевское выражение *boolean_значение* равно **true**.

В цикле с параметром, по традиции, *выражение_1* — начальное выражение, *boolean_значение* — условие выполнения цикла, *выражение_3* — выражение, выполняемое в конце итерации цикла (как правило, это изменение начального значения).

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

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

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

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

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

Некоторые рекомендации при проектировании циклов:

— цикл **for** следует использовать при необходимости выполнения алгоритма строго определенное количество раз. Циклы **while** и **do** {} **while** используются в случаях, когда неизвестно точное число итераций для достижения результата, например, поиск необходимого значения в массиве или коллекции. Цикл **while(true){}** применяется в многопоточных приложениях для организации бесконечных циклов;

— для цикла **for** не рекомендуется в теле цикла изменять индекс цикла;

— в цикле **for** не следует использовать оператор **break**;

— 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 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 самого внутреннего из вложенных циклов. Такое использование является плохим примером проектирования циклов и на практике никогда не встречается.

Массивы

Массив в Java представляет собой класс, при этом имя объекта класса массива является объектной ссылкой на динамическую память, в которой хранятся элементы массива. Элементами массива, в свою очередь, могут быть значения базового типа или объекты. Элементы массива проиндексированы, индексирование элементов начинается с нуля. Для объявления ссылки на массив можно записать пустые квадратные скобки после имени типа, например: **int a[].** Аналогичный результат получится при записи **int []a**.

Массивы в языке Java являются динамическими. Существует два способа создания массива: с помощью оператора **new** или с помощью прямой инициализации присваиванием значений элементам массива в фигурных скобках. Значения элементов неинициализированного массива, для которого выделена память, устанавливаются в значения по умолчанию для массива базового типа или **null** для массива объектных ссылок.

/ # 8 # массивы и ссылки */*

```
int arRef[ ], ar; // объявление ссылки на массив и переменной
float[ ] arRefFloat, arFloat; // два массива!
// объявление с инициализацией нулевыми значениями по умолчанию
     int arDyn[ ] = new int[10]; // 10 нулей
     String str[ ] = new String[7]; // 7 null-ов
     /* объявление с инициализацией */
     int arInt[ ] = { 5, 7, 9, -5, 6, -2 }; // 6 элементов
     arInt[ ] = new int[ ] { 5, 7, 9, -5, 6, -2 }; // идентично предыдущему
     byte arByte[ ] = {1, 3, 5 };// 3 элемента
     /* объявление с помощью ссылки на Object */
     Object arObj = new float[5]; // массив является объектом
     // допустимые присваивания ссылок
     arRef = arDyn;
     arDyn = arInt;
     arRefFloat = (float[ ])arObj;
     arObj = arByte; // источник ошибки для следующей строки
     arRefFloat = (float[ ])arObj; // ошибка выполнения
     // недопустимые присваивания ссылок (нековариантность)
     // arInt = arByte;
     // arInt = (int[ ])arByte;
```

Ссылка на самый верхний в иерархии объект класса **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 **arDyn=arInt** приведет к тому, что значения элементов массива **arDyn** будут утрачены и две ссылки будут установлены на один массив **arInt**, 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**.

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

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. Добавить комментарии в программы в виде /** *комментарий* */, сгенерировать html-файл документации. В заданиях на числа объект можно создавать в виде массива символов.

Ввести *n* чисел с консоли.

1. Найти самое короткое и самое длинное число. Вывести найденные числа и их длину.
2. Упорядочить и вывести числа в порядке возрастания (убывания) значений их длины.
3. Вывести на консоль те числа, длина которых меньше (больше) средней, а также длину.
4. Найти число, в котором число различных цифр минимально. Если таких чисел несколько, найти первое из них.
5. Найти количество чисел, содержащих только четные цифры, а среди них — количество чисел с равным числом четных и нечетных цифр.

6. Найти число, цифры в котором идут в строгом порядке возрастания. Если таких чисел несколько, найти первое из них.
7. Найти число, состоящее только из различных цифр. Если таких чисел несколько, найти первое из них.
8. Среди чисел найти число-палиндром. Если таких чисел больше одного, найти второе.

Вариант B

1. Определить принадлежность некоторого значения *k* интервалам (*n, m*], [*n, m*), (*n, m*), [*n, m*].
2. Вывести числа от 1 до *k* в виде матрицы N x N слева направо и сверху вниз.
3. Найти корни квадратного уравнения. Параметры уравнения передавать с командной строкой.
4. Ввести число от 1 до 12. Вывести на консоль название месяца, соответствующего данному числу. Осуществить проверку корректности ввода чисел.

Вариант C

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 $A_{i, j}$, если $A_{i, j}$ 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 наименьший среди локальных максимумов. (Элемент матрицы называется локальным минимумом, если он строго меньше 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.

Укажите строки, компиляция которых не приведет к ошибке (3):

1) **int** var1 = 356f
2) **double** var2 = 356f
3) **float** var3 = 356f
4) **char** var4 = 356f
5) **long** var5 = 356f
6) **byte** var6 = 356f
7) Integer var7 = 356f
8) Character var8 = 356f
9) Object var9 = 356f

Вопрос 2.2.

Укажите, какие javadoc-комментарии не используются для документирования конструкторов и методов (2):

1) @see
2) @author
3) @param
4) @version

5) @throws
6) @deprecated

Вопрос 2.3.

Дан код:

```
public class Quest4 {
        public static void main (String [] args) {
                double x=0, y=2, z;
                z = y/x;
                System.out.println ("z="+z);
        }
}
```

Что выведется на консоль в результате компиляции и запуска программы (1)?

1) Ошибка компиляции
2) z=Infinity
3) z=NaN
4) Ошибка времени выполнения java.lang.ArithmeticException

Вопрос 2.4.

Что будет результатом компиляции и запуска следующего кода (1)?

```
public class Quest {
        public static void main (String [] args) {
                MedicalStaff medic = new HeadDoctor ();
                if (medic instanceof Nurse) {
                        System.out.println ("Nurse");
                } else if (medic instanceof Doctor) {
                        System.out.println ("Doctor");
                } else if (medic instanceof HeadDoctor) {
                        System.out.println ("HeadDoctor");
                }
        }
}
class MedicalStaff {}
class Doctor extends MedicalStaff {}
class Nurse extends MedicalStaff {}
class HeadDoctor extends Doctor {}
```

1) Nurse
2) Doctor
3) HeadDoctor
4) Ошибка компиляции

Вопрос 2.5.

Дан фрагмент кода **if** (e1) **if** (e2) S1; **else** S2; (e1, e2, S1, S2 — корректные java-выражения). Какому другому фрагменту кода он эквивалентен (2)?

1) **if** (e1) {**if** (e2) S1; **else** S2;}
2) **if** (e1) {**if** (e2) S1;} **else** S2;
3) **if** (e1) **if** (e2) S1; **else**; **else** S2;
4) **if** (e1) **if** (e2) S1; **else** S2; **else**;

Вопрос 2.6.

Какие из фрагментов кода неверно решают задачу «Найти сумму первых 100 натуральных чисел» (2)?

1) i = 1; sum = 0; **for** (; i <= 100; i++) sum += i;
2) sum = 0; **for** (i = 1; i <= 100;) sum += i++;
3) **for** (i = 1, sum = 0; i <= 100; sum += i+, i++);
4) **for** (i = 1, sum = 0; i <= 100; sum += i++);
5) **for** (i = 0, sum = 0; i++, i <= 100; sum += i);

Вопрос 2.7.

Какие утверждения о классах-оболочках корректны (3)?

1) Классы оболочки Double, Long, Float размещаются в пакете java.util
2) Объекты классов оболочек могут хранить те же значения, что и соответствующие им базовые типы
3) Объекты классов-оболочек хранят изменяемые значения аналогично переменным базовых типов
4) Объекты классов-оболочек по умолчанию получают значение null
5) В классах оболочках определены методы преобразования к базовому типу

Вопрос 2.8.

Дан код:

```
class Item {}
1) int [] mas1 = new int [24];
2) Integer mas2 [] = new Integer [24];
3) char [] mas3 = new Character [] {'a', 'b', 'c'};
4) Item [] mas4 = new Item {new Item (), new Item ()};
5) double [] mas5 = {5, 10, 15, 20};
6) int [] mas6 [] = new int [4] [5];
7) int mas7 [] [] = new int [4] [];
```

Компиляция каких строк приведет к ошибке (2)?

Глава 3

КЛАССЫ И ОБЪЕКТЫ

Любая действующая программа устарела.

Первый Закон Программирования

Класс соответствует новому типу данных как описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Классы — основной элемент абстракции, отвечающий за реализацию назначенного ему контракта и обеспечивающий сокрытие реализации. Классы объединяются в пакеты, которые связаны друг с другом только через ограниченное количество методов и классов, не имея никакого представления о процессах, происходящих внутри классов и методов других пакетов. 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. Под элементом следует понимать как физическую сущность (например: *Заказ*, *Товар*), так и логическую (например: *УправлениеСчетом*, *СоставлениеОтчета*).

Определение класса в общем виде без наследования и реализации интерфейсов (наследование по умолчанию только от **Object**) имеет вид:

```
[public] [final] [abstract] class ИмяКласса {
      {} // логические блоки
      // внутренние классы
                        // поля
      private // закрытые
      public // открытые
      // дружественные (по умолчанию)
      protected // защищенные

                        // конструкторы
      public // открытые
      // дружественные (по умолчанию)
      protected // защищенные
      private // закрытые

                        // методы
      public // открытые
```

```
    // дружественные (по умолчанию)
    protected // защищенные
    private // закрытые
}
```

Спецификатор доступа **public** определяет внешний (enclosing) класс, как доступный из других пакетов.

Переменные класса, экземпляра и константы

Класс создается в случае необходимости группировки данных и/или действий (методов) под общим именем, то есть создания нового типа данных, аккумулирующего свойства и действия, связанные с одной предметной областью.

Классы инкапсулируют переменные и методы — члены класса. Переменные в классе объявляются следующим образом:

```
[спецификатор] Тип имя;
```

Со спецификатором **static** объявляется для всего класса статическая переменная класса, которая имеет общее значение для всех экземпляров класса. Без спецификатора **static** объявляются переменные экземпляра класса, имеющие уникальные и независимые значения для каждого объекта класса. Поля класса, как и методы, объявляются также со спецификаторами доступа **public**, **private**, **protected** или по умолчанию без спецификатора. Кроме полей — 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.

Поля класса со спецификатором **final** являются константами и не могут быть изменены после инициализации. Спецификатор **final** можно использовать не только для поля класса, но и для локальной переменной, объявленной в методе, а также для параметра метода. Это единственный спецификатор, применяемый с параметром метода или локальной переменной.

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

/ # 1 # типы атрибутов и переменных # Order.java */*

```
package by.bsu.entity;
// класс доступен из других пакетов
public class Order {
    private int id; // переменная экземпляра класса
    static int bonus; // переменная класса
    public final int MIN_TAX = 8 + (int)(Math.random()*5); // константа экземпляра класса
    public final static int PURCHASE_TAX = 6; // константа класса
```

```
// конструкторы
// метод
    public double calculatePrice(double price, int counter) { /* параметры метода */
        double amount; // локальная переменная метода не получает значения по умолчанию
        // amount++; // ошибка компиляции, значение не задано
        amount = (price - bonus) * counter; // инициализация локальной переменной
        double tax = amount * PURCHASE_TAX /100;
        return amount + tax; // возвращаемое значение
    }
}
```

В примере в качестве переменных экземпляра класса, переменных класса и локальных переменных метода использованы данные базовых типов, не являющиеся ссылками на объекты. Поля классов могут быть ссылками на объект, 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:

- **private** — члены класса доступны только членам данного класса;
- по умолчанию (package-private) — члены класса доступны классам, находящимся в том же пакете;
- **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, конструктор или метод), объявленный **public**, доступен из любого места вне класса. Спецификатор **public** для методов и конструкторов **public**-классов обеспечивает внешний доступ к функциональности пакета, в котором он объявлен. Если данный спецификатор отсутствует, то класс и его методы могут использоваться только в текущем пакете.

Все, что объявлено **private**, доступно только конструкторам и методам внутри класса и нигде больше. Они выполняют служебную или вспомогательную роль в пределах класса, и их функциональность (методов и конструкторов) не предназначена для внешнего использования. Закрытие (**private)** полей обеспечивает инкапсуляцию.

Если у члена класса вообще не указан спецификатор уровня доступа, то такой член класса будет виден и доступен из подклассов и классов того же пакета. Именно такой уровень доступа используется по умолчанию. Такой член класса обеспечивает исполнение **public**-методами **public**-классов своей функциональности.

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**. Спецификатор применим, если поле часто используется в подклассе или если метод предназначен для переопределения и использования его функциональности в другом пакете. Для конструктора наличие **protected** обеспечивает саму возможность наследования от этого класса в другом пакете.

Действие спецификатора доступа распространяется только на тот элемент класса, перед которым стоит такой спецификатор.

Конструкторы

Конструктор — особого вида метод, который по имени автоматически вызывается при создании экземпляра класса с помощью оператора **new**. При корректном проектировании класса конструктор не должен выполнять никаких других обязанностей, кроме инициализации полей класса и проверки непротиворечивости конструирования объекта.

Конструктор имеет то же имя, что и класс; вызывается не просто по имени, а только вместе с ключевым словом **new** при создании экземпляра класса. Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым. Конструкторов в классе может быть несколько, но не менее одного.

Деструктор в языке Java не используется, объект уничтожается сборщиком мусора после определения невозможности его дальнейшего использования (потери ссылки). Некоторым аналогом деструктора является метод **finalize(),** в тело которого помещается код по освобождению занятых объектом ресурсов. Виртуальная машина станет вызывать его каждый раз, когда сборщик мусора будет уничтожать объект класса, которому не соответствует ни одна ссылка.

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

/ # 2 # конструкторы # Account.java */*

```
package by.bsu.transfer.bean;
public class Account {
    private long id;
    private double amount;
    // конструктор без параметров
    public Account() { // наличие этого конструктора некорректно по смыслу класса
        super();
        /* если класс будет объявлен вообще без конструктора, то
        компилятор предоставит его именно в таком виде */
                }
    // конструктор с параметром
    public Account(long id) {
```

```
        super(); /* вызов конструктора суперкласса явным образом
        необязателен, компилятор вставит его автоматически */
        this.id = id;
    }
    // конструктор с параметрами
    public Account(long id, double amount) {
        this.id = id;
        this.amount = amount;
    }
    public double getAmount () {
        return amount;
    }
    public void setAmount (double amount) {
        this.amount = amount;
    }
    public long getId() {
        return id;
    }
    public void setId(long id) {
                // проверка на корректность
        this.id = id;
    }
    public void addAmount (double amount) {
    /* данный метод в общем случае можно объявлять в другом классе */
        this.amount += amount;
    }
}
```

Кроме данных и методов каждый экземпляр класса (объект) имеет неявную ссылку **this** на себя, которая передается также неявно и нестатическим методам класса. После этого каждый 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. Но если в методе объявлена локальная переменная или параметр метода с таким же именем, как и поле класса, то для обращения к полю класса использование **this** обязательно. Без использования указателя обращение всегда будет производиться к локальной переменной, так как просто не существует другого способа ее идентификации.

Объект класса **Account** может быть создан тремя способами, вызывающими один из конструкторов:

```
Account a = new Account(); /* инициализация полей значениями по умолчанию
соответствующего типа, наличие нулевого id неприемлемо для логики класса */
Account b = new Account(71L); /* инициализация одного  поля переданным в конструктор
значением и другого поля по умолчанию */
Account c = new Account(71L, 0.7); /* инициализация полей переданными в конструктор
значениями */
```

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

Если конструктор в классе явно не определен, то компилятор предоставляет конструктор по умолчанию без параметров, который инициализирует каждое поле класса значением по умолчанию, соответствующим его типу, 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 при его создании всегда наделяется возможностью вызова конструктора суперкласса. Этот вызов может быть явным или неявным и всегда располагается в первой строке кода конструктора подкласса. Если конструктору суперкласса нужно передать параметры, то необходим явный вызов из конструктора порожденного класса **super(***список_параметров***).** Подробнее о конструкторе с параметрами будет сказано при рассмотрении вопросов наследования классов.

Методы

Метод — основной элемент структурирования кода. Все методы в Java объявляются только внутри классов и используются для работы с данными класса и передаваемыми параметрами. Простейшее определение метода имеет вид:

```
возвращаемыйТип имяМетода(список параметров) {
     // тело метода
     return значение; /* если нужен возврат значения (если тип
                                 возвращаемого значения не void) */
}
```

Если метод не возвращает значение, ключевое слово **return** отсутствует или записывается без возвращаемого значения в виде:

```
return ;
```

тип возвращаемого методом 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();
```

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 (включая **void**) и имя метода. В приведенном ниже объявлении метода элементы, заключенные в квадратные скобки, являются не обязательными.

```
[доступ] [static] [abstract] [final] [synchronized] [native] [<параметризация>]
                    возвращаемыйТип имяМетода(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.

При распределении обязанностей между классами выделяются так называемые классы бизнес-логики, в которые помещаются методы, обрабатывающие информацию из передаваемых им объектов. Такие классы могут и не иметь атрибутов. Если атрибуты присутствуют, то они обычно играют чисто служебную роль для облегчения методам класса выполнения их функциональности.

/ # 3 # перевод денег со счета на счет # TransferAction.java */*

```
package by.bsu.transfer;
import by.bsu.transfer.bean.Account;
public class TransferAction {
    private double transactionAmount;
    public TransferAction(double amount) { // конструктор по умолчанию не предоставляется
        if (amount > 0) {
                this.transactionAmount = amount;
        } else {
                throw new IllegalArgumentException(); // или собственное исключение
        }
    }
    public boolean transferIntoAccount(Account from, Account to) {
                // определение остатка
        double demand = from.getAmount() - transactionAmount;
                // проверка остатка и перевод суммы
        if (demand >= 0) {
            from.setAsset(demand);
            to.addAsset(transactionAmount);
            return true;
        } else {
            return false;
        }
    }
    public double getTransactionAmount () {
        return transactionAmount;
    }
// вставить метод удержания процента при переводе
}
```

Результат взаимодействия классов для решения поставленной задачи о переводе денег со счета на счет можно представить в следующем коде.

/ # 4 # создание экземпляров и выполнение действий # Runner.java */*

```
package by.bsu.transfer;
import by.bsu.transfer.bean.Account;
public class Runner {
    public static void main(String[ ] args) {
```

```
        Account from = new Account(78031864L, 258.5);
        Account to = new Account(58510009L, 12.1);
        TransferAction action = new TransferAction(52.0);
        boolean complete = action.transferIntoAccount(from, to);
        if (complete) {
            System.out.println("Сумма: " + action.getTransactionAmount() + " переведена успешно");
            System.out.print("На счету клиента ID=" + to.getId());
            System.out.println(" находится сумма: " + to.getAmount());
        } else {
            System.out.println("Транзакция не выполнена.");
            System.out.print("На счету клиента ID=" + from.getId());
            System.out.println(" недостаточно средств.");
        }
    }
}
```

В результате будет выведено:

Сумма: 52.0 переведена успешно
На счету клиента ID=58510009 находится сумма: 64.1

Класс **Runner** имеет недостаток — печатает отчет о выполнении транзакции. Для выполнения этого действия следует создать дополнительный класс **ReportAction** и возложить задачу формирования отчетов на его методы.

Объект класса может быть создан в любом пакете приложения, если конструктор класса объявлен со спецификатором **public**. Спецификатор **private** не позволит создавать объекты вне класса, а спецификатор «по умолчанию» — вне пакета. Спецификатор **protected** позволяет создавать объекты в текущем пакете и делает конструктор доступным для обращения к подклассам данного класса в других пакетах.

Статические методы и поля

Поле данных, объявленное в классе как **static**, является общим для всех объектов класса и называется переменной класса. Может быть использовано без создания экземпляра 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**. Нестатические методы могут обращаться к статическим полям и методам напрямую без всяких дополнительных условий. Статические методы являются методами класса, не привязаны ни к какому объекту и не содержат указателя **this** на конкретный экземпляр, 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 они не «знают», к какому объекту относятся, да и сам экземпляр класса может быть не создан. Для обращения к статическим полям и методам достаточно имени класса, в котором они определены. Класс **TransferAction** можно переработать в класс со статическим полем и методом.

// # 5 # статические метод и поле # TransferAction.java

```
public class TransferAction {
   public static double transactionAmount; // статическое поле
   private int id; // нестатическое поле
   public static boolean transferIntoAccount(Account from, Account to) {
                  // increaseAmount(); // вызвать нельзя – объекта не существует
                  // this.id // использовать this невозможно – объекта не существует
                  // id // недоступен – объекта не существует
      // определение остатка
       double demand = from.getAmount() - transactionAmount;
                  // проверка остатка и перевод суммы
        if (demand >= 0) {
            from.setAsset(demand);
            to.addAsset(transactionAmount);
            return true;
        } else {
            return false;
        }
    }
    public void increaseAmount() { // нестатический метод
        transactionAmount ++;
    }
}
```

Вызов метода **transferIntoAccount()** осуществляется без создания объекта:

```
TransferAction.transferIntoAccount(from, to);
```

но и переопределить статический метод уже нельзя.

Для двух объектов

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

Значение **ob1.*transactionAmount*** и **ob2.*transactionAmount*** равно **0**, поскольку располагается в одной и той же области памяти вне объекта. Такая запись не очень корректна. Поэтому обращаться и изменять значение статического поля следует непосредственно через имя класса:

```
TransferAction.transactionAmount = 70.5;
```

Вызов статического метода всегда следует осуществлять с помощью указания на имя класса, а не объекта. Статический метод можно вызывать также с использованием имени объекта, но такой вызов снижает качество кода,

приводит к появлению соответствующего предупреждения и не будет логически корректным, хотя и не закончится ошибкой компиляции.

Переопределение статических методов невозможно, так как обращение к статическому атрибуту или методу осуществляется посредством задания имени класса, которому они принадлежат.

Статические методы используются при необходимости придать функциональности метода признак «окончательности», «неизменности» реализации алгоритма для данного класса.

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, объявленных со спецификатором **final,** нельзя создавать подклассы. Например:

/ # 6 # final-поля и переменные # Card.java */*

```
package by.bsu.finalvar;
public class Card {
        // инициализированная константа экземпляра
        public final int ID = (int)(Math.random() * 10_000_000);
        // неинициализированная константа
        public final long BANK_ID; // инициализация по умолчанию не производится!
        // { BANK_ID = 11111111L; }  // только один раз!!!
        public Card (long id) {
        // инициализация в конструкторе
                BANK_ID = id; // только один раз!!!
        }
        public final boolean checkRights(final int NUMBER) {
                final int CODE = 72173394; // антишаблон: "Волшебное Число"
                // ID = 1; // ошибка компиляции!
                // NUMBER = 1; // ошибка компиляции!
                // CODE = 1; // ошибка компиляции!
                return CODE == NUMBER + ID;
        }
}
```

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.

/ # 7 # абстрактный класс и метод # AbstractCardAction.java */*

```
public abstract class AbstractCardAction {
       private Long account;
       public AbstractCardAction () { }
       /* тело абстрактного метода отсутствует */
       public abstract void doPayment(double amountPayment);
       public void setAccount(Long account) {
                  this.account = account;
       }
}
```

При этом становится невозможным создание экземпляра

```
AbstractCardAction ap = new AbstractCardAction(); // compile error
```

Спецификатор **abstract** присутствует здесь как в объявлении метода, так и в объявлении класса.

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 «Наследование и полиморфизм».

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**, представляющей 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.

/ # 8 # использование логических блоков при объявлении класса # Department.java # DemoLogic.java */*

```
package by.bsu.logic;
public class Department {
        {
                System.out.println("logic (1) id=" + this.id);
                // проверка и инициализация параметров конкретного объекта
        }
        static {
                System.out.println("static logic");
                /* проверка и инициализация базовых параметров, необходимых
                для функционирования приложения (класса) */
        }
        private int id = 7;
        public Department(int id) {
                this.id = id;
                System.out.println("конструктор id=" + id);
        }
```

```
        public int getId() {
                return id;
        }
        { /* не очень хорошее расположение логического блока */
                System.out.println("logic (2) id=" + id);
        }
}
package by.bsu.logic;
public class DemoLogic {
                public static void main(String[ ] args) {
                                new Department(71);
                                new Department(17);
                }
}
```

В результате выполнения этой программы будет выведено:

static logic
logic (1) id=0
logic (2) id=7
конструктор id=71
logic (1) id=0
logic (2) id=7
конструктор id=17

Во второй строке вывода поле **id** получит значение по умолчанию, так как память для него выделена при создании объекта, а значение еще не проинициализировано. В третьей строке выводится значение поля **id**, равное **7**, так как после инициализации атрибута класса был вызван логический блок, получивший его значение.

Перегрузка методов

Метод называется перегруженным, если существует несколько его версий с одним и тем же именем, но с разным списком параметров. Перегрузка реализует «раннее связывание», то есть версия вызываемого метода определяется на этапе компиляции. Перегрузка может ограничиваться одним классом. Методы с одинаковыми именами, но с различными списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут перегруженными. Если списки параметров идентичны, то имеет место механизм динамического полиморфизма — 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.

/ # 9 # вызов перегруженных методов # NumberInfo.java */*

```
package by.bsu.overload;
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 надо 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 параметризации (generic) классов и методов, позволяющей использовать 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.

Ниже приведен пример generic-класса с двумя параметрами:

/ # 10 # обьявление класса с двумя параметрами # Post.java */*

```
package by.bsu.forum;
public class Post <T1, T2 extends Number> {
          private T1 message;
          private T2 id;
          // методы
}
```

Здесь **T1**, **T2** — фиктивные объектные типы, которые используются при объявлении членов класса и обрабатываемых данных. В качестве типа **T2** допустимо использовать только подклассы класса **Number**. В качестве параметров классов запрещено применять базовые типы.

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

```
Post<String, Short> post1 = new Post<String, Short>();
Post<StringBuilder, Long> post2 = new Post<StringBuilder, Long>();
```

или

```
Post<StringBuilder, Long> post2 = new Post<>(); // оператор diamond в Java 7
```

Параметризированные типы обеспечивают типобезопасность. Присваивание **post1=post2** приводит к ошибке компиляции.

При создании объекта компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект, при этом все внешние признаки

параметризации исчезнут, то есть проверка на принадлежность типу осуществима только в виде:

```
post1 instanceof Post
```

тогда как, будет ошибочно

```
post1 instanceof Post<String, Short>
```

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

/ # 11 # создание и использование обьектов параметризованного класса # Message.java # Runner.java */*

```
package by.bsu.template;
public class Message <T> {
          private T value;
          public Message() {
          }
          public Message (T value) {
                    this.value = value;
          }
          public T getValue() {
                    return value;
          }
          public void setValue(T value) {
                    this.value = value;
          }
          public String toString() {
                    if (value == null) {
                              return null;
                    }
              return value.getClass().getName() + " :" + value;
          }
}
package by.bsu.template;
public class Runner {
          public static void main(String[ ] args) {
          // параметризация типом Integer
                    Message<Integer> ob1 = new Message<Integer>();
                    ob1.setValue(1); // возможен только тип Integer для метода setValue
                    int v1 = ob1.getValue();
                    System.out.println(v1);
          // параметризация типом String
                    Message<String> ob2 = new Message<String>("Java");
                    String v2 = ob2.getValue();
                    System.out.println(v2);

          // ob1 = ob2; // ошибка компиляции – параметризация нековариантна
```

```
        // параметризация по умолчанию – Object
                Message ob3 = new Message(); // warning – raw type
                ob3 = ob1; // нет ошибки компиляции – нет параметризации
                System.out.println(ob3.getValue());
                ob3.setValue(new Byte((byte)1));
                ob3.setValue("Java SE 7");
                System.out.println(ob3); /* выводится тип объекта,
                                            а не тип параметризации */
                ob3.setValue(71);
                System.out.println(ob3);
                ob3.setValue(null);
         }
}
```

В результате выполнения этой программы будет выведено:

```
1
Java
null
java.lang.String: Java SE 7
java.lang.Integer: 71
```

В рассмотренном примере были созданы объекты типа **Message**: **ob1** на основе типа **Integer** и **ob2** на основе типа **String** при помощи различных конструкторов. При компиляции вся информация о generic-типах стирается и заменяется 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 ValueExt <T extends Тип> {
       private T value;
       // поля, конструкторы, методы
}
```

Такая запись говорит о том, что в качестве типа **T** разрешено применять только классы, являющиеся наследниками (подклассами) реального 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.

/ # 12 # использование метасимвола в параметризованном классе # Exam.java # Runner.java */*

```
package by.bsu.exam;
public class Exam<T extends Number> {
          private String name;
          private T mark; // параметр поля
          public Exam(T mark, String name) { // параметр конструктора
                    this.name = name;
                    this.mark = mark;
          }
          public T getMark() { // параметр метода
                    return mark;
          }
          private int roundMark() {
                    return Math.round(mark.floatValue()); // метод класса Number
          }
          public boolean equalsToMark(Exam<T> ob) { // параметр метода
                    return roundMark() == ob.roundMark();
          }
}
package by.bsu.exam;
public class Runner {
          public static void main(String[ ] args) {
          Exam<Double> md1 = new Exam<Double>(71.41D,"Progr");// 71.5d
          Exam<Double> md2 = new Exam<Double>(71.45D, "Progr");// 71.5d
                    System.out.println(md1.equalsToMark(md2));
                    Exam<Integer> mi = new Exam<Integer>(71,"Progr");
                    // md1.equalsToMark(mi); // ошибка компиляции: несовместимые типы
          }
}
```

В результате будет выведено:

true

Метод с параметром **Exam<T>** может принимать исключительно объекты с инициализацией того же типа, что и вызывающий метод объект. Чтобы метод **equalsToMark()** мог распространить свои возможности на экземпляры класса **Exam**, инициализированные любым допустимым типом, его следует переписать с использованием метасимвола «**?**» в виде:

```
public boolean equalsToMark(Exam<?> ob) {
             return roundMark() == ob.roundMark();
}
```

Тогда при вызове **md1.equalsToMark(mi)** ошибки компиляции не возникнет и метод выполнит свою расширенную функциональность по сравнению объектов класса **Exam**, инициализированных объектами различных допустимых типов. В противном случае было бы необходимо создавать новые перегруженные методы.

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

```
class FailedOne <T> {
   private 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:

/ # 13 # неправильное обьявление и использование полей параметризованного класса # FailedTwo.java */*

```
class FailedTwo <T1, T2> {
          static T1 value;
          T2 id;
          static T1 takeValue() {
             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> T[ ] methodName(int count, 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 generic-класса. Причем такому методу разрешено быть статическим, так как параметризацию обеспечивает сам метод, а не класс, в котором он объявлен. Метасимволы применимы и к generic-методам.

/ # 14 # параметризованные конструкторы и методы # SimpleActionCourse.java */*

```
package by.bsu.genericmethod;
public class SimpleActionCourse {
       public <T1 extends Course> SimpleActionCourse(T1 course) { // конструктор
         // реализация
       }
       public <T2> SimpleActionCourse() { // конструктор
         // реализация
       }
       public <T3 extends Course> float calculateMark(T3 course) {
         // реализация
       }
       public <T4> boolean printReport(T4 course) {
         // реализация
       }
       public <T5> void check() {
         // реализация
       }
}
```

Создание экземпляра с параметром и вызов параметризованного метода с параметром выглядят следующим образом:

```
SimpleActionCourse sap = new SimpleActionCourse(new Course());
sap.printReport(new Course(7112));
```

Создание экземпляра с использованием параметризованного конструктора без параметров требует указания типа параметра перед именем конструктора

```
SimpleActionCourse sa = new <String>SimpleActionCourse();
```

Аналогично для метода без параметров

```
sa.<Integer>check();
```

Методы с переменным числом параметров

Возникают ситуации, когда заранее неизвестно количество передаваемых экземпляров класса в метод. В обычной ситуации пришлось бы создавать несколько перегруженных методов с разным числом параметров одного типа. Другим решением будет один метод с параметром в виде массива или коллекции, что потребует предварительной организации соответствующего объекта массива или коллекции.

Начиная с версии Java 5, появилась возможность передачи в метод нефиксированного числа параметров, что позволяет отказаться от предварительного создания сложного объекта для его последующей передачи в метод. Набор объектов, переданный в такой метод, преобразуется в массив с типом и именем, которые указаны в качестве параметров метода. Метод **printf()** с переменным числом аргументов уже применялся неоднократно в примерах предыдущих глав.

Список параметров метода выглядит в общем случае:

```
(Тип... args)
```

а в случае необходимости передачи параметров других типов

```
(Тип1 t1, Тип2 t2, ТипN tn, Тип... args)
```

/ # 15 # определение количества параметров метода # DemoVarargs.java */*

```
package by.bsu.varargs;
public class DemoVarargs {
          public static int defineArgCount(Integer... args) {
                    if (args.length == 0) {
                              System.out.print("No arg ");
                    } else {
                              for (int element : args) {
                                        System.out.printf("arg:%d ", element);
                              }
                    }
                    return args.length;
          }
          public static void main(String … args) {
            System.out.println("N=" + defineArgCount(7, 71, 555));
            Integer[] arr = { 1, 2, 3, 4, 5, 6, 7 };
            System.out.println("N=" + defineArgCount(arr));
            System.out.println(defineArgCount());
            // defineArgCount(arr, arr); // ошибка компиляции
            // defineArgCount(71, arr); // ошибка компиляции
          }
}
```

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 **defineArgCount()** 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:

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

Методы с переменным числом аргументов могут быть перегружены:

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

Недопустимый способ:

```
void methodName(Integer ... args) { }
void methodName(Integer[ ] 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:

/ # 16 # передача массивов # DemoOverload.java */*

```
package by.bsu.overload;
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 при перегрузке!
          }
}
```

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, указанный параметр будет ближайшим. Метод с параметром **Integer[]…args** не вызывается, потому что 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 единственным и последним в объявлении списка параметров метода, то есть записи вида: **(Тип1… args, Тип2 obj)** и **(Тип1… args, Тип2… args)** приведут к ошибке компиляции.

Перечисления

Типобезопасные перечисления (typesafe enums) в Java представляют собой классы и являются подклассами абстрактного класса **java.lang.Enum**. Вместо слова **class** при описании перечисления используется слово **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:

/ # 17 # применение перечисления # MusicStyle.java # MelomanRunner.java */*

```
package by.bsu.enums;
public enum MusicStyle {
          JAZZ, CLASSIC, ROCK, BLUES
}
package by.bsu.enums;
public class MelomanRunner {
          public static void main(String[ ] args) {
                    MusicStyle ms = MusicStyle.CLASSIC; // инициализация
                    System.out.print (ms);
                    switch (ms) {
                      case JAZZ:
                              System.out.println(" is Jazz");
                              break;
                      case CLASSIC:
                              System.out.println(" is Classic");
                              break;
```

```
                    case ROCK:
                            System.out.println(" is Rock");
                            break;
                    default:
                            System.out.println(" Unknown music style: " + ms);
                }
        }
}
```

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 элемент **enum** может использовать методы:

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

static <T extends Enum<T>> T valueOf(Class<T> enumType, String arg) — создает элемент перечисления, соответствующий заданному типу и значению передаваемой строки;

static enumType valueOf(String arg) — создает элемент перечисления, соответствующий значению передаваемой строки;

int ordinal() — возвращает позицию элемента перечисления;

String name() — возвращает имя элемента;

int compareTo(T obj) — сравнивает элементы на больше-меньше либо равно.

А именно:

```
MusicStyle ms = MusicStyle.valueOf("RocK".toUpperCase());
```

Класс перечисления может содержать методы, и, следовательно, экземпляры перечисления могут к этим методам обращаться.

/ # 18 # обьявление перечисления с методом # Shape.java */*

```
package by.bsu.enums;
public enum Shape {
        RECTANGLE, TRIANGLE, CIRCLE;
        // метод класса перечисления
        public double defineSquare(double ... x) {
                // проверка параметров
                switch (this) {
                  case RECTANGLE:
                         return x[0] * x[1];
                  case TRIANGLE:
                         return x[0] * x[1] / 2;
                  case CIRCLE:
```

```
                        return Math.pow(x[0], 2) * Math.PI;
                default:
                        throw new EnumConstantNotPresentException(
                                                        this.getDeclaringClass(),this.name());
                }
        }
}
```

/ # 19 # применение перечисления # ShapeRunner.java */*

```
package by.bsu.enums;
public class ShapeRunner {
        public static void main(String args[ ]) {
                double x = 2, y = 3;
                Shape sh;
                sh = Shape.RECTANGLE;
                System.out.printf("%10s = %5.2f%n", sh, sh.defineSquare (x, y));
                sh = Shape.TRIANGLE;
                System.out.printf("%10s = %5.2f%n", sh, sh.defineSquare (x, y));
                sh = Shape.CIRCLE;
                System.out.printf("ё %10s = %5.2f%n", sh, sh.defineSquare (x));
        }
}
```

В результате будет выведено:

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

Каждый из элементов перечисления в данном случае содержит арифметическую операцию, ассоциированную с методом **defineSquare().** 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 введении нового элемента желательно добавлять соответствующий ему **case**.

Перечисление является классом, поэтому в его теле можно объявлять кроме методов также поля и конструкторы. Все конструкторы вызываются автоматически при инициализации любого из элементов. Конструктор не может быть объявлен со спецификаторами **public** и **protected**, так как не вызывается явно и перечисление не может быть суперклассом. Поля перечисления используются для сохранения дополнительной информации, связанной с его элементом.

/ # 20 # конструкторы и члены перечисления # TaxiStation.java # TaxiRunner.java */*

```
package by.bsu.enums;
public enum TaxiStation {
        MERCEDES(10), TOYOTA(7), VOLVO;
```

```
        private int freeCabs; // поле класса перечисления
        TaxiStation() { // конструктор класса перечисления
        }
        TaxiStation(int cabs) { // конструктор класса перечисления
                freeCabs = cabs;
        }
        public int getFreeCabs() {
                return freeCabs;
        }
        public void setFreeCabs(int cabs) {
                freeCabs = cabs;
        }
        @Override
        public String toString() {
                return String.format("%s : free cabs = %d", name(), freeCabs);
        }
}
package by.bsu.enums;
public class TaxiRunner {
        public static void main(String[ ] args) {
               TaxiStation ts = TaxiStation.valueOf(TaxiStation.class,"Volvo".toUpperCase());
               System.out.println(ts + " : ordinal -> " + ts.ordinal());
               ts.setFreeCabs(3);
               TaxiStation[ ] station = TaxiStation.values();
               for (TaxiStation element : station) {
                        System.out.println(element);
               }
        }
}
```

В результате будет выведено:

VOLVO: free cabs = 0: ordinal -> 2
MERCEDES: free cabs = 10
TOYOTA: free cabs = 7
VOLVO: free cabs = 3

Метод **toString()** реализован в классе **Enum** для вывода элемента в виде строки. Если переопределить метод **toString()** в конкретной реализации перечисления, то можно выводить не только значение элемента, но и значения его полей, то есть предоставить полную информацию об объекте, как и определяется контрактом метода.

Однако на перечисления накладывается целый ряд ограничений.

Им запрещено:

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

Immutable

Если в системе необходим объект, внутреннее состояние которого нельзя изменить, то процедура реализации такой задачи представляется в виде:

/ # 21 # класс для создания неизменяемых обьектов # ImmutableObject.java */*

```
package by.bsu.immutable;
public class ImmutableObject {
    private String name;
    private int id;
    public ImmutableObject (String name, int id) {
        this.name = name;
        this.id = id;
    }
    public int getName() {
        return name;
    }
    public int getId() {
        return id;
    }
}
```

Такой объект от создания и до уничтожения не может быть изменен, что уменьшает затраты на безопасность при использовании в конкурирующих операциях. Классов с неизменяемым внутренним состоянием в стандартной реализации Java достаточно много. Следует вспомнить класс **String**.

Класс с поведением **Immutable**, тем не менее, может содержать методы для создания объекта того же типа с внутренним состоянием, отличающимся от исходного, что оправданно с точки зрения ресурсов, только если такие изменения происходят не слишком часто.

Декомпозиция

Корпоративные информационные системы предоставляют пользователю огромное количество сервисов и манипулируют очень большим количеством самой разнообразной информации. В разработке таких сложных и многообразных систем участвуют зачастую сотни программистов-разработчиков. Такие системы состоят из большого количества подсистем и программных модулей, которые взаимодействуют между собой через ограниченное число интерфейсов (методов). Главное же заключается в том, что система состоит из сотен тысяч (миллионов) строк кода, не может создаваться, существовать, развиваться и поддерживаться, если при ее разработке не использовались принципы объектно-ориентированного программирования, в частности, декомпозиция.

Под *декомпозицией* следует понимать определение физических и логических сущностей, а также принципов их взаимодействия на основе анализа предметной области создаваемой системы.

Все системы состоят из классов. Все классы системы взаимодействуют с теми или иными классами этой же системы.

Объяснение принципов декомпозиции можно рассмотреть на простейшем примере. Пусть требуется решить следующую задачу: *создать систему, позволяющую умножать целочисленные матрицы друг на друга*.

Начинающий программист, знающий о том, что существуют классы, конструкторы и методы, может предложить решение поставленной проблемы в следующем виде:

/ # 22 # произведение двух матриц # Matrix.java */*

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

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

Программа полностью работоспособна, но следует взглянуть на нее внимательнее:

- создан только один класс; маловато, но и задача невелика;
- класс обладает лишними полями, значения которых зависят от значений других полей;
- в классе объявлены два конструктора, оба выделяют память под матрицу и заполняют ее элементами, переданными или сгенерированными. Оба конструктора решают похожие задачи и не проверяют на корректность входные значения и решают слишком обширные задачи;
- определен метод **show()** для вывода матрицы на консоль, что ограничивает способы общения класса с внешними для него классами;
- задача умножения решается в методе **main(),** и класс является одноразовым, т. е. для умножения двух других матриц придется код умножения копировать в другое место;
- реализован только основной положительный сценарий, например, не выполняется проверка размерности при умножении, и, как следствие, отсутствует реакция приложения на некорректные данные.

Ниже приведена попытка переработки (рефакторинга) созданного приложения таким образом, чтобы существовала возможность поддержки и расширения возможности системы при возникновении сопутствующих задач.

/ # 23 # класс хранения матрицы # Matrix.java */*

```
package by.bsu.task.entity;
import by.bsu.task.exceptions.MatrixException;
public class Matrix {
```

```
    private int[ ][ ] a;
    public Matrix(int n, int m) throws MatrixException {
           // проверка на отрицательные значения размерности матрицы
           if ((n < 1) || (m < 1)) {
                      throw new MatrixException();
           }
           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) throws MatrixException {
       if (checkRange(i, j)) { // проверка i и j
           return a[i][j];
       }
           throw new MatrixException();
    }
    public void setElement(int i, int j, int value) throws MatrixException {
       if (checkRange(i, j)) { // проверка i и j
             a[i][j] = value;
       }
           throw new MatrixException();
    }
    @Override
    public String toString() {
      StringBuilder s = new StringBuilder("\nMatrix : " + a.length + "x" + a[0].length + "\n");
      for (int [ ] row : a) {
          for (int value : row) {
              s.append(value + " ");
          }
          s.append("\n");
      }
      return s.toString();
    }
    // проверка возможности выхода за пределы матрицы
    private boolean checkRange(int i, int j) {
        if ( i < 0 || i > a.length - 1 || j < 0 || j > a[0].length - 1) {
            return false;
        } else {
            return true;
        }
    }
}
```

В классе **Matrix** объявлен **private**-метод **checkRange(int i, int j)** для проверки параметров на предельные допустимые значения во избежание невынужденных

ошибок. Однако условный оператор **if** в этом методе выглядит запутанным, во-первых, нарушая правило положительного сценария, то есть разумно ожидать, что параметры метода с индексами элемента матрицы будут корректными, и, исходя из этого, строить условие, а в представленном варианте все наоборот; во-вторых, при значении **true** в условии оператора метод возвращает значение **false**, что выглядит противоречивым.

Метод следует переписать в виде

```
private boolean checkRange(int i, int j) {
          if ( i >= 0 && i < a.length && j >= 0 && j < a[0].length ) {
                     return true;
          } else {
                     return false;
          }
}
```

заменив условие в операторе на противоположное и возвращая непротиворечащее результате значение.

/ # 24 # класс-создатель матрицы # MatrixCreator.java */*

```
package by.bsu.task.creator;
import by.bsu.task.entity.Matrix;
import by.bsu.task.exceptions.MatrixException;
public class MatrixCreator {
    public void fillRandomized(Matrix t, int start, int end) {
           int v = t.getVerticalSize();
           int h = t.getHorizontalSize();
      for(int i = 0; i < v; i++) {
         for(int j = 0; j < h; j++) {
           try {
                       int value = (int)(Math.random() * (end - start) + start));
                       t.setElement(i, j, value);
           } catch (MatrixException e) {
                    /* в данном случае exception невозможен, поэтому обработка отсутствует */
           }
       }
    }
  }
          // public void fillFromFile(Matrix t, File f) { /* код*/ }
          // public void fillFromStream(Matrix t, InputStream input) { /* код*/ }
          // public void fillFromDataBase(Matrix t, Connection conn) { /* код*/ }
}
```

Инициализация элементов матрицы различными способами вынесена в отдельный класс, методы которого могут в зависимости от условий извлекать значения для инициализации элементов из различных источников.

/ # 25 # произведение двух матриц # Multiplicator.java */*

```
package by.bsu.task.action;
import by.bsu.task.entity.Matrix;
import by.bsu.task.exceptions.MatrixException;
public class Multiplicator {
    public Matrix multiply(Matrix p, Matrix q) throws MatrixException {
        int v = p.getVerticalSize();
        int h = q.getHorizontalSize();
        int temp = p.getHorizontalSize();
        // проверка возможности умножения
        if (temp != q.getVerticalSize()) {
            throw new MatrixException(); // Incompatible matrices
        }
        // создание матрицы результата
        Matrix result = new Matrix(v, h);
        try {
            // 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);
                }
            }
        } catch (MatrixException e) {
                // exception невозможен по логике кода
                // обработка опущена
        }
                return result;
    }
}
```

Все манипуляции взаимодействия объектов-матриц между собой и другими объектами должны быть сгруппированы и вынесены в отдельные логические классы, что повышает возможность другому разработчику быстро разобраться в смысле класса и модернизировать его.

/ # 26 # исключительная ситуация при индексировании обьекта-матрицы # MatrixException.java */*

```
package by.bsu.task.exceptions;
public class MatrixException extends Exception {}
```

Создание собственных исключений позволяет разработчику при их возникновении быстро локализовать и исправить ошибку.

/ # 27 # класс, запускающий приложение # Runner.java */*

```
package by.bsu.task;
import by.bsu.task.action.Multiplicator;
import by.bsu.task.creator.MatrixCreator;
import by.bsu.task.entity.Matrix;
import by.bsu.task.exceptions.MatrixException;
public class Runner {
    public static void main(String[ ] args) {
        try {
            Matrix p = new Matrix(2, 3);
            MatrixCreator.fillRandomized(p, 2, 8);
            System.out.println("Matrix first is: " + p);
            Matrix q = new Matrix(3, 4);
            MatrixCreator.fillRandomized(q, 2, 7);
            System.out.println("Matrix second is: " + q);
            Multiplicator mult = new Multiplicator();
            System.out.println("Matrices product is " + mult.multiply(p, q));
        } catch (MatrixException ex) {
                System.err.println("Error of creating matrix " + ex);
        }
    }
}
```

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

Matrix first is:
Matrix: 2x3
3 4 7
4 7 2
Matrix second is:
Matrix: 3x4
3 3 6 6
5 6 5 5
4 3 3 4
Matrices product is
Matrix: 2x4
57 54 59 66
55 60 65 67

Выше был приведен пример простейшей декомпозиции. При разработке приложений любой сложности всегда следует производить анализ предметной области, определять абстракции и разделять задачу на логические взаимодействующие части. Тем не менее границы, отделяющие хороший дизайн приложения от посредственного, достаточно размыты и зависят от общего уровня компетентности команды разработчиков и правил, принятых в проекте.

Рекомендации при проектировании классов

При создании класса следует давать ему такое имя, чтобы его пользователю была понятна роль класса.

Класс должен быть разработан так, чтобы внесение в него изменений было относительно простой задачей.

Каждый класс должен иметь простое назначение.

Код конструктора должен заниматься только инициализацией объекта. Следует избегать вызовов из конструктора других методов, за исключением **final**. Метод может быть переопределен в подклассе и исказить процесс инициализации объекта.

Если класс отвечает за хранение информации, то функциональность работы с этой информацией должна быть базовой. Манипулированием информацией через объект должны заниматься другие классы, которых может оказаться очень много.

Использовать инкапсуляцию нестатических и неконстантных полей.

Применять для доступа к полям классов, хранящих информацию, корректные методы типа **get**, **set, is**, а также желательно реализовать методы **equals(), hashCode(), clone(), toString()** и имплементировать интерфейсы **Comparable** и **Serializable**.

Если разрабатываемый класс кажется сложным, следует разбить его на несколько простых.

По возможности избегать слишком длинных методов. От тридцати строк — длинный метод. Следует разбить метод на несколько, или даже создать для этой цели новый класс.

Если метод используется только другими методами этого класса, следует объявлять его как **private**.

Определить и распределить по разным классам функциональности, которые могут изменяться в процессе разработки, от тех, которые будут постоянными.

Хороший дизайн кода отличается высоким уровнем декомпозиции.

Если в разных участках класса или нескольких классов востребован один и тот же фрагмент кода, следует выделить его в отдельный метод.

Избегать длинного списка аргументов. Приближаясь к числу семь, список аргументов становится не воспринимаемым при чтении. Возможно, следует объединить группы аргументов в новый тип данных.

Не использовать «волшебные числа», «волшебные строки». Логичнее вынести их за пределы кода, например, в файл.

Задания к главе 3

Вариант А

Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы **set***Tun***(), get***Tun***(), toString().** Определить дополнительно

методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль. В каждом классе, обладающем информацией, должно быть объявлено несколько конструкторов.

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 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, у которых сумма баллов выше заданной;
   c) выбрать заданное число *n* абитуриентов, имеющих самую высокую сумму баллов (вывести также полный список абитуриентов, имеющих полупроходную сумму).
5. **Book**: id, Название, Автор (ы), Издательство, Год издания, Количество страниц, Цена, Тип переплета.
   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 заданного срока;
    c) список автобусов, пробег у которых больше заданного расстояния.
12. **Airline**: 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.

Вариант В

Реализовать методы сложения, вычитания, умножения и деления объектов (для тех классов, объекты которых могут поддерживать арифметические действия).

1. Определить класс **Дробь** (**Рациональная Дробь**) в виде пары чисел *m* и *n*. Объявить и инициализировать массив из *k* дробей, ввести/вывести значения для массива дробей. Создать массив/список/множество объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента.
2. Определить класс **Комплекс**. Создать массив/список/множество размерности n из комплексных координат. Передать его в метод, который выполнит сложение/умножение его элементов.
3. Определить класс **Квадратное уравнение**. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать массив/список/множество объектов и определить наибольшие и наименьшие по значению корни.
4. Определить класс **Полином** степени *n*. Объявить массив/список/множество из *m* полиномов и определить сумму полиномов массива.
5. Определить класс **Интервал** с учетом включения/невключения концов. Создать методы по определению пересечения и объединения интервалов, причем интервалы, не имеющие общих точек, пересекаться/объединятся не могут. Объявить массив/список/множество и *n* интервалов и определить расстояние между самыми удаленными концами.
6. Определить класс **Точка** на плоскости (в пространстве) и во времени. Задать движение точки в определенном направлении. Создать методы по определению скорости и ускорения точки. Проверить для двух точек возможность пересечения траекторий. Определить расстояние между двумя точками в заданный момент времени.
7. Определить класс **Треугольник** на плоскости. Определить площадь и периметр треугольника. Создать массив/список/множество объектов и подсчитать количество треугольников разного типа (равносторонний, равнобедренный, прямоугольный, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
8. Определить класс **Четырехугольник** на плоскости. Определить площадь и периметр четырехугольника. Создать массив/список/множество объектов и подсчитать количество четырехугольников разного типа (квадрат, прямоугольник, ромб, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
9. Определить класс **Окружность** на плоскости. Определить площадь и периметр. Создать массив/список/множество объектов и определить группы окружностей, центры которых лежат на одной прямой. Определить наибольший и наименьший по площади (периметру) объект.

10. Определить класс **Прямая** на плоскости (пространстве). Определить точки пересечения прямой с осями координат. Определить координаты пересечения двух прямых. Создать массив/список/множество объектов и определить группы параллельных прямых.

Вариант С

1. Определить класс **Полином** с коэффициентами типа **Рациональная Дробь**. Объявить массив/список/множество из *n* полиномов и определить сумму полиномов массива.
2. Определить класс **Прямая** на плоскости (в пространстве), параметры которой задаются с помощью **Рациональной Дроби**. Определить точки пересечения прямой с осями координат. Определить координаты пересечения двух прямых. Создать массив/список/множество объектов и определить группы параллельных прямых.
3. Определить класс **Полином** с коэффициентами типа **Комплексное число**. Объявить массив/список/множество из m полиномов и определить сумму полиномов массива.
4. Определить класс **Дробь** в виде пары (*m*, *n*) с коэффициентами типа **Комплексное число**. Объявить и инициализировать массив из *k* дробей, ввести/вывести значения для массива дробей. Создать массив/список/множество объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента.
5. Определить класс **Комплекс**, действительная и мнимая часть которой представлены в виде **Рациональной Дроби**. Создать массив/список/множество размерности n из комплексных координат. Передать его в метод, который выполнит сложение/умножение его элементов.
6. Определить класс **Окружность** на плоскости, координаты центра которой задаются с помощью **Рациональной Дроби**. Определить площадь и периметр. Создать массив/список/множество объектов и определить группы окружностей, центры которых лежат на одной прямой. Определить наибольший и наименьший по площади (периметру) объект.
7. Определить класс **Точка** в пространстве, координаты которой задаются с помощью **Рациональной Дроби**. Создать методы по определению расстояния между точками и расстояния до начала координат. Проверить для трех точек возможность нахождения на одной прямой.
8. Определить класс **Точка** в пространстве, координаты которой задаются с помощью **Комплексного числа**. Создать методы по определению расстояния между точками и расстояния до начала координат.
9. Определить класс **Треугольник** на плоскости, вершины которого имеют тип **Точка**. Определить площадь и периметр треугольника. Создать массив/список/множество объектов и подсчитать количество треугольников разного

типа (равносторонний, равнобедренный, прямоугольный, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.

10. Определить класс **Четырехугольник** на плоскости, вершины которого имеют тип **Точка**. Определить площадь и периметр четырехугольника. Создать массив/список/множество объектов и подсчитать количество четырехугольников разного типа (квадрат, прямоугольник, ромб, произвольный). Определить для каждой группы наибольший и наименьший по площади (периметру) объект.
11. Определить класс **Вектор**. Реализовать методы 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.
12. Определить класс **Вектор**. 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.
13. 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 компланарные векторы.
14. Определить класс **Булева матрица** (**BoolMatrix**). Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.
15. Построить класс **Булев вектор** (**BoolVector**). Реализовать методы для выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.
16. Определить класс **Множество символов**. 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.
17. Определить класс **Нелинейное уравнение** для двух переменных. Реализовать метод определения корней методом биекции.
18. Определить класс **Определенный интеграл** с аналитической подынтегральной функцией. 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).

19. Определить класс **Массив**. 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).

Тестовые задания к главе 3

Вопрос 3.1.

Какие описания класса содержат синтаксическую ошибку? Код написан в файле Quest1.java (3)

1) **public class** Quest1 {}
2) **public static class** Quest1 {}
3) **public abstract final class** Quest1 {}
4) **private class** Quest1 {}
5) **final class** Quest1 {}

Вопрос 3.2.

Выберите правильное утверждение, подходящее для окончания фразы «Константное поле может быть проинициализировано …» (3):

1) только один раз;
2) один раз при объявлении, а затем в конструкторе класса;
3) в логическом блоке инициализации;
4) в статическом блоке инициализации;
5) при объявлении или в конструкторе класса.

Вопрос 3.3.

Дан код:

```
public class Quest3 {
        public static int method () {
                final int loc;
                System.out.println (loc);//1
                loc=4;//2
                return loc+1;//3
        }
        public static void main (String [] args) {
                method (); method (); method ();
                System.out.println (method ());
        }
}
```

Каким будет результат компиляции и запуска программы (1)?

1) на консоль выведется число 4
2) на консоль выведется число 0
3) ошибка компиляции в строке 1
4) ошибка компиляции в строке 2
5) ошибка компиляции в строке 3

Вопрос 3.4.

Выберите утверждения, корректно характеризующие модификаторы доступа (2):

1) статические private-члены класса доступны только статическим методам этого класса;
2) статические public-члены классы доступны всем методам этого класса;
3) protected-члены класса доступны подклассам другого пакета;
4) поле — член класса, объявленное без модификатора доступа, доступно в классах другого пакета.

Вопрос 3.5.

Дан код:

```
public class Quest5 {
        public Quest5 () {}
        public Quest5 (int i) {this (i, i);}
        public Quest5 (int i, int j) {this ();}
        public static void main (String [] args) {
                Quest5 q = new Quest5 (2,3); //1
        }
}
```

Сколько конструкторов вызовется при создании объекта в строке 1 (1)?

1) один;
2) два;
3) три;
4) ошибка компиляции.

Вопрос 3.6.

Дан код (1):

```
public class Quest6 {
        public void meth (Number obj) {System.out.print ("1");}
        public void meth (Character obj) {System.out.print ("2");}
        private static void meth (Integer obj) {System.out.print ("3");}
```

```
        public void meth (int i) {System.out.print ("4");}
        public void meth (double d) {System.out.print ("5");}
        public static void main (String [] args) {
                Quest6 q = new Quest6 ();
                Number n = 67;
                Integer i = 78;
                q.meth (n);
                q.meth (i);
        }
}
```

Что выведется на консоль после компиляции и запуска этой программы?

1) 14
2) 11
3) 33
4) 44
5) 13
6) ошибка компиляции
7) ошибка выполнения

Вопрос 3.7.

Дан код:

```
public class Quest7<T> {
        private T pole;
        public Quest7 (T pole) {this.pole = pole;} //1
        public void setPoleDefault () {pole.setTime (1000);} //2
        public static void main (String [] args) {
                Quest7<Date> obj = new Quest7<Date> (new Date ());//3
                obj.setPoleDefault ();
        }
}
```

Каким будет результат компиляции и запуска программы (1)?

1) компиляция и запуск пройдут без ошибок
2) ошибка компиляции в строке 1
3) ошибка компиляции в строке 2
4) ошибка компиляции в строке 3

Вопрос 3.8.

Укажите корректый способ создания экземпляра класса

```
public class Quest8<T1, T2> {}? (2)
```

1) Quest8 obj = **new** Quest8 ()
2) Quest8<Object> obj = **new** Quest8<Object> ()
3) Quest8<Object, Object> obj = **new** Quest8<Object, Object> ()
4) Quest8<…, Object> obj = **new** Quest8<…, Object> ()
5) Quest8<Object, Integer> obj = **new** Quest8<Integer, Object> ()
6) Quest8<Number, Integer> obj = **new** Quest8<Integer, Integer> ()

Вопрос 3.9.

Дан код:

```
enum Numbers {ONE, TWO, THREE, FOUR, FIVE}
public class Quest9 {
        public static void main (String [] args) {
                Numbers n1 = Numbers.ONE;
                Numbers n2 = Numbers.ONE;//1
                if (n1 == n2) {System.out.print ("true");}
                else {System.out.print ("false");}
                System.out.println (Numbers.FIVE.ordinal ());//2
        }
}
```

Что выведется на консоль в результате компиляции и запуска приложения (1)?

1) false4
2) true4
3) false5
4) true5
5) произойдет ошибка компиляции в строке 1
6) произойдет ошибка компиляции в строке 2

Вопрос 3.10.

Выберите неправильные утверждения (3):

1) перечисление является классом
2) при объявлении перечисления его необходимо явно наследовать от класса **java.lang.Enum**
3) в теле перечисления можно объявлять только методы
4) конструктор перечисления может быть объявлен со спецификатором **public**

Глава 4

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Если делегированию полномочий уделять внимание,
ответственность накопится внизу, подобно осадку.

Закон делегирования Раска

Наследование

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

Подкласс наследует поля и методы суперкласса, используя ключевое слово **extends**. Класс может также реализовать 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, аналог предоставляет реализация интерфейсов, которые не являются классами и содержат описание набора методов, задающих поведение объекта класса, реализующего эти интерфейсы. Наличие общих методов, которые должны быть реализованы в разных классах, обеспечивают им сходную функциональность.

Подкласс дополняет члены суперкласса своими полями и методами. Если имена методов совпадают, а параметры различаются, то такое явление называется перегрузкой методов (статическим полиморфизмом).

Если же совпадают имена и параметры методов, то это называется динамическим полиморфизмом. То есть 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 **doPayment()** находится в двух классах **CardAction** и **CreditCardAction**. В соответствии с принципом полиморфизма вызывается метод, наиболее близкий к текущему объекту.

/ # 1 # наследование класса и переопределение метода # CardAction.java # CreditCardAction.java # CardRunner.java */*

```
package by.bsu.inheritance;
public class CardAction {
        // поля, конструкторы, методы
        public void doPayment(double amountPayment) {
                // реализация
                System.out.println("complete from debt card");
        }
}
package by.bsu.inheritance;
public class CreditCardAction extends CardAction {
        // поля, конструкторы, методы
        public boolean checkCreditLimit() { // собственный метод
                return true; // stub
        }
        @Override // аннотация указывает на полиморфную природу метода
        public void doPayment(double amountPayment) { // переопределенный метод
                // реализация
                System.out.println("complete from credit card");
        }
}
package by.bsu.inheritance;
public class CardRunner {
    public static void main(String[ ] args) {
        CardAction dc1 = new CardAction();
        CardAction dc2 = new CreditCardAction();
        CreditCardAction cc = new CreditCardAction();
        // CreditCardAction cca = new CardAction(); // ошибка компиляции
        dc1.doPayment(15.5); // метод класса CardAction
        dc2.doPayment(21.2); // полиморфный метод класса CreditCardAction
        cc.doPayment(7.0); // полиморфный метод класса CreditCardAction
        cc.checkCreditLimit(); // неполиморфный метод класса CreditCardAction
        // dc2.checkCreditLimit(); // ошибка компиляции – неполиморфный метод
        ((CreditCardAction)dc1).checkCreditLimit(); // ошибка времени выполнения
        ((CreditCardAction)dc2).checkCreditLimit(); // ок
    }
}
```

Объект **dc1** создается при помощи вызова конструктора класса **CardAction**, и, соответственно, при вызове метода **doPayment()** вызывается версия метода из класса **CardAction**. При создании объекта **dc2** ссылка типа **CardAction** инициализируется объектом типа **CreditCardAction**. 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, т. е. существуют в одном объекте независимо друг от друга. Такое решение является плохим примером кода, который не используется в практическом программировании. Не следует использовать вызов полиморфных методов в конструкторе. Эти действия могут привести к использованию и получению недостоверной информации при работе метода. Для доступа к полям текущего объекта можно использовать указатель **this**, для доступа к полям суперкласса — указатель **super**. Другие возможности рассмотрены в следующем примере:

/ # 2 # вызов полиморфного метода из конструктора # Dumb.java # Dumber.java */*

```
package by.bsu.dumb;
class Dumb extends Object {
        { this.id = 6; }
        int id;
        Dumb() {
                System.out.println("конструктор класса Dumb ");
                // вызов потенциально полиморфного метода - плохо
                id = this.getId();
                System.out.println(" id=" + id);
        }
        int getId() { // 1
                System.out.println("getId() класса Dumb ");
                return id;
        }
}
class Dumber extends Dumb {
        int id = 9; // получится два поля с одинаковыми именами
        Dumber() {
                System.out.println("конструктор класса Dumber ");
                id = this.getId();
                System.out.println(" id=" + id);
        }
        @Override
        int getId() { // 2
                  System.out.println("getId() класса Dumber ");
                  return id;
        }
}
```

В результате создания экземпляра **Dumb objA = new Dumber()** последовательно будет выведено:

конструктор класса Dumb
getId() класса Dumber
id=0

конструктор класса Dumber
getId() класса Dumber
id=9

Метод **getId()** содержится как в классе **Dumb**, так и в классе **Dumber** и является переопределенным. Перед вызовом конструктора **Dumber()** вызывается конструктор класса **Dumb**. Но так как в обоих случаях создается объект класса **Dumber**, то вызывается метод **getId(),** объявленный в классе **Dumber**, который в свою очередь оперирует полем **id**, еще не проинициализированным для класса **Dumber**. В результате **id** получит значение по умолчанию, т. е. нуль.

Воспользовавшись преобразованием типов вида **((Dumber) objA).id**, легко можно получить доступ к полю **id** из подкласса.

Классы и методы final

Нельзя переопределить метод в порожденном классе, если в суперклассе он объявлен со спецификатором **final**:

```
class ConstMethod {
        // метод method() не может быть полиморфным
        final void method() {}
}
class Sub extends ConstMethod {
        // следующий метод невозможен
        @Override
        void method() {} // ошибка компиляции
}
```

Если разработчик объявляет метод как **final**, следовательно, он считает, что его версия метода окончательна и совершенствованию не подлежит.

Процесс инициализации экземпляра должен быть строго определен и не подвергаться изменениям. Исключить подмену реализации метода, вызываемого в конструкторе, следует объявлением метода как **final**, т. е. при этом метод не может быть переопределен в подклассе. Такое объявление гарантирует обращение именно к этой реализации. Корректное определение вызова метода класса из конструктора представлено ниже:

/ # 3 # вызов нестатического final-метода из конструктора # Coin.java */*

```
package by.bsu.fund.entity;
public class Coin {
        private double diameter;
        // поля
        public Coin(double diameter) {
                  super();
                  initDiameter(diameter); // обращение к final-методу
        }
```

```
        public final void initDiameter(double value) { // можно public final заменить на private
                  if (value > 0) {
                            diameter = value;
                  } else {
                            System.out.println("Отрицательный диаметр!");
                  }
        }
        // методы
}
```

Рекомендуется при разработке классов из конструкторов вызывать методы, на которые не распространяются принципы полиморфизма. Метод может быть еще объявлен как **private** с таким же результатом.

Нельзя создать порожденный класс для класса, объявленного со спецификатором **final**:

```
// класс String не может быть суперклассом
public final class String { /* код */ }
// следующий класс невозможен
class MegaString extends String { /* код */ } // ошибка компиляции
```

Если необходимо создать собственную реализацию **final**-класса, то создается класс-оболочка, где в качестве поля представлен **final**-класс. В свою очередь необходимые имена методов делегируются из **final**-класса, но им придается необходимая разработчику функциональность.

Такой подход гарантирует невозможность прямого использования класса-оболочки вместо оборачиваемого класса и наоборот.

// # 4 # класс-оболочка для класса String # WrapperString.java

```
package by.bsu.wrapper;
public class WrapperString {
          private String str;
          public  WrapperString() {
                    str = new String();
          }
          public  WrapperString(String s) {
                    str = new String(s);
          }
          public int indexOf(int arg) { // делегированный метод
                    // новая функциональность
                    return arg;
          }
          // другие делегированные методы
}
```

Класс **WrapperString** не является наследником класса **String,** и его объект не может быть использован для передачи по ссылке на класс **String**. Класс **WrapperString** может быть суперклассом, поэтому его поведение можно изменять.

Использование super и this

Ключевое слово **super** применяется для обращения к конструктору суперкласса и для доступа к полю или методу суперкласса. Например:

```
super(список_параметров); /* обращение к конструктору суперкласса
                                  с передачей параметров или без нее*/
super.id = 35; /* обращение к атрибуту суперкласса */
super.getId(); // вызов метода суперкласса
```

Первая форма **super** применяется только в конструкторах для обращения к конструктору суперкласса только в качестве первой строки кода и только один раз.

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 он не будет найден.

Во всех случаях с использованием **super** можно обратиться только к ближайшему суперклассу, т. е. «перескочить» через суперкласс, чтобы обратиться к его суперклассу, невозможно.

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

// # 5 # super и this в конструкторе # Point1D.java # Point2D.java # Point3D.java

```
package by.bsu.point;
public class Point1D {
          private int x;
          public Point1D(int x) {
                    this.x = x;
          }
}
package by.bsu.point;
public class Point2D extends Point1D {
          private int y;
          public Point2D(int x, int y) {
                    super(x);
                    this.y = y;
          }
}
package by.bsu.point;
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
        }
}
```

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, если в методе объявлены локальные переменные с тем же именем, что и переменные экземпляра класса. Локальная переменная имеет преимущество перед полем класса и закрывает к нему доступ. Чтобы получить доступ к полю класса, требуется воспользоваться явной ссылкой **this** перед именем поля, так как поле класса является частью объекта, а локальная переменная нет.

Инструкция **this()** должна быть единственной в вызывающем конструкторе и быть первой по счету выполняемой операцией, т. е. обращение к конструктору суперкласса становится невозможным.

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

/ # 6 # динамическое связывание методов # Point1D.java # Point2D.java # Point3D.java # PointReport.java # Runner.java */*

```
package by.bsu.point;
public class Point1D {
          private int x;
          public Point1D(int x) {
                    this.x = x;
          }
          public double length() {
                    return Math.abs(x);
          }
          @Override
          public String toString() {
                    return " x=" + x;
          }
}
package by.bsu.point;
public class Point2D extends Point1D {
          private int y;
          public Point2D(int x, int y) {
                    super(x);
                    this.y = y;
          }
          @Override
          public double length() {
                    return Math.hypot(super.length(), y);
                    /* просто length() нельзя, т.к. метод будет вызывать сам себя, что
                       приведет к бесконечной рекурсии и ошибке во время выполнения  */

          }
          @Override
          public String toString() {
                    return super.toString() + " y=" + y;
          }
}
package by.bsu.point;
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
          }
          @Override
          public double length() {
                    return Math.hypot(super.length(), z);
          }
```

```
        @Override
        public String toString() {
                return super.toString() + " z=" + z;
        }
}
package by.bsu.point;
public class PointReport {
        public void printReport(Point1D p) {
                System.out.printf("length=%.2f %s%n", p.length(), p);
                // вызовы out.print(p.toString()) и out.print(p) для объекта идентичны !
        }
}
package by.bsu.point;
public class Runner {
        public static void main(String[] args) {
                PointReport d = new PointReport();
                Point1D p1 = new Point1D(-7);
                d.printReport(p1);
                Point2D p2 = new Point2D(3, 4);
                d.printReport(p2);
                Point3D p3 = new Point3D(3, 4, 5);
                d.printReport(p3);
        }
}
```

Результат:

length=7.00 x=-7
length=5.00 x=3 y=4
length=7.07 x=3 y=4 z=5

Использование аннотации **@Override** позволяет выделить в коде переопределенный метод и сгенерирует ошибку компиляции в случае, если программист допустит грамматическую ошибку (опечатку) в описании сигнатуры полиморфного метода.

Следует помнить, что при вызове **toString()** обращение **super** всегда происходит к ближайшему суперклассу. Переадресовать вызов, минуя суперкласс, невозможно! 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 можно перегружать и «переопределять» в подклассах, но их доступность всегда зависит от типа ссылки и атрибута доступа, и никогда — от типа самого объекта.

Методы подставки

После выхода 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.

/ # 7 # методы-подставки # Point1DCreator.java # Point2DCreator.java # Point3DCreator.java # BuildRunner.java*/*

```
package by.bsu.creator;
public class Point1DCreator {
          public Point1D createPoint() {
                    System.out.println("Point1D");
                    return new Point1D(1);
          }
}
package by.bsu.creator;
public class Point2DCreator extends Point1DCreator {
          @Override
          public Point2D createPoint() { // метод - подставка
                    System.out.println("Point2D");
                    return new Point2D(2, 5);
          }
}
package by.bsu.creator;
public class Point3DCreator extends Point2DCreator {
          @Override
          public Point3D createPoint() { // метод - подставка
                    System.out.println("Point3D");
                    return new Point3D(3, 7, 8);
          }
}
package by.bsu.creator;
public class BuildRunner {
          public static void main(String[ ] args) {
                    Point2DCreator br = new Point3DCreator();
                    // Point3D p = br.createPoint(); // ошибка компиляции
                    Point2D p = br.createPoint(); // "раннее связывание"
                    System.out.println(br.createPoint().x);
                    System.out.println(br.createPoint().y);
                    // System.out.println(br.createPoint().z);
          }
}
```

В результате будет выведено:

Point3D
Point3D

3
Point3D
7

В данной ситуации при компиляции в подклассе **Point3DCreator** создаются три метода **createPoint().** Один имеет возвращаемое значение **Point3D**, другие (явно невидимые) — **Point1D** и **Point2D**. При обращении к методу **createPoint()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении срабатывает полиморфизм и вызывается метод с возвращаемым значением **Point3D**. Обращение к полю также производится по типу объекта, возвращаемого методом **createPoint(),** т. е. к полю класса **Point3D**. Для результатов обращения к полям следует в классах **Point1D** и его наследниках убрать спецификатор **private**.

На практике методы подставки могут использоваться для расширения возможностей класса по прямому извлечению (без преобразования) объектов подклассов, инициализированных в ссылке на суперкласс.

«Переопределение» статических методов

Для статических методов принципы «позднего связывания» не используются. Динамический полиморфизм к статическим методам класса неприменим, так как обращение к статическому атрибуту или методу осуществляется по типу ссылки, а не по типу объекта, через который производится обращение. Версия вызываемого статического метода всегда определяется на этапе компиляции. При использовании ссылки для доступа к статическому члену компилятор при выборе метода учитывает тип ссылки, а не тип объекта, ей присвоенного.

/ # 8 # поведение статического метода при «переопределении» # Runner.java */*

```
package by.bsu.sample;
class Base {
          public static void go() {
                    System.out.println("метод из Base");
          }
}
class Sub extends Base {
          public static void go() {
                    System.out.println("метод из Sub");
          }
}
public class Runner {
          public static void main(String[ ] args) {
                    Base ob = new Sub(); // !!!
```

```
                // нестатический вызов статического метода
                ob.go(); // предупреждение компилятора о некорректном вызове
        }
}
```

В результате выполнения данного кода будет выведено:

метод из Base

При таком способе инициализации объекта **ob** метод **go()** будет вызван из класса **Base**. Если же спецификатор **static** убрать из объявления методов, то вызов необходимо осуществлять в соответствии с принципами полиморфизма.

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

```
Base.go();
Sub.go();
```

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

Абстракция и абстрактные классы

Множество моделей предметов реального мира обладают 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 с помощью оператора **new**, но можно создать объекты подклассов, которые реализуют все 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. Объект класса **Number** нельзя создать явно при помощи его собственного конструктора.

/ # 9 # абстрактный класс и метод # AbstractCardAction.java */*

```
package by.bsu.inheritance;
public abstract class AbstractCardAction {
          private int id;
          public AbstractCardAction() { // конструктор
          }
          // more methods
          public boolean checkLimit() { // собственный метод
                    return true; // stub
          }
          public abstract void doPayment(double amountPayment);
}
```

/ # 10 # подкласс абстрактного класса # CreditCardAction.java # Runner.java */*

```
package by.bsu.inheritance;
public class CreditCardAction extends AbstractCardAction {
          // поля, конструкторы, методы
          @Override // аннотация указывает на полиморфную природу метода
          // метод должен быть реализован в подклассе
          public void doPayment(double amountPayment) { // переопределенный метод
                    // реализация
                    System.out.println("complete from credit card!");
          }
}
package by.bsu.inheritance;
public class Runner {
          public static void main(String[ ] args) {
                    AbstractCardAction action; // можно объявить ссылку
                    // action = new AbstractCardAction(); нельзя создать объект!
                    action = new CreditCardAction();
                    action.doPayment (100);
          }
}
```

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

Расширение функциональности системы

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

В следующем примере приведение к базовому типу происходит в выражении:

```
AbstractQuest quest1 = new DragnDropQuest();
AbstractQuest quest2 = new SingleChoiceQuest();
```

Базовый класс **AbstractQuest** предоставляет общий интерфейс для своих подклассов. Порожденные классы **DragnDropQuest** и **SingleChoiceQuest** перекрывают эти определения для обеспечения уникального поведения.

/ # 11 # полиморфизм # AbstractQuest.java # DragnDropQuest.java # SingleChoiceQuest.java # Answer.java # QuestFactory.java # Runner.java*/*

```
package by.bsu.scalability;
public abstract class AbstractQuest {
    private long id;
    private String questContent;
            // конструкторы и методы
    public abstract boolean check(Answer ans);
}
package by.bsu.scalability;
public class DragnDropQuest extends AbstractQuest {
            // поля, конструкторы и методы
    @Override
    public boolean check(Answer ans) {
        System.out.println("Drag'n'Drop quest");
            // проверка корректности ответа (true или false)
        return true; // stub
    }
}
package by.bsu.scalability;
public class SingleChoiceQuest extends AbstractQuest {
           // поля, конструкторы и методы
    @Override
    public boolean check(Answer ans) {
        System.out.println("Single choice quest");
           // проверка корректности ответа true или false
        return true; // stub
    }
}
```

```
package by.bsu.scalability;
public class Answer {
                    // поля и методы
}
package by.bsu.scalability;
public class QuestFactory { // шаблон Factory Method (упрощенный)
          public static AbstractQuest getQuestFromFactory(int mode) {
                    switch (mode) {
                              case 0:
                                        return new DragnDropQuest();
                              case 1:
                                        return new SingleChoiceQuest();
                              default :
                                        throw new IllegalArgumentException("illegal mode");
                                        // assert false; // плохо
                                        // return null;  // еще хуже
                    }
          }
}
package by.bsu.scalability;
import java.util.Random;
public class TestAction {
          public AbstractQuest[] generateTest(final int NUMBER_QUESTS, int maxMode) {
                    AbstractQuest[ ] test = new AbstractQuest[NUMBER_QUESTS];
                    for (int i = 0; i < test.length; i++) {
                              int mode = new Random().nextInt(maxMode); // stub
                              /* заполнение массива объектами-вопросами */
                              test[i] = QuestFactory.getQuestFromFactory(mode);
                    }
                    return test;
          }
          public int checkTest(AbstractQuest[] test) {
                    int counter = 0;
                    for (AbstractQuest s : test) {
                              // вызов полиморфного метода
                              counter = s.check(new Answer()) ? ++counter : counter;
                    }
                    return counter;
          }
}
package by.bsu.scalability;
public class TestRunner {
public static void main(String[ ] args) {
          TestAction bt = new TestAction();
          AbstractQuest[ ] test = bt.generateTest(60, 2); // 60 вопросов 2-х видов
          // здесь должен быть код процесса прохождения теста …
          bt.checkTest(test); // проверка теста
    }
}
```

В процессе выполнения приложения будет случайным образом сформирован массив-тест из вопросов разного типа, и информация об ответах на них будет выведена на консоль.

Класс **QuestFactory** содержит метод **getQuestFromFactory(int numMode),** который возвращает ссылку на случайно выбранный объект подкласса класса **AbstractQuest** каждый раз, когда он вызывается. Приведение к базовому типу производится оператором **return**, который возвращает ссылку на **DragnDropQuest** или **SingleChoiceQuest**. Метод **main()** содержит массив из ссылок **AbstractQuest**, заполненный с помощью вызова **getQuestFromFactory().** 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 **check()** вызывается для каждого случайным образом выбранного объекта.

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

Невозможно приравнивать ссылки на классы, находящиеся в разных ветвях наследования, так как не существует никакого способа привести один такой тип к другому.

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;

public boolean equals(Object ob) — предназначен для использования и переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа;

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

protected void finalize() — автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта;

public int hashCode() — вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта);

public String toString() — возвращает представление объекта в виде строки.

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

Если при создании класса предполагается проверка логической эквивалентности объектов, которая не выполнена в суперклассе, следует переопределить два метода: **boolean equals(Object ob)** и **int hashCode().** Кроме того, переопределение 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. Реализация метода в классе **Object** возвращает истину только в том случае, если обе ссылки указывают на один и тот же объект, а конкретно:

```
public boolean equals(Object obj) {
     return (this == obj);
}
```

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 **int 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:

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

Один из способов создания правильного метода **hashCode(),** гарантирующий выполнение соглашений, приведен ниже, в примере # 12.

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 (т. е. 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.

/ # 12 # переопределение методов equals(), hashCode(), toString() # Student.java */*

```
package by.bsu.entity;
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 (getClass() != obj.getClass())
                              return false;
                    Student other = (Student) obj;
                    if (age != other.age)
                              return false;
                    if (id != other.id)
                              return false;
                    if (name == null) {
                              if (other.name != null)
                                        return false;
```

```
                } else if (!name.equals(other.name))
                        return false;
                return true;
        }
        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**, а также сравнить содержимое полей **id**, **name** и **age** соответственно у вызывающего метод объекта и объекта, передаваемого в качестве параметра. Для подкласса всегда придется создавать собственную реализацию метода.

Клонирование объектов

Объекты в методы передаются по ссылке, в результате чего в метод передается ссылка на объект, находящийся вне метода. Если в методе изменить значение поля объекта, 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.

```
/* # 13 # класс, поддерживающий клонирование # Student.java */
package by.bsu.entity;
public class Student implements Cloneable { /* включение интерфейса */
          private int id = 71;
          private String name;
          private int age;
          /* конструкторы, методы */
          public int getId() {
                    return id;
          }
          public void setId(int id) {
                    this.id = id;
          }
          @Override
          public Object clone() throws CloneNotSupportedException { // переопределение
                              return super.clone(); // вызов базового метода
          }
}
```

```
/* # 14 # безопасная передача по ссылке # CloneRunner.java */
public class CloneRunner {
          private static void mutation(Student p) {
                    try {
                              p = (Student)p.clone(); // клонирование
                    } catch (CloneNotSupportedException e) {
                              e.printStackTrace();
                    }
                    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());
                    mutation(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

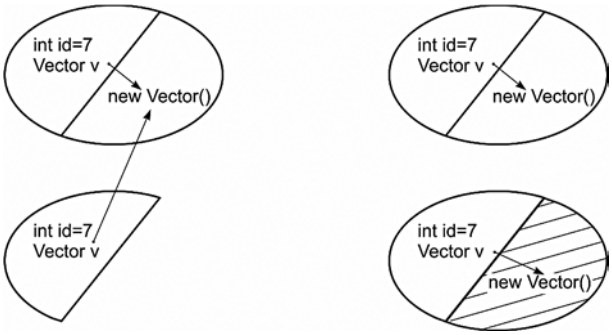

Рис. 4.1. *«Неглубокое» и «глубокое» клонирование*

Решение эффективно только в случае, когда 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, если он сам поддерживает клонирование.

/ # 15 # глубокое клонирование # Student.java */*

```
package by.bsu.entity;
import java.util.Vector;
public class Student implements Cloneable {
          private int id = 7;
          private String name;
          private int age;
          private Vector<Byte> v = new Vector<Byte>(); // список оценок – изменяемое поле
          /* конструкторы, методы */
          public Student clone() { // метод-подставка
                    Student copy = null;
                    try {
                              copy = (Student)super.clone();
                              copy.v = (Vector<Byte>)v.clone();
                    } catch (CloneNotSupportedException e) {
                              e.printStackTrace("не реализован интерфейс Cloneable !");
                    }
                    return copy;
          }
}
```

Клонирование возможно лишь, 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.

```
private static void mutation(Student p) {
                    p = p.clone(); // клонирование
                    p.setId(1000);
                    System.out.println("->id = " + p.getId());
}
```

Если заменить объявление **Vector<Byte>** на **Vector<Mark>**, где **Mark** — изменяемый тип, то клонирование должно затрагивать и внутреннее состояние списка.

«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, т. е. 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» выполнить ее 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 autocloseable. Указанные способы являются предпочтительными, абсолютно надежными и будут рассмотрены в девятой главе.

Запуск стандартного 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 **protected void finalize(),** вызываемого непосредственно перед освобождением памяти, следует определить действия, которые должны быть выполнены до уничтожения объекта.

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, т. е. 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.

/ # 16 # класс Manager с поддержкой finalization # Manager.java # FinalizeDemo.java */*

```
package by.bsu.gc;
public class Manager {
          private int id;
          public Manager(int value) {
                    id = value;
          }
          protected void finalize() throws Throwable {
             try {
             // код освобождения ресурсов
                    System.out.println("объект будет удален, id=" + id);
             } finally {
                     super.finalize();
             }
     }
}
package by.bsu.gc;
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;
              // просьба выполнить "сборку мусора"
              System.gc();
       }
}
```

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: если при выполнении данного метода возникнет исключительная ситуация, она будет проигнорирована и приложение продолжит выполняться, что также представляет опасность для его корректной работы.

Пакеты

Любой класс Java относится к определенному пакету, который может быть неименованным (unnamed или default package), если оператор **package** отсутствует. Оператор **package**, помещаемый в начале исходного программного файла, определяет именованный пакет, т. е. область в пространстве имен классов, в которой определяются имена классов, содержащихся в этом файле. Действие оператора **package** указывает на месторасположение файла относительно корневого каталога проекта. Например:

```
package by.bsu.eun.entity;
```

При этом программный файл будет помещен в подкаталог с названием **by.bsu.eun.entity**. Имя пакета при обращении к классу из другого пакета присоединяется к имени класса:

```
by.bsu.eun.entity.Student
```

В проектах пакеты часто именуются следующим образом:

- обратный Интернет-адрес производителя или заказчика программного обеспечения, а именно для www.bsu.by получится **by.bsu**;
- далее следует имя проекта (обычно сокращенное), например, **eun**;
- затем располагаются пакеты, определяющие собственно приложение.

Общая форма файла, содержащего исходный код Java, может быть следующая:

- одиночный оператор **package** (необязателен, но крайне желателен);
- любое количество операторов **import** (необязательны);
- одиночный открытый (**public**) класс (необязателен);
- любое количество классов пакета (необязательны и нежелательны).

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,

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. 4.2. *Организация пакетов приложения*

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.

Каждый класс добавляется в указанный пакет при компиляции. Например:

/ # 17 # применение пакета # CommonObject.java */*

```
package by.bsu.eun.objects;
public class CommonObject implements Cloneable {
        // more code
}
```

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 при создании объектов классов, 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 еще одним способом (не очень рекомендуемым):

/ # 18 # применение полного имени пакета при наследовании # UserStatistic.java */*

```
package by.bsu.eun.usermng;
public class UserStatistic extends by.bsu.eun.objects.CommonObject {
        // more code
}
```

Такая запись используется, если в классе нужен доступ к классам, имеющим одинаковые имена.

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.

/ # 19 # применение полного пути к классу при импорте # CreatorStatistic.java */*

```
package by.bsu.eun.action;
import by.bsu.eun.objects.CommonObject;
import by.bsu.eun.usermng.UserStatistic;
public class CreatorStatistic extends CommonObject {
        public UserStatistic us;
        // more code
}
```

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

При вызове статических методов и обращении к статическим константам приходится использовать в качестве префикса имя класса, что утяжеляет код и снижает скорость его восприятия.

// # 20 # обращение к статическому методу и константе # ImportDemo.java

```
package by.bsu.stat;
public class ImportDemo {
	public static void main(String[ ] args) {
		System.out.println(2 * Math.PI * 3);
		System.out.println(Math.floor(Math.cos(Math.PI / 3)));
	}
}
```

Статические константы и статические методы класса можно использовать без указания принадлежности к классу, если применить статический импорт,

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

как это показано в следующем примере.

// # 21 # статический импорт методов и констант # ImportDemoLux.java

```
package by.bsu.stat;
import static java.lang.Math.*;
public class ImportDemoLux {
	public static void main(String[ ] args) {
		System.out.println(2 * PI * 3);
		System.out.println(floor(cos(PI / 3)));
	}
}
```

Если необходимо получить доступ только к одной статической константе или методу, то импорт производится в следующем виде:

```
import static java.lang.Math.E; // для одной константы
import static java.lang.Math.cos; // для одного метода
```

Рекомендации при проектировании иерархии

При построении иерархии необходимо помнить, что отношение между классами можно выразить как «is-a», или «является». *Студент* «является» *Человеком*. Поля класса находятся с классом в отношении «has-a», или «содержит». *Студент* «содержит» *Номер зачетной книжки*.

Наследование и переопределение методов используются для реализации отличий поведения. Если наследование можно заменить агрегацией, то следует

так и поступить. Нет смысла создавать подкласс *Студент-заочник*, если можно в подкласс *Студент* добавить поле *Форма обучения*.

При наследовании в новые классы добавляются новые возможности в виде полей и методов или переопределения методов. Если новых возможностей не обнаруживается, то использование наследования, как правило, не имеет для этого оснований.

Базовая функциональность должна определяться в вершине иерархии проектируемых классов. Если в подклассе добавляются новые методы, характеризующие поведение иерархии в целом, следует заняться перепроектированием.

Но нельзя учесть все возможные изменения иерархии в процессе разработки. Избыточный функционал придется поддерживать. Лучше на поздних стадиях добавить методы в суперкласс, чем пытаться понять, зачем нужен метод, которой никто еще не использовал.

Не использовать значения переменных для характерного изменения поведения. Для этих целей следует создать подкласс и переопределить метод.

Для различных семантических сущностей не создавать общую иерархию, даже если действия для них идентичны по форме. Блокировка *Теста* и блокировка *Студента* — действия суть похожие, но отличные по своему функционалу и последствиям.

Задания к главе 4

Вариант А

Создать приложение, удовлетворяющее требованиям, приведенным в задании. Наследование применять только в тех заданиях, в которых это логически обосновано. Аргументировать принадлежность классу каждого создаваемого метода и корректно переопределить для каждого класса методы **equals(), hashCode(), toString().**

1. Создать объект класса **Текст**, используя классы **Предложение, Слово**. Методы: дополнить текст, вывести на консоль текст, заголовок текста.
2. Создать объект класса **Автомобиль**, используя классы **Колесо, Двигатель**. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса **Самолет**, используя классы **Крыло, Шасси, Двигатель**. Методы: летать, задавать маршрут, вывести на консоль маршрут.
4. Создать объект класса **Государство**, используя классы **Область, Район, Город**. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса **Планета**, используя классы **Материк, Океан, Остров**. 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**, **Оперативная память, Процессор**. 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 **Щенок**, используя классы **Животное, Собака.** Методы: вывести на консоль имя, подать голос, прыгать, бегать, кусать.
11. Создать объект класса **Наседка**, используя классы **Птица, Кукушка.** Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса **Текстовый файл**, используя классы **Файл, Директория.** Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса **Одномерный массив**, используя классы **Массив, Элемент.** 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 **Цветок**, используя классы **Лепесток, Бутон.** Методы: расцвести, завять, вывести на консоль цвет бутона.
17. Создать объект класса **Дерево**, используя классы **Лист, Ветка.** Методы: зацвести, опасть листьям, покрыться инеем, пожелтеть листьям.
18. Создать объект класса **Пианино**, используя классы **Клавиша, Педаль.** Методы: настроить, играть на пианино, нажимать клавишу.
19. Создать объект класса **Фотоальбом**, используя классы **Фотография, Страница.** 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 классы **Крылья, Клюв.** Методы: летать, садиться, питаться, атаковать.

23. Создать объект класса **Хищник**, используя классы **Когти, Зубы.** Методы: рычать, бежать, спать, добывать пищу.
24. Создать объект класса **Гитара**, используя класс **Струна, Скворечник.** Методы: играть, настраивать, заменять струну.

Вариант В

Создать консольное приложение, удовлетворяющее следующим требованиям:

- Использовать возможности ООП: классы, наследование, полиморфизм, инкапсуляция.
- Каждый класс должен иметь отражающее смысл название и информативный состав.
- Наследование должно применяться только тогда, когда это имеет смысл.
- При кодировании должны быть использованы соглашения об оформлении кода java code convention.
- Классы должны быть грамотно разложены по пакетам.
- Консольное меню должно быть минимальным.
- Для хранения параметров инициализации можно использовать файлы.

1. **Цветочница.** Определить иерархию цветов. Создать несколько объектов-цветов. Собрать букет (используя аксессуары) с определением его стоимости. Провести сортировку цветов в букете на основе уровня свежести. Найти цветок в букете, соответствующий заданному диапазону длин стеблей.
2. **Новогодний подарок.** Определить иерархию конфет и прочих сладостей. Создать несколько объектов-конфет. Собрать детский подарок с определением его веса. Провести сортировку конфет в подарке на основе одного из параметров. Найти конфету в подарке, соответствующую заданному диапазону содержания сахара.
3. **Домашние электроприборы.** Определить иерархию электроприборов. Включить некоторые в розетку. Подсчитать потребляемую мощность. Провести сортировку приборов в квартире на основе мощности. Найти прибор в квартире, соответствующий заданному диапазону параметров.
4. **Шеф-повар.** Определить иерархию овощей. Сделать салат. Подсчитать калорийность. Провести сортировку овощей для салата на основе одного из параметров. Найти овощи в салате, соответствующие заданному диапазону калорийности.
5. **Звукозапись.** Определить иерархию музыкальных композиций. Записать на диск сборку. Подсчитать продолжительность. Провести перестановку композиций диска на основе принадлежности к стилю. Найти композицию, соответствующую заданному диапазону длины треков.
6. **Камни.** Определить иерархию драгоценных и полудрагоценных камней. Отобрать камни для ожерелья. Подсчитать общий вес (в каратах) и стоимость.

Провести сортировку камней ожерелья на основе ценности. Найти камни в ожерелье, соответствующие заданному диапазону параметров прозрачности.

7. **Мотоциклист.** Определить иерархию амуниции. Экипировать мотоциклиста. Подсчитать стоимость. Провести сортировку амуниции на основе веса. Найти элементы амуниции, соответствующие заданному диапазону параметров цены.
8. **Транспорт.** Определить иерархию подвижного состава железнодорожного транспорта. Создать пассажирский поезд. Подсчитать общую численность пассажиров и багажа. Провести сортировку вагонов поезда на основе уровня комфортности. Найти в поезде вагоны, соответствующие заданному диапазону параметров числа пассажиров.
9. **Авиакомпания.** Определить иерархию самолетов. Создать авиакомпанию. Посчитать общую вместимость и грузоподъемность. Провести сортировку самолетов компании по дальности полета. Найти самолет в компании, соответствующий заданному диапазону параметров потребления горючего.
10. **Таксопарк.** Определить иерархию легковых автомобилей. Создать таксопарк. Подсчитать стоимость автопарка. Провести сортировку автомобилей парка по расходу топлива. Найти автомобиль в компании, соответствующий заданному диапазону параметров скорости.
11. **Страхование.** Определить иерархию страховых обязательств. Собрать из обязательств деривати. Подсчитать стоимость. Провести сортировку обязательств в деривативе на основе уменьшения степени риска. Найти обязательство в деривативе, соответствующее заданному диапазону параметров.
12. **Мобильная связь.** Определить иерархию тарифов мобильной компании. Создать список тарифов компании. Подсчитать общую численность клиентов. Провести сортировку тарифов на основе размера абонентской платы. Найти тариф в компании, соответствующий заданному диапазону параметров.
13. **Фургон кофе.** Загрузить фургон определенного объема грузом на определенную сумму из различных сортов кофе, находящихся, к тому же, в разных физических состояниях (зерно, молотый, растворимый в банках и пакетиках). Учитывать объем кофе вместе с упаковкой. Провести сортировку товаров на основе соотношения цены и веса. Найти в фургоне товар, соответствующий заданному диапазону параметров качества.
14. **Игровая комната.** Подготовить игровую комнату для детей разных возрастных групп. Игрушек должно быть фиксированное количество в пределах выделенной суммы денег. Должны встречаться игрушки родственных групп: маленькие, средние и большие машины, куклы, мячи, кубики. Провести сортировку игрушек в комнате по одному из параметров. Найти игрушки в комнате, соответствующие заданному диапазону параметров.
15. **Налоги.** Определить множество и сумму налоговых выплат физического лица за год с учетом доходов с основного и дополнительного мест работы,

авторских вознаграждений, продажи имущества, получения в подарок денежных сумм и имущества, переводов из-за границы, льгот на детей и материальной помощи. Провести сортировку налогов по сумме.

16. **Счета.** Клиент может иметь несколько счетов в банке. Учитывать возможность блокировки/разблокировки счета. Реализовать поиск и сортировку счетов. Вычисление общей суммы по счетам. Вычисление суммы по всем счетам, имеющим положительный и отрицательный балансы отдельно.
17. **Туристические путевки.** Сформировать набор предложений клиенту по выбору туристической путевки различного типа (отдых, экскурсии, лечение, шопинг, круиз и т. д.) для оптимального выбора. Учитывать возможность выбора транспорта, питания и числа дней. Реализовать выбор и сортировку путевок.
18. **Кредиты.** Сформировать набор предложений клиенту по целевым кредитам различных банков для оптимального выбора. Учитывать возможность досрочного погашения кредита и/или увеличения кредитной линии. Реализовать выбор и поиск кредита.

Тестовые задания к главе 4

Вопрос 4.1.

Дан класс:

```
package ch04.q01;
class Quest41 {}
```

Укажите правильные варианты наследования от этого класса (2):

1) **package** ch04.q01; **class** Quest4 **extends** Quest41 {}
2) **package** ch04.q01._2; **public class** Quest42 **extends** Quest41 {}
3) **package** ch04.q01; **public class** Quest43 **implements** Quest41 {}
4) **package** ch04.q01._2; **import** ch04.q01.Quest41;
5) **public class** Quest44 **extends** Quest41 {}
6) **package** ch04.q01; **public class** Quest45 **extends** Quest41 {}

Вопрос 4.2.

Выберите правильные утверждения (3):

1) Класс может быть использован в качестве суперкласса для себя самого.
2) В конструкторе класса можно совместно использовать вызовы this и super.
3) Статические методы можно определять в подклассах с той же сигнатурой, что и в базовом классе.
4) Статические методы можно перегружать в подклассах.
5) Динамическое связывание определяет версию вызываемого метода на этапе выполнения.

Вопрос 4.3.

Дан код:

```
package ch04.q03;
public class Quest43 {
        private final void method () {} //1
}
class Quest431 extends Quest43 {
        public void method () {} //2
}
```

Что произойдет в результате компиляции этого кода (1)?

1) ошибка компиляции в строке 1
2) ошибка компиляции в строке 2
3) компиляция без ошибок

Вопрос 4.4.

Дан код двух классов:

```
// класс 1
package ch04.q04;
public class Quest41 {}
// класс 2
package ch04.q04._2;
import ch04.q04.Quest41;
public class Quest43 extends Quest41 {
        public Quest43 () {
                super ();
}}
```

С каким атрибутом доступа объявлен конструктор по умолчанию в базовом классе Quest41 (1)?

1) public
2) private
3) protected
4) friendly

Вопрос 4.5.

Дан код:

```
package ch04.q05;
public class Quest51 {
        public String toString () {
```

```
            return getClass ().getSimpleName ();
        }
        public static void main (String [] args) {
            Quest53 q = new Quest53 ();
            System.out.println (q.toString ());
        }}
class Quest52 extends Quest51 {}
class Quest53 extends Quest52 {}
```

Что выведется на консоль в результате компиляции и запуска программы (1)?

1) Quest52
2) Quest53
3) Quest51
4) ошибка компиляции

Вопрос 4.6.

Дан код:

```
package ch04.q06;
class Item {
        public int item;
        Item (int item) {
            this.item = item;
        }
}
public class Quest61 {
        public static void main (String [] args) {
            Item ar1 [] = {new Item (1), new Item (2), new Item (3)};
            Item ar2 [] = ar1.clone ();
            ar2 [0].item = 4;
            System.out.println (ar1 [0].item +" " + ar1 [1].item + " " + ar1 [2].item);
        }
}
```

Что выведется на консоль после компиляции и запуска этой программы (1)?

1) 1 2 3
2) 1 4 3
3) 4 2 3
4) ошибка компиляции
5) ошибка выполнения

Глава 5

ВНУТРЕННИЕ КЛАССЫ

Внутри каждой большой задачи сидит маленькая, пытающаяся пробиться наружу.

Закон больших задач Хоара

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. Решение о включении одного класса внутрь другого может быть принято при слишком тесном и частом взаимодействии двух классов.

Одной из серьезных причин использования внутренних классов является возможность быть подклассом любого класса независимо от того, подклассом какого класса является внешний класс. Фактически при этом реализуется ограниченное множественное наследование со своими преимуществами и проблемами.

При необходимости доступа к защищенным полям и методам некоторого класса может появиться множество подклассов в разных пакетах системы. В качестве альтернативы можно сделать этот подкласс внутренним и методами класса-владельца предоставить доступ к интересующим защищенным полям и методам.

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, т. е. запустить *Двигатель* можно только посредством использования объекта *Корабль* из его 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 соответствующий 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). Доступ к элементам внутреннего класса возможен из внешнего 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:

/ # 1 # обьявление внутреннего класса и использование его в качестве поля как поля # Ship.java */*

```
package by.bsu.inner;
public class Ship {
       // поля и конструкторы
       private Engine eng;
       // abstract, final, private, protected - допустимы
       public class Engine { // определение внутреннего (inner) класса
              // поля и методы
              public void launch() {
                  System.out.println("Запуск двигателя");
              }
       } // конец обьявления внутреннего класса
       public final void init() { //  метод внешнего класса
          eng = new Engine();
          eng.launch();
       }
}
```

Внутреннему классу совершенно не обязательно быть полем класса владельца. Внутренний класс может быть использован любым членом своего внешнего класса, а может и не использоваться вовсе.

/ # 2 # обьявление внутреннего класса # Ship.java */*

```
package by.bsu.inner;
public class Ship {
       // поля и конструкторы внешнего класса
       public class Engine { // определение внутреннего класса
              // поля и методы
              public void launch() {
                   System.out.println("Запуск двигателя");
              }
       } // конец обьявления внутреннего класса
       // методы внешнего класса
}
```

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.

При компиляции создается объектный модуль, соответствующий внутреннему классу, который 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. Доступ будет разрешен по имени в том числе и к полям, объявленным как **private**. 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:

/ # 3 # наследование от внешнего и внутреннего классов # RaceShip.java */*

```
public class RaceShip 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 был объявлен.

/ # 4 # наследование от внутреннего классf # Motor.java */*

```
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 на примере определения логически самостоятельной сущности **PhoneNumber**, дополняющей описание внешней, глобальной для нее сущности **Abonent**, приведен ниже. Внутренний класс **PhoneNumber** может оказаться достаточно сложным и обширным, поэтому простое включение его полей и методов в класс **Abonent** сделало бы последний весьма громоздким и трудным для восприятия.

/ # 4 # сокрытие реализации во внутреннем классе # Abonent.java */*

```
package by.bsu.inner;
public class Abonent {
          private long id;
          private String name;
          private String tariffPlan;
          private PhoneNumber phoneNumber; // ссылка на внутренний класс
          public Abonent (long id, String name) {
                    this.id = id;
                    this.name = name;
          }
          // объявление внутреннего класса
          private class PhoneNumber {
                    private int countryCode;
                    private int netCode;
                    private int number;
                    public void setCountryCode(int countryCode) {
                              // проверка на допустимые значения кода страны
                              this.countryCode = countryCode;
```

```
                }
                public void setNetCode(int netCode) {
                        // проверка на допустимые значения кода сети
                        this.netCode = netCode;
                }
                public int generateNumber() {
                        int temp = new java.util.Random().nextInt(10_000_000);
                        // проверка значения temp на совпадение в БД
                        number = temp;
                        return number;
                }
        } // окончание внутреннего класса
        public long getId() {
                return id;
        }
        public String getName() {
                return name;
        }
        public String getTariffPlan() {
                return tariffPlan;
        }
        public void setTariffPlan(String tariffPlan) {
                this.tariffPlan = tariffPlan;
        }
        public String getPhoneNumber() {
                if (phoneNumber != null) {
                        return ("+" + phoneNumber.countryCode + "-"
                                + phoneNumber.netCode + "-" + phoneNumber.number);
                } else {
                        return ("phone number is empty!");
                }
        }
        // соответствует шаблону Façade
        public void obtainPhoneNumber(int countryCode, int netCode) {
                phoneNumber = new PhoneNumber();
                phoneNumber.setCountryCode(countryCode);
                phoneNumber.setNetCode(netCode);
                phoneNumber.generateNumber();
        }
        @Override
        public String toString() {
                StringBuilder s = new StringBuilder(100);
                s.append("Abonent '" + name + "':\n");
                s.append("    ID - " + id + "\n");
                s.append("    Tariff Plan -  " + tariffPlan + "\n");
                s.append("    Phone Number - " + getPhoneNumber() + "\n");
                return s.toString();
        }
}
```

/ # 5 # инициализация и использование экземпляра Abonent # MobilMain.java */*

```
package by.bsu.inner.run;
import by.bsu.inner.Abonent;
public class MobilMain {
          public static void main(String[] args) {
                    Abonent abonent = new Abonent(819002, "Timofey Balashov");
                    abonent.setTariffPlan("free");
                    abonent.obtainPhoneNumber(375, 25);
                    System.out.println(abonent);
          }
}
```

В результате будет выведено:

Abonent 'Timofey Balashov**':**
ID - 819002
Tariff Plan - Free
Phone Number - +375–25–7492407

Внутренний класс определяет сущность предметной области «номер телефона» (класс **PhoneNumber**), которая обычно непосредственно связана в информационной системе с объектом класса **Abonent**. Класс **PhoneNumber** в данном случае определяет только способы доступа и изменения своих атрибутов и совершенно невидим вне класса **Abonent**, включающего метод по созданию и инициализации объекта внутреннего класса с составным номером телефона, способ построения которого скрыт от всех других классов.

Внутренний класс может быть объявлен также внутри метода или логического блока внешнего (owner) 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.

/ # 6 # внутренний класс, обьявленный внутри метода # AbstractTeacher.java # TeacherCreator.java # Teacher.java # TeacherLogic.java # Runner.java */*

```
package by.bsu.inner.study;
public abstract class AbstractTeacher {
         private int id;
         public AbstractTeacher(int id) {
                   this.id = id;
         }
         /* методы */
         public abstract boolean excludeStudent(String name);
}
```

```
package by.bsu.inner.study;
public class Teacher extends AbstractTeacher {
        public Teacher(int id) {
                super(id);
        }
        /* методы */
        @Override
        public boolean excludeStudent(String name) {
                 return false;
        }
}
package by.bsu.inner.study;
public class TeacherCreator {
        public static AbstractTeacher createTeacher(int id) {
           // объявление класса внутри метода
           class Rector extends AbstractTeacher {
                  Rector (int id) {
                    super(id);
                  }
                  @Override
                  public boolean excludeStudent(String name) {
                              if (name != null) { // изменение статуса студента в базе данных
                                    return true;
                              } else {
                                    return false;
                              }
                            }
           } // конец внутреннего класса
                  if (isRectorId(id)) {
                            return new Rector(id);
                  } else {
                            return new Teacher(id);
                  }
        }
        private static boolean isRectorId(int id) {
                  // проверка id
                  return id  == 6; // stub
        }
}
package by.bsu.inner.study;
public class TeacherLogic {
        public void excludeProcess(int rectorId, String nameStudent) {
                 AbstractTeacher teacher = TeacherCreator.createTeacher(rectorId);

                 System.out.println("Студент: " + nameStudent
                 + " отчислен:" + teacher.excludeStudent(nameStudent));
        }
}
package by.bsu.inner.study;
public class Runner {
```

```
        public static void main(String[ ] args) {
                TeacherLogic tl = new TeacherLogic();
                tl.excludeProcess(777, "Олейников");
                tl.excludeProcess(6, "Олейников");
        }
}
```

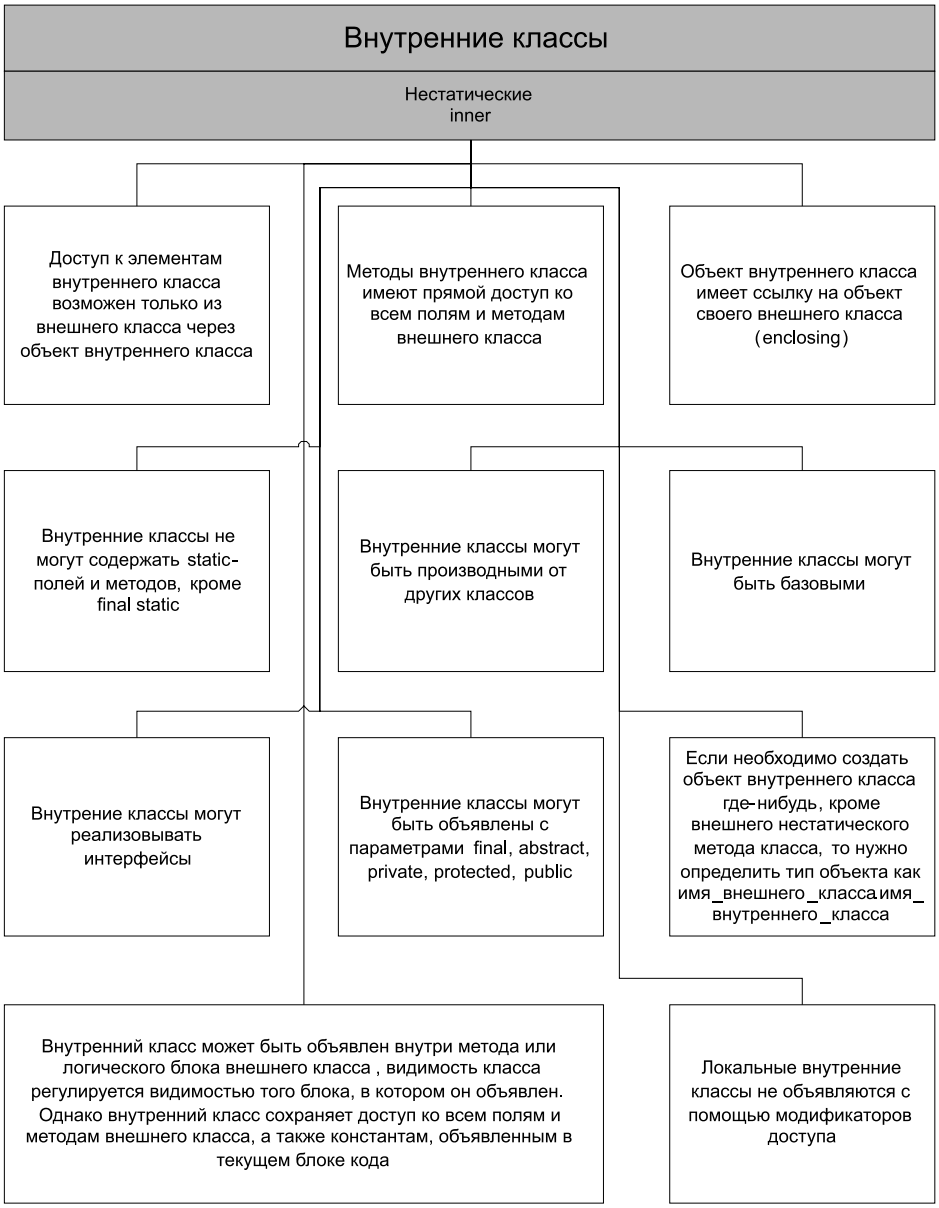

Рис. 5.1. *Внутренние классы*

В результате будет выведено:

Студент: Олейников отчислен: false
Студент: Олейников отчислен: true

Класс **Rector** объявлен в методе **createTeacher(int id)**, и, соответственно, объекты этого класса можно создавать только внутри 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, как в данном случае, при наследовании внутренним классом функциональности обычного класса, в частности, **AbstractTeacher**. При компиляции данного кода с внутренним классом ассоциируется объектный модуль со сложным именем **TeacherCreator$1Rector.class**, тем не менее однозначно определяющим связь между внешним и внутренним классами. Цифра **1** в имени говорит о том, что потенциально в других методах класса могут быть объявлены внутренние классы с таким же именем.

Вложенные (nested) классы

Если не существует жесткой 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. Если предполагается использовать внутренний класс в качестве подкласса, следует исключить использование в его теле любых прямых обращений к членам класса владельца.

/ # 7 # вложенный класс # Ship.java # RunnerShip.java */*

```
package by.bsu.nested;
public class Ship {
          private int id;
```

```
        // abstract, final, private, protected - допустимы
        public static class LifeBoat {
                private int boatId;
                public static void down() {
                        System.out.println("шлюпки на воду!");
                }
                public void swim() {
                        System.out.println("отплытие шлюпки");
                }
        }
}
package by.bsu.nested;
public class RunnerShip {
        public static void main(String[ ] args) {
                // вызов статического метода
                Ship.LifeBoat.down();
                // создание объекта статического класса
                Ship.LifeBoat lifeBoat = new Ship.LifeBoat();
                // вызов обычного метода
                lifeBoat.swim();
        }
}
```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему. Объект **lifeBoat** 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.

/ # 8 # класс, вложенный в интерфейс # LearningDepartment.java # University.java */*

```
package by.bsu.nested;
public interface University {
        int NUMBER_FACULTY = 20;
        void create();
        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.

Рис. 5.2. *Вложенные классы*

Анонимные (anonymous) классы

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому экземпляру, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков 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.

Конструктор анонимного класса определить невозможно.

/ # 9 # анонимные классы # WrapperString.java # Runner.java */*

```
package by.bsu.anonym;
public class WrapperString {
          private String str;
          public WrapperString(String str) {
                    this.str = str;
          }
          public String getStr() {
                    return str;
          }
          public String replace(char oldChar, char newChar) { // замена первого символа
                    char[] array = new char[str.length()];
                    str.getChars(0, str.length(), array, 0);
                    for (int i = 0; i < array.length; i++) {
                              if (array[i] == oldChar) {
                                        array[i] = newChar;
                                        break;
                              }
                    }
                    return new String(array);
          }
}
package by.bsu.anonym;
public class Runner {
          public static void main(String[ ] args) {
                     String ob = "qweRtRRR";
                     WrapperString wrFirst = new WrapperString(ob);
                     // анонимный класс #1
                     WrapperString wrLast = new WrapperString(ob) {
                               // замена последнего символа
                               @Override
                               public String replace(char oldChar, char newChar) {
                                        char[] array = new char[getStr().length()];
                                        getStr().getChars(0, getStr().length(), array, 0);
                                        for (int i = array.length - 1; i > 0; i--) {
                                                  if (array[i] == oldChar) {
                                                            array[i] = newChar;
                                                            break;
                                                  }
                                        }
                                        return new String(array);
                               }
                     }; // конец объявления анонимного класса
                     WrapperString wr2 = new WrapperString(ob) { // анонимный класс #2
                               private int position = 2; // собственное поле
                               // замена символа по позиции
                               public String replace(char oldChar, char newChar) {
                                        int counter = 0;
```

```
                                        char[] array = new char[getStr().length()];
                                        getStr().getChars(0, getStr().length(), array, 0);
                                        if (verify(oldChar, array)) {
                                                for (int i = 0; i < array.length; i++) {
                                                        if (array[i] == oldChar) {
                                                                counter++;
                                                                if (counter == position) {
                                                                        array[i] = newChar;
                                                                                break;
                                                                }
                                                        }
                                                }
                                        }
                                        return new String(array);
                                }
                                // собственный метод
                                public boolean verify(char oldChar, char[] array){
                                        int counter = 0;
                                        for (char c : array) {
                                                if (c == oldChar) {
                                                    counter++;
                                                }
                                        }
                                        return counter >= position;
                                }
                        }; // конец объявления анонимного класса
                        System.out.println(wrLast.replace('R', 'Y'));
                        System.out.println(wr2.replace('R', 'Y'));
                        System.out.println(wrFirst.replace('R', 'Y'));
                }
}
```

В результате будет выведено:

qweRtRRY
qweRtYRR
qweYtRRR

При запуске приложения происходит объявление объекта **wrLast** с применением анонимного класса, в котором переопределяется метод **replace().** Вызов данного метода на объекте **wrLast** приводит к вызову версии метода из анонимного класса, который компилируется в объектный модуль с именем **Runner$1.class**. Процесс создания второго объекта с анонимным типом применяется в программировании значительно чаще, особенно при реализации классов-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.

/ # 10 # анонимный класс в перечислении # Shape.java # EnumRunner.java */*

```
package by.bsu.enums;
public enum Shape {
        RECTANGLE, SQUARE, TRIANGLE { // анонимный класс
                public double computeSquare() { // версия для TRIANGLE
                        return this.getA() * this.getB() / 2;
                }
        };
        private double a;
        private double b;
        public double getA() {
                return a;
        }
        public double getB() {
                return b;
        }
        public void setShape(double a, double b) {
                if ((a <= 0 || b <= 0) || a != b && this == SQUARE) {
                        throw new IllegalArgumentException();
                }
                this.a = a;
                this.b = b;
        }
        public double computeSquare() { // версия для RECTANGLE и SQUARE
                return a * b;
        }
        public String toString() {
                return name() + "-> 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 + " площадь= " + f.computeSquare());
                }
        }
}
```

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

Рис. 5.3. *Анонимные классы*

Объектный модуль для такого анонимного класса будет скомпилирован с именем **Shape$1**.

Ситуации, в которых следует использовать внутренние классы:

- выделение самостоятельной логической части сложного класса;
- сокрытие реализации;
- одномоментное использование переопределенных методов. В том числе обработка событий;
- запуск потоков выполнения;
- отслеживание внутреннего состояния, например, с помощью **enum**.

Анонимные классы, как и остальные внутренние, допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными, поэтому в практике программирования данная техника не используется.

Задания к главе 5

Вариант А

1. Создать класс **Notepad** с внутренним классом или классами, с помощью объектов которого могут храниться несколько записей на одну дату.

2. Создать класс **Payment** с внутренним классом, с помощью объектов которого можно сформировать покупку из нескольких товаров.
3. Создать класс **Account** 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** 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** с внутренним классом, с помощью объектов которого можно хранить информацию об истории выдач книги читателям.
7. Создать класс **Европа** с внутренним классом, с помощью объектов которого можно хранить информацию об истории изменения территориального деления на государства.
8. Создать класс **City** с внутренним классом, с помощью объектов которого можно хранить информацию о проспектах, улицах, площадях.
9. Создать класс **BlueRayDisc** 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** с внутренним классом, с помощью объектов которого можно хранить информацию об отделах, товарах и услугах.
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** с внутренним классом, с помощью объектов которого можно хранить информацию об операционной системе, процессоре и оперативной памяти.
16. Создать класс **Park** с внутренним классом, с помощью объектов которого можно хранить информацию об аттракционах, времени их работы и стоимости.
17. Создать класс **Cinema** с внутренним классом, с помощью объектов которого можно хранить информацию об адресах кинотеатров, фильмах и времени начала 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.

Тестовые задания к главе 5

Вопрос 5.1.

Дан код:

```
public class Quest1 {
public static void main (String [] args) {
        for (Numbers num: Numbers.values ()) {
                System.out.print (num.getNumber ());
                }
        }}
enum Numbers {
        ONE (1), TWO (2) {public int getNumber () {return x + x;}
        }, THREE (3) {public int getNumber () {return x + x + x;}
        }, FOUR (4), FIVE (5);
        int x;
        Numbers (int x) {
                this.x = x;
        }
        public int getNumber () {
                return x;
        }}
```

Что будет результатом компиляции и выполнения данного кода (1)?

1) строка 12345
2) строка 54321
3) строка 14945
4) строка 54941
5) строка 12945
6) строка 54921
7) строка 14345
8) строка 54341
9) ошибка компиляции

Вопрос 5.2.

Дан код:

```
public class Quest3 {
public static void main (String [] args) {
        Outer obj = new Outer ().new Inner1 ();
                obj.print ();
        }}
class Outer {
        public void print () {}
        class Inner1 _____ // line 1
        {       public void print () {
                        System.out.println ("In inner.");
                }
        }
}
```

Что необходимо дописать в line 1, чтобы при компиляции и запуске на консоль вывелась строка In inner (1)?

1) ничего, код написан верно
2) **implements** Outer
3) **extends** Outer
4) нет верного варианта

Вопрос 5.3.

Выберите правильный вариант доступа из внутреннего Inner класса к экземпляру его внешнего Outer класса (1):

1) Outer.**class**.**this;**
2) **new** Outer ().**this;**
3) Inner.Outer.**class**.**this;**
4) Outer.**class**.newInstance ().**this;**
5) Outer.**this.**

Вопрос 5.4.

Даны два фрагмента кода:

Фрагмент 1

```
new Object () {
        public void hello () {
                System.out.print ("Hello!");
        }
}.hello ();
```

Фрагмент 2

```
Object obj = new Object () {
        public void hello () {
                System.out.print ("Hello!");
        }
};
obj.hello ();
```

Каким будет результат компиляции и запуска этих фрагментов кода (1)?

1) и первый, и второй фрагменты кода скомпилируются и выведут на консоль строку «Hello!»;
2) первый фрагмент кода скомпилируется и выведет на консоль строку «Hello!», при компиляции второго фрагмента возникнет ошибка;
3) второй фрагмент кода скомпилируется и выведет на консоль строку «Hello!», при компиляции первого фрагмента возникнет ошибка;
4) ни первый, ни второй фрагмент кода не скомпилируются.

Вопрос 5.5.

Выберите неправильные утверждения (5):

1) методы внутреннего (нестатического) класса имеют прямой доступ только к статическим полям и методам внешнего класса;
2) доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса;
3) внутренние классы могут содержать static-поля;
4) внутренние классы не могут реализовывать интерфейсы;
5) внутренние классы могут быть объявлены только с параметрами final, abstract, public, protected, private;
6) внутренний локальный класс обладает доступом только к полям метода, в котором он объявлен;
7) внутренние классы могут содержать static-поля, только если они final static;
8) внутренние классы могут быть производными классами, но не базовыми.

Глава 6

ИНТЕРФЕЙСЫ И АННОТАЦИИ

Решение сложной задачи поручайте ленивому сотруднику – он найдет более легкий путь.

Закон Хлейда

Интерфейсы

Назначение интерфейса — описание или спецификация функциональности, которую должен реализовывать каждый класс, его имплементирующий. Класс, реализующий интерфейс, предоставляет к использованию объявленный интерфейс в виде набора **public**-методов в полном объеме.

Интерфейсы подобны полностью абстрактным классам, хотя и не являются классами. Ни один из объявленных методов не может быть реализован внутри интерфейса. В языке Java существует два вида интерфейсов: интерфейсы, определяющие функциональность для классов посредством описания методов, но не их реализации; и интерфейсы, реализация которых автоматически придает классу определенные свойства. К последним относятся, например, интерфейсы **Cloneable** и **Serializable**, отвечающие за клонирование и сохранение объекта в информационном потоке соответственно.

Общее определение интерфейса имеет вид:

```
[public] interface Имя [extends Имя1, Имя2,…, ИмяN] { /* реализация интерфейса */ }
```

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.

Если необходимо определить набор функциональности для какого-либо рода деятельности, например, для управления счетом в банке, то следует использовать интерфейс вида:

/ # 1 # обьявление интерфейса управления банковским счетом # AccountAction.java */*

```
package by.bsu.proj.accountlogic;
public interface AccountAction {
        // по умолчанию все методы public abstract
```

```
        boolean openAccount(); // объявление сигнатуры метода
        boolean closeAccount();
        void blocking();
        void unBlocking();
        double depositInCash(int accountNumber, int amount);
        boolean withdraw(int accountNumber, int amount);
        boolean convert(double amount);
        boolean transfer(double amount);
}
```

В интерфейсе обозначены, но не реализованы, действия, которые может производить клиент со своим счетом. Реализация не представлена из-за возможности различного способа выполнения действия в конкретной ситуации. А именно: счет может блокироваться автоматически, по требованию клиента или администратором банковской системы. В каждом из трех указанных случаев реализация метода **blocking()** будет уникальной и никакого базового решения предложить невозможно. С другой стороны, наличие в интерфейсе метода заставляет класс, его имплементирующий, предоставить реализацию методу. Программист получает повод задуматься о способе реализации функциональности, так как наличие метода в интерфейсе говорит о необходимости той или иной функциональности всем классам, реализующим данный интерфейс.

Интерфейс можно сделать ориентированным на выполнение близких по смыслу задач, например, отделить действия по созданию, закрытию и блокировке счета от действий по снятию и пополнению средств. Такое разделение разумно в ситуации, когда клиенту посредством Интернет-системы не предоставляется возможность открытия, закрытия и блокировки счета. Тогда вместо одного общего интерфейса можно записать два специализированных: один для администратора, второй — для клиента.

/ # 2 # общее управление банковским счетом # AccountBaseAction.java */*

```
package by.bsu.proj.accountlogic;
public interface AccountBaseAction {
        boolean openAccount();
        boolean closeAccount();
        void blocking();
        void unBlocking();
}
```

/ # 3 # операционное управление банковским счетом # AccountOperationManager.java */*

```
package by.bsu.proj.accountlogic;
public interface AccountOperationManager {
        double depositInCash(int accountNumber, int amount);
        boolean withdraw(int accountNumber, int amount);
        boolean convert(double amount);
        boolean transfer(int accountNumber, double amount);
}
```

Реализация интерфейса при реализации всех методов выглядит следующим образом:

/ # 4 # реализация общего управления банковским счетом # AccountBaseActionImpl.java */*

```
package by.bsu.proj.accountlogic;
public class AccountBaseActionImpl implements AccountBaseAction {
          public boolean openAccount() {
            // more code
          }
          public boolean closeAccount() {
            // more code
          }
          public void blocking() {
            // more code
          }
          public void unBlocking() {
            // more code
          }
}
```

Если по каким-либо причинам метод для данного класса не имеет реализации или его реализация нежелательна, рекомендуется генерация исключения в теле метода, а именно:

```
public boolean blocking() {
          throw new UnsupportedOperationException(); // лучше собственное исключение
}
```

Менее хорошим примером будет реализация в виде:

```
public boolean blocking() {
          return false;
}
```

так как пользователь метода будет считать реализацию корректной.

В интерфейсе не могут быть объявлены поля без инициализации и методы с реализацией, интерфейс в качестве полей содержит только константы, в качестве методов — только абстрактные методы.

```
interface WrongInterface { // нежелательно использование слова Interface при объявлении
          int id; // ошибка, инициализации "по умолчанию" не существует
          void method() {} // ошибка, так как абстрактный метод не может иметь тела!
}
```

Множественное наследование между интерфейсами допустимо. Классы, в свою очередь, интерфейсы только реализуют. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов. В языке Java интерфейсы обеспечивают большую часть той функциональности,

которая в C++ представляется с помощью механизма множественного наследования.

Например:

/ # 5 # обьявление интерфейсов # ILineGroupAction.java # IShapeAction.java */*

```
package by.bsu.shapes.action;
public interface ILineGroupAction {
          double computePerimeter(AbstractShape shape);
}
```

/ # 6 # наследование интерфейсов # IShapeAction.java */*

```
package by.bsu.shapes.action;
public interface IShapeAction extends ILineGroupAction {
          double computeSquare(AbstractShape shape);
}
```

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 **ShapeAction** следует записать **IShapeAction**. В конец названия может добавляться **able** в случае, если в названии присутствует действие (глагол). В конец имени класса, реализующего интерфейс, для указания на источник действий часто добавляется слово **Impl**.

Класс, который будет реализовывать интерфейс **IShapeAction**, должен будет определить все методы из цепочки наследования интерфейсов. В данном случае это методы **computePerimeter()** и **computeSquare().**

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, атрибут доступа по умолчанию, могут использоваться только в этом пакете и нигде более.

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 того, 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**.

/ # 7 # реализация интерфейса # RectangleAction.java */*

```
package by.bsu.shapes.action;
import by.bsu.shapes.entity.AbstractShape;
import by.bsu.shapes.entity.Rectangle;
public class RectangleAction implements IShapeAction {
```

```
        @Override // реализация метода из интерфейса
        public double computeSquare(AbstractShape shape)  { // площадь прямоугольника
                double square = 0;
                // необходимо проверить тип
                if (shape instanceof Rectangle) {
                        Rectangle rectangle = (Rectangle) shape;
                        square = rectangle.getA() * rectangle.getB();
                } else {
                        throw new IllegalArgumentException("Incompatible shape:"
                                                        + shape.getClass());
                }
                return square;
        }
        @Override // реализация метода из интерфейса
        public double computePerimeter(AbstractShape shape) { // периметр прямоугольника
                double perimeter = 0;
                if (shape instanceof Rectangle) {
                        Rectangle rectangle = (Rectangle) shape;
                        perimeter = 2 * (rectangle.getA() + rectangle.getB());
                } else {
                        throw new IllegalArgumentException("Incompatible shape:"
                                                        + shape.getClass());
                }
                return perimeter;
        }
}
```

/ # 8 # реализация интерфейса # TriangleAction.java */*

```
package by.bsu.shapes.action;
import by.bsu.shapes.entity.AbstractShape;
import by.bsu.shapes.entity.Triangle;
public class TriangleAction implements IShapeAction {
        @Override
        public double computeSquare(AbstractShape shape) {
                double square = 0;
                if (shape instanceof Triangle) {
                        Triangle triangle = (Triangle) shape;
                        square = 1 / 2 * triangle.getA() * triangle.getB()
                                        * Math.sin(triangle.getAngle());
                } else {
                        throw new IllegalArgumentException("Incompatible shape"
                                                        + shape.getClass());
                }
                return square;
        }
        @Override
        public double computePerimeter(AbstractShape shape) {
                double perimeter = 0;
                if (shape instanceof Triangle) {
```

```
                    Triangle triangle = (Triangle) shape;
                    perimeter = triangle.getA() + triangle.getB() + triangle.getC();
                } else {
                    throw new IllegalArgumentException("Incompatible shape"
                                                        + shape.getClass());
                }
                return perimeter;
        }
}
```

При неполной реализации интерфейса класс должен быть объявлен как абстрактный, что говорит о необходимости реализации абстрактных методов уже в его подклассе.

/ # 9 # неполная реализация интерфейса # PentagonAction.java */*

```
package by.bsu.shapes.action;
import by.bsu.shapes.entity.AbstractShape;
/* метод computeSquare() в данном абстрактном классе не реализован */
public abstract class PentagonAction implements IShapeAction {
        // поля, конструкторы
        @Override
        public double computePerimeter(AbstractShape shape) {
                // проверка и вычисление периметра
        }
}
```

Классы для определения фигур, используемые в интерфейсах:

/ # 10 # абстрактная фигура, прямоугольник, треугольник # AbstractShape.java # Rectangle.java # Triangle.java */*

```
package by.bsu.shapes.entity;
public abstract class AbstractShape {
        private double a;
        public AbstractShape(double a) {
                this.a = a;
        }
        public double getA() {
                return a;
        }
}
package by.bsu.shapes.entity;
public class Rectangle extends AbstractShape {
        private double b;
        public Rectangle(double a, double b) {
                super(a);
                this.b = b;
        }
```

```
        public double getB() {
                return b;
        }
}
package by.bsu.shapes.entity;
public class Triangle extends AbstractShape {
        private double b;
        private double angle; // угол между сторонами в радианах
        public Triangle(double a, double b, double angle) {
                super(a);
                this.b = b;
                this.angle = angle;
        }
        public double getAngle() {
                return angle;
        }
        public double getB() {
                return b;
        }
        public double getC() {
                double c = // stub : вычисление по теореме косинусов
                return c;
        }
}
```

/ # 11 # свойства ссылки на интерфейс # ActionMain.java */*

```
package by.bsu.shapes;
import by.bsu.shapes.action.IShapeAction;
import by.bsu.shapes.action.RectangleActionImpl;
import by.bsu.shapes.action.TriangleActionImpl;
import by.bsu.shapes.entity.AbstractShape;
import by.bsu.shapes.entity.Rectangle;
import by.bsu.shapes.entity.Triangle;
import static java.lang.Math.PI;
public class ActionMain {
   public static void main(String[ ] args) {
      IShapeAction action;
      try {
          Rectangle rectShape = new Rectangle(2, 3);
          action = new RectangleAction();
          System.out.println("Square rectangle: " + action.computeSquare(rectShape));
          System.out.println("Perimeter rectangle: " + action.computePerimeter(rectShape));

          Triangle trShape = new Triangle(3, 4, PI/6);
          action = new TriangleAction ();
          System.out.println("Square triangle: " + action.computeSquare(trShape));
          System.out.println("Perimeter triangle: " + action.computePerimeter(trShape));
          action.computePerimeter(rectShape); // ошибка времени выполнения
```

```
        } catch (IllegalArgumentException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

В результате будет выведено:

Square rectangle: 6.0
Perimeter rectangle: 10.0
Square triangle: 3.0
Perimeter triangle: 8.0
Incompatible shape class by.bsu.shapes.entity.Rectangle

Допустимо объявление ссылки на интерфейсный тип или использование ее в качестве параметра метода. 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.

Параметризация интерфейсов

Реализация для интерфейсов, параметры методов которых являются ссылками на иерархически организованные классы, в данном случае (предыдущий пример) представлена достаточно тяжеловесно. Необходимость проверки принадлежности обрабатываемого объекта к допустимому типу снижает гибкость программы и увеличивает количество кода, в том числе и на генерацию и обработку исключений.

Сделать реализацию интерфейса удобной, менее подверженной ошибкам и практически исключающей проверки на принадлежность типу можно достаточно легко, если при описании интерфейса добавить параметризацию в виде:

/ # 12 # параметризация интерфейса # IShapeAction.java */*

```
package by.bsu.shapes.action;
public interface IShapeAction <T extends AbstractShape> {
          double computeSquare(T shape);
          double computePerimeter(T shape);
}
```

Параметризованный тип **T extends AbstractShape** указывает, что в качестве параметра методов может использоваться только подкласс **AbstractShape**, что, в общем, мало чем отличается от случая, когда тип параметра метода указывался явно. Но когда дело доходит до реализации интерфейса, то при реализации

интерфейса указывается конкретный тип объектов, являющийся подклассом **AbstractShape**, которые будут обрабатываться методами данного класса, а в качестве параметра метода также прописывается тот же самый конкретный тип:

/ # 13 # реализация интерфейса с указанием типа параметра # RectangleAction.java # TriangleAction.java */*

```
package by.bsu.shapes.action;
public class RectangleAction implements IShapeAction<Rectangle> {
          @Override
          public double computeSquare(Rectangle shape) {
                    return shape.getA() * shape.getB();
          }
          @Override
          public double computePerimeter(Rectangle shape) {
                    return 2 * (shape.getA() + shape.getB());
          }
}
package by.bsu.shapes.action;
public class TriangleAction implements IShapeAction<Triangle> {
          @Override
          public double computeSquare(Triangle shape) {
                    return 0.5 * shape.getA() * shape.getB() * Math.sin(shape.getAngle());
          }
          @Override
          public double computePerimeter(Triangle shape) {
                    return shape.getA() + shape.getB() + shape.getC();
          }
}
```

На этапе компиляции исключается возможность передачи в метод объекта, который не может быть обработан, т. е. код **action.computePerimeter(rectShape);** спровоцирует ошибку компиляции, если **action** инициализирован объектом класса **TriangleAction**.

Применение параметризации при объявлении интерфейсов в данном случае позволяет избавиться от лишних проверок и преобразований типов при реализации непосредственно самого интерфейса и использовании созданных на их основе классов.

/ # 14 # использование параметризованных интерфейсов # ShapeMain.java */*

```
package by.bsu.shapes.action;
public class ShapeMain {
    public static void main(String[ ] args) {
          Rectangle rectShape = new Rectangle(2, 3);
          IShapeAction<Rectangle> rectAction = new RectangleAction();
          Triangle trShape = new Triangle(3, 4, PI / 6);
          IShapeAction<Triangle> trAction = new TriangleAction();
```

```
        System.out.println("Square rectangle: " + rectAction.computeSquare(rectShape));
        System.out.println("Perimeter rectangle: " + rectAction.computePerimeter(rectShape));
        System.out.println("Square triangle: " + trAction.computeSquare(trShape));
        System.out.println("Perimeter triangle: " + trAction.computePerimeter(trShape));
        // trAction.computePerimeter(rectShape); // ошибка компиляции
        }
}
```

Аннотации

Аннотации — это своего рода метатеги, которые добавляются к коду и применяются к объявлению пакетов, классов, конструкторов, методов, полей, параметров и локальных переменных. Аннотации всегда обладают некоторой информацией и связывают эти «дополнительные данные» и все перечисленные конструкции языка. Фактически аннотации представляют собой их дополнительные модификаторы, применение которых не влечет за собой изменений ранее созданного кода. Аннотации позволяют избежать создания шаблонного кода во многих ситуациях, активируя утилиты для его генерации из аннотаций в исходном коде.

В языке Java SE определено несколько встроенных аннотаций, четыре типа — **@Retention**, **@Documented**, **@Target** и **@Inherited** — из пакета **java.lang.annotation**. Из оставшиеся выделяются — **@Override**, **@Deprecated** и **@SuppressWarnings** — из пакета **java.lang**. Широкое использование аннотаций в различных технологиях и фреймворках обуславливается возможностью сокращения кода и снижения его связанности.

В следующем коде приведено объявление аннотации.

/ # 15 # многочленная аннотация класса # BaseAction.java */*

```
@Target(ElementType.TYPE)
public @interface BaseAction {
        int level();
        String sqlRequest();
}
```

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

После объявления аннотации ее можно использовать для аннотирования объявлений класса. Объявление практически любого типа может иметь аннотацию, связанную с ним. Даже к аннотации можно добавить аннотацию. Во всех случаях аннотация предшествует объявлению.

Применяя аннотацию, нужно задавать значения для ее методов-членов, если при объявлении аннотации не было задано значение по умолчанию. Далее приведен фрагмент, в котором аннотация **BaseAction** сопровождает объявление класса:

/ # 16 # примитивное использование аннотации класса # Base.java */*

```
@BaseAction (
          level = 2,
          sqlRequest = "SELECT * FROM phonebook"
          )
public class Base {
          public void doAction() {
                              Class<Base> f = Base.class;
                              BaseAction a = (BaseAction)f.getAnnotation(BaseAction.class);
                              System.out.println(a.level());
                              System.out.println(a.sqlRequest());
          }
}
```

Данная аннотация помечает класс **Base**. 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 **"SELECT * FROM phonebook"** присваивается методу **sqlRequest(),** члену аннотации типа **BaseAction**. При этом в операции присваивания после имени **sqlRequest** нет круглых скобок. Когда методу-члену передается инициализирующее значение, используется только имя метода. Следовательно, в данном контексте методы-члены выглядят как поля.

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 **Annotation** из пакета **java.lang.annotation**. В этом интерфейсе приведен метод **annotationType(),** который возвращает объект типа **Class**, представляющий вызывающую аннотацию.

Если необходим доступ к аннотации в процессе функционирования приложения, то перед объявлением аннотации задается правило сохранения **RUNTIME** в виде кода

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

предоставляющее 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 *имя***.class**, но недоступна в JVM во время выполнения.

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 MarkerAnnotation {}
```

Для проверки наличия аннотации 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.

Реализация обработки аннотации, приведенная в методе **doAction()** класса **Base**, крайне примитивна. Разработчику каждый раз при использовании аннотаций придется писать код по извлечению значений полей-членов и реакцию метода, класса и поля на значение аннотации. Необходимо привести реализацию аннотации таким образом, чтобы программисту достаточно было только аннотировать класс, метод или поле и передать нужное значение. Реакция системы на аннотацию должна быть автоматической. Ниже приведен пример работы с аннотацией, реализованной на основе ReflectionAPI. Примеры корректного использования аннотаций короткими не бывают.

Аннотация **BankingAnnotation** предназначена для задания уровня проверки безопасности при вызове метода.

/ # 17 # одночленная аннотация метода # BankingAnnotation.java */*

```
package by.bsu.proj.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface BankingAnnotation {
          SecurityLevelEnum securityLevel() default SecurityLevelEnum.NORMAL;
}
```

/ # 18 # возможные значения уровней безопасности # SecurityLevelEnum.java */*

```
package by.bsu.proj.annotation;
public enum SecurityLevelEnum {
          LOW, NORMAL, HIGH
}
```

Методы класса логики системы выполняют действия с банковским счетом. В зависимости от различных причин конкретным реализациям интерфейса **AccountOperationManager** могут понадобиться дополнительные действия помимо основных.

/ # 19 # аннотирование методов # AccountOperationManagerImpl.java */*

```
package by.bsu.proj.accountlogic;
import by.bsu.proj.annotation.BankingAnnotation;
import by.bsu.proj.annotation.SecurityLevelEnum;
public class AccountOperationManagerImpl implements AccountOperationManager {
          @BankingAnnotation(securityLevel = SecurityLevelEnum.HIGH)
          public double depositInCash(int accountNumber, int amount) {
                    // зачисление на депозит
                    return 0; // stub
          }
          @BankingAnnotation(securityLevel = SecurityLevelEnum.HIGH)
          public boolean withdraw(int accountNumber, int amount) {
                    // снятие суммы, если не превышает остаток
                    return true; // stub
          }
          @BankingAnnotation(securityLevel = SecurityLevelEnum.LOW)
          public boolean convert(double amount) {
                    // конвертировать сумму
                    return true; // stub
          }
          @BankingAnnotation
          public boolean transfer(int accountNumber, double amount) {
                    // перевести сумму на счет
                    return true; // stub
          }
}
```

/ # 20 # конфигурирование и запуск # AnnoRunner.java */*

```
package by.bsu.proj.run;
import by.bsu.proj.accountlogic.AccountOperationManager;
import by.bsu.proj.accountlogic.AccountOperationManagerImpl;
import by.bsu.proj.annotation.logic.SecurityFactory;
public class AnnoRunner {
          public static void main(String[] args) {
                    AccountOperationManager account = new AccountOperationManagerImpl();
                    // "регистрация класса" для включения аннотаций в обработку.
                    AccountOperationManager securityAccount =
                                                  SecurityFactory.createSecurityObject(account);
                    securityAccount.depositInCash(10128336, 6);
                    securityAccount.withdraw(64092376, 2);
```

```
                securityAccount.convert(200);
                securityAccount.transfer(64092376, 300);
        }
}
```

Подход с вызовом некоторого промежуточного метода для включения в обработку аннотаций достаточно распространен и, в частности, используется в технологии JPA.

Вызов метода **createSecurityObject(),** регистрирующего методы класса для обработки аннотаций, можно разместить в конструкторе некоторого промежуточного абстрактного класса, реализующего интерфейс **AccountOperationManager**, или в статическом логическом блоке самого интерфейса. Тогда реагировать на аннотации будут все методы всех реализаций интерфейса **AccountOperationManager.**

Класс, определяющий действия приложения в зависимости от значения **SecurityLevelEnum**, передаваемого в аннотированный метод.

/ # 21 # логика обработки значения аннотации # SecurityLogic.java */*

```
package by.bsu.proj.annotation.logic;
import java.lang.reflect.Method;
import by.bsu.proj.annotation.SecurityLevelEnum;
public class SecurityLogic {
        public void onInvoke(SecurityLevelEnum level, Method method, Object[] args) {
                StringBuilder argsString = new StringBuilder();
                for (Object arg : args) {
                        argsString.append(arg.toString() + " :");
                }
                argsString.setLength(argsString.length() - 1);
                System.out.println(String.format(
                        "Method %S was invoked with parameters : %s", method.getName(),
                                argsString.toString()));
                switch (level) {
                  case LOW:
                       System.out.println("Низкий уровень проверки безопасности: " + level);
                          break;
                  case NORMAL:
                       System.out.println("Обычный уровень проверки безопасности: " + level);
                          break;
                  case HIGH:
                       System.out.println("Высокий уровень проверки безопасности: " + level);
                          break;
                }
        }
}
```

Статический метод **createSecurityObject(Object targetObject)** класса-фабрики создает для экземпляра класса **AccountOperationManagerImpl** экземпляр-заместитель, обладающий кроме всех свойств оригинала возможностью

неявного обращения к логике, выбор которой определяется выбором значения для поля аннотации.

/ # 22 # создание прокси-экземпляра, включающего функциональность SecurityLogic # SecurityFactory.java */*

```
package by.bsu.proj.annotation.logic;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import by.bsu.proj.accountlogic.AccountOperationManager;
import by.bsu.proj.annotation.BankingAnnotation;
public class SecurityFactory {
          public static AccountOperationManager createSecurityObject(
                                    AccountOperationManager targetObject) {
                    return (AccountOperationManager)Proxy.newProxyInstance(
                                    targetObject.getClass().getClassLoader(),
                                    targetObject.getClass().getInterfaces(),
                                    new SecurityInvokationHandler(targetObject));
          }
          private static class SecurityInvokationHandler implements InvocationHandler {
                    private Object targetObject = null;
                    public SecurityInvokationHandler(Object targetObject) {
                              this.targetObject = targetObject;
                    }
                    public Object invoke(Object proxy, Method method, Object[ ] args)
                                    throws Throwable {
                              SecurityLogic logic = new SecurityLogic();
                              Method realMethod = targetObject.getClass().getMethod(
                                                  method.getName(),
                                                  (Class[]) method.getGenericParameterTypes());

                              BankingAnnotation annotation = realMethod
                                                  .getAnnotation(BankingAnnotation.class);
                              if (annotation != null) {
                                 // доступны и аннотация и параметры метода
                                  logic.onInvoke(annotation.securityLevel(), realMethod, args);
                                    try {
                                              return method.invoke(targetObject, args);
                                    } catch (InvocationTargetException e) {
                                              System.out.println(annotation.securityLevel());
                                              throw e.getCause();
                                    }
                              } else {
                              /* в случае если аннотирование метода обязательно следует
                              генерировать исключение на факт ее отсутствия */
                              /* throw new InvocationTargetException(null, "method "
                                        + realMethod + " should be annotated "); */
```

```
                        // в случае если допустимо отсутствие аннотации у метода
                        return null;
                        }
                }
        }
}
```

Использование аннотации может быть обязательным или опциональным. Если метод **invoke()** при отсутствии аннотации у метода генерирует исключение, то это говорит об обязательности явного использования аннотации всеми без исключения методами класса. Если метод возвращает в этой ситуации значение **null**, то аннотированными могут быть только те методы, которые необходимы программисту. Последний вариант использования предпочтителен, так как не требует обязательного использования аннотации для методов общего назначения, например: **equals()** или **toString().**

Задания к главе 6

Вариант А

Создать и реализовать интерфейсы, также использовать наследование и полиморфизм для следующих предметных областей:

1. interface Издание ← abstract class Книга ← class 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.

Что будет результатом компиляции и запуска следующего кода (1)?

```
interface Quest10{  Number returner();  }
abstract class Quest100{
        public abstract Integer returner();
}
public class Quest1 extends Quest100 implements Quest10{//line 1
        @Override //line 2
        public Integer returner() {// line 3
                return new Integer(6);
        }
        public static void main(String[] args) {
                Quest1 quest = new Quest1();
                Quest10 quest10 = quest;
                Quest100 quest100 = quest;
                System.out.println(quest10.returner()+""+quest100.returner());
        }}
```

1) ошибка компиляции в строке 1
2) ошибка компиляции в строке 2
3) ошибка компиляции в строке 3
4) компиляция и запуск программы осуществится без ошибок

Вопрос 6.2.

Даны объявления интерфейсов. Которые из них скомпилируются с ошибкой (2)?

1)
```
interface Quest20{
        class Inner{
                private int x;
        }
}
```
2)
```
interface Quest21{
        static class Inner{  int x;  }
}
```
3)
```
class Quest22{
        interface Inner{  int x;  }
}
```

4)
```
class Quest23{
        static interface Inner{
                int x;
        }
}
```

Вопрос 6.3.

Дан код:
```
interface Inter1{
        void f();
}
interface Inter2 extends Inter1{
        void f();
}
class Cl1 implements Inter1{
        public void f(){
                System.out.println("one");
        }
}
class Cl2 implements Inter2{
        public void f(){
                System.out.println("two");
        }
}
```

Какой метод f() будет вызван при выполнении следующего кода (1)?
```
Inter2 obj = new Cl2();
((Inter1) obj).f();
```

1) метод f() класса Cl1
2) метод f() класса Cl2
3) произойдет ошибка компиляции, т. к. ссылка типа Inter2 не может ссылаться на объект Cl2
4) произойдет ошибка компиляции, т. к. ссылку obj1 нельзя привести к типу Inter1

Вопрос 6.4.

Укажите, какое ключевое слово используется для объявления аннотации:

1) @interface;
2) @class;
3) @annotation;
4) @custom_annotation.

Контрольная работа к части 1

Вариант А

Ф.И._____ «__» гр.

0. Если возможно, создать подкласс для класса. Ответ обосновать.

```
class Policy {
        private int a;
        Policy(int a){
                this.a=a;}}
```

1. В строки, помеченные комментариями //1 и //2, записать оптимальные вызовы метода execute()

```
public class Hope {
        public static void execute(){}
        private void taste(){//1
}}
public class Dream {
        private Float taste(){//2
}}
```

2. Записать реализацию интерфейса

```
interface Application {
        void hello();}
```

3. Если возможно, создать подкласс абстрактного класса. Ответ обосновать

```
abstract class Cook {
        protected abstract void sell();}
```

4. Реализовать метод hashCode для класса

```
class Quest {
        private int a;
        private short b;}
```

5. Реализовать метод equals для класса

```
class Bus {
        protected String type;}
```

6. Дана строка вида String num = "1234"; преобразовать к int n четырьмя способами.

Часть 2

Использование классов и библиотек

Во второй части книги рассмотрены вопросы использования классов Java при работе со строками и файлами, хранения объектов, многопоточного и сетевого программирования с использованием классов из пакетов ***java.util***, ***java.net***, ***java.io*** *и др.*

Из-за ограниченности объема книги детальное рассмотрение библиотек классов невозможно. Подробное описание классов и методов можно найти в документации по языку, которой необходимо пользоваться каждому разработчику.

Глава 7

СТРОКИ

Строка — это застывшая структура данных, и повсюду, куда она передается, происходит значительное дублирование процесса. Это идеальное средство для сокрытия информации.

Алан Дж. Перлис

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. 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 (заключенная в двойные апострофы), является экземпляром 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.

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.getBytes(), "UTF-8")
```

можно установить кодировку создаваемому экземпляру в качестве второго параметра конструктора. Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект-литерал 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 = "oracle.com";
String s2 = new String("oracle.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 их регистра. Метод осуществляет вычитание кодов первых различных символов вызывающей и передаваемой строки 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**;

char[] getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) — извлечение символов строки в массив 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), String[] 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**.

Эффективной демонстрацией работы методов класса служит преобразование строки в массив объектов типа **String** и их сортировка в алфавитном порядке.

Ниже рассмотрена сортировка массива строк методом выбора, таким образом демонстрируются возможности методов класса.

// # 1 # сортировка # SortArray.java

```
package by.bsu.strings;
public class SortArray {
          public static void main(String[ ] args) {
             String names = "   Alena   Alice     alina albina     Anastasya   ALLA ArinA     ";
             names = names.trim(); // удаление пробелов по краям строки
             // разбиение строки на подстроки, где "пробел" – разделитель
             String a[ ] = names.split(" "); // массив содержит элементы нулевой длины
             for(int j = 0; j < a.length – 1; j++) {
                    for(int i = j + 1; i < a.length; i++) {
                       if(a[i].compareToIgnoreCase(a[j]) < 0) {
                              String temp = a[j];
                              a[j] = a[i];
                              a[i] = temp;
                       }
                    }
             }
             for (String arg : a) {
                    if (!arg.isEmpty()) {
                              System.out.print(arg + " ");
                    }
             }
          }
}
```

albina Alena Alice alina ALLA Anastasya ArinA

Вызов метода **trim()** обеспечивает удаление всех начальных и конечных символов пробелов. Метод **compareToIgnoreCase()** выполняет лексикографическое сравнение строк между собой по правилам Unicode. Оператор **if(!arg.isEmpty())** не позволяет выводить пустые строки.

При использовании методов класса **String**, изменяющих строку, создается новый обновленный объект класса **String**. Сохранить произведенные изменения экземпляра класса **String** можно только с применением оператора присваивания, т. е. установкой ссылки на вновь созданный объект. В следующем примере будет подтвержден тезис о неизменяемости экземпляра типа **String**.

/ # 2 # передача строки по ссылке # RefString.java */*

```
package by.bsu.strings;
public class RefString {
          public static void changeStr(String s) {
                    s = s.concat(" Certified"); // создается новая строка
                    // или  s.concat(" Certified");
                    // или s += " Certified";
```

```
        }
        public static void main(String[ ] args) {
                String str = new String("Java");
                changeStr(str);
                System.out.print(str);
        }
}
```

В результате будет выведена строка:

Java

Поскольку объект был передан по ссылке, любое изменение объекта в методе должно сохраняться и для исходного объекта, так как обе ссылки равноправны. Это не происходит по той причине, что вызов метода **concat(String s)** приводит к созданию нового объекта.

При создании экземпляра класса **String** путем присваивания его ссылки на литерал, последний помещается в так называемый «пул литералов». Если в дальнейшем будет создана еще одна ссылка на литерал, эквивалентный ранее объявленному, то будет произведена попытка добавления его в «пул литералов». Так как идентичный литерал там уже существует, то дубликат не может быть размещен, и вторая ссылка будет на существующий литерал. Аналогично в случае, если литерал является вычисляемым. То есть компилятор воспринимает литералы **"Java"** и **"J" + "ava"** как эквивалентные.

// # 3 # сравнение ссылок и обьектов # EqualStrings.java

```
package by.bsu.strings;
public class EqualStrings {
        public static void main(String[ ] args) {
                String s1 = "Java";
                String s2 = "Java";
                String s3 = new String("Java");
                String s4 = new String(s1);
                System.out.println(s1 + "==" + s2 + " : " + (s1 == s2)); // true
                System.out.println(s3 + "==" + s4 + " : " + (s3 == s4)); // false
                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
        }
}
```

В результате, например, будет выведено:

Java==Java : true
Java==Java : false
Java==Java : false
Java equals Java : true
Java equals Java : true

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

Так как 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, т. е. 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()**.

// # 4 # занесение в пул литералов # DemoIntern.java

```
package by.bsu.strings;
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 в «пуле литералов» 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.

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 отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего. Более высокая скорость обработки есть следствие отсутствия потокобезопасности класса **StringBuilder**. Его 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.

/ # 5 # свойства объекта StringBuffer # DemoStringBuffer.java */*

```
package by.bsu.strings;
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());

                sb.append("Internationalization");
                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
строка —> JavaInternationalization
длина —> 24
размер —> 34
реверс —> noitazilanoitanretnIavaJ

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 емкость объекта автоматически увеличивается, оставляя при этом некоторый резерв для дальнейших изменений. Методом **reverse()** можно быстро изменить порядок символов в объекте.

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

/ # 6 # изменение объекта StringBuilder # RefStringBuilder.java */*

```
package by.bsu.strings;
public class RefStringBuilder {
          public static void changeStr(StringBuilder s) {
                    s.append(" Certified");
          }
          public static void main(String[ ] args) {
                    StringBuilder str = new StringBuilder("Oracle");
                    changeStr(str);
                    System.out.println(str);
          }
}
```

В результате выполнения этого кода будет выведена строка:

Oracle Certified

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

Для классов **StringBuffer** и **StringBuilder** не переопределены методы **equals()** и **hashCode()**, т. е. сравнить содержимое двух объектов невозможно, следовательно хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**. При идентичном содержимом у двух экземпляров, размеры буфера каждого могут отличаться, поэтому сравнение на эквивалентность объектов представляется неоднозначным.

/ # 7 # сравнение объектов StringBuffer и их хэш-кодов # EqualsStringBuffer.java */*

```
package by.bsu.strings;
public class EqualsStringBuffer {
```

```
        public static void main(String[ ] args) {
                StringBuffer sb1 = new StringBuffer();
                StringBuffer sb2 = new StringBuffer(48);
                sb1.append("Java");
                sb2.append("Java");
                System.out.print(sb1.equals(sb2));
                System.out.print(sb1.hashCode() == sb2.hashCode());
        }
}
```

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

Сравнить содержимое можно следующим образом:

```
sb1.toString().contentEquals(sb2)
```

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 некоторые логические конструкции для задания шаблона.

Если необходимо, чтобы в строке, проверяемой на соответствие, 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]]	между **a** и **d**, или между **m** и **p**

Кроме стандартных классов символов существуют предопределенные классы символов:

.	любой символ
\d или \p{Digit}	**[0-9]**
\D	**[^0-9]**
\s или \p{Space}	**[\t\n\x0B\f\r]**
\S	**[^\s]**
\w	**[0-9_A-Za-z]**
\W	**[^\w]**

\p{Lower}	[**a-z**]
\p{Upper}	[**A-Z**]
\p{Punkt}	!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Blank}	Пробел или табуляция

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**;

boolean matches(String regex, CharSequence input) — проверяет на соответствие строки **input** шаблону **regex**;

String[] split(CharSequence input) — разбивает строку **input**, учитывая, что разделителем является шаблон;

Matcher matcher(CharSequence input) — возвращает **Matcher**, с помощью которого можно находить соответствия в строке **input**.

С помощью метода **matches()** класса **Pattern** можно проверять на соответствие шаблону целую строку, 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 информация шаблону;

boolean lookingAt() — поиск последовательности символов, начинающейся с начала строки и соответствующей шаблону;

boolean find() или **boolean find(int start)** — ищет последовательность символов, соответствующих шаблону, в любом месте строки. Параметр **start** указывает на начальную позицию поиска.

Иногда необходимо сбросить состояние экземпляра **Matcher** 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)**.

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.

String group() — возвращает всю подпоследовательность, удовлетворяющую шаблону;

int start() — возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону;

int start(int group) — возвращает индекс первого символа указанной группы;

int end() — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону;

int end(int group) — возвращает индекс последнего символа указанной группы;

String group(int group) — возвращает конкретную группу по позиции;

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.

/ # 8 # обработка строк с помощью шаблонов # DemoRegular.java */*

```
package by.bsu.regex;
import java.util.regex.Pattern;
```

```
import java.util.regex.Matcher;
import java.util.Arrays;
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{6,})@(\\w+\\.)([a-z]{2,4})";
                    String s = "адреса эл.почты:blinov@gmail.com, romanchik@bsu.by!";
                    Pattern p2 = Pattern.compile(regex);
                    Matcher m2 = p2.matcher(s);
                    while (m2.find()) {
                              System.out.println("e-mail: " + m2.group());
                    }
// разбиение строки на подстроки с применением шаблона в качестве разделителя
                    Pattern p3 = Pattern.compile("\\d+\\s?");
                    String[ ] words = p3.split("java5tiger 77 java6mustang");
                    System.out.print(Arrays.toString(words));
          }
}
```

В результате будет выведено:

true
e-mail: blinov@gmail.com
e-mail: romanchik@bsu.by
[java, tiger, java, mustang]

Использование групп, а также собственных и неполных квантификаторов.

/ # 9 # группы и квантификаторы # Groups.java */*

```
package by.bsu.regex;
public class Groups {
          public static void main(String[ ] args) {
                    String input = "abdcxyz";
                              simpleMatches ("([a-z]*)([a-z]+)", input);
                              simpleMatches ("([a-z]?)([a-z]+)", input);
                              simpleMatches ("([a-z]+)([a-z]*)", input);
                              simpleMatches ("([a-z]?)([a-z]?)", input);
          }
          public static void simpleMatches(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)+ "\n");
```

```
                } else {
                    System.out.println("nothing\n");
                }
        }
}
```

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.

Интернационализация приложения

Класс **java.util.Locale** позволяет учесть особенности региональных представлений алфавита, символов, чисел, дат 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 всех остальных объект **Locale** нужно создавать с помощью конструктора, например:

```
Locale rus = new Locale("ru", "RU");
```

Определить текущий вариант региональных параметров можно следующим образом:

```
Locale current = Locale.getDefault();
```

А можно и изменить для текущего экземпляра (instance) JVM:

```
Locale.setDefault(Locale.CANADA);
```

Если, например, в ОС установлен регион «Беларусь» или в приложении с помощью **new Locale("be", "BY")**, 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:

BY
Беларусь
be
белорусский

Для создания приложений, поддерживающих несколько языков, существует целый ряд решений. Самое логичное из них — дублирование сообщений на разных языках в разных файлах с эквивалентными ключами с последующим извлечением информации на основе значения заданной локали. Данное решение основано на взаимодействии классов **java.util.ResourceBundle** и **Locale**. Класс **ResourceBundle** предназначен для взаимодействия с текстовыми файлами свойств (расширение.**properties**). Каждый объект **ResourceBundle** представляет собой набор 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_it_CH.properties** соответствует объекту **Locale**, заданному кодом итальянского языка (**it**) и кодом страны Швейцарии (**CH**).

```
text.properties
text_ru_RU.properties
text_it_CH.properties
text_fr_CA.properties
```

Чтобы выбрать определенный объект **ResourceBundle**, следует вызвать один из статических перегруженных методов **getBundle(***параметры***)**. Следующий фрагмент выбирает **text** объекта **ResourceBundle** для объекта **Locale**, который соответствует французскому языку и стране Канада.

```
Locale locale = new Locale("fr", "CA");
ResourceBundle rb = ResourceBundle.getBundle("text", locale);
```

Если объект **ResourceBundle** для заданного объекта **Locale** не существует, то метод **getBundle()** извлечет наиболее общий. Если общее определение файла ресурсов не задано, то метод **getBundle()** генерирует исключительную ситуацию **MissingResourceException**. Чтобы это не произошло, необходимо обеспечить

наличие базового файла ресурсов без суффиксов, а именно: **text.properties** в дополнение к частным случаям вида:

```
text_en_US.properties
text_ru_RU.properties
```

В файлах свойств информация должна быть организована по принципу:

```
#Комментарий
group1.key1 = value1
group1.key2 = value2
group2.key1 = value3…
```

Например:

```
label.button = submit
label.field = login
message.welcome = Welcome!
```

или

```
label.button = принять
label.field = логин
message.welcome = Добро пожаловать!
```

В классе **ResourceBundle** определен ряд полезных методов, в том числе метод **getKeys(),** возвращающий объект **Enumeration**, который применяется для последовательного обращения к элементам. Множество **Set<String>** всех ключей — методом **keySet()**. Конкретное значение по конкретному ключу извлекается методом **getString(String key)**. Отсутствие запрашиваемого ключа приводит к генерации исключения. Проверить наличие ключа в файле можно методом **boolean containsKey(String key)**. Методы **getObject(String key)** и **getStringArray(String key)** извлекают соответственно объект и массив строк по передаваемому ключу.

В следующем примере в зависимости от выбора пользователя известная фраза будет выведена на одном из трех языков.

// # 10 # поддержка различных языков # HamletInternational.java

```
package by.bsu.resource;
import java.io.IOException;
import java.util.Locale;
import java.util.ResourceBundle;
public class HamletInternational {
          public static void main(String[ ] args) {
              for (int i = 0; i < 3: i++) {
                    System.out.println("1 – английский /n 2 – белорусский \n любой – русский ");
                    char i = 0;
                    try {
                              i = (char) System.in.read();
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
                String country = "";
                String language = "";
                switch (i) {
                    case '1':
                          country = "US";
                          language = "EN";
                          break;
                    case '2':
                          country = "BY";
                          language = "BE";
                          break;
                }
                Locale current = new Locale(language, country);
                ResourceBundle rb = ResourceBundle.getBundle("property.text", current);
                          String s1 = rb.getString("str1");
                          System.out.println(s1);

                          String s2 = rb.getString("str2");
                          System.out.println(s2);
                    }
        }
}
```

Все файлы следует разместить в каталоге **property** в корне проекта на одном уровне с пакетами приложения.

Файл **text_en_US.properties** содержит следующую информацию:

str1 = To be or not to be?
str2 = This is a question.

Файл **text_be_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.

Если в результате выполнения приложения на консоль результаты выдаются в нечитаемом виде, то следует изменить кодировку файла или символов.

Для взаимодействия с **properties**-файлами можно создать специальный класс, экземпляр которого позволит не только извлекать информацию по ключу, но и изменять значение локали, что сделает его удобным для использования при интернационализации приложений.

// # 11 # менеджер ресурсов # ResourceManager.java

```
package by.bsu.resource;
import java.util.Locale;
import java.util.ResourceBundle;
public enum ResourceManager {
          INSTANCE;
          private ResourceBundle resourceBundle;
          private final String resourceName = "property.text";
          private ResourceManager() {
                 resourceBundle = ResourceBundle.getBundle(resourceName, Locale.getDefault());
          }
          public void changeResource(Locale locale) {
                 resourceBundle = ResourceBundle.getBundle(resourceName, locale);
          }
          public String getString(String key) {
                 return resourceBundle.getString(key);
          }
}
```

Экземпляр класса может быть создан только один, и все приложение пользуется его возможностями.

/ # 12 # извлечение информации из файла ресурсов и смена локали # ResourceManagerRun.java */*

```
package by.bsu.resource;
import java.util.Locale;
public class ResourceManagerRun {
public static void main(String[ ] args) {
                    ResourceManager manager = ResourceManager.INSTANCE;
                    System.out.println(manager.getString("str1"));

                    manager.changeResource(new Locale("be","BY"));
                    System.out.println(manager.getString("str1"));
          }
}
```

Качественно разработанное приложение обычно не содержит литералов типа **String**. Все необходимые сообщения хранятся вне системы, в частности, в **properties** файлах. Что позволяет без перекомпиляции кода безболезненно изменять любое сообщение или информацию, хранящуюся вне классов системы.

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.

// # 13 # региональные представления чисел # DemoNumberFormat.java

```
package by.bsu.num;
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 = "1.234,5"; // 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) {
                              System.err.print("Error position: " + e.getErrorOffset());
                    }
          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("\nUs " + sUs + "\nFr " + sFr);
        }
}
```

Результат работы программы:

iGe = 1234,500000
iUs = 1,234000
iFr = 1,000000
Us 1,234.5
Fr 1 234,5

Аналогично выполняются переходы от одного регионального стандарта к другому при отображении денежных сумм с добавлением символа валюты.

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 s = df.format(new Date());
```

Метод **Date parse(String source)** преобразовывает переданную в виде строки дату в объектное представление конкретного регионального формата, например:

```
String str = "April 9, 2012";
Date d = df.parse(str);
```

Класс **DateFormat** содержит большое количество методов, позволяющих выполнять разнообразные манипуляции с датой и временем.

В качестве примера рассмотрено преобразование заданной даты в различные региональные форматы.

// # 14 # региональные представления дат # DemoDateFormat.java

```
package by.bsu.dates;
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 9, 2012";
                       try {
                              d = df.parse(str);
                              System.out.println(d);
                       } catch (ParseException e) {
                              System.err.print("Error position: " + e.getErrorOffset());
                       }
                       df = DateFormat.getDateInstance(DateFormat.LONG, new Locale("ru","RU"));
                       System.out.println(df.format(d));

                       df = DateFormat.getDateInstance(DateFormat.FULL,Locale.GERMAN);
                       System.out.println(df.format(d));
}
```

Результат работы программы:

Mon Apr 9 00:00:00 EEST 2012
9 Апрель 2012 г.
Montag, 9. April 2012

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 будет выведена пара сотен строк, каждая из которых представляет текущую дату в соответствии с региональным стандартом, выводимым перед датой с помощью инструкции **loc.toString()**.

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 C.

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(OutputStream outStrm)
Formatter(PrintStream printStrm)
```

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.

Некоторые методы класса:

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. Существует перегруженная версия метода без использования локализации;

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 **FormatterClosedException**.

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 **%**

Также возможны спецификаторы с заглавными буквами: **%A** (эквивалентно **%a**). Форматирование с их помощью обеспечивает перевод символов в верхний регистр.

/ # 15 # форматирование строки при помощи метода format() # SimpleFormatString.java */*

```
package by.bsu.format;
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", '8');
                    System.out.print(f);
          }
}
```

В результате выполнения этого кода будет выведено:

This book is about
JAVA 8

/ # 16 # форматирование чисел с использованием спецификаторов %x, %o, %a, %g # FormatterDemoNumber.java */*

```
package by.bsu.format;
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"**

/ # 17 # форматирование даты и времени # FormatterDemoTimeAndDate.java */*

```
package by.bsu.format;
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 использования флагов форматирования.

/ # 18 # применение флагов форматирования # FormatterDemoFlags.java */*

```
package by.bsu.format;
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
```

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 тексте каждую букву заменить ее порядковым номером в алфавите. При выводе в одной строке печатать текст с двумя пробелами между буквами, в следующей строке внизу под каждой буквой печатать ее номер.
3. В тексте после буквы Р, если она не последняя в слове, ошибочно напечатана буква А вместо О. Внести исправления в текст.
4. В тексте после *k*-го символа вставить заданную подстроку.
5. После **каждого** слова текста, заканчивающегося заданной подстрокой, вставить указанное слово.
6. 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.
7. Из текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.
8. Удалить из текста его часть, заключенную между двумя символами, которые вводятся (например, между скобками «(» и «)» или между звездочками «*» и т. п.).
9. Определить, сколько раз повторяется в тексте каждое слово, которое встречается в нем.

10. В тексте найти и напечатать *n* символов (и их количество), встречающихся наиболее часто.
11. Найти, каких букв, гласных или согласных, больше в каждом предложении текста.
12. В стихотворении найти количество слов, начинающихся и заканчивающихся гласной буквой.
13. Напечатать без повторения слова текста, у которых первая и последняя буквы совпадают.
14. В тексте найти и напечатать все слова максимальной и все слова минимальной длины.
15. Напечатать квитанцию об оплате телеграммы, если стоимость одного слова задана.
16. В стихотворении найти одинаковые буквы, которые встречаются во всех словах.
17. В тексте найти первую подстроку максимальной длины, не содержащую букв.
18. В тексте определить все согласные буквы, встречающиеся не более чем в двух словах.
19. Преобразовать текст так, чтобы каждое слово, не содержащее неалфавитных символов, начиналось с заглавной буквы.
20. Подсчитать количество содержащихся в данном тексте знаков препинания.
21. В заданном тексте найти сумму всех встречающихся цифр.
22. Из кода Java удалить все комментарии (//, /*, /**).
23. Все слова текста встречаются четное количество раз, за исключением одного. Определить это слово. При сравнении слов регистр не учитывать.
24. Определить сумму всех целых чисел, встречающихся в заданном тексте.
25. Из текста удалить все лишние пробелы, если они разделяют два различных знака препинания и если рядом с ними находится еще один пробел.
26. Строка состоит из упорядоченных чисел от 0 до 100000, записанных подряд без пробелов. Определить, что будет подстрокой от позиции *n* до *m*.
27. Определить количество вхождений заданного слова в текст, игнорируя регистр символов и считая буквы «е», «ё», и «и», «й» одинаковыми.
28. Преобразовать текст так, чтобы только первые буквы каждого предложения были заглавными.
29. Заменить все одинаковые рядом стоящие в тексте символы одним символом.
30. Вывести в заданном тексте все слова, расположив их в алфавитном порядке.
31. Подсчитать, сколько слов в заданном тексте начинается с прописной буквы.
32. Подсчитать, сколько раз заданное слово входит в текст.

Вариант В

Создать программу обработки текста учебника по программированию с использованием классов: *Символ*, *Слово*, *Предложение*, *Абзац*, *Лексема*, *Листинг*, *Знак препинания* и др. Во всех задачах с формированием текста заменять табуляции и последовательности пробелов одним пробелом.

Предварительно текст следует разобрать на составные части, выполнить одно из перечисленных ниже заданий и вывести полученный результат.

1. Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
2. Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.
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. Напечатать слова текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.
7. Рассортировать слова текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).
8. Слова текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по первой согласной букве слова.
9. Все слова текста рассортировать по возрастанию количества заданной буквы в слове. Слова с одинаковым количеством расположить в алфавитном порядке.
10. Существует текст и список слов. Для каждого слова из заданного списка найти, сколько раз оно встречается в каждом предложении, и рассортировать слова по убыванию общего количества вхождений.
11. В каждом предложении текста исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.
12. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
13. Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства — по алфавиту.
14. В заданном тексте найти подстроку максимальной длины, являющуюся палиндромом, т. е. читающуюся слева направо и справа налево одинаково.
15. Преобразовать каждое слово в тексте, удалив из него все следующие (предыдущие) вхождения первой (последней) буквы этого слова.
16. В некотором предложении текста слова заданной длины заменить указанной подстрокой, длина которой может не совпадать с длиной слова.

Вариант С

1. 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° 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.

2. Исключить из текста подстроку максимальной длины, начинающуюся и заканчивающуюся одним и тем же символом.
3. Вычеркнуть из текста минимальное количество предложений так, чтобы у любых двух оставшихся предложений было хотя бы одно общее слово.
4. 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.
5. Распаковать текст, сжатый по правилу из предыдущего задания.
6. Определить, удовлетворяет ли имя файла маске. Маска может содержать символы «?» (произвольный символ) и «*» (произвольное количество произвольных символов).
7. 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.
8. 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.
9. Изменить программу из предыдущего примера так, чтобы она осуществляла форматирование с выравниванием по обоим краям. Для этого добавить дополнительные пробелы между словами.
10. 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.
11. Пусть текст содержит миллион символов и необходимо сформировать из них строку путем конкатенации. Определить время работы кода. Ускорить процесс, используя класс StringBuilder.
12. 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.

13. Восстановить слово по его преобразованию Барроуза — Уиллера. К примеру, получив на вход VJAA, в результате работы программа должна выдать слово JAVA.
14. В Java код добавить корректные getter и setter-методы для всех полей данного класса при их отсутствии.
15. В тексте нет слов, начинающихся одинаковыми буквами. Напечатать слова текста в таком порядке, чтобы последняя буква каждого слова совпадала с первой буквой следующего слова. Если все слова нельзя напечатать в таком порядке, найти такую цепочку, состоящую из наибольшего количества слов.
16. 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.
17. 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.
18. Все слова текста рассортировать в порядке убывания их длин, при этом все слова одинаковой длины рассортировать в порядке возрастания в них количества гласных букв.

Тестовые задания к главе 7

Вопрос 7.1.

Дан код:

```
String s1 = "Minsk";
String s2 = new String("Minsk");
if(s1.equals(s2.intern())){
        System.out.print("true");
} else {
        System.out.print("false");
}
if(s1 == s2){
        System.out.print("true");
```

```
} else{
        System.out.print("false");
}
```

В результате при компиляции и запуске будет выведено (1):

1) truefalse
2) falsetrue
3) truetrue
4) falsefalse
5) ошибка компиляции: заданы некорректные параметры для метода equals()

Вопрос 7.2.

Дан код:

```
StringBuilder sb1 = new StringBuilder("I like Java.");//1
StringBuilder sb2 = new StringBuilder(sb1);//2
if (sb1.equals(sb2)){
        System.out.println("true");
} else {
        System.out.println("false");
}
```

В результате при компиляции и запуске будет выведено (1):

1) true
2) false
3) ошибка компиляции в строке 1
4) ошибка компиляции в строке 2

Вопрос 7.3.

Что будет результатом компиляции и выполнения следующего кода?

```
Pattern p = Pattern.compile("(1*)0");
Matcher m = p.matcher("111110");
System.out.println(m.group(1));
```

1) вывод на консоль строки «111110»;
2) вывод на консоль строки «11111»;
3) ошибка компиляции;
4) ошибка выполнения.

Вопрос 7.4.

Что выведется на консоль при компиляции и выполнении следующих строчек кода (1)?

```
Locale loc = new Locale("ru", "RU");
System.out.println(loc.getDisplayCountry(Locale.US));
```

1) United States;
2) Соединенные Штаты;
3) Россия;
4) Russia;
5) ошибка компиляции.

Вопрос 7.5.

Укажите, какому пакету принадлежат классы, позволяющие форматировать числа и даты: NumberFormat и DateFormat.

1) java.text
2) java.util.text
3) java.util
4) java.lang

Глава 8

ИСКЛЮЧЕНИЯ И ОШИБКИ

Существуют только ошибки.

Аксиома Робертса

Что для одного ошибка, для другого — исходные данные.

Следствие Бермана из аксиомы Робертса

Никогда не выявляйте в программе ошибки, если не знаете, что с ними делать.

Руководство Штейнбаха

Иерархия исключений и ошибок

Исключительные ситуации (исключения) и ошибки возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте и невозможно продолжение работы программы. Примерами «популярных» ошибок являются: попытка индексации вне границ массива, вызов метода на нулевой ссылке или деление на нуль. При возникновении исключения в приложении 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 обязан включить в код метода обработку исключений, которые могут генерироваться в этом методе, или передать обработку исключения на более высокий уровень методу, вызвавшему данный метод.

Исключение не должно восприниматься как нечто вредное, от которого следует избавиться любой ценой. Исключение — это источник дополнительной информации о ходе выполнения приложения. Такая информация позволяет лучше адаптировать код к конкретным условиям его использования, а также на ранней стадии выявить ошибки или защититься от их возникновения в будущем. В противном случае «подавление» исключений приведет к тому, что

о возникшей ошибке никто не узнает или узнает на стадии некорректно обработанной информации. Поиск места возникновения может быть затруднительным.

Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при исключительной ситуации. 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**.

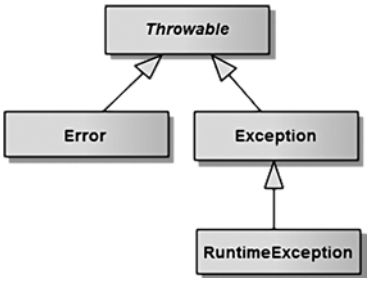

Рис. 8.1. *Иерархия основных классов исключений*

Исключительные ситуации типа **Error** возникают только во время выполнения программы. Такие исключения связаны с серьезными ошибками, к примеру, с переполнением стека, не подлежат исправлению и не могут обрабатываться приложением. Некоторые классы из иерархии наследуемых от класса **Error** приведены на рис. 8.2.

Ниже приведена иерархия классов проверяемых исключений, наследуемых от класса **Exception** при отсутствии в цепочке наследования класса **RuntimeException**. Возможность возникновения проверяемого исключения может быть отслежена еще на этапе компиляции кода. Компилятор проверяет, может ли данный метод генерировать или обрабатывать исключение.

Проверяемые исключения должны быть обработаны в методе, который может их генерировать, или включены в **throws**-список метода для дальнейшей обработки в вызывающих методах.

Во время выполнения могут генерироваться также исключения, которые могут быть обработаны без ущерба для выполнения программы. Список этих исключений приведен на рис. 8.4. В отличие от проверяемых исключений, класс **RuntimeException** и порожденные от него классы относятся к непроверяемым исключениям. Компилятор не проверяет, может ли генерировать и/или обрабатывать метод эти исключения. Исключения типа **RuntimeException** генерируются при возникновении ошибок во время выполнения приложения.

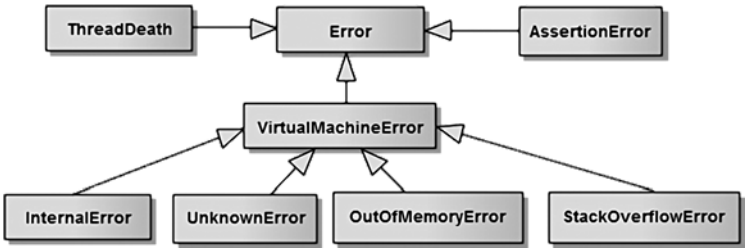

Рис. 8.2. *Некоторые классы исключений, наследуемые от класса Error*

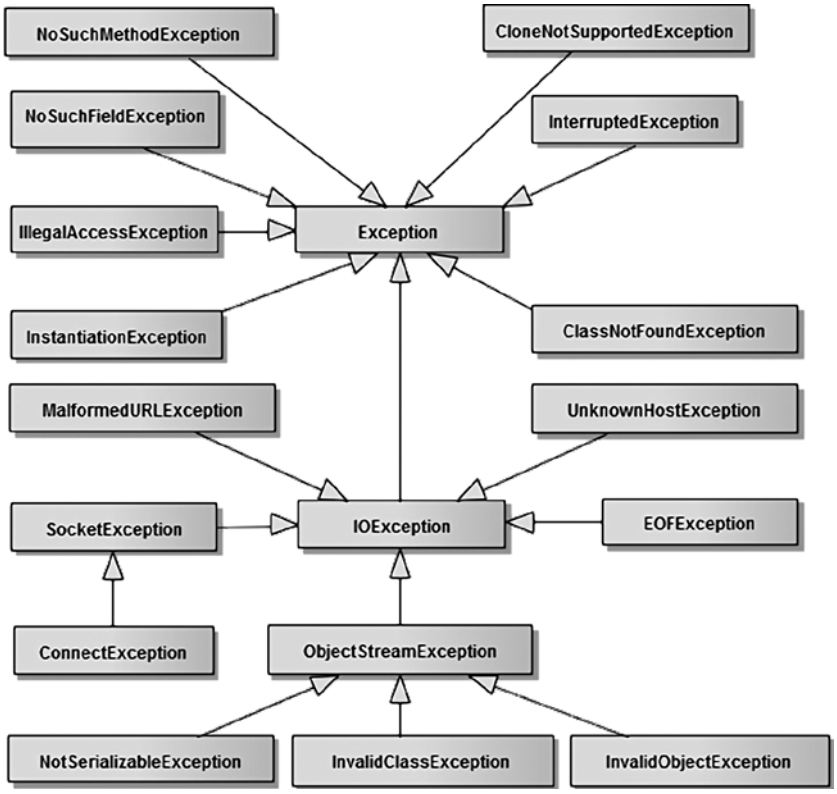

Рис. 8.3. *Иерархия классов проверяемых (checked) исключительных ситуаций*

Почему возникла необходимость деления исключений на проверяемые и непроверяемые? Представим, что следующие ситуации проверяются на этапе компиляции, а именно:

- деление в целочисленных типах вида **a/b** при **b=0** генерирует исключение **ArithmeticException**;
- индексация массивов. Выход за пределы массива приводит к исключению **ArrayIndexOfBoundException**;
- вызов метода на ссылке вида **obj.toString()**, если **obj** ссылается на **null**.

Если бы возможность появления перечисленных исключений проверялась на этапе компиляции, то любая попытка индексации массива или каждый вызов метода требовали бы или блока **try-catch**, или секции **throws**. Такой код был бы практически непригоден для понимания и поддержки, поэтому часть исключений была выделена в группу непроверяемых и ответственность за защиту приложения от последствий их возникновения возложена на программиста.

Ниже приведен список часто встречаемых в практике программирования непроверяемых исключений, знание причин возникновения которых необходимо при создании качественного кода.

Исключение	Значение
ArithmeticException	Арифметическая ошибка: деление на нуль и др.
ArrayIndexOutOfBoundsException	Индекс массива находится вне границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
ConcurrentModificationException	Некорректная модификация коллекции
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора на разблокированном экземпляре
IllegalStateException	Среда или приложение находятся в некорректном состоянии
IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексации вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

Рис. 8.4. *Классы непроверяемых исключений, наследуемых от класса RuntimeException*

Способы обработки исключений

Если при возникновении исключения в текущем методе обработчик не будет обнаружен, то его поиск будет продолжен в методе, вызвавшем данный метод, и так далее вплоть до метода **main()** для консольных приложений или другого метода, запускающего соответствующий вид приложения. Если же и там исключение не будет перехвачено, то JVM выполнит аварийную остановку приложения с вызовом метода **printStackTrace()**, выдающего данные трассировки.

Для проверяемого исключения возможность его генерации отслеживается. Передача обработки вызывающему методу осуществляется с помощью оператора **throws**. В конце концов исключение будет передано в метод **main()**, где и должна находиться крайняя точка обработки. Добавлять оператор **throws** методу **main()** представляется дурным тоном программирования, как безответственное действие программиста, не обращающего никакого внимания на альтернативное выполнение программы.

На практике 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 (в первую очередь для проверяемых исключений);
- использование собственных исключений.

Первый подход можно рассмотреть на следующем примере. При преобразовании содержимого строки к числу в определенных ситуациях может возникать проверяемое исключение типа **ParseException**. Например:

```
public double parseFromFrance(String numberStr) {
          NumberFormat nfFr = NumberFormat.getInstance(Locale.FRANCE);
          try {
                    double numFr = nfFr.parse(numberStr).doubleValue();
                    return numFr;
          } catch (ParseException e) { // проверяемое исключение
// 1. генерация стандартного исключения, н-р: IllegalArgumentException() – не очень хорошо
// 2. генерация собственного исключения
// 3. return 0 или другого значения по умолчанию; – нежелательно
          }
}
```

Исключительная ситуация возникнет в случае, если переданная строка содержит нечисловые символы или не является числом. Генерируется объект исключения, и управление передается соответствующему блоку **catch**, в котором он обрабатывается, 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 вызывающий метод мог защитить себя от этих исключений. В вызывающем методе должна быть предусмотрена или обработка этих исключений, или последующая передача вызывающему методу.

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

```
public double parseFromFrance(String numberStr) throws ParseException {
          NumberFormat nfFr = NumberFormat.getInstance(Locale.FRANCE);
          double numFr = nfFr.parse(numberStr).doubleValue();
          return numFr;
}
```

Ключевое слово **throws** позволяет разобраться с исключениями методов «чужих» классов, код которых отсутствует. Обрабатывать исключение при этом должен будет метод, вызывающий **parseFromFrance()**:

```
public void doAction() {
              // same code here
          try {
                    parseFromFrance(numberStr);
          } catch (ParseException e) {
                    // обработка
          }
}
```

Создание и применение собственных исключений будет рассмотрено позже в этой главе.

Обработка нескольких исключений

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.

/ # 1 # обработка двух типов исключений # TwoExceptionAction.java */*

```
package by.bsu.exception;
public class TwoExceptionAction {
          public void doAction() {
                    try {
                              int a = (int)(Math.random() * 2);
                              System.out.println("a = " + a);
                              int c[] = { 1/a }; // опасное место #1
                              c[a] = 71; // опасное место #2
                    } catch(ArithmeticException e) {
                              System.err.println("деление на 0" + e);
                    } catch(ArrayIndexOutOfBoundsException e) {
                              System.err.println("out of bound: " + e);
                    } // окончание try-catch блока
                    System.out.println("after 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 допустимые 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(IllegalArgumentException e) {
} catch(PatternSyntaxException e) { } /* никогда не может быть вызван: ошибка компиляции */
```

где класс **PatternSyntaxException** представляет собой подкласс класса **IllegalArgumentException**. Корректно будет просто поменять местами блоки **catch**:

```
try { /* код, который может вызвать исключение */
} catch(PatternSyntaxException e) {
} catch(IllegalArgumentException e) {
}
```

На практике иногда возникают ситуации, когда инструкций **catch** несколько и обработка производится идентичная, например, вывод сообщения об исключении в журнал.

```
try {
          // some operations
} catch(NumberFormatException e) {
           e.printStackTrace();
} catch(ClassNotFoundException e) {
           e.printStackTrace();
} catch(InstantiationException e) {
           e.printStackTrace();
}
```

В версии Java 7 появилась возможность объединить все идентичные инструкции в одну, используя для разделения оператор «|».

```
try {
          // some operations
} catch(NumberFormatException | ClassNotFoundException | InstantiationException e) {
           e.printStackTrace();
}
```

Такая запись позволяет избавиться от дублирования кода.

Введено понятие более точной переброски исключений (more precise rethrow). Это решение применимо в случае, если обработка возникающих исключений не предусматривается в методе и должна быть передана вызывающему данный метод методу.

До введения этого понятия код выглядел так:

```
public double parseFromFileBefore(String filename)
                              throws FileNotFoundException, ParseException, IOException {
                    NumberFormat nfFr = NumberFormat.getInstance(Locale.FRANCE);
                    double numFr = 0;
                    BufferedReader buff = null;
                    try {
                              FileReader fr = new FileReader(filename);
                              buff = new BufferedReader(fr);
                              String number = buff.readLine();
                              numFr = nfFr.parse(number).doubleValue();
                    } catch (FileNotFoundException e) {
                              // запись лога
```

```
                    throw e;
                } catch (IOException e) {
                    // запись лога
                    throw e;
                } catch (ParseException e) {
                    // запись лога
                    throw e;
                } finally {
                    if(buff != null) {
                        buff.close();
                    }
                }
        return numFr;
}
```

More precise rethrow разрешает записать в единственную инструкцию **catch** более общее исключение, чем может быть генерировано в инструкции **try**, с последующей генерацией перехваченного исключения для его передачи в вызывающий метод.

```
public double parseFromFile(String filename)
                    throws FileNotFoundException, ParseException, IOException {
                NumberFormat nfFr = NumberFormat.getInstance(Locale.FRANCE);
                double numFr = 0;
                BufferedReader buff = null;
                try {
                    FileReader fr = new FileReader(filename);
                    buff = new BufferedReader(fr);
                    String number = buff.readLine();
                    numFr = nfFr.parse(number).doubleValue();
                } catch (final Exception e) { // final – необязателен
                    // запись лога
                    throw e; // more precise rethrow
                } finally {
                    if(buff != null) {
                        buff.close();
                    }
                }
        return numFr;
}
```

Наличие секции **throws** контролируется компилятором на предмет точного указания списка проверяемых исключений, которые могут быть генерированы в блоке **try-catch**. При возможности возникновения непроверяемых исключений последние в секции **throws** обычно не указываются. Ключевое слово **final** не позволяет подменить экземпляр исключения для передачи за пределы метода. Однако данную конструкцию можно использовать и без **final**.

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**.

/ # 2 # вложенные блоки try-catch # NestedTryCatchRunner.java */*

```
package by.bsu.exception;
public class NestedTryCatchRunner {
          public void doAction() {
                    try { // внешний блок
                        int a = (int) (Math.random() * 2) – 1;
                        System.out.println("a = " + a);
                        try { // внутренний блок
                              int b = 1/a;
                              StringBuilder sb = new StringBuilder(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. Вкладывание блоков **try-catch** друг в друга загромождает код, поэтому такими конструкциями следует пользоваться с осторожностью.

Оператор throw

При разработке кода возникают ситуации, когда в приложении необходимо инициировать генерацию исключения для указания, например, на заведомо ошибочный результат выполнения операции, на некорректные значения параметра метода и др. Для генерации исключительной ситуации и создания экземпляра исключения используется оператор **throw**. В качестве исключения должен быть использован объект подкласса класса **Throwable**, 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 IIlegalArgumentException();
```

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.

В ситуации, когда получение методом достоверной информации критично для выполнения им своей функциональности, у программиста может возникнуть необходимость в генерации исключения, так как метод не может выполнить ожидаемых от него действий, основываясь на некорректных или ошибочных данных.

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

/ # 3 # генерация исключений # Connector.java # Runner.java # SameResource.java */*

```
package by.bsu.conn;
public class Connector {
    public static void loadResource(SameResource f) {
            if (f == null || !f.exists() || !f.isCreate()) {
                    throw new IllegalArgumentException(); /* генерация исключения */
                    // или собственное, н-р, throw new IllegalResourceException();
                    }
                    // more code
            }
}
package by.bsu.conn;
public class Runner {
    public static void main(String[ ] args) {
            SameResource f = new SameResource(); // SameResource f = null;
          try {// необязателен только при гарантированной корректности значения параметра
              Connector.loadResource(f);
          } catch(IllegalArgumentException e) {
              System.err.print("обработка unchecked-исключения вне метода: " + e);
          }
}
package by.bsu.conn;
public class SameResource {
            // поля, конструкторы
            public boolean isCreate() {
                    // more code
            }
            public boolean exists() {
                    // more code
            }
            public void execute() {
                    // more code
            }
            public void close() {
                    // more code
          }
}
```

Вызываемый метод **loadResource()** может (при отсутствии требуемого ресурса или при аргументе **null**) генерировать исключение, перехватываемое обработчиком. В результате экземпляр непроверяемого исключения **IllegalArgumentException** как подкласса класса **RuntimeException** передается обработчику исключений в методе **main()**.

В случае генерации проверяемого исключения **IllegalResourceException** компилятор требует обработки объекта исключения в методе или передачи его с помощью инструкции **throws**. Проверяемое исключение может быть создано как

```
public class IllegalResourceException extends Exception {}
```

Тогда методы **loadResource()** и **main()** будут выглядеть так:

```
public static void loadResource(Resource f) throws IllegalResourceException {
          if (f == null || !f.exists() || !f.isCreate()) {
                    throw new IllegalResourceException();
          }
                    // more code
}
```

В этом случае в методе **main()** блок **try-catch** будет обязателен.

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 может отсутствовать, так как играет только информационную роль.

Блок 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 обязательно выполняется после инструкций **try** или **catch**. Например:

```
try { /* код, который может вызвать исключение */
} catch(OneClassException e) { /* обработка исключения */  // необязателен
} catch(TwoClassException e) { /* обработка исключения */  // 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**.

/ # 4 # выполнение блоков finally # ResourceAction.java */*

```
package by.bsu.conn;
public class ResourceAction {
          public void doAction() {
                    SameResource sr = null;
                    try {
                        // реализация – захват ресурсов
                        sr = new SameResource(); // возможна генерация исключения
                        // реализация – использование ресурсов
                        sr.execute(); // возможна генерация исключения
                        // sr.close(); // освобождение ресурсов (некорректно)
                    } finally {
                        // освобождение ресурсов (корректно)
                        if (sr != null) {
                              sr.close();
                        }
                    }
                    System.out.print("after finally");
          }
}
```

В методе **doAction()** при использовании ресурсов и генерации исключения осуществляется преждевременный выход из блока **try** с игнорированием всего оставшегося в нем кода, но до выхода из метода обязательно будет выполнен раздел **finally**. Освобождение ресурсов в этом случае произойдет корректно. В следующей главе будет рассмотрен новый способ закрытия ресурсов autocloseable.

Собственные исключения

Для повышения качества и скорости восприятия кода разработчик может создать собственное исключение как подкласс класса **Exception** и затем использовать его при обработке ситуации, не являющейся исключением с точки зрения языка, но нарушающей логику вещей. По соглашению наследник любого класса-исключения должен заканчиваться словом **Exception**.

Например, возможность появления объекта типа **Coin** с отрицательным значением поля **diameter** является предлогом для генерации собственного логического исключения **CoinLogicException**, хотя для языка Java появление у объекта поля с отрицательным значением исключением не является и впоследствии к возникновению других исключений само по себе привести не может.

/ # 5 # метод, вызывающий исключение, созданное программистом # Coin.java */*

```
package by.bsu.fund.entity;
import by.bsu.fund.exceptions.CoinLogicException;
public class Coin {
          private double diameter;
          private double weight;
          public double getDiameter() {
                    return diameter;
          }
          public void setDiameter(double value) throws CoinLogicException {
                    if(value <= 0) {
                              throw new CoinLogicException("diameter is incorrect");
                    }
                    diameter = value;
          }
          public double getWeight() {
                    return weight;
          }
          public void setWeight(double value) {
                    weight = value;
          }
}
```

При невозможности присвоить значение генерируется экземпляр **CoinLogicException**, используемый в качестве собственного исключения.

/ # 6 # собственное «логическое» исключение # CoinLogicException.java */*

```
package by.bsu.fund.exceptions;
public class CoinLogicException extends Exception {
          public CoinLogicException() {
          }
          public CoinLogicException(String message, Throwable exception) {
                    super(message, exception);
          }
          public CoinLogicException(String message) {
                    super(message);
          }
          public CoinLogicException (Throwable exception) {
                    super(exception);
          }
}
```

Если же генерируется стандартное исключение или получены значения некоторых параметров — такие, что генерация какого-либо стандартного исключения становится неизбежной немедленно либо сразу по выходе из метода, то следует генерировать собственное техническое исключение **CoinTechnicalException**.

Продемонстрировать процесс генерации и обработки логического и технического собственных исключений можно на примере.

/ # 7 # генерация и обработка собственных исключений */*

```
public void doAction(String value) throws CoinTechnicalException {
                    Coin ob = new Coin();
                    try {
                              double d = Double.parseDouble(value);
                              ob.setDiameter(d);
                    } catch (NumberFormatException e) {
                              throw new CoinTechnicalException("incorrect symbol in string", e);
                    } catch (CoinLogicException e) {
                              System.err.println(e.getCause());
                    }
}
```

У класса-наследника **Exception** обычно определяются четыре конструктора, два из которых в качестве параметра принимают объект типа **Throwable**, что означает генерацию исключения на основе другого исключения. Такая ситуация в приведенном случае возможна, например, при предварительном определении диаметра монеты, в процессе которого произошла ошибка преобразования строки в базовый тип, спровоцировавшая возникновение **NumberFormatException**. Значение диаметра монеты присвоить все равно невозможно, поэтому есть смысл для более точной передачи причин некорректной работы приложения сгенерировать **CoinTechnicalException**, но с вызовом конструктора, обладающего параметром типа **Throwable**.

/ # 8 # собственное «техническое» исключение # CoinTechnicalException.java */*

```
package by.bsu.fund.exceptions;
public class CoinTechnicalException extends Exception {
          public CoinTechnicalException() {
          }
          public CoinTechnicalException(String message, Throwable cause) {
                    super(message, cause);
          }
          public CoinTechnicalException(String message) {
                    super(message);
          }
          public CoinTechnicalException(Throwable cause) {
                    super(cause);
          }
}
```

Один из них — сообщение, которое может быть выведено в поток ошибок; другой — реальное исключение, которое привело к вызову технического исключения. Этот код показывает, как можно сохранить дополнительную информацию внутри пользовательского исключения. Преимущество этого сохранения состоит в том, что если вызываемый метод захочет узнать реальную причину вызова **CoinTechnicalException**, он всего лишь должен вызвать метод

getCause(). Это позволяет вызываемому методу решить, нужно ли работать со специфичным исключением или достаточно обработки **CoinTechnicalException**.

Разработчики программного обеспечения стремятся к высокому уровню повторного использования кода, поэтому они постарались предусмотреть и закодировать все возможные исключительные ситуации. При реальном программировании создание собственных классов исключений позволяет разработчику выделить важные аспекты приложения и обратить внимание на детали разработки.

Приведенные выше классы собственных исключений могут иметь общий суперкласс, например: **CoinException**, что позволит всегда определить, является ли перехваченное исключение собственным или стандартным. Тогда предыдущий пример можно переписать в виде:

/ # 9 # генерация и переброска собственных исключений # */*

```
public void doAction(String value) throws CoinLogicException  {
                    Coin ob = new Coin();
                    try {
                              double d = Double.parseDouble(value);
                              ob.setDiameter(d);
                    } catch (CoinException e) {
                              throw e;
                    }
}
```

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 приложения, 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.

/ # 10 # полиморфизм и исключения # Stone.java # WhiteStone.java # BlackStone.java # StoneAction.java */*

```
package by.bsu.polymorph;
public class Stone { // ранее созданный класс
          public void build(String data) throws ParseException {
                    /* реализация */
          }
}
package by.bsu.polymorph;
public class WhiteStone extends Stone { // ранее созданный класс
          @Override
          public void build(String data) {
                    /* реализация */
                    System.out.println("белый каменный шар");
          }
}
package by.bsu.polymorph;
public class StoneAction { // ранее созданный класс
          public void buildHouse(Stone stone) {
                    try {
                              stone.build("some info");
                    // предусмотрена обработка ParseException и его подклассов
                    } catch(ParseException e) {
                              System.err.print(e);
                    }
          }
}
package by.bsu.polymorph;
public class BlackStone extends Stone { // новый класс
          @Override
          public void build(String data) throws Exception { // 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.

/ # 11 # конструкторы и исключения # Resource.java # ConcreteResource.java */*

```
package by.bsu.construction;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```
class Resource { // ранее созданный класс
          public Resource(String filename) throws FileNotFoundException {
                    // more code
          }
}
class ConcreteResource extends Resource { // ранее созданный класс
          // ранее созданный конструктор
          public ConcreteResource(String name) throws FileNotFoundException {
                    super(name);
                    // more code
          }
          // ранее созданный конструктор
          public ConcreteResource() throws IOException {
                    super("file.txt");
                    // more code
          }
          // новый конструктор
          public ConcreteResource(String name, int mode) { /* ошибка компиляции */
                    super(name);
                    // more code
          }
          public ConcreteResource(String name, int mode, String type) throws ParseException {
                                                              /* ошибка компиляции */
                    super(name);
                    // more code
          }

}
```

Если разрешить создание экземпляра в виде

```
ConcreteResource inCorrect = new ConcreteResource("info", 1);
```

то конструктор суперкласса может генерировать исключение и никаких предварительных действий по его предотвращению принято не будет.

```
try {
          ConcreteResource correct = new ConcreteResource();
} catch (IOException e) { // обработка
}
```

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 **ConcreteResource** не было бы никаких сообщений о возможности генерации исключения, и при возникновении исключительной ситуации ее источник было бы трудно идентифицировать.

Рекомендации по обработке исключений

В любом случае, если есть возможность не генерировать исключение, следует ею воспользоваться. Генерация исключения — процесс ресурсоемкий, и слишком частая генерация исключений оказывает влияние на быстродействие.

- Не обрабатывать конкретное исключение или несколько исключений с использованием в инструкции **catch** исключения более общего типа.

```
try {
      // some code here
      int a = Integer.parseInt(args[0]);
      StringBuilder sb = new StringBuilder(a);
      // some code here
} catch (Exception e) {
      System.err.println(e);
}
```

  Следует классифицировать исключения. Вместо этого следует использовать:

```
try {
      int a = Integer.parseInt(args[0]);
      StringBuilder sb = new StringBuilder(a);
} catch (NegativeArraySizeException e) {
      System.err.println("недопустимый размер буфера: " + e);
} catch (NumberFormatException e) {
      System.err.println("недопустимый символ в числе: " + e);
}
```

- Не оставлять пустыми блоки **catch**. При генерации и перехвате исключения никто не узнает, что исключительная ситуация имела место, и не станет устранять ее причины

```
try {
      // some code here
} catch (NumberFormatException e) {
}
```

- По возможности не использовать одинаковую обработку различных исключений

```
try {
      // some code here
} catch (IOException e) {
      e.printStackTrace();
} catch (SAXException e) {
      e.printStackTrace();
}
```

  Для замены можно воспользоваться конструкцией multi-catch из Java 7 или в каждую инструкцию **catch** помещать уникальную обработку.

- Не создавать класс исключений, эквивалентный по смыслу уже существующему. Прежде чем написать свое исключение, необходимо изучить документацию,

возможно, там найдется что-то подходящее. Например, вместо того, чтобы создавать исключение для информирования о некорректной работе с перечислением вида

```
public class EnumNotPresentException extends Exception { }
```

следует применить класс **EnumConstantNotPresentException**.

- Не создавать избыточное число классов собственных исключений. Прежде чем создавать новый класс исключений, следует подумать, что, возможно, ранее созданный в состоянии его обработать.
- Не использовать исключения, которые могут ввести в заблуждение:

```
public class Human {
      private int year;
      public void setYear(int year) throws IOException {
                if (year <= 0) {
                          throw new IOException();
                }
                          this.year = year;
      }
}
```

- Не допускать, чтобы часть обработки ошибки присутствовала в блоке, генерирующем исключение:

```
public void setDeduce(double deduce) throws TaxException {
      if (deduce < 0) {
                this.deduce = 0; // лишнее
                recalculateAmount(); // совсем лишнее
                System.err.print(DEDUCE_NEGATIVE);
                throw new TaxException("VAT deduce < 0");
      }
                this.deduce = deduce;
                recalculateAmount();
}
```

- Никогда самостоятельно не генерировать **NullPointerException** и избегать случаев, когда такая генерация возможна в принципе. Проверка значения ссылки на **null** позволяет обойтись без генерации исключения. Если по логике приложения необходимо генерировать исключение, следует использовать, например, **IllegalArgumentException** с соответствующей информацией об ошибке или собственное исключение.
- Не следует в общем случае в секцию **throws** помещать unchecked-исключения.
- Не рекомендуется вкладывать блоки **try-catch** друг в друга из-за ухудшения читаемости кода.
- При создании собственных исключений следует проводить наследование от класса **Exception**, либо от другого проверяемого класса исключений, а не от **RuntimeException**.
- Никогда не генерировать исключения в инструкции **finally**:

```
try {
      // some code here
} finally {
      if (условие) {
                throw new ParseException();
      }
}
```

При такой генерации исключения никто в приложении не узнает об исключении и, соответственно, не сможет обработать исключение, ранее сгенерированное в инструкции **try**, в случае, если оно не было обработано в инструкции **catch**. В связи со сказанным никогда не следует использовать в инструкции **finally** операторы **return**, **break**, **continue**.

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) {
   // more code
} 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!!!";
   // more code
```

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);
// more code
switch (mono) {
          case WHITE : // more code
                    break;
          case BLACK : // more code
                    break;
          default :
                    assert false : "Colored!";
}
```

Создатели языка не рекомендуют использовать assertion при проверке параметров **public**-методов. В таких ситуациях лучше обрабатывать возможность генерации 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.

Механизм assertion можно включать для отдельных классов и пакетов при запуске виртуальной машины в виде:

```
java -enableassertions RunnerClass
```

или

```
java -ea RunnerClass
```

Для выключения применяется **-da** или **-disableassertions**.

Задания к главе 8

Вариант А

Выполнить задания на основе варианта А гл. 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 гл. 4, реализуя собственные обработчики исключений и исключения ввода/вывода.

Тестовые задания к главе 8

Вопрос 8.1.

Выберите правильные утверждения (3):

1) Проверяемые (checked) исключения являются наследниками класса java.lang.Exception
2) Непроверяемые (unchecked) исключения являются наследниками класса java.lang.Error
3) Непроверяемые (unchecked) исключения являются наследниками класса java.lang.Exception
4) Проверяемые (checked) исключения обязательно обрабатываются
5) Непроверяемые (unchecked) исключения невозможно обработать

Вопрос 8.2.

Дан код:

```
try {  FileReader fr1 = new FileReader("test1.txt");
try {     FileReader fr2 = new FileReader("test2.txt");
          } catch (IOException e) {
                    System.out.print("test2");
          }
          System.out.print("+");
} catch (FileNotFoundException e) {
          System.out.print("test1");
}
System.out.print("+");
```

Какая строка выведется на консоль при компиляции и запуске этого кода, если файл test1.txt существует и доступен, а test2.txt нет (1)?

1) test1
2) test1+
3) test1++
4) test2
5) test2+
6) test2++
7) ошибка компиляции

Вопрос 8.3.

Дана иерархия исключений:

```
class A extends java.lang.Exception{}
class B extends A{}
class C extends B{}
class D extends A{}
class E extends A{}
class F extends D{}
class G extends D{}
class H extends E{}
```

Выберите цепочки блоков catch, использование которых не приведет к ошибке компиляции, если в соответствующем блоке try могут генерироваться исключения типа C,D,G,H (3):

1) **catch**(C e){}**catch**(D e){} **catch**(H e){}**catch**(A e){}
2) **catch**(C e){}**catch**(D e){}**catch**(E e){}**catch**(A e){}
3) **catch**(C e){}**catch**(D e){}**catch**(G e){}**catch**(A e){}
4) **catch**(A e){}**catch**(D e){}**catch**(G e){}**catch**(H e){}
5) **catch**(E e){}**catch**(D e){}**catch**(B e){}**catch**(A e){}

Вопрос 8.4.

Дан код:

```
class A{
        public void f() throws IOException{}
}
class B extends A{}
```

Каким образом можно переопределить метод f() в классе B, не вызвав при этом ошибку компиляции (4)?

1) **public void** f() **throws** Exception {}
2) **public void** f() **throws** IOException {}
3) **public void** f() **throws** InterruptedException, IOException {}
4) **public void** f() **throws** IOException, FileNotFoundException {}
5) **public void** f() **throws** FileNotFoundException {}
6) **public void** f() **throws** FileNotFoundException, InternalError {}

Вопрос 8.5.

Дан код:

```
public class Quest {
        private int qQ;
        public Quest(int q) {
                qQ = 12 / q;//1
        }
        public int getQQ() {
                return qQ;//2
        }
        public static void main(String[] args) {
                Quest quest = null;
                try {
                        quest = new Quest(0);//3
                } catch (Exception e) {//4
                }
                System.out.println(quest.getQQ());//5
        }
}
```

Укажите строку, выполнение которой приведет к необрабатываемой в данном коде исключительной ситуации (1):

1) 1
2) 2
3) 3
4) 4
5) 5

Глава 9

ПОТОКИ ВВОДА/ВЫВОДА

Вопрос. Я скачал из интернета файл. Теперь мне он больше не нужен. Как закачать его обратно?

Ответ. Вот из-за таких, как ты, скоро в интернете все файлы кончатся.

Цитата из Интернет-форума

Потоки ввода/вывода используются для передачи данных в файловые потоки, на консоль или на сетевые соединения. Потоки представляют собой объекты соответствующих классов. Пакеты ввода/вывода: **java.io, java.nio** предоставляют пользователю большое число классов и методов и постоянно обновляются.

Байтовые и символьные потоки ввода/вывода

При разработке приложения регулярно возникает необходимость извлечения информации из какого-либо источника и хранения результатов. Действия по чтению/записи информации представляют собой стандартный и простой вид деятельности. Самые первые классы ввода/вывода связаны с передачей и извлечением последовательности байтов из потоков.

Все потоки ввода последовательности байтов являются подклассами абстрактного класса **InputStream**, потоки вывода — подклассами абстрактного класса **OutputStream**. При работе с файлами используются подклассы этих классов, соответственно, **FileInputStream** и **FileOutputStream**, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом. Существуют классы-потоки для ввода массивов байтов, строк, объектов, а также для выбора из файлов и сетевых соединений.

Для чтения байта или массива байтов используются реализации абстрактных методов **int read()** и **int read(byte[] b)** класса **InputStream**. Метод **int read()** возвращает **–1**, если достигнут конец потока данных, поэтому возвращаемое значение имеет тип **int**, а 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 классе **FileInputStream** метод **read()** читает один байт из файла, а поток **System.in** как объект подкласса **InputStream** позволяет вводить байт с консоли.

Реализация абстрактного метода **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. Класс **FileOutputStream** используется для вывода одного или нескольких байт информации в файл.

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 рисунках 9.1 и 9.2.

Класс **DataInputStream** предоставляет методы для чтения из потока данных значений базовых типов, но не рекомендуется к использованию. Класс **BufferedInputStream** присоединяет к потоку буфер для упрощения и ускорения следующего доступа.

Для вывода данных в поток используются следующие классы.

Абстрактный класс **FilterOutputStream** используется как шаблон для настройки производных классов. Класс **BufferedOutputStream** присоединяет буфер к потоку для ускорения вывода и ограничения 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

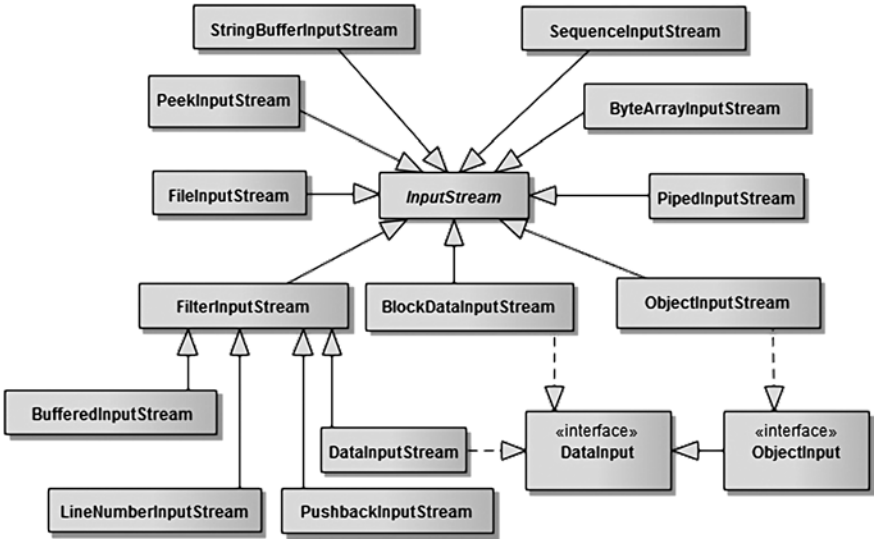

Рис. 9.1. *Иерархия классов байтовых потоков ввода*

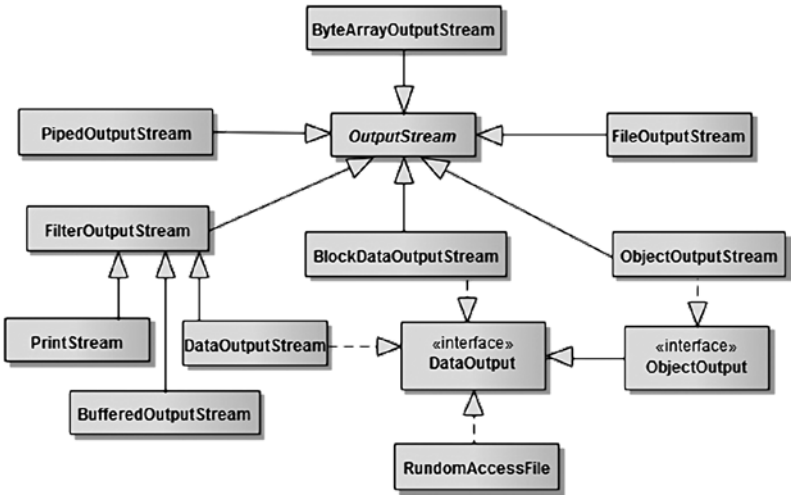

Рис. 9.2. *Иерархия классов байтовых потоков вывода*

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.

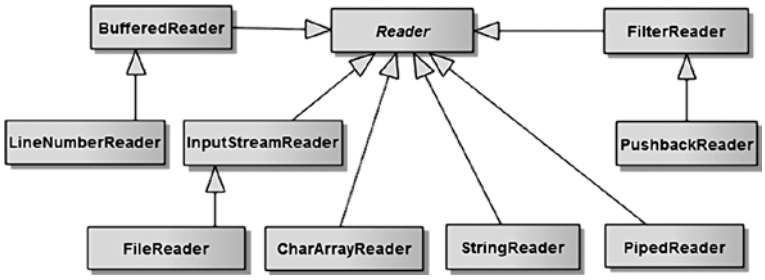

Рис. 9.3. *Иерархия символьных потоков ввода*

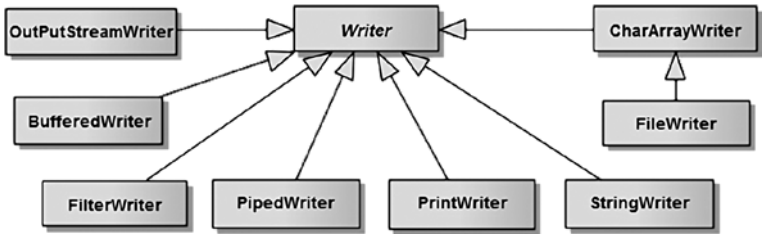

Рис. 9.4. *Иерархия символьных потоков вывода*

/ # 1 # чтение по одному символу (байту) из потока ввода # ReadDemo.java */*

```
package by.bsu.io;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class ReadDemo {
          public static void main(String[] args) {
                    String file = "data\\file.txt";
                    File f = new File(file); // объект для связи с файлом на диске
                    int b, count = 0;
                    FileReader is = null;
                    // FileInputStream is = null; // альтернатива
                    try {
                              is = new FileReader(f);
                              // is = new FileInputStream(f);
                              while ((b = is.read()) != -1) { // чтение
                                        System.out.print((char) b);
                                        count++;
                              }
                              System.out.print("\n число байт = " + count);
                    } catch (IOException e) {
                              System.err.println("Ошибка файла: " + e );
                    } finally {
                              try {
                                        if (is != null) {
                                                  is.close(); // закрытие потока ввода
                                        }
                              } catch (IOException e) {
                                        System.err.println("ошибка закрытия: " + e);
                              }
                    }
          }
}
```

Один из конструкторов **FileReader(file)** или **FileInputStream(file)** открывает поток **is** и связывает его с файлом **file**, где директория **data** в корне проекта должна существовать.

Если файла по указанному пути не существует, то при попытке инициализации потока с несуществующим файлом будет сгенерировано исключение:

Ошибка файла: java.io.FileNotFoundException: data\file.txt (The system cannot find the file specified)

Если файл существует, то информация из него будет считана по одному символу; и результаты чтения, и количество прочитанных символов будут выведены на консоль.

Для закрытия потока используется метод **close()**. Закрытие потока должно произойти при любом исходе чтения: удачном или с генерацией исключения. Гарантировать закрытие потока может только помещение метода **close()** в блок

finally. При чтении из потока можно пропустить **n** байт с помощью метода **long skip(long n)**.

Закрытие потока ввода/вывода в блоке **finally** принято как негласное соглашение и является признаком хорошего кода. Однако выглядит достаточно громоздко. В Java 7 возможно автоматическое закрытие потоков ввода/вывода без явного вызова метода **close()** и блока **finally**:

```
try (FileReader is = new FileReader(new File("data\\file.txt"))) {
          int b = 0;
          while ((b = is.read()) != -1) { /* чтение */
                    System.out.print((char)b)
          }
} catch (IOException e) {
          System.err.println("ошибка файла: " + e);
}
```

С этой версии в список интерфейсов, реализуемых практически всеми классами потоков, добавлен интерфейс **AutoCloseable**. Метод **close()** вызывается неявно для всех потоков, открытых в инструкции

```
try(Поток1: Поток2:…: ПотокN)
```

Например:

```
String dest1 = "filename1";
String dest2 = "filename2";
          try (Writer writer = new FileWriter(dest1):
                    OutputStream out = new FileOutputStream(dest2)) {
              // using writer & out
          } catch (IOException e) {
              e.printStackTrace();
          }
```

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()**.

// # 2 # вывод массива в поток в виде символов и байт # WriteRunner.java

```
package by.bsu.io;
import java.io.*;
public class WriteRunner {
          public static void main(String[] args) {
                    String pArray[] = { "2013 ", "Java SE 8" };
                    File fbyte = new File("data\\byte.data");
                    File fsymb = new File("data\\symbol.txt");
                    FileOutputStream fos = null;
                    FileWriter fw = null;
                    try {
```

```
                    fos = new FileOutputStream(fbyte);
                    fw = new FileWriter(fsymb);
                    for (String a : pArray) {
                        fos.write(a.getBytes());
                        fw.write(a);
                    }
                } catch (IOException e) {
                    System.err.println("ошибка записи: " + e);
                } finally {
                    try {
                        if (fos != null) {
                            fos.close();
                        }
                        // ниже некорректно!
                        // каждому close() требуется свой try-catch
                        if (fw != null) {
                            fw.close();
                        }
                    } catch (IOException e) {
                        System.err.println("ошибка закрытия потока: " + e);
                    }
                }
        }
}
```

В результате будут получены два файла с идентичным набором данных, но созданные различными способами.

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 право 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 obFile = new File("\\com\\file.txt");
File obDir = new File("c:/jdk/src/java/io");
File obFile1 = new File(obDir, "File.java");
File obFile2 = new File("c:\\com", "file.txt");
File obFile3 = 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 file = new File(File.separator + "com" + File.separator + "data.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** принимает значение «**:**», а для Windows — «**;**».

В классе **File** объявлено более тридцати методов, наиболее используемые из них рассмотрены в следующем примере:

/ # 3 # работа с файловой системой: FileTest.java */*

```
package by.bsu.io;
import java.io.File;
import java.util.Date;
import java.io.IOException;
public class FileTest {
   public static void main(String[] args) {
     // с объектом типа File ассоциируется файл на диске FileTest2.java
     File fp = new File("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);
               // 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:                FileTest2.java
Абсолютный путь:             C:\workspace\FileTest2.java
Размер файла:                2091
Последняя модификация:       Fri Nov 21 12:26:50 EEST 2008
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()**.

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**.

Для 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("text\\data.txt"))));
```

В итоге класс **PrintWriter** выступает классом-оберткой для класса **BufferedWriter**, и далее так же, как и класс **BufferedReader** для **FileReader**. По окончании работы с потоками закрывать следует только самый последний класс. В данном случае — **PrintWriter**. Все остальные, в него обернутые, будут закрыты автоматически.

Приведенный ниже пример демонстрирует вывод в файл строк и чисел с плавающей точкой.

// # 4 # буферизованный вывод в файл # DemoWriter.java

```
package by.bsu.io;
import java.io.*;
public class DemoWriter {
          public static void main(String[] args) {
                    File f = new File("data\\res.txt");
                    double[ ] v = { 1.10, 1.2, 1.401, 5.01, 6.017, 7, 8 };
                    FileWriter fw = null;
                    BufferedWriter bw = null;
                    PrintWriter pw = null;
                    try {
                              fw = new FileWriter(f, true);
                              bw = new BufferedWriter(fw);
                              pw = new PrintWriter(bw);
                              for (double version : v) {
                                        pw.printf("Java %.2g%n", version);// запись прямо в файл
                              }
                    } catch (IOException e) {
                              System.err.println("ошибка открытия потока " + e);
                    } finally {
                              if (pw != null) {
```

```
                                        try { // закрывать нужно только внешний поток
                                                pw.close();
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                }
}
```

В итоге в файл **res.txt** будет помещена следующая информация:

Java 1.1
Java 1.2
Java 1.4
Java 5.0
Java 6.0
Java 7.0
Java 8.0

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("res.txt")));
```

Чтение из созданного в предыдущем примере файла с использованием удобной технологии можно произвести следующим образом:

// # 5 # чтение из файла # DemoReader.java

```
package by.bsu.io;
import java.io.*;
public class DemoReader {
        public static void main(String[ ] args) {
                BufferedReader br = null;
                try {
                        br = new BufferedReader(new FileReader("data\\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);
                                        }
                                }
                        } catch (IOException e) {
                                e.printStackTrace();
                        } finally {
                                if (br != null) {
                                        try {
                                                br.close();
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                }
}
```

В консоль будет выведено:

Java
1.1
Java
1.2
Java
1.4
Java
5.0
Java
6.0
Java
7.0
Java
8.0

Сериализация объектов

Кроме данных базовых типов, в поток можно отправлять объекты классов целиком для передачи клиентскому приложению или для хранения.

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 **java.io.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 используются, 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.

/ # 6 # сериализуемый класс # Student.java */*

```
package by.bsu.serial;
import java.io.Serializable;
public class Student implements Serializable {
   protected static String faculty;
   private String name;
   private int id;
   private transient String password;
   private static final long serialVersionUID = 1L;
   /* смысл поля serialVersionUID для класса будет объяснен ниже */
        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;
   }
}
```

/ # 7 # запись сериализованного обьекта в файл и его десериализация # Serializator.java */*

```
package by.bsu.serial;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.io.InvalidClassException;
import java.io.FileNotFoundException;
public class Serializator {
          public boolean serialization(Student s, String fileName) {
                    boolean flag = false;
                    File f = new File(fileName);
                    ObjectOutputStream ostream = null;
                    try {
                              FileOutputStream fos = new FileOutputStream(f);
                              if (fos != null) {
                                        ostream = new ObjectOutputStream(fos);
                                        ostream.writeObject(s); // сериализация
                                        flag = true;
                              }
                    } catch (FileNotFoundException e) {
                              System.err.println("Файл не может быть создан: "+ e);
                    } catch (NotSerializableException e) {
                              System.err.println("Класс не поддерживает сериализацию: " + e);
                    } catch (IOException e) {
                              System.err.println(e);
                    } finally {
                              try {
                                        if (ostream != null) {
                                                  ostream.close();
                                        }
                              } catch (IOException e) {
                                        System.err.println("ошибка закрытия потока");
                              }
                    }
                    return flag;
          }
          public Student deserialization(String fileName) throws InvalidObjectException {
                    File fr = new File(fileName);
                    ObjectInputStream istream = null;
                    try {
                              FileInputStream fis = new FileInputStream(fr);
                              istream = new ObjectInputStream(fis);
                              // десериализация
                              Student st = (Student) istream.readObject();
                              return st;
                    } catch (ClassNotFoundException ce) {
```

```
                    System.err.println("Класс не существует: " + ce);
                } catch (FileNotFoundException e) {
                    System.err.println("Файл для десериализации не существует: "+ e);
                } catch (InvalidClassException ioe) {
                    System.err.println("Несовпадение версий классов: " + ioe);
                } catch (IOException ioe) {
                    System.err.println("Общая I/O ошибка: " + ioe);
                } finally {
                    try {
                        if (istream != null) {
                            istream.close();
                        }
                    } catch (IOException e) {
                        System.err.println("ошибка закрытия потока ");
                    }
                }
                throw new InvalidObjectException("объект не восстановлен");
        }
}
```

/ # 8 # запуск процессов сериализации и десериализации # RunnerSerialization.java */*

```
package by.bsu.serial;
import java.io.InvalidObjectException;
public class RunnerSerialization {
        public static void main(String[] args) {
                // создание и запись объекта
                Student ob = new Student("MMF", "Goncharenko", 1, "G017s9");
                System.out.println(ob);
                String file = "data\\demo.data";
                Serializator sz = new Serializator();
                boolean b = sz.serialization(ob, file);
                Student.faculty = "GEO"; // изменение значения static-поля
                // чтение и вывод объекта
                Student res = null;
                try {
                    res = sz.deserialization(file);
                } catch (InvalidObjectException e) {
                    // обработка
                    e.printStackTrace();
                }
                System.out.println(res);
        }
}
```

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 **res** сохранили значения, которые им были присвоены до записи в файл. Поле **password** со спецификатором **transient** получило значение по умолчанию, соответствующее типу (объектный 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 полем класса является ссылка на другой тип, то необходимо, чтобы агрегированный тип также реализовывал интерфейс **Serializable**, иначе при попытке сериализации объекта такого класса будет сгенерировано исключение **NotSerializableException**.

/ # 9 # класс, сериализация которого невозможна # Student.java */*

```
public class Student implements Serializable {
          protected static String faculty;
          private int id;
          private String name;
          private transient String password;
          private Address addr = new Address(); // не поддерживающее сериализацию поле
          private static final long serialVersionUID = 2L;
          // more code
}
```

Если класс **Address** не имплементирует интерфейс **Serializable**, а именно:

```
public class Address {
          // поля, методы
}
```

то при таком объявлении класса **Address** сериализация объекта класса **Student** невозможна.

При сериализации объекта класса, реализующего интерфейс **Serializable**, учитывается порядок объявления полей в классе. Поэтому при изменении порядка, имен и типов полей или добавлении новых полей в класс структура информации, содержащейся в сериализованном объекте, будет серьезно отличаться от новой структуры класса. Поэтому десериализация может пройти некорректно. Этим обусловлена необходимость добавления программистом в каждый класс, реализующий интерфейс **Serializable**, поля **private static final long serialVersionUID** на стадии разработки класса. Это поле содержит уникальный

идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса — полям, их порядку объявления, методам, их порядку объявления. Для этого применяются специальные программы-генераторы UID.

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 первую очередь полей класса значение поля **serialVersionUID** должно быть изменено программистом или генератором.

Если набор полей класса и их порядок жестко определены, методы класса могут меняться. В этом случае сериализации и десериализации ничто не угрожает**.**

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 конструктор по умолчанию. 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.
```

Класс Scanner

Объект класса **java.util.Scanner** принимает форматированный объект или ввод из потока и преобразует его в двоичное представление. При вводе могут использоваться данные из консоли, файла, строки или любого другого источника, реализующего интерфейсы **Readable**, **InputStream** или **ReadableByteChannel**.

На настоящий момент класс **Scanner** предлагает наиболее удобный и полный интерфейс для извлечения информации практически из любых источников.

Некоторые конструкторы класса:

```
Scanner(String source)
Scanner(File source) throws FileNotFoundException
Scanner(File source, String charset) throws FileNotFoundException
Scanner(InputStream source)
Scanner(InputStream source, String charset)
Scanner(Path source)
Scanner(Path source, String charset),
```

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 символов, выделенный набором разделителей (по умолчанию пробельными символами). В случае ввода из консоли следует определить объект:

```
Scanner con = new Scanner(System.in);
```

После создания объекта его используют для ввода информации в приложение, например лексему или строку,

```
String str1 = con.next();
String str2 = con.nextLine();
```

или типизированных лексем, например, целых чисел,

```
if(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***Tun***()** или **boolean hasNext***Tun***(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 *Tun* **next***Tun***()**. Произвольная лексема считывается методом **String next()**. После извлечения любой лексемы текущий указатель устанавливается перед следующей лексемой.

В качестве примера можно рассмотреть форматированное чтение из файла **scan.txt**, содержащего информацию следующего вида:

2 Java 1,6
true 1.7

// # 10 # разбор текстового файла # ScannerDemo.java

```
package by.bsu.reading;
import java.io.*;
```

```
import java.util.Scanner;
public class ScannerDemo {
          public static void main(String[ ] args) {
                    String filename = "text\\scan.txt";
                    Scanner scan = null;
                    try {
                              FileReader fr = new FileReader(filename);
                              scan = new Scanner(fr);
                              // чтение из файла
                              while (scan.hasNext()) {
                                        if (scan.hasNextInt()) {
                                          System.out.println(scan.nextInt() + " :int");
                                        } else if (scan.hasNextBoolean()) {
                                          System.out.println(scan.nextBoolean() + " :boolean");
                                        } else if (scan.hasNextDouble()) {
                                          System.out.println(scan.nextDouble() + " :double");
                                        } else {
                                          System.out.println(scan.next() + " :String");
                                        }
                              }
                    } catch (FileNotFoundException e) {
                              System.err.println(e);
                    } finally {
                              if (scan != null) {
                                        scan.close();
                              }
                    }
          }
}
```

В результате выполнения кода (при русских региональных установках) будет выведено:

2 :int
Java :String
1.6 :double
true :boolean
1.7 :String

Процедура проверки типа реализована 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.

Объект класса **Scanner** определяет границы лексемы, основываясь на наборе разделителей. Можно задавать разделители с помощью метода **useDelimiter(Pattern pattern)** или **useDelimiter(String regex)**, где **pattern** и **regex** содержит набор разделителей в виде регулярного выражения. Применение метода **useLocale(Locale loc)** позволяет задавать правила чтения информации, принятые в заданной стране или регионе.

/ # 11 # применение разделителей и локалей # ScannerDelimiterDemo.java*/*

```
package by.bsu.scan;
import java.util.Locale;
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.useLocale(Locale.FRANCE);
                    // scan.useLocale(Locale.US);
                    scan.useDelimiter(";\\s*");
                    while (scan.hasNext()) {
                              if (scan.hasNextDouble()) {
                                        sum += scan.nextDouble();
                              } else {
                                        System.out.println(scan.next());
                              }
                    }
                    scan.close();
                    System.out.printf("Сумма чисел = " + sum);
          }
}
```

В результате выполнения программы будет выведено:

Сумма чисел = 36.6

Если заменить **Locale** на американскую, то результат будет иным, так как представление чисел с плавающей точкой отличается.

Использование шаблона **";\\s*"** указывает объекту класса **Scanner**, что символ «**;**» и ноль или более пробелов следует рассматривать как разделитель.

Метод **String findInLine(Pattern pattern)** или **String findInLine(String pattern)** ищет заданный шаблон в текущей 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 существует 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. Некоторые методы класса:

void setMethod(int method) — устанавливает метод сжатия 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.

Пусть необходимо архивировать только файлы в указанной директории. Если директория содержит другие директории, то их файлы архивироваться не будут.

/ # 12 # создание jar-архива # PackJar.java */*

```
package by.bsu.packing;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
import java.util.ArrayList;
import java.util.jar.JarEntry;
import java.util.jar.JarOutputStream;
import java.util.zip.Deflater;
public class PackJar {
          private String jarFileName;
          public final int BUFFER = 2_048;
          public PackJar(String jarFileName) {
                    this.jarFileName = jarFileName;
          }
          public void pack(String dirName) throws FileNotFoundException {
                    // получение списка имен файлов в директории
                    File dir = new File(dirName);
                    if (!dir.exists() || !dir.isDirectory()) {
                              throw new FileNotFoundException(dir + " not found");
                    }
                    File[] files = dir.listFiles();
                    ArrayList<String> listFilesToJar = new ArrayList<>();
                    for (int i = 0; i < files.length; i++) {
                              if (!files[i].isDirectory()) {
                                        listFilesToJar.add(files[i].getPath());
                              }
                    }
                    String[] temp = {};
                    String[] filesToJar = listFilesToJar.toArray(temp);
                    // собственно архивирование
                    try (FileOutputStream fos = new FileOutputStream(jarFileName);
                                        JarOutputStream jos = new JarOutputStream(fos)) {
                              byte[] buffer = new byte[BUFFER];
                              // метод сжатия
                              jos.setLevel(Deflater.DEFAULT_COMPRESSION);
                              for (int i = 0; i < filesToJar.length; i++) {
                                    jos.putNextEntry(new JarEntry(filesToJar[i]));
                                    try (FileInputStream in = new FileInputStream(filesToJar[i])) {
                                              int len;
                                              while ((len = in.read(buffer)) > 0) {
                                                        jos.write(buffer, 0, len);
                                              }
                                              jos.closeEntry();
                                        } catch (FileNotFoundException e) {
                                              System.err.println("Файл не найден" + e);
                                        }
                              }
                    } catch (IllegalArgumentException e) {
                              System.err.println(e + "Некорректный аргумент" + e);
                    } catch (IOException e) {
                              System.err.println("Ошибка доступа" + e);
                    }
          }
}
```

Расширить класс до архивации файлов из вложенных директорий относительно просто.

Класс **JarFile** обеспечивает гибкий доступ к записям, хранящимся в **jar**-файле. Это достаточно 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.

/ # 13 # запуск процесса архивации # PackDemo.java */*

```
package by.bsu.packing;
import java.io.FileNotFoundException;
public class PackDemo {
          public static void main(String[] args) {
                    String dirName = "путь_к_директории";
                    PackJar pj = new PackJar("example.jar");
                    try {
                              pj.pack(dirName);
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    }
          }
}
```

В результате выполнения кода будет создан архивный файл **example.jar**, размещенный в корне проекта.

Класс **JarInputStream** читает данные в **jar**-формате из потока ввода. Он переопределяет метод **read()** таким образом, чтобы любые данные, считываемые из потока, предварительно распаковывались.

Теперь следует распаковать файл из архива и разместить по заданному пути, к которому добавится исходный путь к файлам.

/ # 14 # чтение jar-архива # UnPackJar.java */*

```
package by.bsu.packing;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
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 = 2_048;
        public void unpack(String destinationDirectory, String nameJar) {
                File sourceJarFile = new File(nameJar);
                try (JarFile jFile = new JarFile(sourceJarFile)) {
                        File unzipDir = new File(destinationDirectory);
                        // открытие jar-архива для распаковки
                        Enumeration<JarEntry> jarFileEntries = jFile.entries();
                        while (jarFileEntries.hasMoreElements()) {
                                // извлечение текущей записи из архива
                                JarEntry entry = jarFileEntries.nextElement();
                                String entryname = entry.getName();
                                System.out.println("Extracting: " + entry);
                                destFile = new File(unzipDir, entryname);
                                // определение каталога
                                File destinationParent = destFile.getParentFile();
                                // создание структуры каталогов
                                destinationParent.mkdirs();
                                // распаковывание записи, если она не каталог
                                if (!entry.isDirectory()) {
                                        writeFile(jFile, entry);
                                }
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
        private void writeFile(JarFile jFile, JarEntry entry) {
                int currentByte;
                byte data[] = new byte[BUFFER];
                try (BufferedInputStream is = new BufferedInputStream(
                                                        jFile.getInputStream(entry));
                      FileOutputStream fos = new FileOutputStream(destFile);
                      BufferedOutputStream dest = new BufferedOutputStream(fos,BUFFER)) {
                        // запись файла на диск
                        while ((currentByte = is.read(data, 0, BUFFER)) > 0) {
                                dest.write(data, 0, currentByte);
                        }
                        dest.flush();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

При указании пути к архивному файлу и пути, по которому требуется разместить разархивированные файлы, следует указывать либо абсолютный путь, либо путь относительно корневой директории проекта.

```
/* # 15 # запуск процесса архивации # UnpackDemo.java */
package by.bsu.packing;
public class UnpackDemo {
          public static void main(String[] args) {
                    // расположение и имя архива
                    String nameJar = "example.jar";
                    // куда файлы будут распакованы
                    String destinationPath = "c:\\temp\\";
                    new UnPackJar().unpack(destinationPath, nameJar);
          }
}
```

На консоль будет выведен список разархивированных файлов, например:

Extracting: c:\temp\by\bsu\market\Broker.java
Extracting: c:\temp\by\bsu\market\Main.java
Extracting: c:\temp\by\bsu\market\Market.java

Задания к главе 9

Вариант А

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

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

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

1. В каждой строке найти и удалить заданную подстроку.
2. В каждой строке стихотворения 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 гл. 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 в 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˚ 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.

Укажите классы, которые стоят во главе символьной иерархии потоков ввода/вывода в Java (2):

1) InputStreamReader
2) OutputStreamWriter
3) Reader
4) Writer
5) InputStream
6) OutputStream

Вопрос 9.2.

Дан код:

```
public class Quest {
        public static void main(String[] args) throws IOException {
                Scanner sc1 = new Scanner(System.in);
                int x1 = 0;
                x1 = sc1.nextInt();
                sc1.close();
                int x2=0;
                x2 = System.in.read();
                System.out.println(x1+" "+(char)x2);
        }
}
```

Что будет результатом компиляции и запуска этого кода, если предполагается при первом считывании с консоли ввести 1, а при втором — 2 (1)?

1) компиляция и запуск пройдут успешно, на консоль выведется строка «1 2»;
2) компиляция и запуск пройдут успешно, на консоль выведется строка «1 0»;
3) компиляция и запуск пройдут успешно, на консоль выведется строка «1»;

4) компиляция кода осуществится успешно, а при работе программы возникнет исключительная ситуация;
5) ошибка компиляции.

Вопрос 9.3.

Дан код:

```
class A {
        public int a=0;
        public A() {
                a = 1;
        }
}
class B extends A implements Serializable {
        public int b=0;
        public B() {
                a = 2;
                b = 3;
        }
}
```

Какие значения примут поля объекта, созданного от класса B с помощью конструктора без параметров, на который ссылается ссылка b при десериализации (1)?

1) b.a=1, b.b=3
2) b.a=2, b.b=3
3) b.a=0, b.b=0
4) b.a=0, b.b=3
5) b.a=1, b.b=0
6) десериализация невозможна, так как класс A не реализует интерфейс Serializable

Вопрос 9.4.

Укажите классы байтовых потоков ввода-вывода, для которых не существует аналогичных классов в символьной иерархии потоков ввода-вывода:

1) DataInputStream
2) InputStreamReader
3) ObjectOutputStream
4) StringWriter

Вопрос 9.5.

Компиляция каких строк следующего кода приведет к ошибке (3)?

```
public class Quest {
        public static void main(String[] args) throws IOException {
                File file = new File("files.t1.txt");
                FileWriter obj1 = new FileWriter(file); //1
                ByteArrayInputStream obj2 = new ByteArrayInputStream(file); //2
                InputStreamReader obj3 = new InputStreamReader(file); //3
                BufferedReader obj4 = new BufferedReader(file); //4
                PrintWriter obj5 = new PrintWriter(file); //5
        }
}
```

1) ошибка компиляции в строке 1;
2) ошибка компиляции в строке 2;
3) ошибка компиляции в строке 3;
4) ошибка компиляции в строке 4;
5) ошибка компиляции в строке 5.

Глава 10

КОЛЛЕКЦИИ

Программирование заставило дерево зацвести.

Алан Дж. Перлис

Общие определения

Коллекции — это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных 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: заменить, 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. Современные информационные системы тестируются на примере 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 собой 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, необходимо 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 Java SE 5, 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 коллекций **List, Set**;

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 remove(Object obj) — удаляет **obj** из 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;

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.

При работе с элементами коллекции применяются следующие интерфейсы:

Comparator<T>, Comparable<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 объекта, который обеспечивает доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом **iterator().** Такой объект позволяет просматривать содержимое коллекции последовательно, элемент 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 ссылку на 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().** Если метод **next()** до вызова **remove()** не вызывался, то будет сгенерировано исключение **IllegalStateException**.

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

Класс **ArrayList<E>** — динамический массив объектных ссылок. Реализует интерфейсы **List<E>**, **Collection<E>**, **Iterable<E>**.

```
java.util.AbstractCollection<E>
   └ java.util.AbstractList<E>
      └ java.util.ArrayList<E>
```

В классе объявлены 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**;

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;

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;

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 списков с малым числом подобных действий. С другой стороны, навигация по списку осуществляется очень быстро, поэтому операции поиска производятся за более короткое время.

/ # 1 # создание параметризованной коллекции # DemoGeneric.java */*

```
package by.bsu.collect;
import java.util.ArrayList;
public class DemoGeneric {
          public static void main(String args[ ]) {
                    ArrayList<String> list = new ArrayList<>();
                    // ArrayList<int> b = new ArrayList<int>(); // ошибка компиляции
                    list.add("Java"); /* компилятор "знает"
                                                допустимый тип передаваемого значения */
                    list.add("JavaFX 2");
                    String res = list.get(0); /* компилятор "знает"
                                                допустимый тип возвращаемого значения */
                    // list.add(new StringBuilder("C#")); // ошибка компиляции
                    // компилятор не позволит добавить "посторонний" тип
                    System.out.print(list); // удобный вывод
          }
}
```

В результате будет выведено:

[Java, JavaFX 2]

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. На этом основан принцип типобезопасности, обеспечиваемый параметризацией коллекций. Пусть в системе онлайн-торговли используются понятия **Order** и **Item**. Заказы могут собираться в списки, например, список заказов за день. Возможна ситуация, когда по каким-либо причинам в список заказов будет добавлен экземпляр товара. В этой ситуации даже свести баланс действительно сделанных заказов простым определением размера списка будет невозможно. Придется извлекать каждый объект, определять его тип и проч.

/ # 2 # некорректная(raw) и корректная коллекция # UncheckCheckRun.java */*

```
package by.bsu.collection;
import java.util.ArrayList;
public class UncheckCheckRun {
          public static void main(String[ ] args) {
                    ArrayList raw = new ArrayList() { // "сырая" коллекция – raw type
                              { // логический блок анонимного класса
                                        add(new Order(231, 12.f));
                                        add(new Item(23154, 120.f, "Xerox"));
                                        add(new Order(217, 1.7f));
                              }
                    };
                    // при извлечении требуется приведение типов
                    Order or1 = (Order) raw.get(0);
                    Item or2 = (Item) raw.get(1);
                    Order or3 = (Order) raw.get(2);
                    for (Object ob : raw) {
                              System.out.println("raw " + ob);
                    }
                    ArrayList<Order> orders = new ArrayList<Order>() {
                              {
                                        add(new Order(231, 12.f));
                                        add(new Order(389, 2.9f));
                                        add(new Order(217, 1.7f));
                                        // add(new Item(23154, 120.f, "Xerox"));
                                        // ошибка компиляции: список параметризован
                              }
                    };
                    for (Order ob : orders) {
                              System.out.println("Order: " + ob);
                    }
          }
}
```

В результате будет выведено:

raw Order [idOrder=231, amount=12.0]
raw Item [idItem=231, price=12.0, name=Xerox]

raw Order [idOrder=217, amount=1.7]
Order: Order [idOrder=231, amount=12.0]
Order: Order [idOrder=389, amount=2.9]
Order: Order [idOrder=217, amount=1.7]

где классы **Order** и **Item** представляют собой сущности *Заказ* и *Товар* в следующем виде:

/ # 3 # класс используется здесь и далее для наполнения коллекций # Order.java */*

```
package by.bsu.collection;
public class Order {
          private int orderId;
          private float amount;
          // поля и методы описания подробностей заказа
          public Order(int orderId, float amount) {
                    this.orderId = orderId;
                    this.amount = amount;
          }
          public int getOrderId () {
                    return orderId;
          }
          public float getAmount() {
                    return amount;
          }
          @Override
          public String toString() {
                    return "Order [orderId =" + orderId + ", amount=" + amount + "]";
          }
}
```

/ # 4 # класс используется здесь и далее для наполнения коллекций # Item.java */*

```
package by.bsu.collection;
public class Item {
          private int itemId;
          private float price;
          private String name;
          public Item(int itemId, float price, String name) {
                    this itemId = itemId;
                    this.price = price;
                    this.name = name;
          }
          public int getItemId () {
                    return itemId;
          }
          public float getPrice() {
                    return price;
          }
          public String getName() {
```

```
                return name;
        }
        @Override
        public String toString() {
                return "Item [itemId =" + itemId + ", price=" + price + ", name=" + name + "]\n";
        }
}
```

Чтобы параметризация коллекции была полной, необходимо указывать параметр и при объявлении ссылки, и при создании объекта.

Интерфейс **Iterator** используется для последовательного доступа к элементам коллекции. Ниже приведен пример поиска с помощью итератора заказов, сумма которых превышает заданную.

/ # 5 # методы класса ArrayList и интерфейса Iterator # RunIterator.java # FindOrder.java*/*

```
package by.bsu.collection;
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;
public class RunIterator {
        public static void main(String[ ] args) {
                ArrayList<Order> orders = new ArrayList<Order>() {
                        {
                                add(new Order(231, 12.f));
                                add(new Order(389, 2.9f));
                                add(new Order(217, 1.7f));
                        }
                };
                FindOrder fo = new FindOrder();
                List<Order> res = fo.findBiggerAmountOrder(10f, orders);
                System.out.println(res);
        }
}
package by.bsu.collection;
import java.util.List;
import java.util.Iterator;
public class FindOrder {
        public List<Order> findBiggerAmountOrder(float bigAmount, List<Order> orders) {
                ArrayList<Order> bigPrices = new ArrayList<Order>();
                Iterator<Order> it = orders.iterator();
                while (it.hasNext()) {
                        Order current = it.next();
                        if(current.getAmount() >= bigAmount) {
                                bigPrices.add(current);
                }
            }
        return bigPrices;
        }
}
```

В результате на консоль будет выведено:

[Order [idOrder=231, amount=12.0]]

Метасимвол в коллекциях

Метасимволы используются при параметризации коллекций для расширения возможностей самой коллекции и обеспечения ее типобезопасности. Например, если параметр метода предыдущего примера изменить с **List<Order>** на **List<? extends Order>**, то в метод можно будет передавать коллекции, параметризованные любым допустимым типом, а именно классом **Order** и любым его подклассом, что невозможно при записи без анонимного символа.

Но в методе нельзя будет добавить к коллекции новый элемент, пусть даже и допустимого типа, так как компилятору неизвестен заранее тип параметризации списка.

```
List<Order> findBiggerAmountOrder(float bigAmount, List<? extends Order> orders) {
          // orders.add(new Order(231, 12.f)); // ошибка компиляции
          orders.remove(0); // удалять можно
          // some code here
}
```

Объясняется это тем, что список, передаваемый в качестве параметра метода, может быть параметризован классом **DiscountOrder**, а в методе будет совершаться попытка добавления экземпляра самого класса **Order**, как показано выше, что недопустимо определением самой параметризации передаваемого в метод списка, а именно:

```
findBiggerAmountOrder(10.f, new ArrayList<DiscountOrder>());
```

где

```
public class DiscountOrder extends Order {
          // поля
          public DiscountOrder(int idOrder, float amount) {
                    super(idOrder, amount);
          }
          // методы
}
```

Поэтому добавление к спискам, параметризованным метасимволом с применением **extends**, запрещено всегда.

Параметризация **List<? super Order>** утверждает, что параметр метода или возвращаемое значение может получить список типа **Order** или любого из его суперклассов, в то же время разрешает добавлять туда экземпляры класса **Order** и любых его подклассов.

```
List<Order> findBiggerAmountOrder(float bigAmount, List<? super Order> orders) {
                    orders.add(new Order(231, 12.f));
                    // some code here
}
```

Или, если использовать **super** для определения типа возвращаемого значения:

```
List<? super Order> findBiggerAmountOrder(float bigAmount, List<? extends Order> orders) {
        // some code here
}
```

В этом случае к списку, возвращенному методом, можно будет добавлять экземпляры класса **Order** и его подклассов.

Интерфейс ListIterator

Для доступа к элементам списка может также использоваться интерфейс **ListIterator<E>**, который позволяет получить доступ сразу в необходимую программисту позицию списка вызовом метода **listIterator(int index).** Интерфейс **ListIterator<E>** расширяет интерфейс **Iterator<E>** и предназначен для обработки списков и их вариаций. 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.

При внесении изменений в коллекцию после извлечения итератора гарантирует генерацию исключения **java.util.ConcurrentModificationException** при попытке использовать итератор позднее.

```
ArrayList<Order> orders = new ArrayList<>();
// добавление элементов
Iterator<Order> it = orders.iterator();
orders.add(new Order(12, 12.1f)); // или remove(0);
while (it.hasNext()) {
        System.out.println(it.next());
}
```

Параллельная или конкурентная модификация коллекции различными активностями (самой коллекцией и ее итератором) приводит к неразрешимой проблеме и заканчивается генерацией исключения практически всегда. Проблема заключается в том, что итератор извлек N число позиций, а коллекция изменила число своих экземпляров и итератор перестал соответствовать коллекции. Если модификацию коллекции и работу с итератором нужно выполнять из различных потоков, то для решения такой задачи используются ограниченно потокобезопасные решения для коллекций из пакета **java.util.concurrent**.

При создании класса, содержащего коллекцию элементов, возможны два способа: агрегация коллекции в качестве поля класса или отношение **HAS-A** и наследование от класса, представляющего коллекцию или отношение **IS-A**. Последнее встречается значительно реже.

/ # 6 # класс, агрегирующий список, с реализацией интерфейса Iterable # Order.java*/*

```
import java.util.Iterator;
import java.util.List;
import java.util.Collections;
public class Order implements Iterable<Item> {
          private int orderId;
          private List<Item> listItems;
          // private float amount; // не нужен, т.к. сумму можно вычислить
          public Order(int orderId, List<Item> listItems) {
                    this.orderId = orderId;
                    this.listItems = listItems;
          }
          public int getOrderId () {
                    return orderId;
          }
          public List<Item> getListItems() {
                    return Collections.unmodifiableList(listItems);
          }
          // некоторые делегированные методы интерфейсов List и Collection
          public boolean add(Item e) {
                    return listItems.add(e);
          }
          public Item get(int index) {
                    return listItems.get(index);
          }
          public Item remove(int index) {
                    return listItems.remove(index);
          }
          // создание итератора
          @Override
          public Iterator<Item> iterator() {
                    return listItems.iterator();
          }
}
```

Но можно эту задачу решить еще проще, унаследовав класс коллекции, например, **ArrayList**:

/ # 7 # класс, наследующий список и приобретающий его свойства # Order.java */*

```
package by.bsu.collection;
import java.util.ArrayList;
public class Order extends ArrayList<Item> {
          private int orderId;
          // private float amount; // по прежнему не нужен, т.к. сумму можно вычислить
          public Order(ArrayList<Item> c) {
                    super(c);
          }
          public Order(int orderId, ArrayList<? extends Item> c) {
                    super(c);
```

```
                this.orderId = orderId;
        }
        public int getOrderId() {
                return orderId;
        }
        public void setOrderId (int orderId) {
                this. orderId = orderId;
        }
}
```

При такой реализации нет явной необходимости в переопределении метода класса **ArrayList**, так как можно просто воспользоваться его стандартными методами.

Интерфейс Comparator

Реализация метода **equals()** класса **Object** предоставляет возможность проверить, эквивалентен один экземпляр другому или нет. На практике возникает необходимость сравнения объектов на больше/меньше либо равно.

При реализации интерфейса **java.util.Comparator<T>** появляется возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа. Контракт интерфейса подразумевает реализацию метода **int compare(T ob1, T ob2),** принимающего в качестве параметров два объекта, для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки. Метод **public abstract boolean equals(Object obj),** также объявленный в интерфейсе **Comparator<T>**, очень рекомендуется переопределять, если экземпляр класса будет использоваться для хранения информации. Это необходимо для исключения противоречивой ситуации, когда для двух объектов метод **compare()** возвращает **0**, т. е. сообщает об их эквивалентности, в то же время метод **equals()** для этих же объектов возвращает **false**, так как данный метод не был никем определен и была использована его версия из класса **Object**. Кроме того, наличие метода **equals()** обеспечивает корректную работу метода семантического поиска и проверки на идентичность **contains(Object o),** определенного в интерфейсе **java.util.Collection**, а следовательно, реализованного в любой коллекции.

Метод **compare()** автоматически вызывается при сортировке списка методом: **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 и применяется правило сортировки.

С помощью анонимного типа можно привести простейшую реализацию компаратора.

/ # 8 # сортировка списка по полю, определенному в классе comparator-е # SortItemRunner.java */*

```
package by.bsu.comparison;
import java.util.ArrayList;
```

```
import java.util.Collections;
import java.util.Comparator;
import by.bsu.collection.Item;
public class SortItemRunner {
          public static void main(String[ ] args) {
                    ArrayList<Item> p = new ArrayList<Item>() {
                              {
                                        add(new Item(52201, 9.75f, "T-Shirt"));
                                        add(new Item(52127, 13.99f, "Dress"));
                                        add(new Item(47063, 45.95f, "Jeans"));
                                        add(new Item(90428, 60.9f, "Gloves"));
                                        add(new Item(53295, 31f, "Shirt"));
                                        add(new Item(63220, 14.9f, "Tie"));
                              }
                    };
                    // создание компаратора
                    Comparator<Item> comp = new Comparator<Item>() {
                              // сравнение для сортировки по убыванию цены товара
                              public int compare(Item one, Item two) {
                                        return Double.compare(two.getPrice() - one.getPrice());
                              }
                              // public boolean equals(Object ob) { /* реализация */ }
                    };
                    // сортировка списка объектов
                    Collections.sort(p, comp);
                    System.out.println(p);
          }
}
```

В результате будет выведено:

[Item [itemId=90428, price=60.9, name=Gloves]
, Item [itemId =47063, price=45.95, name=Jeans]
, Item [itemId =53295, price=31.0, name=Shirt]
, Item [itemId =63220, price=14.9, name=Tie]
, Item [itemId =52127, price=13.99, name=Dress]
, Item [itemId =52201, price=9.75, name=T-Shirt]
]

Если в процессе использования необходимы сортировки по различным полям класса, то реализацию компаратора следует вынести в отдельный класс.

Для обеспечения возможности сортировки по любому полю класса **Item** следует создать перечисление со значениями, соответствующими полям этого класса

```
public enum ItemEnum {
          ITEM_ID, PRICE, NAME
}
```

и отдельный класс, реализующий параметризованный интерфейс **Comparator**.

/ # 9 # возможность сортировки по всем полям класса # ItemComparator.java */*

```
package by.bsu.collection;
import java.util.Comparator;
public class ItemComparator implements Comparator<Item> {
          private ItemEnum sortingIndex;
          public ItemComparator(ItemEnum sortingIndex) {
             setSortingIndex(sortingIndex);
          }
          public final void setSortingIndex(ItemEnum sortingIndex) {
             if (sortingIndex == null) {
                throw new IllegalArgumentException();
             }
             this.sortingIndex = sortingIndex;
          }
          public ItemEnum getSortingIndex() {
             return sortingIndex;
          }
          @Override
          public int compare(Item one, Item two) {
             switch (sortingIndex) {
             case ITEM_ID:
                return one.getItemId() - two.getItemId();
             case PRICE:
                return Double.compare(two.getPrice() - one.getPrice());
             case NAME:
                return one.getName().compareTo(two.getName());
             default:
                throw new EnumConstantNotPresentException(ItemEnum.class,sortingIndex.name());
             }
          }
}
```

При необходимости сортировки по полю **itemId** следует изменить значение статической переменной **sortingIndex** и в качестве второго параметра методу **sort()** передать объект класса **ItemComparator**:

```
Collections.sort (p, new ItemComparator (ItemEnum.ITEM_ID));
```

Достаточно легко реализовать возможность сортировки по возрастанию и убыванию для каждого из полей класса, добавив в перечисление, например, поле **ascend** типа **boolean**, от значения которого поставить в зависимость знак числа, возвращаемого методом **compare().** Перечисление **ItemEnum** будет выглядеть следующим образом:

/ # 10 # перечисление, предоставляющее возможность сортировки по убыванию\ возрастанию для всех полей класса # FullItemEnum.java # */*

```
package by.bsu.collection;
public enum FullItemEnum {
          ITEM_ID(true), PRICE(false), NAME(true);
```

```
        private boolean ascend;
        private FullItemEnum(boolean ascend) {
                this.ascend = ascend;
        }
        public void setAscend(boolean ascend) {
                this.ascend = ascend;
        }
        public boolean getAscend() {
                return ascend;
        }
}
```

Класс **ItemComparator** также следует переписать, так как с учетом новых возможностей число кодов блоков **case** увеличится.

Класс-компаратор, являясь логической частью класса-сущности, может быть его частью и представлен в виде статического внутреннего класса

/ # 11 # класс-сущность с внутренними классами-компараторами # Item.java # */*

```
package by.bsu.collection;
import java.util.Comparator;
public class Item {
        private int itemId;
        private float price;
        private String name;
        public Item(int itemId, float price, String name) {
                super();
                this.itemId = itemId;
                this.price = price;
                this.name = name;
        }
        public int getItemId() {
                return itemId;
        }
        public float getPrice() {
                return price;
        }
        public String getName() {
                return name;
        }
        public static class IdComparator implements Comparator<Item> {
                @Override
                public int compare(Item one, Item two) {
                        return one.getItemId() - two.getItemId();
                }
        }
        public static class PriceComparator implements Comparator<Item> {
                @Override
                public int compare(Item one, Item two) {
```

```
                    return Double.compare(two.getPrice() - one.getPrice());
                }
        }
}
```

Экземпляр компаратора создается обычным для внутренних классов способом

```
Item.IdComparator comp = new Item.IdComparator();
```

Объявление внутри класса позволяет манипулировать доступом к его функциональности.

Интерфейс **Comparator** не рекомендуется имплементировать классу-сущности, для сортировки экземпляров которого он предназначается.

Класс LinkedList и интерфейс Queue

Коллекция **LinkedList<E>** реализует связанный список.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.AbstractSequentialList<E>
              └ java.util.LinkedList<E>
```

Реализует кроме интерфейсов, указанных при описании **ArrayList**, также интерфейсы **Queue<E>** и **Deque<E>**.

Связанный список хранит ссылки на объекты отдельно вместе со ссылками на следующее и предыдущее звенья последовательности, поэтому часто называется двунаправленным списком. Операции добавления и удаления выполняются достаточно быстро, в отличие от операций поиска и навигации.

В этом классе объявлены методы, позволяющие манипулировать им как очередью, двунаправленной очередью и т. д. Двунаправленный список кроме обычного имеет особый «нисходящий» итератор, позволяющий двигаться от конца списка к началу, и извлекается методом **descendingIterator().**

Для манипуляций с первым и последним элементами списка в **LinkedList<E>** реализованы методы:

void addFirst(E ob), void addLast(E ob) — добавляющие элементы в начало и конец списка;

E getFirst(), E getLast() — извлекающие элементы;

E removeFirst(), E removeLast() — удаляющие и извлекающие элементы;

E removeLastOccurrence(E elem), E removeFirstOccurrence(E elem) — удаляющие и извлекающие элемент, первый или последний раз встречаемый в списке.

Класс **LinkedList<E>** реализует интерфейс **Queue<E>**, что позволяет списку придать свойства очереди. В компьютерных науках очередь — структура данных, в основе которой лежит принцип FIFO (first in, first out). Элементы добавляются в конец и вынимаются из начала очереди. Но существует возможность

не только добавлять и удалять элементы, также можно просмотреть, что находится в 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>**:

boolean add(E o) — вставляет элемент в очередь, если же очередь полностью заполнена, то генерирует исключение **IllegalStateException**;

boolean offer(E o) — вставляет элемент в очередь, если возможно;

E element() — возвращает, но не удаляет головной элемент очереди;

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 **NoSuchElementException**, если очередь пуста.

```
Queue<Item> queue = new LinkedList<>();
```

Создается очередь простым присваиванием связанного списка ссылке типа **Queue**.

/ # 12 # двунаправленный список и очередь # OrderQueueAction.java */*

```
package by.bsu.collection;
import java.util.LinkedList;
import java.util.Queue;
public class OrderQueueAction {
          public static void main(String[ ] args) {
                    LinkedList<Order> orders = new LinkedList<Order>() {
                              {
                                        add(new Order(231, 12.f));
                                        add(new Order(389, 2.9f));
                                        add(new Order(217, 1.7f));
                              }
                    };
                    Queue<Order> queueA = orders; // создание очереди
                    queueA.offer(new Order(222, 9.7f)); // элемент не добавится
                    orderProcessing(queueA); // обработка очереди
                    if (queueA.isEmpty()) {
                              System.out.println("Queue of Orders is empty");
                    }
          }
          public static void orderProcessing(Queue<Order> queue) { // заменить void -> boolean
                    Order ob = null;
                    // заменить while -> do{} while
                    while ((ob = queue.poll()) != null) { // извлечение с удалением
```

```
                    System.out.println("Order #" + ob.getIdOrder() + " is processing");
                    // verifying and processing
                }
        }
}
```

В результате будет выведено:

Order #231 is processing
Order #389 is processing
Order #217 is processing
Queue of Orders is empty

При всей схожести списков **ArrayList** и **LinkedList** существуют серьезные отличия, которые необходимо учитывать при использовании коллекций в конкретных задачах. Если необходимо осуществлять быструю навигацию по списку, то следует применять **ArrayList**, так как перебор элементов в **LinkedList** осуществляется на порядок медленнее. С другой стороны, если требуется часто добавлять и удалять элементы из списка, то уже класс **LinkedList** обеспечивает значительно более высокую скорость переиндексации. То есть если коллекция формируется в начале процесса и в дальнейшем используется только для доступа к информации, то применяется **ArrayList**, если же коллекция подвергается изменениям на всем протяжении функционирования приложения, то выгоднее **LinkedList**.

Интерфейс **Queue<E>** также реализует класс **PriorityQueue<E>**.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractQueue<E>
          └ java.util.PriorityQueue<E>
```

В такую очередь элементы добавляются не в порядке «живой очереди», а в порядке, определяемом в классе, экземпляры которого содержатся в очереди. Сам порядок следования задан реализацией интерфейсов **Comparator<E>** или **Comparable<E>**.

По умолчанию компаратор размещает элементы в очереди в порядке возрастания. Если порядок в классе не определен, а именно, не реализован ни один из указанных интерфейсов, то генерируется исключительная ситуация **ClassCastException** при добавлении второго элемента в очередь. При добавлении первого элемента компаратор не вызывается.

```
PriorityQueue<Order> orders = new PriorityQueue<Order>();
orders.add(new Order(546, 53.f)); // нормально
orders.add(new Order(146, 13.f)); // ошибка времени выполнения
```

С другой стороны класс **String** реализует интерфейс **Comparable**:

```
PriorityQueue<String> q = new PriorityQueue<String>();
orders.add(new String("Oracle")); // нормально
orders.add(new String("Google")); // нормально
```

Если попытаться заменить тип **String** на **StringBuilder** или **StringBuffer**, то создать очередь **PriorityQueue** так просто не удастся, как и в случае добавления объектов класса **Order**. Решением такой задачи будет создание нового класса-оболочки с полем типа **StringBuilder** или **Order** и реализацией интерфейса **Comparable<T>**.

/ # 13 # пользовательский класс, обьект которого может быть добавлен в очередь PriorityQueue и множество TreeSet # Order.java */*

```
package by.bsu.collection;
public class Order implements Comparable<Order> {
          private int orderId;
          private float amount;
          // поля и методы описания подробностей заказа
          public Order(int orderId, float amount) {
                    super();
                    this.orderId = orderId;
                    this.amount = amount;
          }
          public int getOrderId() {
                    return orderId;
          }
          public float getAmount() {
                    return amount;
          }
          // реализация метода интерфейса Comparable
          @Override
          public int compareTo(Order ob) {
                    return this.orderId - ob.orderId;
          }
          @Override
          public String toString() {
                    return "Order [orderId =" + orderId + ", amount=" + amount + "]";
          }
}
```

Предлагаемое решение универсально для любых пользовательских типов. Предложенное решение применимо для создания упорядоченных множеств вида **TreeSet**, которые используют компаратор для сортировки при добавлении экземпляра в множество.

Интерфейс Deque и класс ArrayDeque

Интерфейс **Deque** определяет «двунаправленную» очередь и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди. Реализацию этого интерфейса можно использовать для моделирования стека. 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. 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**. Объявить двуконечную очередь на основе связанного списка можно, например, следующим образом:

```
Deque<String> dq = new LinkedList<>();
```

Класс **ArrayDeque** быстрее, чем **Stack**, если используется как стек, и быстрее, чем **LinkedList<E>,** если используется в качестве очереди.

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<E>** существенно облегчает поиск элементов, например, расположенных рядом с заданным.

Класс **HashSet<E>** наследуется от абстрактного суперкласса **AbstractSet<E>** и реализует интерфейс **Set<E>**, используя хэш-таблицу для хранения коллекции.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.HashSet<E>
```

Ключ (хэш-код) используется в качестве индекса хэш-таблицы для доступа к объектам множества, что значительно ускоряет процессы поиска, добавления и извлечения элемента. Скорость указанных процессов становится заметной для коллекций с большим количеством элементов. Множество **HashSet** не является сортированным. В таком множестве могут храниться элементы с одинаковыми хэш-кодами в случае, если эти элементы не эквивалентны при сравнении. Для грамотной организации **HashSet** следует следить, чтобы реализации методов **hashCode()** и **equals()** соответствовали контракту.

Конструкторы класса:

```
HashSet()
HashSet(Collection <? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float loadFactor),
```

где **capacity** — число ячеек для хранения хэш-кодов.

/ # 14 # использование множества для вывода всех уникальных слов из файла # DemoHashSet.java */*

```
package by.bsu.set;
import java.io.*;
import java.util.*;
public class DemoHashSet {
          public static void main(String[ ] args) {
                    HashSet<String> words = new HashSet<>(100);
                    long callTime = System.nanoTime();
                    Scanner scan = null;
                    try {
                       scan = new Scanner(new File("texts\\nabokov.txt"));
                              scan.useDelimiter("[^A-я]+");
                              while (scan.hasNext()) {
                                        String word = scan.next();
                                        words.add(word.toLowerCase());
                              }
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    }
                    Iterator<String> it = words.iterator();
                    while (it.hasNext()) {
                              System.out.println(it.next());
                    }
                    TreeSet<String> ts = new TreeSet<>(words);
                    System.out.println(ts);
                    long totalTime = System.nanoTime() - callTime;
                    System.out.println("различных слов: " + words.size() + ", "
                              + totalTime + " наносекунд");
          }
}
```

Класс **TreeSet<E>** для хранения объектов использует бинарное дерево.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.TreeSet<E>
```

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что класс реализует интерфейс **SortedSet**, где правило сортировки добавляемых элементов определяется в самом классе, сохраняемом в множестве, который в большинстве случаев реализует интерфейс **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 **subSet(E from, E to)**, **tailSet(E from)** и **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.

/ # 15 # создание множества из списка и его методы # DemoTreeSet.java */*

```
package by.bsu.set;
import java.util.*;
public class DemoTreeSet {
          public static void main(String[ ] args) {
                    ArrayList<String> list = new ArrayList<>();
                    boolean b;
                    for (int i = 0; i < 6; i++) {
                              list.add((int) (Math.random() * 10) + "Y ");
                    }
                    System.out.println(list + "список");
                    TreeSet<String> set = new TreeSet<String>(list);
                    System.out.println(set + "множество");

                    System.out.println(set.comparator()); // null – String реализует Comparable

                    // извлечение наибольшего и наименьшего элементов
                    System.out.println(set.last() + " " + set.first());

                    HashSet<String> hSet = new HashSet<>(set);
                    for (String str : hSet) {
                              System.out.println(str + "    " + str.hashCode());
                    }
          }
}
```

В результате может быть выведено:

[6Y , 5Y , 2Y , 5Y , 4Y , 8Y]список
[2Y , 4Y , 5Y , 6Y , 8Y]множество
null
8Y 2Y
2Y 50841
6Y 54685
8Y 56607
4Y 52763
5Y 53724

Множество инициализируется списком и сортируется сразу же в процессе создания. С помощью итератора элемент может быть найден и удален из множества. Для множества, состоящего из обычных строк, используется лексикографическая

сортировка, задаваемая реализацией интерфейса **Comparable**, поэтому метод **comparator()** возвращает **null**.

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от абстрактного класса **AbstractSet**.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.EnumSet<E>
```

Класс 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 нумерованых констант с указанным типом элемента, метод **allOf(Class<E> elementType)** создает множество нумерованых 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 указанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**.

/ # 16 # использование множества enum-типов # CarManufacturer.java # UseEnumSet.java */*

```
package by.bsu.set;
public enum CarManufacturer {
          AUDI, BMW, VW, TOYOTA, HONDA, ISUZU, SUZUKI, VOLVO, RENAULT
}
package by.bsu.set;
import java.util.EnumSet;
public class UseEnumSet {
          public static void main(String[ ] args) {
                    /* множество japanAuto содержит элементы типа
                    enum из интервала, определенного двумя элементами */
                    EnumSet <CarManufacturer> japanAuto =
                    EnumSet.range(CarManufacturer.TOYOTA, CarManufacturer.SUZUKI);
/* множество other будет содержать все элементы, не содержащиеся в множестве japanAuto */
                    EnumSet <CarManufacturer> other = EnumSet.complementOf(japanAuto);
                    System.out.println(japanAuto);
                    System.out.println(other);
                              action("audi", japanAuto);
```

```
                    action("suzuki", japanAuto);
        }
        public static boolean action(String auto, EnumSet <CarManufacturer> set) {
                CarManufacturer cm = CarManufacturer.valueOf(auto.toUpperCase());
                boolean ok = false;
                if(ok = set.contains(cm)) {
                        // обработка
                        System.out.println("Обработан: " + cm);
                } else {
                        System.out.println("Обработка невозможна: " + cm);
                }
                return ok;
        }
}
```

В результате будет выведено:

[TOYOTA, HONDA, ISUZU, SUZUKI]
[AUDI, BMW, VW, VOLVO, RENAULT]
Обработка невозможна: AUDI
Обработан: SUZUKI

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()** или реализацией интерфейсов **Comparable**, **Comparator** пользовательским классом. Классы карт отображений:

AbstractMap<K, V> — реализует интерфейс **Map<K, V>**, является суперклассом для всех перечисленных карт отображений;

HashMap<K, V> — использует хэш-таблицу для работы с ключами;

TreeMap<K, V> — использует дерево, где ключи расположены в виде дерева поиска в определенном 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> — образует дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

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 навигации и поиска по ключу.

Интерфейс **Map<K, V>** содержит следующие методы:

V get(Object obj) — возвращает значение, связанное с ключом **obj.** Если элемент с указанным ключом отсутствует в карте, то возвращается значение **null**;

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**;

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<K> keySet() — возвращает множество ключей;

Set<Map.Entry<K, V>> entrySet() — возвращает множество, содержащее значения карты в виде пар «ключ–значение»;

Collection<V> values() — возвращает коллекцию, содержащую значения карты.

В коллекциях, возвращаемых тремя последними методами, можно только удалять элементы, добавлять нельзя. Данное ограничение обуславливается параметризацией возвращаемого методами значения.

Интерфейс **Map.Entry<K, V>** представляет пару «ключ–значение» и 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.

/ # 17 # создание хэш-карты и замена элемента по ключу # DemoHashMap.java */*

```
package by.bsu.maps;
public class DemoHashMap {
          public static void main(String[] args) {
                    HashMap<String, Integer> hm = new HashMap<String, Integer>(3) {
                              {
                                        this.put("Сырок", 3);
                                        this.put("Пряник", 5);
```

```
                                        this.put("Молоко", 1);
                                        this.put("Хлеб", 1);
                                }
                        };
                        System.out.println(hm);
                        hm.put("Пряник", 4); // замена или добавление при отсутствии ключа
                        System.out.println(hm + "после замены элемента");
                        Integer a = hm.get("Хлеб");
                        System.out.println(a + " - найден по ключу 'Хлеб'");
                        // вывод хэш-таблицы с помощью методов интерфейса Map.Entry<K,V>
                        Set<Map.Entry<String, Integer>> setv = hm.entrySet();
                        System.out.println(setv);
                        Iterator<Map.Entry<String, Integer>> i = setv.iterator();
                        while (i.hasNext()) {
                                Map.Entry<String, Integer> me = i.next();
                                System.out.println(me.getKey() + " : " + me.getValue());
                        }
                        Set<Integer> val = new HashSet<Integer>(hm.values());
                        System.out.println(val);
        }
}
```

{Пряник=5, Хлеб=1, Сырок=3, Молоко=1}
{Пряник=4, Хлеб=1, Сырок=3, Молоко=1}после замены элемента
1 - найден по ключу 'Хлеб'
[Пряник=4, Хлеб=1, Сырок=3, Молоко=1]
Пряник : 4
Хлеб : 1
Сырок : 3
Молоко : 1
[1, 3, 4]

Метод **put()** не проверяет наличия пары с таким же ключом, как и у добавляемой пары и просто заменяет значение по ключу на новое. Если критично наличие всех ранее добавленных элементов, следует перед добавлением пары выполнять проверку на наличие идентичных ключей. В простейшем варианте это выглядит:

```
HashMap<String, Integer> hm = new HashMap<String, Integer>() {
                        {
                                this.put("Сырок", 3);
                                this.put("Пряник", 5);
                                this.put("Молоко", 1);
                                this.put("Хлеб", 1);
                        }
        };
        if( !hm.containsKey("Пряник") ) { // замена не произойдет, если ключ существует
                        hm.put("Пряник", 4);
        }
```

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.

/ # 18 # пример работы с классом EnumMap # GASStation.java # Gases.java */*

```
package by.bsu.enummap;
enum GASStation {
          DT(50), A80(30), A92(30), A95(50), GAS(40);
          private Integer maxVolume;
          private GASStation (int maxVolume) {
                    this.maxVolume = maxVolume;
          }
          public Integer getMaxVolume() {
                    return maxVolume;
          }
}
import java.util.EnumMap;
public class Gases {
          public static void main(String[ ] args) {
                    EnumMap<GASStation, Integer> station1 =
                              new EnumMap<GASStation, Integer>(GASStation.class);
                    station1.put(GASStation.DT, 10);
                    station1.put(GASStation.A80, 5);
                    station1.put(GASStation.A92, 30);
                    EnumMap<GASStation, Integer> station2 =
                              new EnumMap<GASStation, Integer>(GASStation.class);
                    station2.put(GASStation.DT, 25);
                    station2.put(GASStation.A95, 25);
                    System.out.println(station1);
                    System.out.println(station2);
                    }
}
```

В результате будет выведено:

{DT=10, A80=5, A92=30}
{DT=25, A95=25}

Унаследованные коллекции

В ряде распределенных приложений, например с использованием сервлетов, до сих пор применяются коллекции, более медленные в обработке, но при этом потокобезопасные (thread-safety), существовавшие в языке Java с момента его создания, а именно карта **Hashtable<K, V>**, список **Vector<E>** и перечисление (аналог итератора) **Enumeration<E>**. Все они также были параметризованы, но сохраняют возможность одновременного доступа из конкурирующих потоков.

Класс **Hashtable<K, V>** реализует интерфейс **Map**,

```
java.util.Dictionary<K,V>
      └ java.util.Hashtable<K,V>
```

обладает также несколькими специфичными по сравнению с другими коллекциями методами:

Enumeration<V> elements() — возвращает перечисление для значений карты;
Enumeration<K> keys() — возвращает перечисление для ключей карты.

/ # 19 # создание хэш-таблицы и поиск элемента по ключу # HashTableDemo.java */*

```
package by.bsu.legacy;
import java.util.*;
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 **Properties** предназначен для хранения карты свойств, где и ключ, и значения являются экземплярами класса **String**. Значения пары можно загружать и хранить в файле.

/ # 20 # создание экземпляра и файла properties # PropertiesDemoWriter.java */*

```
package by.bsu.collection;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
public class PropertiesDemoWriter {
   public static void main(String[ ] args) {
      Properties props = new Properties();
      try {
          // установка значений экземпляру
          props.setProperty("db.driver", "com.mysql.jdbc.Driver");
          // props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");
          props.setProperty("db.user", "root");
          props.setProperty("db.password", "pass");
          props.setProperty("db.poolsize", "5");
          // запись properties-файла в папку prop проекта
          props.store(new FileWriter("prop" + File.separator + "database.properties"),null);
      } catch (IOException e) {
          e.printStackTrace();
      }
   }
}
```

В результате в файле **database.properties** будет размещена следующая информация:

#Wed Aug 01 03:40:01 FET 2012
db.password=pass
db.user=root
db.poolsize=5
db.driver=com.mysql.jdbc.Driver

Символ «=» служит по умолчанию разделителем ключа и значения в файле.**properties**, также в этом качестве можно использовать символ «:». Эти два специальных символа при записи в файл в качестве части ключа или значения получают впереди символ «\», чтобы в дальнейшем при чтении не быть воспринятым как разделитель «ключа–значения». Например, при задании свойства вида

```
props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");
```

в файл будет записано

db.url=jdbc\:mysql\://127.0.0.1\:3306/testphones

Извлечь информацию из файла достаточно просто.

/ # 21 # загрузка файла properties в экземпляр и доступ к содержимому # PropertiesDemo.java */*

```
package by.bsu.collection;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;
public class PropertiesDemo {
          public static void main(String[ ] args) {
                    Properties props = new Properties();
                    try {
                              // загрузка пар ключ-значение через поток ввода-вывода
                              props.load(new FileReader("prop\\database.properties"));
                    } catch (IOException e) {
                              e.printStackTrace();
                    }
                    String driver = props.getProperty("db.driver");
                    // следующих двух ключей в файле нет
                    String maxIdle = props.getProperty("db.maxIdle"); // будет присвоен null
                    // значение "20" будет присвоено ключу, если он не будет найден в файле
                    String maxActive = props.getProperty("db.maxActive", "20");
                    System.out.println(driver);
                    System.out.println(maxIdle);
                    System.out.println(maxActive);
          }
}
```

В результате будет выведено:

com.mysql.jdbc.Driver
null
20

Алгоритмы класса Collections

Класс **java.util.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:

```
static <E> Collection <E> checkedCollection(Collection<E> c, Class<E> type)
```

Этот метод создает коллекцию, проверяемую на этапе выполнения, т. е. в случае добавления «постороннего» объекта генерируется исключение **ClassCastException**:

/ # 22 # проверяемая коллекция # SafeSetRun.java */*

```
package by.bsu.check;
import java.util.*;
public class SafeSetRun {
          public static void main(String args[ ]) {
                    HashSet orders;
                    // orders = new HashSet();  // заменяемый код на jdk1.4 и ниже
                    orders = Collections.checkedSet(new HashSet<Order>(), Order.class);
                    orders.add(new Order(параметры));
                    // some code here
                    orders.add(new Item(параметры)); // runtime error
          }
}
```

В этот же класс добавлен целый ряд статических методов, специализированных для проверки конкретных типов коллекций, а именно: **checkedList(), checkedSortedMap(), checkedMap(), checkedSortedSet(), checkedCollection(),** а также

<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> 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;

singleton(T o), singletonList(T o), singletonMap(K key, V value) — создают множество, список и карту отображения, позволяющие добавлять только один элемент;

<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 с использованием **Comparable** или **Comparator** соответственно;

void swap(List<?> list, int i, int j) — меняет местами элементы списка, стоящие на заданных позициях;

<T> List<T> unmodifiableList(List<? extends T> list) — возвращает ссылку на список с запрещением его модификации. Аналогичные методы есть и для других видов коллекций.

/ # 23 # применение некоторых алгоритмов # AlgorithmDemo.java */*

```
package by.bsu.collect;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class AlgoritmDemo {
          public static void main(String[ ] args) {
                    Comparator<Integer> comp = new Comparator<Integer>() {
                              public int compare(Integer n, Integer m) {
                                        return m.intValue() - n.intValue();
                              }
                    };
                    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
          }
          private static void print(List<Integer> c) {
                    for (int i : c) {
                              System.out.print(i + " ");
                    }
                    System.out.println();
          }
}
```

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
```

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 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, больше *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,

построить дерево, отображающее систему дорог республики, а затем, продвигаясь по дереву, определить минимальный по длине путь из г. Минска в другой заданный город. Предусмотреть возможность 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 действительно эффективным, протестировав программу для случая, когда оба ключа 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 двух 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 четырех 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 класс коллекции позволяет наращивать и сокращать размер, предоставляет индексный доступ к элементам, но его методы несинхронизированы (1)?

1) java.util.HashSet
2) java.util.ArrayList
3) java.util.LinkedHashSet

Вопрос 10.2.

Дан класс:

```
class X{
        private int x;
        public X(int x){this.x = x;}
        public int hashCode(){
                return 2;
        }
}
```

Каков будет размер коллекции set после выполнения следующего кода (1)?

```
X obj1 = new X(1);      X obj2 = new X(1);
Set<X> set = new HashSet<X>();
set.add(obj1);
set.add(obj2);
```

1) 0;
2) 2;
3) 1;
4) при выполнении данного кода возникнет исключительная ситуация.

Вопрос 10.3.

Даны два фрагмента кода:

1.

```
Vector<String> vct = new Vector<String>();
vct.add("One"); vct.add("Two");
vct.add("Three"); vct.add("Four");
vct.add("Five"); vct.add("Six");
Iterator itr = vct.iterator();
vct.remove(0);
System.out.println(itr.next());
```

2.
```
Vector<String> vct = new Vector<String>();
vct.add("One"); vct.add("Two");
vct.add("Three"); vct.add("Four");
vct.add("Five"); vct.add("Six");
Enumeration enm = vct.elements();
vct.remove(0);
System.out.println(enm.nextElement());
```

Укажите разницу при выполнении первого и второго фрагмента кода (1):

1) при выполнении первого фрагмента на консоль выведется строка «Two», а при выполнении второго произойдет исключительная ситуация;
2) при выполнении второго фрагмента на консоль выведется строка «Two», а при выполнении первого произойдет исключительная ситуация;
3) выполнение двух фрагментов кода приведет к исключительной ситуации;
4) при выполнении двух фрагментов кода на консоль выведется строка «Two»;
5) при компиляции первого фрагмента приведет к ошибке;
6) компиляция второго фрагмента кода приведет к ошибке;
7) компиляция двух фрагментов кода приведет к ошибке.

Вопрос 10.4.

Дан код:

```
import java.util.*;
enum PCounter {UNO, DOS, TRES, CUATRO, CINCO, SEIS, SIETE};
public class Quest {
public static void main(String[] args) {
EnumSet<PCounter> enst1 = EnumSet.range(PCounter.TRES, PCounter.CINCO);//1
EnumSet<PCounter> enst2 = EnumSet.complementOf(enst1);//2
System.out.println(enst2);
}}
```

Что будет выведено при попытке компиляции и запуска программы (1)?

1) [UNO, DOS, TRES, CINCO, SEIS, SIETE]
2) [DOS, TRES, CUATRO, CINCO, SEIS]
3) [UNO, DOS, SEIS, SIETE]
4) ошибка компиляции в строке 1
5) ошибка компиляции в строке 2
6) ничего из перечисленного

Вопрос 10.5.

Что будет результатом компиляции и запуска следующей программы (1)?

```
import java.util.*;
public class Quest {
public static void main(String[] args) {
        NavigableMap<String, Number> nmap = new TreeMap<String, Number>();
        nmap.put("one", new Integer(1)); nmap.put("two", new Integer(2));
        nmap.put("three", new Integer(3)); nmap.put("four", new Integer(4));
        Map<String, Number> map = nmap.headMap("three");
        System.out.println(map);
}}
```

1) {four=4, one=1}
2) {one=2, two=2}
3) {three=3, four=4}
4) {four=4, one=1, three=3}
5) {one=2, two=2, three=3}

Глава 11

ПОТОКИ ВЫПОЛНЕНИЯ

Соседняя очередь всегда движется быстрее.

Наблюдение Этторе

Как только вы перейдете в другую очередь,
ваша начнет двигаться быстрее.

Наблюдение О'Брайена

Класс Thread и интерфейс Runnable

К большинству современных распределенных приложений (Rich Client) и веб-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, каждой в независимом потоке. Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существует два способа создания и запуска потока: на основе расширения класса **Thread** или реализации интерфейса **Runnable**.

// # 1 # расширение класса Thread # TalkThread.java

```
package by.bsu.threads;
public class TalkThread extends Thread {
        @Override
        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println("Talking");
                        try {
                                Thread.sleep(7); // остановка на 7 миллисекунд
                        } catch (InterruptedException e) {
                                System.err.print(e);
                        }
                }
        }
}
```

При реализации интерфейса **Runnable** необходимо определить его единственный абстрактный метод **run().** Например:

/ # 2 # реализация интерфейса Runnable # WalkRunnable.java # WalkTalk.java */*

```
package by.bsu.threads;
public class WalkRunnable implements Runnable {
          @Override
          public void run() {
                    for (int i = 0; i < 10; i++) {
                              System.out.println("Walking");
                              try {
                                        Thread.sleep(7);
                              } catch (InterruptedException e) {
                                        System.err.println(e);
                              }
                    }
          }
}
package by.bsu.threads;
public class WalkTalk {
          public static void main(String[ ] args) {
                    // новые объекты потоков
                    TalkThread talk = new TalkThread();
                    Thread walk = new Thread(new WalkRunnable());
                    // запуск потоков
                    talk.start();
                    walk.start();
                    // WalkRunnable w = new WalkRunnable(); // просто объект, не поток
                    // w.run(); или talk.run(); // выполнится метод, но поток не запустится!
          }
}
```

Запуск двух потоков для объектов классов **TalkThread** непосредственно и **WalkRunnable** через инициализацию экземпляра **Thread** приводит к выводу строк: **Talking Walking**. Порядок вывода, как правило, различен при нескольких запусках приложения.

Интерфейс **Runnable** не имеет метода **start(),** а только единственный метод **run().** Поэтому для запуска такого потока, как **WalkRunnable** следует создать экземпляр класса **Thread** с передачей экземпляра **WalkRunnable** его конструктору. Однако при прямом вызове метода **run()** поток не запустится, выполнится только тело самого метода.

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.

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;

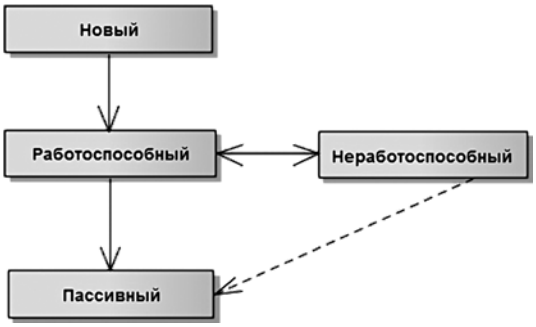

Рис. 11.1. *Состояния потока*

TIMED_WAITING — поток некоторое время ждет окончания другого потока;

TERMINATED — поток завершен.

Получить текущее значение состояния потока можно вызовом метода **getState().**

Поток переходит в состояние «неработоспособный» в режиме ожидания (**WAITING**) вызовом методов **join(), 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 по времени (**TIMED_WAITING**) с помощью методов **yield(), sleep(long millis), join(long timeout)** и **wait(long timeout),** при выполнении которых 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, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока. Метод **interrupt()** успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток неработоспособен, например, находится в состоянии **TIMED_WAITING**, то метод инициирует исключение **InterruptedException**. Чтобы это не происходило, следует предварительно вызвать метод **isInterrupted(),** который проверит возможность завершения работы потока. При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.

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 потока можно с помощью метода **getPriority().**

// # 3 # демонстрация приоритетов # PriorityRunner.java # PriorThread.java

```
package by.bsu.priority;
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 {
                                        Thread.sleep(1); // попробовать sleep(0),sleep(10)
                              } catch (InterruptedException e) {
                                        System.err.print(e);
                              }
                    }
          }
}
package by.bsu.priority;
public class PriorityRunner {
          public static void main(String[ ] args) {
                    PriorThread min = new PriorThread("Min");
                    PriorThread max = new PriorThread("Max");
                    PriorThread norm = new PriorThread("Norm");
                    min.setPriority(Thread.MIN_PRIORITY); // 1
                    max.setPriority(Thread.MAX_PRIORITY); // 10
                    norm.setPriority(Thread.NORM_PRIORITY); // 5
                    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 в группу, 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 приоритета 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(int millis)** 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 или не истечет время ожидания при обращении к методу **join(long timeout).**

// # 4 # задержка потока # JoinRunner.java

```
package by.bsu.management;
class JoinThread extends Thread {
          public JoinThread (String name) {
                    super(name);
          }
          public void run() {
                    String nameT = getName();
                    long timeout = 0;
                    System.out.println("Старт потока " + nameT);
                    try {
                              switch (nameT) {
                                  case "First":
                                        timeout = 5_000;
                                  case "Second":
                                        timeout = 1_000;
                              }
                              Thread.sleep(timeout);
                              System.out.println("завершение потока " + nameT);
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
          }
}
public class JoinRunner {
              static {
                    System.out.println("Старт потока main");
                    }
          public static void main(String[ ] args) {
                    JoinThread t1 = new JoinThread("First");
                    JoinThread t2 = new JoinThread("Second");
```

```
                t1.start();
                t2.start();
                try {
                    t1.join(); // поток main остановлен до окончания работы потока t1
                } catch (InterruptedException e) {
                                e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()); // имя текущего потока
        }
}
```

Возможно, будет выведено:

Старт потока main
Старт потока First
Старт потока Second
завершение потока Second
завершение потока First
main

Несмотря на вызов метода **join()** для потока **t1**, поток **t2** будет работать, в отличие от потока **main**, который сможет продолжить свое выполнение только по завершении потока **t1**.

Если вместо метода **join()** без параметров использовать версию **join(long timeout),** то поток **main** будет остановлен только на указанный промежуток времени. При вызове **t1.join(500)** вывод будет другим:

Старт потока First
Старт потока Second
main
завершение потока Second
завершение потока First

Статический метод **currentThread()** возвращает ссылку на текущий поток, т. е. на поток, в котором данный метод был вызван.

Вызов статического метода **yield()** для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, чтобы другие потоки могли выполнять свои действия. Например, в случае потока с высоким приоритетом после обработки части пакета данных, когда следующая еще не готова, стоит уступить часть времени другим потокам. Однако если требуется надежная остановка потока, то следует использовать его крайне осторожно или вообще применить другой способ.

// # 5 # задержка потока # YieldRunner.java

```
package by.bsu.yield;
public class YieldRunner {
        public static void main(String[ ] args) {
```

```
                new Thread() { // анонимный класс
                        public void run() {
                                System.out.println("старт потока 1");
                                Thread.yield();
                                System.out.println("завершение 1");
                        }
                }.start(); // запуск потока
                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 используются для работы в фоновом режиме вместе с программой, но не являются неотъемлемой частью логики программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon(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.

// # 6 # запуск и выполнение потока-демона # SimpleThread.java # DaemonRunner.java

```
package by.bsu.daemons;
public class SimpleThread extends Thread {
        public void run() {
                try {
                        if (isDaemon()) {
                                System.out.println("старт потока-демона");
                                Thread.sleep(10_000); // заменить параметр на 1
```

```
                    } else {
                        System.out.println("старт обычного потока");
                    }
                } catch (InterruptedException e) {
                    System.err.print(e);
                } finally {
                    if (!isDaemon()) {
                        System.out.println("завершение обычного потока");
                    } else {
                        System.out.println("завершение потока-демона");
                    }
                }
        }
}
package by.bsu.daemons;
public class DaemonRunner {
        public static void main(String[ ] args) {
                SimpleThread usual = new SimpleThread();
                SimpleThread daemon = new SimpleThread();
                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 и исключения

В процессе функционирования потоки являются в общем случае независимыми друг от друга. Прямым следствием такой независимости будет корректное продолжение работы потока **main** после аварийной остановки запущенного из него потока после генерации исключения.

/ # 7 # генерация исключения в созданном потоке # ExceptThread.java # ExceptionThreadDemo.java */*

```
package by.bsu.thread;
public class ExceptThread extends Thread {
          public void run() {
                    boolean flag = true;
                    if (flag) {
                              throw new RuntimeException();
                    }
                    System.out.println("end of ExceptThread");
          }
}
package by.bsu.thread;
public class ExceptionThreadDemo {
          public static void main(String[ ] args) throws InterruptedException {
                    new ExceptThread().start();
                    Thread.sleep(1000);
                    System.out.println("end of main");
          }
}
```

Основной поток избавлен от необходимости обрабатывать исключения в порожденных потоках.

В данной ситуации верно и обратное: если основной поток прекратит свое выполнение из-за необработанного исключения, то это не скажется на работоспособности порожденного им потока.

/ # 8 # генерация исключения в потоке main # SimpleThread.java # ExceptionMainDemo.java */*

```
package by.bsu.thread;
public class SimpleThread extends Thread {
          public void run() {
                    try {
                              Thread.sleep(1000);
                    } catch (InterruptedException e) {
                              System.err.print(e);
                    }
                    System.out.println("end of SimpleThread");
          }
}
package by.bsu.thread;
public class ExceptionMainDemo {
          public static void main(String[ ] args) {
                    new SimpleThread().start();
                    System.out.println("end of main with exception");
                    throw new RuntimeException();
          }
}
```

Атомарные типы и модификатор volatile

Все данные приложения находятся в основном хранилище данных. При запуске нового потока создается копия хранилища и именно ею пользуется этот поток. Изменения, произведенные в копии, могут не сразу находить отражение в основном хранилище, и наоборот. Для получения актуального значения следует прибегнуть к синхронизации. Наиболее простым приемом будет объявление поля класса с модификатором **volatile**. Данный модификатор вынуждает потоки производить действия по фиксации изменений достаточно быстро. То есть другой заинтересованный поток, скорее всего, получит доступ к уже измененному значению. Для базовых типов до 32 бит этого достаточно. При использовании со ссылкой на объект — синхронизировано будет только значение самой ссылки, а не объект, на который она ссылается. Синхронизация ссылки будет эффективной в случае, если она указывает на перечисление, так как все элементы перечисления существуют в единственном экземпляре. Решением проблемы с доступом к одному экземпляру из разных потоков является блокирующая синхронизация. Модификатор **volatile** обеспечивает неблокирующую синхронизацию.

Существует целая группа классов пакета **java.util.concurrent.atomic**, обеспечивающая неблокирующую синхронизацию. Атомарные классы созданы для организации неблокирующих структур данных. Классы атомарных переменных **AtomicInteger**, **AtomicLong**, **AtomicReference** и др. расширяют нотацию **volatile** значений, полей и элементов массивов. Все атомарные классы являются изменяемыми в отличие от соответствующих им классов-оболочек. При реализации классов пакета использовались эффективные атомарные инструкции машинного уровня, которые доступны на современных процессорах. В некоторых ситуациях могут применяться варианты внутреннего блокирования.

Экземпляры классов, например, **AtomicInteger** и **AtomicReference,** предоставляют доступ и разного рода обновления к одной-единственной переменной соответствующего типа. Каждый класс также обеспечивает набор методов для этого типа. В частности, класс **AtomicInteger** — атомарные методы инкремента и декремента. Инструкции при доступе и обновлении атомарных переменных, в общем, следуют правилам для **volatile**.

Не следует классы атомарных переменных использовать как замену соответствующих классов-оболочек.

Пусть имеется некоторая торговая площадка, представленная классом **Market**, работающая в непрерывном режиме и информирующая о разнонаправленных изменениях биржевого индекса (поле **index** типа **AtomicLong**) дважды за один цикл с интервалом до 500 миллисекунд. Изменения поля **index** фиксируются методом **addAndGet(long delta)** атомарного добавления переданного значения к текущему.

/ # 9 # класс с атомарным полем # Market.java */*

```
package by.bsu.market;
import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;
public class Market extends Thread {
          private AtomicLong index;
          public Market(AtomicLong index) {
                    this.index = index;
          }
          public AtomicLong getIndex() {
                    return index;
          }
          @Override
          public void run() {
                    Random random = new Random();
                    try {
                              while (true) {
                                        index.addAndGet(random.nextInt(10));
                                        Thread.sleep(random.nextInt(500));
                                        index.addAndGet(-1 * random.nextInt(10));
                                        Thread.sleep(random.nextInt(500));
                              }
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
          }
}
```

Имеется класс **Broker**, запрашивающий значение поля **index** с некоторым интервалом в миллисекундах.

/ # 10 # получатель значения атомарного поля # Broker.java */*

```
package by.bsu.market;
import java.util.Random;
public class Broker extends Thread {
          private Market market;
          private static final int PAUSE = 500; // in millis
          public Broker(Market market) {
                    this.market = market;
          }
          @Override
          public void run() {
                    try {
                              while (true) {
                                        System.out.println("Current index: " + market.getIndex());
                                        Thread.sleep(PAUSE);
                              }
```

```
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Количество экземпляров класса **Broker** может быть любым, и они постоянно с заданным интервалом запрашивают текущее значение **index**.

/ # 11 # запуск потоков изменения атомарного поля и его отслеживания несколькими потоками # AtomicDemo.java */*

```
package by.bsu.market;
import java.util.concurrent.atomic.AtomicLong;
public class AtomicDemo {
        private static final int NUMBER_BROKERS = 30;
        public static void main(String[ ] args) {
                Market market = new Market(new AtomicLong(100));
                market.start();
                for (int i = 0; i < Main. NUMBER_BROKERS; i++) {
                        new Broker(market).start();
                }
        }
}
```

Атомарность поля обеспечивает получение экземплярами класса **Broker** идентичных текущих значений поля **index**.

Методы synchronized

Нередко возникает ситуация, когда несколько потоков имеют доступ к некоторому объекту, проще говоря, пытаются использовать общий ресурс и начинают мешать друг другу. Более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток. Для контролирования процесса записи может использоваться разделение ресурса с применением ключевого слова **synchronized**.

В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе **main()** класса **SynchroRun** создаются два потока. В этом же методе создается экземпляр класса **Resource**, содержащий поле типа **FileWriter**, связанное с файлом на диске. Экземпляр **Resource** передается в качестве параметра обоим потокам. Первый поток записывает строку методом **writing()** в экземпляр класса **Resource**. Второй поток также пытается сделать запись строки в тот же самый объект **Resource**. Во избежание одновременной записи такие методы объявляются как **synchronized**. Синхронизированный метод изолирует объект, после чего он становится недоступным для других потоков. Изоляция снимается, когда поток полностью

выполнит соответствующий метод. Другой способ снятия изоляции — вызов метода **wait()** из изолированного метода — будет рассмотрен позже.

В примере продемонстрирован вариант синхронизации файла для защиты от одновременной записи информации в файл двумя различными потоками.

/ # 12 # синхронизация записи информации в файл # SyncThread.java # Resource.java # SynchroRun.java */*

```
package by.bsu.synch;
import java.io.*;
public class Resource {
          private FileWriter fileWriter;
          public Resource (String file) throws IOException {
                    // проверка наличия файла
                    fileWriter = new FileWriter(file, true);
          }
          public synchronized void writing(String str, int i) {
                    try {
                              fileWriter.append(str + i);
                              System.out.print(str + i);
                                        Thread.sleep((long)(Math.random() * 50));
                              fileWriter.append("->" + i + " ");
                              System.out.print("->" + i + " ");
                    } catch (IOException e) {
                              System.err.print("ошибка файла: " + e);
                    } catch (InterruptedException e) {
                              System.err.print("ошибка потока: " + e);
                    }
          }
          public void close() {
                    try {
                              fileWriter.close();
                    } catch (IOException e) {
                              System.err.print("ошибка закрытия файла: " + e);
                    }
          }
}
package by.bsu.synch;
public class SyncThread extends Thread {
          private Resource rs;
          public SyncThread(String name, Resource rs) {
                    super(name);
                    this.rs = rs;
          }
          public void run() {
                    for (int i = 0; i < 5; i++) {
                              rs.writing(getName(), i); // место срабатывания синхронизации
                    }
          }
}
```

```
package by.bsu.synch;
import java.io.IOException;
public class SynchroRun {
          public static void main(String[ ] args)  {
                    Resource s = null;
                    try {
                              s = new Resource ("data\\result.txt");
                              SyncThread t1 = new SyncThread("First", s);
                              SyncThread t2 = new SyncThread("Second", s);
                              t1.start();
                              t2.start();
                              t1.join();
                              t2.join();
                    } catch (IOException e) {
                              System.err.print("ошибка файла: " + e);
                    } catch (InterruptedException e) {
                              System.err.print("ошибка потока: " + e);
                    } finally {
                              s.close();
                    }
          }
}
```

В результате в файл будет, например, 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()** в случае его вызова 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. Такая синхронизация позволяет сузить область синхронизации, т. е. вывести за пределы синхронизации код, в ней не нуждающийся. Обычные методы на синхронизацию внимания не обращают, поэтому ответственность за грамотную блокировку объектов ложится на программиста.

/ # 13 # блокировка обьекта потоком # TwoThread.java */*

```
package by.bsu.instruction;
public class TwoThread {
          static int counter = 0;
          public static void main(String args[ ]) {
                    final StringBuilder s = new StringBuilder();
                    new Thread() {
                              public void run() {
                                        synchronized (s) {
                                                  do {
                                                            s.append("A");
                                                            System.out.println(s);
                                                  try {
                                                            Thread.sleep(100);
                                                  } catch (InterruptedException e) {
                                                            System.err.print(e);
                                                  }
                                                } while (TwoThread.counter++ < 2);
                                        } // конец synchronized
                              }
                    }.start();
                    new Thread() {
                              public void run() {
                                        synchronized (s) {
                                                  while (TwoThread.counter++ < 6) {
                                                            s.append("B");
                                                            System.out.println(s);
                                        }
                              } // конец synchronized
                              }
                    }.start();
          }
}
```

В результате компиляции и запуска, скорее всего (например, второй поток может заблокировать объект первым), будет 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.

Данный пример можно немного изменить для демонстрации «потокобезопасности» класса **StringBuffer** при вызове метода **append()** на синхронизированном экземпляре.

/ # 14 # потокобезопасность класса StringBuffer # BufferThread.java */*

```
package by.bsu.synchro;
public class BufferThread {
          static int counter = 0;
          static StringBuffer s = new StringBuffer(); // заменить на StringBuilder
          public static void main(String args[ ]) throws InterruptedException {
                    new Thread() {
                              public void run() {
                                        synchronized (s) {
                                                  while (BufferThread.counter++ < 3) {
                                                            s.append("A");
                                                            System.out.print("> " + counter + " ");
                                                            System.out.println(s);
                                                            Thread.sleep(500);
                                                  }
                                        } // конец synchronized-блока
                              }
                    }.start();
                    Thread.sleep(100);
                    while (BufferThread.counter++ < 6) {
                              System.out.print("< " + counter + " ");
          // в этом месте поток main будет ждать освобождения блокировки объекта s
                              s.append("B");
                              System.out.println(s);
                    }
          }
}
```

Вызов метода на синхронизированном другим потоком объекте класса **StringBuffer** приведет к остановке текущего потока до тех пор, пока объект не будет разблокирован. То есть выведено будет следующее:

> 1 A
< 2 > 3 AA
AAB
< 5 AABB
< 6 AABBB

Если заменить **StringBuffer** на **StringBuilder**, то остановки потока на заблокированном объекте не произойдет и вывод будет таким:

> 1 A
< 2 AB
< 3 ABB
< 4 ABBB
< 5 ABBBB
< 6 ABBBBB

Монитор

Контроль за доступом к объекту-ресурсу обеспечивает понятие монитора. Монитор экземпляра может иметь только одного владельца. При попытке конкурирующего доступа к объекту, чей монитор имеет владельца, желающий заблокировать объект-ресурс поток должен подождать освобождения монитора этого объекта и только после этого завладеть им и начать использование объекта-ресурса. Каждый экземпляр любого класса имеет монитор. Методы **wait(), wait(long inmillis), notify(), notifyAll()** корректно срабатывают только на экземплярах, чей монитор уже кем-то захвачен. Статический метод захватывает монитор экземпляра класса **Class**, того класса, на котором он вызван. Существует в единственном экземпляре. Нестатический метод захватывает монитор экземпляра класса, на котором он вызван.

Методы wait(), notify() и notifyAll()

Эти методы никогда не переопределяются и используются только в исходном виде. Вызываются только внутри синхронизированного блока или метода на объекте, монитор которого захвачен текущим потоком. Попытка обращения к данным методам вне синхронизации или на несинхронизированном объекте (со свободным монитором) приводит к генерации исключительной ситуации **IllegalMonitorStateException**. В примере #15 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. Возвратить блокировку объекта потоку можно вызовом метода **notify()** для одного потока или **notifyAll()** для всех потоков. Если ожидающих потоков несколько, то после вызова метода **notify()** невозможно определить, какой поток из ожидающих потоков заблокирует объект. Вызов может быть осуществлен только из другого потока, заблокировавшего в свою очередь тот же самый объект.

Проиллюстрировать работу указанных методов можно с помощью примера, когда инициализация полей и манипуляция их значениями производится в различных потоках.

/ # 15 # взаимодействие wait() и notify() # Payment.java # PaymentRunner.java */*

```
package by.bsu.synchro;
import java.util.Scanner;
public class Payment {
          private int amount;
          private boolean close;
          public int getAmount() {
                    return amount;
          }
          public boolean isClose() {
                    return close;
          }
          public synchronized void doPayment() {
                    try {
                              System.out.println("Start payment:");
                    while (amount <= 0) {
                              this.wait(); //  остановка потока и освобождение блокировки
                              // после возврата блокировки выполнение будет продолжено
                    }
                    // код выполнения платежа
                    close = true;
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
                    System.out.println("Payment is closed : " + close);
          }
          public void initAmount() {
                    Scanner scan = new Scanner(System.in);
                    amount = scan.nextInt();
          }
}
package by.bsu.synchro;
public class PaymentRunner {
          public static void main(String[] args) throws InterruptedException {
                    final Payment payment = new Payment();
                    new Thread() {
                              public void run() {
                                        payment.doPayment(); // вызов synchronized метода
                              }
                    }.start();
                    Thread.sleep(200);
                    synchronized (payment) { // 1-ый блок
                              System.out.println("Init amount:");
                              payment.initAmount();
```

```
                    payment.notify(); // уведомление о возврате блокировки
                }
                synchronized (payment) { // 2-ой блок
                    payment.wait(1_000);
                    System.out.println("ok");
                }
        }
}
```

В результате компиляции и запуска при вводе корректного значения для инициализации поля **amount** будет запущен процесс проведения платежа.

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 результатом вычислений, скорее всего, будет ноль, так как вычисление будет произведено до инициализации полей объекта.

Новые способы управления потоками

Java всегда предлагала широкие возможности для многопоточного программирования: потоки — это основа языка. Однако очень часто использование этих возможностей вызывало трудности: создать и отладить для корректного функционирования многопоточную программу достаточно сложно.

В версии Java SE 5 языка добавлен пакет **java.util.concurrent**, возможности классов которого обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков concurrent классов. Кроме этого усовершенствован вызов утилит синхронизации, добавлены классы семафоров и блокировок.

Возможности синхронизации существовали и ранее. Практически это означало, что синхронизированные экземпляры блокировались, хотя необходимо это было далеко не всегда. Например, поток, изменяющий одну часть объекта **Hashtable**, блокировал работу других потоков, которые хотели бы 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 аналоги существующих синхронизированных классов-коллекций **ConcurrentHashMap**, **ConcurrentLinkedQueue** — эффективные аналоги **Hashtable** и **LinkedList**;
- классы **CopyOnWriteArrayList** и **CopyOnWriteArraySet**, копирующие свое содержимое при попытке его изменения, причем ранее полученный итератор будет корректно продолжать работать с исходным набором данных;

- блокирующие очереди **BlockingQueue** и **BlockingDeque**, гарантирующие остановку потока, запрашивающего элемент из пустой очереди до появления в ней элемента, доступного для извлечения, а также блокирующего поток, пытающийся вставить элемент в заполненную очередь, до тех пор, пока в очереди не освободится позиция;
- механизм управления заданиями, основанный на возможностях класса **Executor**, включающий организацию запуска пула потоков и службы их планирования;
- классы-барьеры синхронизации, такие как **CountDownLatch** (заставляет потоки ожидать завершения заданного числа операций, по окончании чего все ожидающие потоки освобождаются), **Semaphore** (предлагает потоку ожидать завершения действий в других потоках), **CyclicBarrier** (предлагает нескольким потокам ожидать момента, когда они все достигнут какой-либо точки, после чего барьер снимается), **Phaser** (барьер, контракт которого является расширением возможностей **CyclicBarrier**, а также частично согласовывается с возможностями **CountDownLatch**);
- класс **Exchanger** позволяет потокам обмениваться объектами.

В дополнение к перечисленному выше в пакете **java.util.concurrent.locks** содержатся дополнительные реализации моделей синхронизации потоков, а именно:

- интерфейс **Lock**, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством интерфейса **Condition**;
- класс семафор **ReentrantLock,** добавляющий ранее не существующую функциональность по отказу от попытки блокировки объекта с возможностью многократного повторения запроса на блокировку и отказа от нее;
- класс **ReentrantReadWriteLock** позволяет изменять объект только одному потоку, а читать в это время — нескольким.

Перечисление TimeUnit

Представляет различные единицы измерения времени. В **TimeUnit** реализован ряд методов по преобразованию между единицами измерения и по управлению операциями ожидания в потоках в этих единицах. Используется для информирования методов, работающих со временем, о том, как интерпретировать заданный параметр времени.

Перечисление **TimeUnit** может представлять время в семи размерностях-значениях: **NANOSECONDS**, **MICROSECONDS**, **MILLISECONDS**, **SECONDS**, **MINUTES**, **HOURS**, **DAYS**.

Кроме методов преобразования единиц времени представляют интерес методы управления потоками:

void timedWait(Object obj, long timeout) — выполняет метод **wait(long time)** для объекта **obj** класса **Object,** используя данные единицы измерения;

void timedJoin(Thread thread, long timeout) — выполняет метод **join(long time)** на потоке **thread**, используя данные единицы измерения.

void sleep(long timeout) — выполняет метод **sleep(long time)** класса **Thread**, используя данные единицы измерения.

Блокирующие очереди

Реализации интерфейсов **BlockingQueue** и **BlockingDeque** предлагают методы по добавлению/извлечению элементов с задержками, а именно:

void put(E e) — добавляет элемент в очередь. Если очередь заполнена, то ожидает, пока освободится место;

boolean offer(E e, long timeout, TimeUnit unit) — добавляет элемент в очередь. Если очередь заполнена, то ожидает время **timeout**, пока освободится место; если за это время место не освободилось, то возвращает **false**, не выполнив действия по добавлению;

boolean offer(E e) — добавляет элемент в очередь. Если очередь заполнена, то возвращает **false**, не выполнив действия по добавлению;

E take() — извлекает и удаляет элемент из очереди. Если очередь пуста, то ожидает, пока там появится элемент;

E poll(long timeout, TimeUnit unit) — извлекает и удаляет элемент из очереди. Если очередь пуста, то ожидает время **timeout**, пока там появится элемент, если за это время очередь так и осталась пуста, то возвращает **null**;

E poll() — извлекает и удаляет элемент из очереди. Если очередь пуста, то возвращает **null.**

Максимальный размер очереди должен быть задан при ее создании, а именно, все конструкторы класса **ArrayBlockingQueue** принимают в качестве параметра **capacity** длину очереди. Пусть объявлена очередь из пяти элементов. Изначально в ней размещены три элемента. В первом потоке производится попытка добавления трех элементов. Два добавятся успешно, а при попытке добавления третьего поток будет остановлен до появления свободного места в очереди. Только когда второй поток извлечет один элемент и освободит место, первый поток получит возможность добавить свой элемент.

/ # 16 # демонстрация возможностей блокирующей очереди # RunBlocking.java */*

```
package by.bsu.blocking;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class RunBlocking {
          public static void main(String[] args) {
                    final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
                    new Thread() {
```

```
                    public void run() {
                        for (int i = 1; i < 4; i++) {
                            try {
                            queue.put("Java" + i); // добавление 3-х
                            System.out.println("Element " + i + " added");
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                        }
                    }
                }.start();
                new Thread() {
                    public void run() {
                        try {
                        Thread.sleep(1_000);
                        // извлечение одного
                        System.out.println("Element " + queue.take() + " took");
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }.start();
        }
}
```

В результате будет выведено:

Element 1 added
Element 2 added
Element Java1 took
Element 3 added

Семафоры

Семафор позволяет управлять доступом к ресурсам или просто работой потоков на основе запрещений-разрешений. Семафор всегда устанавливается на предельное положительное число потоков, одновременное функционирование которых может быть разрешено. При превышении предельного числа все желающие работать потоки будут приостановлены до освобождения семафора одним из работающих по его разрешению потоков. Уменьшение счетчика доступа производится методами **void acquire()** и его оболочки **boolean tryAcquire()**. Оба метода занимают семафор, если он свободен. Если же семафор занят, то метод **tryAcquire()** возвращает ложь и пропускает поток дальше, что позволяет при необходимости отказаться от дальнейшей работы потоку, который не смог получить семафор. Метод **acquire()** при невозможности захвата семафора остановит поток до тех пор, пока хотя бы другой поток не освободит семафор. Метод **boolean tryAcquire(long timeout,TimeUnit unit)** возвращает

ложь, если время ожидания превышено, т. е. за указанное время поток не получил от семафора разрешение работать и пропускает поток дальше. Метод **release()** освобождает семафор и увеличивает счетчик на единицу. Простое надежное стандартное взаимодействие методов **acquire()** и **release()** демонстрирует следующий фрагмент.

/ # 17 # базовое решение при использовании семафора */*

```
public void run() {
          try {
                    semaphore.acquire();
                    // код использования защищаемого ресурса
          } catch (InterruptedException e) {
          } finally {
                    semaphore.release(); // освобождение семафора
          }
}
```

Методы **acquire()** и **release()** в общем случае могут и не вызываться в одном методе кода. Тогда за корректное и своевременное возвращение семафора будет ответственен разработчик. Метод **acquire()** не пропустит поток до тех пор, пока счетчик семафора имеет значение ноль.

Методы **acquire(), tryAquire()** и **release()** имеют перегруженную версию с параметром типа **int**. В такой метод можно передать число, на которое изменится значение счетчика семафора при успешном выполнении метода, в отличие от методов без параметров, которые всегда изменяют значение счетчика только на единицу.

Для демонстрации работы семафора предлагается задача о пуле ресурсов с ограниченным числом, в данном случае аудиоканалов, и значительно бóльшим числом клиентов, желающих воспользоваться одним из каналов. Каждый клиент получает доступ к каналу, причем пользоваться можно только одним каналом. Если все каналы заняты, то клиент ждет в течение заданного интервала времени. Если лимит ожидания превышен, генерируется исключение и клиент уходит, так и не воспользовавшись услугами пула.

Класс **ChannelPool** объявляет семафор и очередь из каналов. В методе **getResource()** производится запрос к семафору, и в случае успешного его прохождения метод извлекает из очереди канал и выдает его в качестве возвращаемого значения метода. Метод **returnResource()** добавляет экземпляр-канал к очереди на выдачу и освобождает семафор.

Реализация принципов пула предоставляет возможность повторного использования объектов в ситуациях, когда создание нового объекта — дорогостоящая процедура с точки зрения задействованных для этого ресурсов виртуальной машины. Поэтому при возможности следует объект после использования не уничтожать, а возвратить его в так называемый «пул объектов» для повторного использования. Данная стратегия широко используется при организации пула соединений с базой данных. Реализаций организации пулов существует

достаточно много с различающимися способами извлечения и возврата объектов, а также способа контроля за объектами и за заполняемостью пула. Поэтому выбрать какое-либо решение как абсолютно лучшее для всех случаев невозможно.

// # 18 # пул ресурсов # ChannelPool.java

```
package by.bsu.resource.pool;
import java.util.Queue;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
import java.util.LinkedList;
import by.bsu.resource.exception.ResourceException;
public class ChannelPool <T> {
          private final static int POOL_SIZE = 5; // размер пула
          private final Semaphore semaphore = new Semaphore(POOL_SIZE, true);
          private final Queue<T> resources = new LinkedList<T>();
          public ChannelPool(Queue<T> source) {
                    resources.addAll(source);
          }
          public T getResource(long maxWaitMillis) throws ResourceException {
                    try {
                              if (semaphore.tryAcquire(maxWaitMillis, TimeUnit.MILLISECONDS)) {
                                        T res = resources.poll();
                                        return res;
                              }
                    } catch (InterruptedException e) {
                              throw new ResourceException(e);
                    }
                    throw new ResourceException(":превышено время ожидания");
          }
          public void returnResource(T res) {
                    resources.add(res); // возвращение экземпляра в пул
                    semaphore.release();
          }
}
```

Класс **AudioChannel** предлагает простейшее описание канала и его использования.

// # 19 # канал — ресурс: обычный класс с некоторой информацией # AudioChannel.java

```
package by.bsu.resource.pool;
public class AudioChannel {
          private int channellId;
          public AudioChannel(int id) {
                    super();
                    this.channellId = id;
          }
          public int getChannellId() {
                    return channellId;
```

```
        }
        public void setChannellId(int id) {
                this.channellId = id;
        }
        public void using() {
                try {
                        // использование канала
                        Thread.sleep(new java.util.Random().nextInt(500));
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Класс **ResourceException** желателен в такого рода задачах, чтобы точно описать возникающую проблему при работе ресурса, используемого конкурирующими потоками.

// # 20 # исключение, информирующее о сбое в поставке ресурса # ResourceException.java

```
package by..resource.exception;
public class bsu ResourceException extends Exception {
        public ResourceException() {
                super();
        }
        public ResourceException(String message, Throwable cause) {
                super(message, cause);
        }
        public ResourceException(String message) {
                super(message);
        }
        public ResourceException(Throwable cause) {
                super(cause);
        }
}
```

Класс **Client** представляет поток, запрашивающий ресурс из пула, использующий его некоторое время и возвращающий его обратно в пул.

// # 21 # поток, работающий с ресурсом # Client.java

```
package by.bsu.resource.pool;
import by.bsu.resource.exception.ResourceException;
public class Client extends Thread {
        private boolean reading = false;
        private ChannelPool<AudioChannel> pool;
        public Client (ChannelPool<AudioChannel> pool) {
                this.pool = pool;
        }
        public void run() {
```

```
                AudioChannel channel = null;
                try {
                        channel = pool.getResource(500); // изменить на 100
                        reading = true;
                        System.out.println("Channel Client #" + this.getId()
                                + " took channel #" + channel.getChannellId());
                        channel.using();
                } catch (ResourceException e) {
                        System.out.println("Client #" + this.getId() + " lost ->"
                                + e.getMessage());
                } finally {
                        if (channel != null) {
                                reading = false;
                        System.out.println("Channel Client #" + this.getId() + " : "
                                + channel.getChannellId() + " channel released");
                                pool.returnResource(channel);
                        }
                }
        }
        public boolean isReading() {
                return reading;
        }
}
```

Класс **Runner** демонстрирует работу пула ресурсов аудиоканалов. При заполнении очереди каналов в данном решении необходимо следить, чтобы число каналов, передаваемых списком в конструктор класса **ChannelPool**, совпадало со значением константы **POOL_SIZE** этого же класса. Константа используется для инициализации семафора и при большем или меньшем размерах передаваемого списка возникают коллизии, которые, кстати, есть смысл спровоцировать и разобраться в причинах и следствиях.

// # 22 # запуск и использование пула # Runner.java

```
package by.bsu.resource.main;
import java.util.LinkedList;
import by.bsu.resource.pool.AudioChannel;
import by.bsu.resource.pool.ChannelPool;
import by.bsu.resource.pool.Client;
public class Runner {
        public static void main(String[] args) {
                LinkedList<AudioChannel> list = new LinkedList<AudioChannel>() {
                        {
                                this.add(new AudioChannel(771));
                                this.add(new AudioChannel(883));
                                this.add(new AudioChannel(550));
                                this.add(new AudioChannel(337));
                                this.add(new AudioChannel(442));
                        }
```

```
                    };
                    ChannelPool<AudioChannel> pool = new ChannelPool<>(list);
                    for (int i = 0; i < 20; i++) {
                              new Client(pool).start();
                    }
          }
}
```

Результатом может быть вывод:

Channel Client #8 took channel #771
Channel Client #10 took channel #550
Channel Client #12 took channel #337
Channel Client #14 took channel #442
Channel Client #9 took channel #883
Channel Client #9 : 883 channel released
Channel Client #16 took channel #883
Channel Client #12 : 337 channel released
Channel Client #18 took channel #337
Channel Client #10 : 550 channel released
Channel Client #11 took channel #550
Channel Client #11 : 550 channel released
Channel Client #13 took channel #550
Channel Client #18 : 337 channel released
Channel Client #15 took channel #337
Channel Client #14 : 442 channel released
Channel Client #17 took channel #442
Channel Client #8 : 771 channel released
Channel Client #20 took channel #771
Client #19 lost ->:превышено время ожидания
Client #26 lost ->:превышено время ожидания
Client #24 lost ->:превышено время ожидания
Client #22 lost ->:превышено время ожидания
Client #23 lost ->:превышено время ожидания
Client #25 lost ->:превышено время ожидания
Client #27 lost ->:превышено время ожидания
Client #21 lost ->:превышено время ожидания
Channel Client #16 : 883 channel released
Channel Client #13 : 550 channel released
Channel Client #17 : 442 channel released
Channel Client #15 : 337 channel released
Channel Client #20 : 771 channel released

Барьеры

Многие задачи могут быть разделены на подзадачи и выполняться параллельно. По достижении некоторой данной точки всеми параллельными потоками подводится итог и определяется общий результат. Если стоит задача задержать заданное число потоков до достижения ими определенной точки синхронизации, то используются классы-барьеры. После того, как все потоки достигли этой самой точки, они будут разблокированы и могут продолжать выполнение. Класс **CyclicBarrier** определяет минимальное число потоков, которое может быть остановлено барьером. Кроме этого барьер сам может быть проинициализирован потоком, который будет запускаться при снятии барьера. Методы **int await()** и **int await(long timeout, TimeUnit unit)** останавливают поток, использующий барьер до тех пор, пока число потоков достигнет заданного числа в классе-барьере. Метод **await()** возвращает порядковый номер достижения потоком барьерной точки. Метод **boolean isBroken()** проверяет состояние барьера. Метод **reset()** сбрасывает состояние барьера к моменту инициализации. Метод **int getNumberWaiting()** позволяет определить число ожидаемых барьером потоков до его снятия. Экземпляр **CyclicBarrier** можно использовать повторно.

Процесс проведения аукциона подразумевает корректное использование класса **CyclicBarrier**. Класс **Auction** определяет список конкурирующих предложений от клиентов и размер барьера. Чтобы приложение работало корректно, необходимо, чтобы размер списка совпадал со значением константы **BIDS_NUMBER**. Барьер инициализируется потоком определения победителя торгов, который запустится после того, как все предложения будут объявлены. Если потоков будет запущено больше чем размер барьера, то «лишние» предложения могут быть не учтены при вычислении победителя, если же потоков будет меньше, то приложение окажется в состоянии deadlock. Для предотвращения подобных ситуаций следует использовать метод **await()** с параметрами.

// # 23 # определение барьера и действия по его окончании # Auction.java

```
package by.bsu.auction;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.concurrent.CyclicBarrier;
public class Auction {
  private ArrayList<Bid> bids;
  private CyclicBarrier barrier;
  public final int BIDS_NUMBER = 5;
  public Auction() {
    this.bids = new ArrayList<Bid>();
      this.barrier = new CyclicBarrier(this.BIDS_NUMBER, new Runnable() {
        public void run() {
          Bid winner = Auction.this.defineWinner();
```

```
            System.out.println("Bid #" + winner.getBidId() + ", price:" + winner.getPrice() + " win!");
          }
        });
      }
      public CyclicBarrier getBarrier() {
        return barrier;
      }
      public boolean add(Bid e) {
        return bids.add(e);
      }
      public Bid defineWinner() {
          return Collections.max(bids, new Comparator<Bid>() {
                              @Override
                              public int compare(Bid ob1, Bid ob2) {
                                        return ob1.getPrice() - ob2.getPrice();
                              }
          });
     }
}
```

Класс **Bid** определяет предложение клиента на аукционе и запрашивает барьер, после которого клиент либо заплатит за лот, либо будет продолжать работать дальше.

// # 24 # поток, использующий барьер # Bid.java

```
package by.bsu.auction;
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class Bid extends Thread {
          private Integer bidId;
          private int price;
          private CyclicBarrier barrier;
          public Bid(int id, int price, CyclicBarrier barrier) {
                    this.bidId = id;
                    this.price = price;
                    this.barrier = barrier;
          }
          public Integer getBidId() {
                    return bidId;
          }
          public int getPrice() {
                    return price;
          }
          @Override
          public void run() {
                    try {
                              System.out.println("Client " + this.bidId + " specifies a price.");
                              Thread.sleep(new Random().nextInt(3000)); // время на раздумье
                              // определение уровня повышения цены
```

```
                    int delta = new Random().nextInt(50);
                    price += delta;
                    System.out.println("Bid " + this.bidId + " : " + price);
                    this.barrier.await(); // остановка у барьера
                    System.out.println("Continue to work..."); // проверить кто выиграл
                                        // и оплатить в случае победы ставки
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }
}
```

// # 25 # инициализация аукциона и его запуск # AuctionRunner.java

```
package by.bsu.auction;
import java.util.Random;
public class AuctionRunner {
        public static void main(String[ ] args) {
                Auction auction = new Auction();
                int startPrice = new Random().nextInt(100);
                for (int i = 0; i < auction.BIDS_NUMBER; i++) {
                    Bid thread = new Bid(i, startPrice, auction.getBarrier());
                    auction.add(thread);
                    thread.start();
                }
        }
}
```

Результаты работы аукциона:

Client 0 specifies a price.
Client 2 specifies a price.
Client 1 specifies a price.
Client 3 specifies a price.
Client 4 specifies a price.
Bid 4 : 87
Bid 0 : 81
Bid 1 : 93
Bid 2 : 81
Bid 3 : 96
Bid #3, price:96 win!
Continue to work...
Continue to work...
Continue to work...
Continue to work...
Continue to work...

«Щеколда»

Еще один вид барьера представляет класс **CountDownLatch**. Экземпляр класса инициализируется начальным значением числа ожидающих снятия «щеколды» потоков. В отличие от **CyclicBarrier**, метод **await()** просто останавливает поток без всяких изменений значения счетчика. Значение счетчика снижается вызовом метода **countDown()**, т. е. «щеколда» сдвигается на единицу. Когда счетчик обнулится, барьеры, поставленные методом **await()**, снимаются для всех ожидающих разрешения потоков. Крайне желательно, чтобы метод **await()** был вызван раньше, чем метод **countDown()**. Последнему безразлично, вызывался метод **await()** или нет, счетчик все равно будет уменьшен на единицу. Если счетчик равен нулю, то «лишние» вызовы метода **countDown()** будут проигнорированы.

Демонстрацией возможностей класса **CountDownLatch** может служить задача выполнения студентами набора заданий (тестов). *Студенту* предлагается для выполнения набор заданий. Он выполняет их и переходит в режим ожидания оценок по всем заданиям, чтобы вычислить среднее значение оценки. *Преподаватель* (**Tutor**) проверяет задание и после каждого проверенного задания сдвигает «щеколду» на единицу. Когда все задания студента проверены, счетчик становится равным нулю и барьер снимается, производятся необходимые вычисления в классе **Student**.

// # 26 # поток-студент, выполняющий задания и ожидающий их проверки # Student.java

```
package by.bsu.learning;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.Random;
public class Student extends Thread {
          private Integer idStudent;
          private List<Task> taskList;
          private CountDownLatch countDown;
          public Student(Integer idStudent, int numberTasks) {
                    this.idStudent = idStudent;
                    this.countDown = new CountDownLatch(numberTasks);
                    this.taskList = new ArrayList<Task>(numberTasks);
          }
          public Integer getIdStudent() {
                    return idStudent;
          }
          public void setIdStudent(Integer idStudent) {
                    this.idStudent = idStudent;
          }
          public CountDownLatch getCountDownLatch() {
                    return countDown;
          }
```

```
        public List<Task> getTaskList() {
                return taskList;
        }
        public void addTask(Task task) {
                        taskList.add(task);
        }
        public void run() {
                int i = 0;
                for (Task inWork : taskList) {
                        // на выполнение задания требуется некоторое время
                        try {
                                Thread.sleep(new Random().nextInt(100));
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        // отправка ответа
                        inWork.setAnswer("Answer #" + ++i);
                        System.out.println("Answer #" + i + " from " + idStudent);
                }
                try {
                        countDown.await(); // ожидание проверки заданий
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                // подсчет средней оценки за все задачи
                float averageMark = 0;
                for (Task inWork : taskList) {
                        // выполнение задания
                        averageMark += inWork.getMark(); // отправка ответа
                }
                averageMark /= taskList.size();
                System.out.println("Student " + idStudent + ": Average mark = "
                                + averageMark);
        }
}
```

// # 27 # поток-тьютор, проверяющий задания # Tutor.java

```
package by.bsu.learning;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
public class Tutor extends Thread {
        private Integer idTutor;
        private List<Student> list;
        public Tutor() {
                this.list = new ArrayList<>();
        }
        public Tutor(List<Student> list) {
                this.list = list;
        }
```

```
        public Integer getIdTutor() {
                return idTutor;
        }
        public void setIdTutor(Integer id) {
                this.idTutor = id;
        }
        public void run() {
                for (Student st : list) {
                        // проверить, выданы ли студенту задания
                        List<Task> tasks = st.getTaskList();
                        for (Task current : tasks) {
                                // проверить наличие ответа!
                                int mark = 3 + new Random().nextInt(7);
                                current.setMark(mark);
                                System.out.println(mark + " for student N "
                                                + st.getIdStudent());
                                st.getCountDownLatch().countDown();
                        }
                        System.out.println("All estimates made for " + st.getIdStudent());
                }
        }
}
```

// # 28 # класс-носитель информации # Task.java

```
package by.bsu.learning;
public class Task {
        private String content;
        private String answer;
        private int mark;
        public Task(String content) {
                this.content = content;
        }
        public String getContent() {
                return content;
        }
        public void setContent(String content) {
                this.content = content;
        }
        public String getAnswer() {
                return answer;
        }
        public void setAnswer(String answer) {
                this.answer = answer;
        }
        public int getMark() {
                return mark;
        }
        public void setMark(int mark) {
                this.mark = mark;
        }
}
```

/ # 29 # запуск формирования группы студентов и выполнения проверки их заданий # RunLearning.java */*

```
package by.bsu.learning;
import java.util.ArrayList;
public class RunLearning {
          public static void main(String[] args) {
                    final int NUMBER_TASKS_1 = 5;
                    Student student1 = new Student(322801, NUMBER_TASKS_1);
                    for (int i = 0; i < NUMBER_TASKS_1; i++) {
                              Task t = new Task("Task #" + i);
                              student1.addTask(t);
                    }
                    final int NUMBER_TASKS_2 = 4;
                    Student student2 = new Student(322924, NUMBER_TASKS_2);
                    for (int i = 0; i < NUMBER_TASKS_2; i++) {
                              Task t = new Task("Task ##" + i);
                              student2.addTask(t);
                    }
                    ArrayList<Student> lst = new ArrayList<Student>();
                    lst.add(student1);
                    lst.add(student2);
                    Tutor tutor = new Tutor(lst);
                    student1.start();
                    student2.start();
                    try { // поток проверки стартует с задержкой
                              Thread.sleep(5000);
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
                    tutor.start();
          }
}
```

В результате будет выведено:

Answer #1 from 322801
Answer #2 from 322801
Answer #1 from 322924
Answer #2 from 322924
Answer #3 from 322801
Answer #4 from 322801
Answer #5 from 322801
Answer #3 from 322924
Answer #4 from 322924
3 for student N%322801
6 for student N%322801
9 for student N%322801
3 for student N%322801

6 for student N%322801
All mark for 322801 accepted
9 for student N%322924
8 for student N%322924
3 for student N%322924
9 for student N%322924
All mark for 322924 accepted
Student 322924: Average mark = 7.25
Student 322801: Average mark = 5.4

Следует отметить, что в этом и предыдущих заданиях не полностью выполнены все условия по обеспечению корректной работы приложений во всех возможных ситуациях, возникающих в условиях многопоточности. Следует рассматривать данный пример в качестве каркаса для построения работоспособного приложения.

Обмен блокировками

Существует возможность безопасного обмена объектами, в том числе и синхронизированными. Функционал обмена представляет класс **Exchanger** с его единственным методом **T exchange(T ob)**. Возвращаемый параметр метода — объект, который будет принят из другого потока, передаваемый параметр **ob** метода — собственный объект потока, который будет отдан другому потоку.

Поток **Producer** представляет информацию о количестве произведенного товара, поток **Consumer** — о количестве проданного. В результате обмена производитель снизит план производства, если количество проданного товара меньше произведенного. Потребитель, к тому же, снижает цену на товар, так как поступления товара больше, чем продано за время, предшествующее обмену.

/ # 30 # содержит Exchanger и представляет основу для производителя и потребителя # Subject.java */*

```
package by.bsu.exchanger;
import java.util.concurrent.Exchanger;
public class Subject {
          protected static Exchanger<Item> exchanger = new Exchanger<>();
          private String name;
          protected Item item;
          public Subject(String name, Item item) {
                    this.name = name;
                    this.item = item;
          }
          public String getName() {
                    return name;
          }
```

```
        public Item getItem() {
                return item;
        }
        public void setName(String name) {
                this.name = name;
        }
        public void setItem(Item item) {
                this.item = item;
        }
}
```

/ # 31 # поток-производитель товара, обменивающегося информацией о планах производства с потребителем # Producer.java */*

```
package by.bsu.exchanger;
public class Producer extends Subject implements Runnable {
        public Producer(String name, Item item) {
                super(name, item);
        }
        public void run() {
                try {
                synchronized(item) { // блок синхронизации не нужен, но показателен
                        int proposedNumber = this.getItem().getNumber();
                        // обмен синхронизированными экземплярами
                        item = exchanger.exchange(item);
                        if (proposedNumber <= item.getNumber()) {
                                System.out.println("Producer " + getName()
                                + " повышает план производства товара");
                        } else {
                                System.out.println("Producer " + getName()
                                + " снижает план производства товара");
                        }
                }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

/ # 32 # поток-потребитель товара, обменивающийся уровнем продаж с производителем # Consumer.java */*

```
package by.bsu.exchanger;
public class Consumer extends Subject implements Runnable {
        public Consumer(String name, Item item) {
                super(name, item);
        }
        public void run() {
                try {
                synchronized(item) { // блок синхронизации не нужен, но показателен
```

```
                    int requiredNumber = item.getNumber();
                    item = exchanger.exchange(item); // обмен
                    if (requiredNumber >= item.getNumber()) {
                              System.out.println("Consumer " + getName()
                              + " повышает стоимость товара");
                    } else {
                              System.out.println("Consumer " + getName()
                              + " снижает стоимость товара");
                    }
          }
          } catch (InterruptedException e) {
                    e.printStackTrace();
          }
     }
}
```

/ # 33 # класс-описание товара # Item.java */*

```
package by.bsu.exchanger;
public class Item {
     private Integer id;
     private Integer number;
     public Item(Integer id, Integer number) {
          super();
          this.id = id;
          this.number = number;
     }
     public Integer getId() {
          return id;
     }
     public Integer getNumber() {
          return number;
     }
}
```

/ # 34 # процесс обмена # RunExchange.java */*

```
package by.bsu.exchanger;
public class RunExchange {
     public static void main(String[ ] args) {
          Item ss1 = new Item(34, 2200);
          Item ss2 = new Item(34, 2100);
          new Thread(new Producer("HP ", ss1)).start();
          new Thread(new Consumer("RETAIL Trade", ss2)).start();
     }
}
```

В результате будет получено:

Consumer RETAIL Trade снижает стоимость товара
Producer HP снижает план производства товара

Альтернатива synchronized

Синхронизация ресурса ключевым словом **synchronized** накладывает достаточно жесткие правила на освобождение этого ресурса. Интерфейс **Lock** представляет собой некоторое обобщение синхронизации. Появляется возможность провести опрос о блокировании, установить время ожидания блокировки и условия ее прерывания. Интерфейс также оптимизирует работу JVM с процессами конкурирования за освобождаемые ресурсы.

Класс **ReentrantLock** представляет два основных метода:

void lock() — получает блокировку экземпляра. Если экземпляр блокирован другим потоком, то поток отключается и бездействует до освобождения экземпляра;

void unlock() — освобождает блокировку экземпляра. Если текущий поток не является обладателем блокировки, генерируется исключение **IllegalMonitorStateException**.

Шаблонное применение этих методов после объявления экземпляра **locking** класса **ReentrantLock**:

```
        locking.lock();
try {
        // some code here
} finally {
        locking.unlock();
}
```

Данная конструкция копирует функциональность блока **synchronized**. Гибкость классу предоставляют методы:

boolean tryLock() — получает блокировку экземпляра. Если блокировка выполнена другим потоком, то метод сразу возвращает **false**;

boolean tryLock(long timeout, TimeUnit unit) — получает блокировку экземпляра. Если экземпляр блокирован другим потоком, то метод приостанавливает поток на время **timeout**, и если блокировка становится возможна в течение этого интервала, то поток ее получает, если же блокировка недоступна, то метод возвращает **false**.

Метод **lock()** и оба метода **tryLock()** при повторном успешном получении блокировки в одном и том же потоке увеличивают счетчик блокировок на единицу, что требует соответствующего числа снятий блокировки.

Класс **Condition** предназначен для управления блокировкой. Ссылку на экземпляр можно получить только из объекта типа **Lock** методом **newCondition()**. Расширение возможностей происходит за счет методов **await()** и **signal()**, функциональность которых подобна действию методов **wait()** и **notify()** класса **Object**.

Пусть необходим нереляционный способ сохранения информации в коллекции, когда неделимым квантом информации считается пара или более следующих друг за другом элементов. То есть добавление и удаление элементов может

осуществляться только парами и другой поток не может добавить/удалить свои элементы, пока заблокировавший коллекцию поток полностью не выполнит свои действия.

/ # 35 # ресурсы добавляются и удаляются только парами # DoubleResource.java */*

```
package by.bsu.lock;
import java.util.Deque;
import java.util.LinkedList;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class DoubleResource {
          private Deque<String> list = new LinkedList<String>();
          private Lock lock = new ReentrantLock();
          private Condition isFree = lock.newCondition();
          public void adding(String str, int i) {
                    try {
                              lock.lock();
                              list.add(i + "<" + str);
                              TimeUnit.MILLISECONDS.sleep(new Random().nextInt(50));
                              list.add(str + ">" + i);
                              isFree.signal();
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    } finally {
                              lock.unlock();
                    }
          }
          public String deleting() {
                    lock.lock();
                    String s = list.poll();
                    s += list.poll();
                    isFree.signal();
                    lock.unlock();
                    return s;
          }
          public String toString() {
                    return list.toString();
          }
}
```

/ # 36 # поток доступа к ресурсу # ResThread.java */*

```
package by.bsu.lock;
import java.util.Random;
public class ResThread extends Thread {
          private DoubleResource resource;
          public ResThread(String name, DoubleResource rs) {
```

```
                super(name);
                resource = rs;
        }
        public void run() {
                for (int i = 1; i < 4; i++) {
                        if (new Random().nextInt(2) > 0) {
                                resource.adding(getName(), i);
                        } else {
                                resource.deleting();
                        }
                }
        }
}
```

/ # 37 # запуск процессов доступа к ресурсу # SynchroMain.java */*

```
package by.bsu.lock;
import java.util.concurrent.TimeUnit;
public class SynchroMain {
        public static void main(String[ ] args) throws InterruptedException {
                for (int i = 0; i < 5; i++) {
                        DoubleResource resource = new DoubleResource();
                        new ResThread("a", resource).start();
                        new ResThread("Z", resource).start();
                        new ResThread("#", resource).start();
                        TimeUnit.MILLISECONDS.sleep(1_000);
                        System.out.println(resource);
                }
        }
}
```

В результате может быть получено:

[1<a, a>1, 2<a, a>2, 1<Z, Z>1, 1<#, #>1, 3<Z, Z>3, 3<#, #>3]
[]
[2<a, a>2, 3<a, a>3, 1<Z, Z>1, 3<Z, Z>3, 2<#, #>2]
[3<a, a>3, 3<Z, Z>3, 3<#, #>3]
[1<a, a>1, 2<a, a>2, 3<a, a>3, 3<#, #>3]

где результат с пустыми скобками свидетельствует, что попыток изъятия пар было больше, чем попыток добавления. Нерезультативные попытки не фиксировались.

ExecutorService и Callable

В альтернативной системе управления потоками разработан механизм исполнителей, функции которого заключаются в запуске отдельных потоков и их групп, а также в управлении ими: принудительной остановке, контроле числа работающих потоков и планирования их запуска.

Класс **ExecutorService** методом **execute(Runnable thread)** запускает традиционные потоки, метод же **submit(Callable<T> task)** запускает потоки с возвращаемым значением. Метод **shutdown()** останавливает все запущенные им ранее потоки и прекращает действие самого исполнителя. Статические методы **newSingleThreadExecutor()** и **newFixedThreadPool(int numThreads)** класса **Executors** определяют правила, по которым работает **ExecutorService**, а именно первый позволяет исполнителю запускать только один поток, второй — не более чем указано в параметре **numThreads**, ставя другие потоки в очередь ожидания окончания уже запущенных потоков.

/ # 38 # поток с возвращением результата # CalcCallable.java */*

```
package by.bsu.future;
import java.util.Random;
import java.util.concurrent.Callable;
public class CalcCallable implements Callable<Number> {
          @Override
          public Number call() throws Exception {
                    Number res = new Random().nextGaussian(); // имитация вычислений
                    return res;
          }
}
```

/ # 39 # запуск потока и извлечение результата его выполнения # CalcRunner.java */*

```
package by.bsu.future;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CalcRunner {
          public static void main(String[] args) {
                    ExecutorService es = Executors.newSingleThreadExecutor();
                    Future<Number> future = es.submit(new CalcCallable());
                    es.shutdown();
                    try {
                              System.out.println(future.get());
                    } catch (InterruptedException | ExecutionException e) {
                              e.printStackTrace();
                    }
          }
}
```

Интерфейс **Callable** представляет поток, возвращающий значение вызывающему потоку. Определяет один метод **V call() throws Exception**, в код реализации которого и следует поместить решаемую задачу. Результат выполнения метода **call()** может быть получен через экземпляр класса **Future**, методами **V get()** или **V get(long timeout, TimeUnit unit)**, как и продемонстрировано в предыдущем

примере. Перед извлечением результатов работы потока **Callable** можно проверить, завершилась ли задача успешно, методами **isDone()** и **isCancelled()** соответственно.

/ # 40 # список обрабатываемых объектов # ProductList.java */*

```
package by.bsu.future;
import java.util.ArrayDeque;
public class ProductList {
          private static ArrayDeque<String> arr = new ArrayDeque<String>() {
                    {
                              this.add("Product 1");
                              this.add("Product 2");
                              this.add("Product 3");
                              this.add("Product 4");
                              this.add("Product 5");
                    }
          };
          public static String getProduct() {
                    return arr.poll();
          }
}
```

/ # 41 # поток обработки экземпляра продукта # BaseCallable.java */*

```
package by.bsu.future;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;
public class BaseCallable implements Callable<String> {
          @Override
          public String call() throws Exception {
                    String product = ProductList.getProduct();
                    String result = null;
                    if (product != null) {
                              result = product + " done";
                    } else {
                              result = "productList is empty";
                    }
                    TimeUnit.MILLISECONDS.sleep(100);
                    System.out.println(result);
                    return result;
          }
}
```

/ # 42 # запуск пула потоков и извлечение результатов их работы # RunExecutor.java */*

```
package by.bsu.future;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
import java.util.concurrent.Future;
public class RunExecutor {
          public static void main(String[] args) throws Exception {
                    ArrayList<Future<String>> list = new ArrayList<Future<String>>();
                    ExecutorService es = Executors.newFixedThreadPool(3);
                    for (int i = 0; i < 7; i++) {
                              list.add(es.submit(new BaseCallable()));
                    }
                    es.shutdown();
                    for (Future<String> future : list) {
                              System.out.println(future.get() + " result fixed");
                    }
          }
}
```

В результате выполнения будет, возможно, выведено:

Product 3 done
Product 1 done
Product 2 done
Product 1 done result fixed
Product 3 done result fixed
Product 2 done result fixed
productList is empty
Product 5 done
Product 4 done
Product 4 done result fixed
Product 5 done result fixed
productList is empty result fixed
productList is empty
productList is empty result fixed

Phaser

Более сложное поведение этого синхронизатора **Phaser** напоминает поведение **CyclicBarrier**, однако число участников синхронизации может меняться. Участвующие стороны сначала должны зарегистрироваться phaser-объектом. Регистрация осуществляется с помощью методов **register()**, **bulkRegister(int parties)** или подходящего конструктора. Выход из синхронизации phaser-объектом производит метод **arriveAndDeregister()**, причем выход из числа синхронизируемых сторон может быть и в случае, когда поток завершил выполнение, и в случае, когда поток все еще выполняется. Основным назначением класса **Phaser** является синхронизация потоков, выполнение которых требуется разбить на отдельные этапы (фазы), а эти фазы, в свою очередь, необходимо синхронизовать. **Phaser** может как задержать поток, пока другие потоки не достигнут конца

текущей фазы методом **arriveAndAwaitAdvance()**, так и пропустить поток, отметив лишь окончание какой-либо фазы методом **arrive()**.

/ # 43 # поток # Truck.java */*

```
package by.bsu.phaser;
import java.util.ArrayDeque;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.Phaser;
public class Truck implements Runnable {
          private Phaser phaser;
          private String number;
          private int capacity;
          private Storage storafeFrom;
          private Storage storageTo;
          private Queue<Item> bodyStorage;
          public Truck(Phaser phaser, String name, int capacity, Storage stFrom,
                              Storage stTo) {
                    this.phaser = phaser;
                    this.number = name;
                    this.capacity = capacity;
                    this.bodyStorage = new ArrayDeque<Item>(capacity);
                    this.storafeFrom = stFrom;
                    this.storageTo = stTo;
                    this.phaser.register();
          }
          public void run() {
                    loadTruck();
                    phaser.arriveAndAwaitAdvance();

                    goTruck();
                    phaser.arriveAndAwaitAdvance();

                    unloadTruck();
                    phaser.arriveAndDeregister();
          }
          private void loadTruck() {
                    for (int i = 0; i < capacity; i++) {
                              Item g = storafeFrom.getGood();
                              if (g == null) { // если в хранилище больше нет товара,
                                                  // загрузка грузовика прерывается
                                        return;
                              }
                              bodyStorage.add(g);
                              System.out.println("Грузовик " + number + " загрузил товар №"
                                                  + g.getRegistrationNumber());
                              try {
                                        Thread.sleep(50);
                              } catch (InterruptedException e) {
                                        e.printStackTrace();
```

```
                    }
                }
        }
        private void unloadTruck() {
                int size = bodyStorage.size();
                for (int i = 0; i < size; i++) {
                        Item g = bodyStorage.poll();
                        storageTo.setGood(g);
                        System.out.println("Грузовик " + number + " разгрузил товар №"
                                        + g.getRegistrationNumber());
                        try {
                                Thread.sleep(50);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
        private void goTruck() {
                try {
                        Thread.sleep(new Random(100).nextInt(500));
                        System.out.println("Грузовик " + number + " начал поездку.");
                        Thread.sleep(new Random(100).nextInt(1000));
                        System.out.println("Грузовик " + number + " завершил поездку.");
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

/ # 44 # перевозимый товар # Item.java */*

```
package by.bsu.phaser;
public class Item {
        private int registrationNumber;
        public Item(int number) {
                this.registrationNumber = number;
        }
        public int getRegistrationNumber() {
                return registrationNumber;
        }
}
```

/ # 45 # склад-коллекция # Storage.java */*

```
package by.bsu.phaser;
import java.util.Iterator;
import java.util.List;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;
public class Storage implements Iterable<Item> {
```

```
        public static final int DEFAULT_STORAGE_CAPACITY = 20;
        private Queue<Item> goods = null;
        private Storage() {
                goods =
                new LinkedBlockingQueue<Item>(DEFAULT_STORAGE_CAPACITY);
        }
        private Storage(int capacity) {
                goods = new LinkedBlockingQueue<Item>(capacity);
        }
        public static Storage createStorage(int capacity) {
                Storage storage = new Storage(capacity);
                return storage;
        }
        public static Storage createStorage(int capacity, List<Item> goods) {
                Storage storage = new Storage(capacity);
                storage.goods.addAll(goods);
                return storage;
        }
        public Item getGood() {
                return goods.poll();
        }
        public boolean setGood(Item good) {
                return goods.add(good);
        }
        @Override
        public Iterator<Item> iterator() {
                return goods.iterator();
        }
}
```

/ # 46 # запуск процесса # PhaserDemo.java */*

```
package by.bsu.phaser;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.Phaser;
public class PhaserDemo {
        public static void main(String[] args) {
                // создание коллекцию товаров
                Item[] goods = new Item[20];
                for (int i = 0; i < goods.length; i++) {
                        goods[i] = new Item(i + 1);
                }
                List<Item> listGood = Arrays.asList(goods);
                // создание склада, из которого забирают товары
                Storage storageA = Storage.createStorage(listGood.size(), listGood);
                // создание склада, куда перевозят товары
                Storage storageB = Storage.createStorage(listGood.size());
                // создание фазера для синхронизации движения колонны грузовиков
```

```
                Phaser phaser = new Phaser();
                phaser.register();
                int currentPhase;
                // создание колонны грузовиков
                Thread tr1 = new Thread(new Truck(phaser, "tr1", 5, storageA, storageB));
                Thread tr2 = new Thread(new Truck(phaser, "tr2", 6, storageA, storageB));
                Thread tr3 = new Thread(new Truck(phaser, "tr3", 7, storageA, storageB));
                printGoodsToConsole("Товары на складе A", storageA);
                printGoodsToConsole("Товары на складе B", storageB);
                // запуск колонны грузовиков на загрузку на одном складе + поездку +
                // разгрузку на другом складе
                tr1.start();
                tr2.start();
                tr3.start();
                // синхронизация загрузки
                currentPhase = phaser.getPhase();
                phaser.arriveAndAwaitAdvance();
                System.out.println("Загрузка колонны завершена. Фаза " + currentPhase
                                + " завершена.");
                // синхронизация поездки
                currentPhase = phaser.getPhase();
                phaser.arriveAndAwaitAdvance();
                System.out.println("Поездка колонны завершена. Фаза " + currentPhase
                                + " завершена.");
                // синхронизация разгрузки
                currentPhase = phaser.getPhase();
                phaser.arriveAndAwaitAdvance();
                System.out.println("Разгрузка колонны завершена. Фаза " + currentPhase
                                + " завершена.");
                phaser.arriveAndDeregister();
                if (phaser.isTerminated()) {
                        System.out.println("Фазы синхронизированы и завершены.");
                }
                printGoodsToConsole("Товары на складе A", storageA);
                printGoodsToConsole("Товары на складе B", storageB);
        }
        public static void printGoodsToConsole(String title, Storage storage) {
                System.out.println(title);
                Iterator<Item> goodIterator = storage.iterator();
                while (goodIterator.hasNext()) {
                        System.out.print(goodIterator.next().getRegistrationNumber() + " ");
                }
                System.out.println();
        }
}
```

Товары на складе A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Товары на складе B
Грузовик tr1 загрузил товар №1

Грузовик tr2 загрузил товар №2
Грузовик tr3 загрузил товар №3
Грузовик tr2 загрузил товар №4
Грузовик tr1 загрузил товар №5
Грузовик tr3 загрузил товар №6
Грузовик tr2 загрузил товар №7
Грузовик tr1 загрузил товар №8
Грузовик tr3 загрузил товар №9
Грузовик tr2 загрузил товар №10
Грузовик tr3 загрузил товар №11
Грузовик tr1 загрузил товар №12
Грузовик tr2 загрузил товар №13
Грузовик tr3 загрузил товар №14
Грузовик tr1 загрузил товар №15
Грузовик tr2 загрузил товар №16
Грузовик tr3 загрузил товар №17
Грузовик tr3 загрузил товар №18
Загрузка колонны завершена. Фаза 0 завершена.
Грузовик tr2 начал поездку.
Грузовик tr1 начал поездку.
Грузовик tr3 начал поездку.
Грузовик tr1 завершил поездку.
Грузовик tr3 завершил поездку.
Грузовик tr2 завершил поездку.
Грузовик tr2 разгрузил товар №2
Грузовик tr1 разгрузил товар №1
Грузовик tr3 разгрузил товар №3
Поездка колонны завершена. Фаза 1 завершена.
Грузовик tr2 разгрузил товар №4
Грузовик tr1 разгрузил товар №5
Грузовик tr3 разгрузил товар №6
Грузовик tr2 разгрузил товар №7
Грузовик tr1 разгрузил товар №8
Грузовик tr3 разгрузил товар №9
Грузовик tr2 разгрузил товар №10
Грузовик tr1 разгрузил товар №12
Грузовик tr3 разгрузил товар №11
Грузовик tr2 разгрузил товар №13
Грузовик tr1 разгрузил товар №15
Грузовик tr3 разгрузил товар №14
Грузовик tr2 разгрузил товар №16
Грузовик tr3 разгрузил товар №17

Грузовик tr3 разгрузил товар №18
Разгрузка колонны завершена. Фаза 2 завершена.
Фазы синхронизированы и завершены.
Товары на складе A
19 20
Товары на складе B
2 3 1 4 5 6 7 8 9 10 12 11 13 15 14 16 17 18

Задания к главе 11

Вариант A

Разработать многопоточное приложение.

Использовать возможности, предоставляемые пакетом java.util.concurrent.

Не использовать слово synchronized.

Все сущности, желающие получить доступ к ресурсу, должны быть потоками.

Использовать возможности ООП.

Не использовать графический интерфейс. Приложение должно быть консольным.

1. **Порт**. Корабли заходят в порт для разгрузки/загрузки контейнеров. Число контейнеров, находящихся в текущий момент в порту и на корабле, должно быть неотрицательным и превышающим заданную грузоподъемность судна и вместимость порта. В порту работает несколько причалов. У одного причала может стоять один корабль. Корабль может загружаться у причала, разгружаться или выполнять оба действия.
2. **Маленькая библиотека**. Доступны для чтения несколько книг. Одинаковых книг в библиотеке нет. Некоторые выдаются на руки, некоторые только в читальный зал. Читатель может брать на руки и в читальный зал несколько книг.
3. **Автостоянка**. Доступно несколько машиномест. На одном месте может находиться только один автомобиль. Если все места заняты, то автомобиль не станет ждать больше определенного времени и уедет на другую стоянку.
4. **CallCenter**. В организации работает несколько операторов. Оператор может обслуживать только одного клиента, остальные должны ждать своей очереди. Клиент может положить трубку и перезвонить еще раз через некоторое время.
5. **Автобусные остановки**. На маршруте несколько остановок. На одной остановке может останавливаться несколько автобусов одновременно, но не более заданного числа.
6. **Свободная касса**. В ресторане быстрого обслуживания есть несколько касс. Посетители стоят в очереди в конкретную кассу, но могут перейти в другую очередь при уменьшении или исчезновении там очереди.

7. **Тоннель**. В горах существует два железнодорожных тоннеля, по которым поезда могут двигаться в обоих направлениях. По обоим концам тоннеля собралось много поездов. Обеспечить безопасное прохождение тоннелей в обоих направлениях. Поезд можно перенаправить из одного тоннеля в другой при превышении заданного времени ожидания на проезд.
8. **Банк.** Имеется банк с кассирами, клиентами и их счетами. Клиент может снимать/пополнять/переводить/оплачивать/обменивать денежные средства. Кассир последовательно обслуживает клиентов. Поток-наблюдатель следит, чтобы в кассах всегда были наличные, при скоплении денег более определенной суммы, часть их переводится в хранилище, при истощении запасов наличных происходит пополнение из хранилища.
9. **Аукцион.** На торги выставляется несколько лотов. Участники аукциона делают заявки. Заявку можно корректировать в сторону увеличения несколько раз за торги одного лота. Аукцион определяет победителя и переходит к следующему лоту. Участник, не заплативший за лот в заданный промежуток времени, отстраняется на несколько лотов от торгов.
10. **Биржа.** На торгах брокеры предлагают акции нескольких фирм. На бирже совершаются действия по купле-продаже акций. В зависимости от количества проданных-купленных акций их цена изменяется. Брокеры предлагают к продаже некоторую часть акций. От активности и роста-падения котировок акций изменяется индекс биржи. Биржа может приостановить торги при резком падении индекса.
11. **Аэропорт**. Посадка/высадка пассажиров может осуществляться через конечное число терминалов и наземным способом через конечное число трапов. Самолеты бывают разной вместимости и дальности полета. Организовать функционирование аэропорта, если пунктов назначения 4–6, и зон дальности 2–3.

Вариант В

Для заданий варианта В гл. 4 организовать синхронизированный доступ к ресурсам (файлам). Для каждого процесса создать отдельный поток выполнения.

Тестовые задания к главе 11

Вопрос 11.1.

Дан класс:

```
class InThread implements Runnable{
        public void run() {
                System.out.println("running...");
        }
}
```

Укажите правильные варианты создания потокового объекта (1):
1) **new** Thread().**new** InThread();
2) **new** Runnable(**new** InThread());
3) **new** Thread(Intread);
4) **new** Thread(**new** InThread());
5) **new** InThread().

Вопрос 11.2.

Укажите методы, определенные в классе java.lang.Thread (4):
1) join()
2) getPrioroty()
3) wait()
4) notifyAll()
5) sleep()
6) getName()

Вопрос 11.3.

Укажите состояния потока, при вызове на которых метод isAlive() класса java.lang.Thread вернет значение true (4):
1) NEW
2) RUNNABLE
3) BLOCKED
4) WAITING
5) TIMED_WAITING
6) TERMINATED

Вопрос 11.4.

Дан код:

```
class InThread implements Runnable{
        public void run() {System.out.println("running...");        }
}
public class Quest {
public static void main(String[] args) {
ExecutorService exec = Executors.newFixedThreadPool(2);
exec.execute(new InThread());   exec.execute(new InThread());
exec.execute(new InThread());   exec.execute(new InThread());
exec.execute(new InThread());   exec.execute(new InThread());
exec.shutdown(); while (!exec.isTerminated()) {   }
}}
```

Сколько потоков выполнит объект exec при запуске этого кода (1)?

1) 2
2) 4
3) 0
4) 6
5) столько, сколько успеет до завершения метода main()

Вопрос 11.5.

Дан класс Lamp(лампочка). Расставьте указанные ниже строки кода метода turnOn() так, чтобы не допустить ситуации включения лампочки, когда она уже включена (поле lamp имеет значение true, когда лампочка включена):

```
class Lamp {
        private boolean lamp = false;
        public synchronized void turnOn() throws InterruptedException {
                ____________________
}}
                a)  lamp = true;
                b)  notify();
                c)  while (lamp == true) wait();
```

Выберите один вариант (1):

1) abc;
2) bca;
3) acb;
4) cba;
5) cab.

Глава 12

JDBC

Пространство — иллюзия,
дисковое пространство — тем более.

Драйверы, соединения и запросы

API JDBC (Java DataBase Connectivity) — стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. 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 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, но присутствует драйвер ODBC.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Последовательность действий для выполнения первого запроса.

1. *Подключение библиотеки с классом-драйвером базы данных.*

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, поместив ее предварительно в папку **/lib** приложения.

mysql-connector-java-[*номер версии***]-bin.jar** для СУБД MySQL,
ojdbc[*номер версии*].**jar** для СУБД Oracle.

2. *Установка соединения с БД.*

Для установки соединения с БД вызывается статический метод **getConnection()** класса **java.sql.DriverManager**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Загрузка класса драйвера базы данных при отсутствии ссылки на экземпляр этого класса в JDBC 4.1 происходит автоматически при установке соединения экземпляром **DriverManager**. 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. Соответственно:

```
Connection cn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testphones",
                                                                    "root", "pass");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@//localhost:1521:testphones",
                                                                    "system", "pass");
```

В результате будет возвращен объект **Connection** и будет одно установленное соединение с БД с именем **testphones**. Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных. С помощью метода **getDrivers()** можно получить список всех доступных драйверов.

До появления JDBC 4.0 объект драйвера СУБД нужно было создавать явно с помощью вызова соответственно:

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("oracle.jdbc.OracleDriver");
```

или зарегистрировать драйвер

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

В большинстве случаев в этом нет необходимости, так как экземпляр драйвера загружается автоматически при попытке получения соединения с БД.

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.

4. *Выполнение запроса.*

Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов **execute(String sql)**, **executeBatch()**, **executeQuery(String sql)** или **executeUpdate(String sql)**. Результаты выполнения запроса помещаются в объект **ResultSet**:

```
/* выборка всех данных таблицы phonebook */
ResultSet rs = st.executeQuery("SELECT * FROM phonebook");
```

Для добавления, удаления или изменения информации в таблице запрос помещается в метод **executeUpdate()**.

5. *Обработка результатов выполнения запроса* производится методами интерфейса **ResultSet**, где самыми распространенными являются **next(), first(), previous(), last()** для навигации по строками таблицы результатов и группа методов по доступу к информации вида **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 имени в строке результатов методами типа **int getInt(String columnLabel)**, **String getString(String columnLabel)**, **Object getObject(String columnLabel)** и подобными им. Интерфейс располагает большим числом методов по доступу к таблице результатов, поэтому рекомендуется изучить его достаточно тщательно.

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, statement*

```
st.close(); // закрывает также и ResultSet
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. В Java 7 для объектов-ресурсов, требующих закрытия, реализована технология try with resources.

СУБД MySQL

СУБД MySQL совместима с JDBC и будет применяться для создания учебных баз данных. Версия СУБД может быть загружена с сайта **www.mysql.com**. Для корректной установки необходимо следовать инструкциям мастера. В процессе установки следует создать администратора СУБД с именем **root** и паролем, например, **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 базы данных и ее таблиц используются команды языка SQL.

Простое соединение и простой запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем **testphones** и одной таблицей **PHONEBOOK**. Таблица должна содержать три поля: числовое (первичный ключ) — **IDPHONEBOOK**, символьное — **LASTNAME** и числовое — **PHONE** и несколько занесенных записей.

IDPHONEBOOK	LASTNAME	PHONE
1	Каптур	7756544
2	Artukevich	6861880

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение символов кириллицы.

Приложение, осуществляющее простейший запрос на выбор всей информации из таблицы, выглядит следующим образом.

/ # 1 # простое соединение с БД и простой запрос # SimpleJDBCRunner.java */*

```
package by.bsu.data.main;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Properties;
import by.bsu.data.subject.Abonent;
```

```
public class SimpleJDBCRunner {
          public static void main(String[ ] args) {
                    String url = "jdbc:mysql://localhost:3306/testphones";
                    Properties prop = new Properties();
                    prop.put("user", "root");
                    prop.put("password", "pass");
                    prop.put("autoReconnect", "true");
                    prop.put("characterEncoding", "UTF-8");
                    prop.put("useUnicode", "true");
                    Connection cn = null;
                    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
                    try { // 1 блок
                              cn = DriverManager.getConnection(url, prop);
                              Statement st = null;
                              try { // 2 блок
                                        st = cn.createStatement();
                                        ResultSet rs = null;
                                        try { // 3 блок
                                            rs = st.executeQuery("SELECT * FROM phonebook");
                                            ArrayList<Abonent> lst = new ArrayList<>();
                                                  while (rs.next()) {
                                                            int id = rs.getInt(1);
                                                            int phone = rs.getInt(3);
                                                            String name = rs.getString(2);
                                                            lst.add(new Abonent(id, phone, name));
                                                  }
                                                  if (lst.size() > 0) {
                                                            System.out.println(lst);
                                                  } else {
                                                            System.out.println("Not found");
                                                  }
                                        } finally { // для 3-го блока try
                                                  /*
                                                   * закрыть ResultSet, если он был открыт
                                                   * или ошибка произошла во время
                                                   * чтения из него данных
                                                   */
                                                  if (rs != null) { // был ли создан ResultSet
                                                            rs.close();
                                                  } else {
                                                            System.err.println(
                                                            "ошибка во время чтения из БД");
                                                  }
                                        }
                              } finally {
                                        /*
                                         * закрыть Statement, если он был открыт или ошибка
                                         * произошла во время создания Statement
                                         */
                                        if (st != null) { // для 2-го блока try
```

```
                                        st.close();
                                } else {
                                        System.err.println("Statement не создан");
                                }
                        }
                } catch (SQLException e) { // для 1-го блока try
                        System.err.println("DB connection error: " + e);
                                        /*
                                        * вывод сообщения о всех SQLException
                                        */
                } finally {
                        /*
                         * закрыть Connection, если он был открыт
                        */
                        if (cn != null) {
                                try {
                                      cn.close();
                                } catch (SQLException e) {
                                      System.err.println("Connection close error: " + e);
                                }
                        }
                }
        }
}
```

В несложном приложении достаточно контролировать закрытие соединения, так как незакрытое или «провисшее» соединение снижает быстродействие системы.

Класс **Abonent**, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

/ # 2 # класс с информацией # Entity.java # Abonent.java */*

```
package by.bsu.data.subject;
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable {
          private int id;
          public Entity() {
          }
          public Entity(int id) {
                    this.id = id;
          }
          public int getId() {
                    return id;
          }
          public void setId(int id) {
                    this.id = id;
          }
}
```

```
package by.bsu.data.subject;
public class Abonent extends Entity {
          private int phone;
          private String lastname;
          public Abonent() {
          }
          public Abonent(int id, int phone, String lastname) {
                    super(id);
                    this.phone = phone;
                    this.lastname = lastname;
          }
          public int getPhone() {
                    return phone;
          }
          public void setPhone(int phone) {
                    this.phone = phone;
          }
          public String getLastname() {
                    return lastname;
          }
          public void setLastname(String lastname) {
                    this.lastname = lastname;
          }
          @Override
          public String toString() {
                    return "Abonent [id=" + id + ", phone=" + phone +
                    ", lastname=" + lastname + "]";
          }
}
```

Параметры соединения можно задавать несколькими способами: с помощью прямой передачи значений в коде класса, а также с помощью файлов **properties** или **xml**. Окончательный выбор производится в зависимости от конфигурации проекта.

Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Класс **ConnectorDB** использует файл ресурсов **database.properties**, в котором хранятся, как правило, параметры подключения к БД, такие, как логин и пароль доступа. Например:

```
db.driver   = com.mysql.jdbc.Driver
db.user     = root
db.password = pass
db.poolsize = 32
db.url  = jdbc:mysql://localhost:3306/testphones
db.useUnicode = true
db.encoding = UTF-8
```

Код класса **ConnectorDB** выглядит следующим образом:

/ # 3 # установка соединения с БД # ConnectorDB.java */*

```
package by.bsu.data.connect;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ResourceBundle;
public class ConnectorDB {
          public static Connection getConnection() throws SQLException {
                    ResourceBundle resource = ResourceBundle.getBundle("database");
                    String url = resource.getString("db.url");
                    String user = resource.getString("db.user");
                    String pass = resource.getString("db.password");
                    return DriverManager.getConnection(url, user, pass);
          }
}
```

В таком случае получение соединения с БД сведется к вызову

```
Connection cn = ConnectorDB.getConnection();
```

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 существует 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 при многократном выполнении однотипных запросов. Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

При использовании **PreparedStatement** невозможен sql injection attacks. То есть если существует возможность передачи в запрос информации в виде строки, то следует использовать для выполнения такого запроса объект **PreparedStatement**.

Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод **prepareStatement(String sql)** интерфейса **Connection**, возвращающий объект **PreparedStatement**.

```
String sql = "INSERT INTO phonebook(idphonebook, lastname, phone) VALUES(?, ?, ?)";
PreparedStatement ps = cn.prepareStatement(sql);
```

Установка входных значений конкретных параметров этого объекта производится с помощью методов **setString(int index, String x)**, **setInt(int index, int x)** и подобных им, после чего и осуществляется непосредственное выполнение запроса методами **int executeUpdate()**, **ResultSet executeQuery()**.

/ # 4 # подготовка запроса на добавление информации # DataBaseHelper.java */*

```
package by.bsu.data.connect;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import by.bsu.data.subject.Abonent;
public class DataBaseHelper {
          private final static String SQL_INSERT =
                         "INSERT INTO phonebook(idphonebook, lastname, phone ) VALUES(?,?,?)";
          private Connection connect;
                    public DataBaseHelper() throws SQLException {
                    connect = ConnectorDB.getConnection();
          }
          public PreparedStatement getPreparedStatement(){
                    PreparedStatement ps = null;
                    try {
                              ps = connect.prepareStatement(SQL_INSERT);
                    } catch (SQLException e) {
```

```
                    e.printStackTrace();
                }
                return ps;
        }
        public boolean insertAbonent(PreparedStatement ps, Abonent ab) {
                boolean flag = false;
                try {
                    ps.setInt(1, ab.getId());
                    ps.setString(2, ab.getName());
                    ps.setInt(3, ab.getPhone());
                    ps.executeUpdate();
                    flag = true;
                } catch (SQLException e) {
                    e.printStackTrace();
                }
                return flag;
        }
        public void closeStatement(PreparedStatement ps) {
                if (ps != null) {
                    try {
                        ps.close();
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
        }
}
```

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.

/ # 5 # добавление нескольких записей в БД # PreparedJDBCRunner.java */*

```
package by.bsu.data.main;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.ArrayList;
import by.bsu.data.connect.DataBaseHelper;
import by.bsu.data.subject.Abonent;
public class PreparedJDBCRunner {
        public static void main(String[] args) {
                ArrayList<Abonent> list = new ArrayList<Abonent>() {
                    {
                       add(new Abonent(87, 1658468, "Кожух Дмитрий"));
                       add(new Abonent(51, 8866711, "Буйкевич Александр"));
                    }
                };
                DataBaseHelper helper = null;
```

```
                    PreparedStatement statement = null;
                    try {
                              helper = new DataBaseHelper();
                              statement = helper.getPreparedStatement();
                              for (Abonent abonent : list) {
                                        helper.insertAbonent(statement, abonent);
                              }
                    } catch (SQLException e) {
                              e.printStackTrace();
                    } finally {
                              helper.closeStatement(statement);
                    }
          }
}
```

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 с помощью метода **registerOutParameter()**. После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Пусть в БД существует хранимая процедура **getlastname**, которая по уникальному номеру телефона для каждой записи в таблице **phonebook** будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getlastname (p_phone IN INT, p_lastname OUT VARCHAR) AS
BEGIN
SELECT lastname INTO p_lastname FROM phonebook WHERE phone = p_phone;
END
```

Тогда для получения имени через вызов данной процедуры необходимо исполнить java-код вида:

```
final String SQL = "{call getlastname (?, ?)}";
CallableStatement cs = cn.prepareCall(SQL);
```

```
// передача значения входного параметра
cs.setInt(1, 1658468);
// регистрация возвращаемого параметра
cs.registerOutParameter(2, java.sql.Types.VARCHAR);
cs.execute();
String lastName = cs.getString(2);
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
// turn off autocommit
cn.setAutoCommit(false);
Statement st = con.createStatement();
st.addBatch("INSERT INTO phonebook VALUES (55, 5642032, 'Гончаров')");
st.addBatch("INSERT INTO location VALUES (260, 'Minsk')");
st.addBatch("INSERT INTO student_department 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 **executeBatch()** интерфейса **PreparedStatement** возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует список объектов типа **Abonent** со стандартным набором методов **get***Tun***()/set***Tun***()** для каждого из его полей, и необходимо внести их значения в БД. Многократное выполнение методов **execute()** или **executeUpdate()** становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
          ArrayList<Abonent> abonents = new ArrayList<>();
          // заполнение списка
          PreparedStatement statement =
                    con.prepareStatement("INSERT INTO phonebook VALUES(?,?,?)");
          for (Abonent abonent : abonents) {
                    statement.setInt(0, abonent.getId());
                    statement.setInt(1, abonent.getPhone());
                    statement.setString(2, abonent.getLastname());
                    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, допускающая 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.

Транзакция, или деловая операция, определяется как единица работы, обладающая 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()**.

/ # 6 # транзакция по переводу денег со счета на счет # SingletonEngine.java */*

```
package com.epam.logic;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class SingletonEngine {
          private Connection connectionTo;
          private Connection connectionFrom;
          private static SingletonEngine instance = null;
          public synchronized static SingletonEngine getInstance() {
                    if (instance == null) {
                              instance = new SingletonEngine();
                              instance.getConnectionTo();
                              instance.getConnectionFrom();
                    }
                    return instance;
          }
          private Connection getConnectionFrom() {
                    try {
                              Class.forName("com.mysql.jdbc.Driver");
                              connectionFrom = DriverManager.getConnection(
                                        "jdbc:mysql://localhost:3306/testFrom", "root", "pass");
                              connectionFrom.setAutoCommit(false);
                    } catch (SQLException e) {
                              System.err.println("SQLException: " + e.getMessage()
                              + "SQLState: " + e.getSQLState());
                    } catch (ClassNotFoundException ex) {
                              System.out.println("Driver not found");
                    }
                    return connectionFrom;
          }
          private Connection getConnectionTo() {
                    final String connectToAdress = "jdbc:mysql://10.162.4.151:3306/testTo";
                    try {
                              Class.forName("com.mysql.jdbc.Driver");
                              connectionTo = DriverManager.getConnection(
                                                  connectToAdress, "root", "pass");
                              connectionTo.setAutoCommit(false);
                    } catch (SQLException e) {
                              System.err.println("SQLException: " + e.getMessage()
                              + "SQLState: " + e.getSQLState());
                    } catch (ClassNotFoundException e) {
                              System.err.println("Driver not found");
```

```
            }
            return connectionTo;
    }
    public void transfer(String summa) throws SQLException {
            Statement stFrom = null;
            Statement stTo = null;
            try {
                    int sum = Integer.parseInt(summa);
                    if (sum <= 0) {
                            throw new NumberFormatException("less or equals zero");
                    }
                    stFrom = connectionFrom.createStatement();
                    stTo = connectionTo.createStatement();
                    // транзакция из 4-х запросов
                    ResultSet rsFrom =
                            stFrom.executeQuery("SELECT balance from table_from");
                    ResultSet rsTo =
                            stTo.executeQuery("SELECT balance from table_to");
                            int accountFrom = 0;
                    while (rsFrom.next()) {
                            accountFrom = rsFrom.getInt(1);
                    }
                    int resultFrom= 0;
                    if (accountFrom >= sum) {
                            resultFrom = accountFrom - sum;
                    } else {
                            throw new SQLException("Invalid balance");
                    }
                    int accountTo = 0;
                    while (rsTo.next()) {
                            accountTo = rsTo.getInt(1);
                    }
                    int resultTo = accountTo + sum;
                    stFrom.executeUpdate(
                            "UPDATE table_from SET balance=" + resultFrom);
                    stTo.executeUpdate("UPDATE table_to SET balance=" + resultTo);
                    // завершение транзакции
                    connectionFrom.commit();
                    connectionTo.commit();
                    System.out.println("remaining on :" + resultFrom + " rub");
            } catch (SQLException e) {
                    System.err.println("SQLState: " + e.getSQLState()
                    + "Error Message: " + e.getMessage());
                    // откат транзакции при ошибке
                    connectionFrom.rollback();
                    connectionTo.rollback();
            } catch (NumberFormatException e) {
                    System.err.println("Invalid summa: " + summa);
            } finally {
                    if (stFrom != null) {
```

```
                                        try {
                                                stFrom.close();
                                        } catch (SQLException ex) {
                                                ex.printStackTrace();
                                        }
                                }
                                if (stTo != null) {
                                        try {
                                                stTo.close();
                                        } catch (SQLException ex) {
                                                ex.printStackTrace();
                                        }
                                }
                        }
                }
}
```

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;
- неповторяющееся 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 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, неповторяющееся и фантомное чтения;
- **TRANSACTION_READ_COMMITTED** — означает, что любое изменение, сделанное в транзакции, не видно вне ее, пока она не сохранена. Это предотвращает грязное чтение, но разрешает неповторяющееся и фантомное;
- **TRANSACTION_REPEATABLE_READ** — запрещает грязное и неповторяющееся чтение, но фантомное разрешено;

- **TRANSACTION_SERIALIZABLE** — определяет, что грязное, неповторяющееся 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 необходимый уровень.

Точки сохранения

Точки сохранения, представляемые классом **java.sql.Savepoint**, дают дополнительный контроль над транзакциями, привязывая изменения СУБД к конкретной точке в области транзакции. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных. Таким образом, если произойдет ошибка, можно вызвать метод **rollback(Savepoint point)** 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()** возвращают объект **Savepoint** интерфейса **Connection** и используются для установки именованой или неименованой точки сохранения во время текущей транзакции. При этом новая транзакция будет начата, если в момент вызова **setSavepoint()** не будет активной транзакции.

Data Access Object

При создании информационной системы выявляются некоторые слои, которые отвечают за взаимодействие различных частей приложения. Связь с базой данных является важной частью любой системы, поэтому всегда выделяется часть кода, ответственная за передачу запросов в БД и обработку полученных от нее ответов. Общее определение шаблона Data Access Object трактует его как прослойку между приложением и СУБД. DAO абстрагирует бизнес-сущности системы и отражает их на записи в БД. DAO определяет общие способы использования соединения с БД, моменты его открытия и закрытия или извлечения и возвращения в пул. В общем случае DAO можно определять таким образом, чтобы была возможность подмены одной модели базы

данных другой. Например: реляционную заменить на объектную или, что проще, MySQL на Oracle. В практическом программировании такие глобальные задачи ставятся крайне редко, поэтому будет приведено несколько способов организации взаимодействия с БД, отличающихся уровнем использования коннекта к БД и организацией работы с бизнес-сущностями.

Вершина иерархии DAO представляет собой класс или интерфейс с описанием общих методов, которые будут использоваться при взаимодействии с таблицей или группой таблиц. Как правило, это методы выбора, поиска сущности по признаку, добавление, удаление и замена информации.

/ # 7 # общие методы взаимодействия с моделью данных # AbstractDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import by.bsu.subject.Entity;
public abstract class AbstractDAO <K, T extends Entity> {
          public abstract List<T> findAll();
          public abstract T findEntityById(K id);
          public abstract boolean delete(K id);
          public abstract boolean delete(T entity);
          public abstract boolean create(T entity);
          public abstract T update(T entity);
}
```

Набор методов может варьироваться в зависимости от логики приложения. Параметр **K** описывает, как правило, ключ в таблице, так как редкая таблица, содержащая описание сущности, обходится без первичного колюча. Параметр **Entity** определяет общую бизнес-сущность, от которой наследуются все бизнес-сущности системы. Класс может содержать также методы возвращения соединения в пул или его закрытия, а также закрытия экземпляра **Statement** в виде:

```
public void close(Statement st) {
                    try {
                              if (st != null) {
                                        st.close();
                              }
                    } catch (SQLException e) {
                              // лог о невозможности закрытия Statement
                    }
          }
public void close(Connection connection) {
                    try {
                              if (connection != null) {
                                        connection.close();
```

```
                    }
                } catch (SQLException e) {
                    // генерация исключения, т.к. нарушается работа пула
                }
        }
```

DAO. Уровень метода

Реализация DAO для конкретного бизнес-объекта имеет шанс выглядеть следующим образом. Часть методов может остаться нереализованной, кроме того, могут добавляться собственные методы, определить которые в более общем классе невозможно из-за узкой области применения. В данном случае это метод **Abonent findAbonentByLastName(String name)**.

Реализация на уровне метода предполагает использование соединения для выполнения единственного запроса, т. е. соединение будет получено из пула в начале работы метода и возвращено по его окончании, что в общем случае не является экономным решением.

/ # 8 # конкретная реализация взаимодействия с моделью данных # AbonentDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import by.bsu.pool.ConnectionPool;
import by.bsu.subject.Abonent;
public class AbonentDAO extends AbstractDAO <Integer, Abonent> {
        public static final String SQL_SELECT_ALL_ABONENTS = "SELECT * FROM phonebook";
        public static final String SQL_SELECT_ABONENT_BY_LASTNAME =
                "SELECT idphonebook,phone FROM phonebook WHERE lastname=?";
        @Override
        public List<Abonent> findAll() {
                List<Abonent> abonents = new ArrayList<>();
                Connection cn = null;
                Statement st = null;
                try {
                        cn = ConnectionPool.getConnection();
                        st = cn.createStatement();
                        ResultSet resultSet =
                                st.executeQuery(SQL_SELECT_ALL_ABONENTS);

                        while (resultSet.next()) {
                                Abonent abonent = new Abonent();
                                abonent.setId(resultSet.getInt("idphonebook"));
```

```
                    abonent.setPhone(resultSet.getInt("phone"));
                    abonent.setLastName(resultSet.getString("lastname"));
                    abonents.add(abonent);
                }
        } catch (SQLException e) {
                System.err.println("SQL exception (request or table failed): " + e);
        } finally {
                close(st);
                // код возвращения экземпляра Connection в пул
        }
        return abonents;
}
@Override
public Abonent findEntityById(Integer id) {
        throw new UnsupportedOperationException();
}
@Override
public boolean delete(Integer id) {
        throw new UnsupportedOperationException();
}
@Override
public Abonent create(Abonent entity) {
        throw new UnsupportedOperationException();
}
@Override
public Abonent update(Abonent entity) {
        throw new UnsupportedOperationException();
}
// собственный метод DAO
public Abonent findAbonentByLastName(String name) {
        Abonent abonent = new Abonent();
        Connection cn = null;
        PreparedStatement st = null;
        try {
                cn = ConnectionPool.getConnection();
                st =
                cn.prepareStatement(SQL_SELECT_ABONENT_BY_LASTNAME);
                st.setString(1, name);
                ResultSet resultSet =st.executeQuery();
                resultSet.next();
                        abonent.setId(resultSet.getInt("idphonebook"));
                        abonent.setPhone(resultSet.getInt("phone"));
                        abonent.setLastName(name);
        } catch (SQLException e) {
                System.err.println("SQL exception (request or table failed): " + e);
        } finally {
                close(st);
                // код возвращения экземпляра Connection в пул
        }
        return abonent;
```

```
        }
        @Override
        public boolean delete(Abonent entity) {
                throw new UnsupportedOperationException();
        }
}
```

SQL-запросы размещаются в статических константах класса. В редких случаях запросы могут храниться вне системы в **xml** или **properties** файлах.

В данном примере использовался пул соединений, конфигурация которого задается в файле конфигурации **context.xml**.

/ # 9 # стандартный пул соединений # ConnectionPool.java */*

```
package by.bsu.pool;
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
public class ConnectionPool {
    private static final String DATASOURCE_NAME = "jdbc/testphones";
    private static DataSource dataSource;
    static {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context) initContext.lookup("java:/comp/env");
            dataSource = (DataSource) envContext.lookup(DATASOURCE_NAME);
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
    private ConnectionPool() { }
    public static Connection getConnection() throws SQLException {
        Connection connection = dataSource.getConnection();
        return connection;
    }
// метод возвращения Connection в пул
}
```

Особенности конфигурирования и использования стандартного пула соединений приведены в конце главы 16.

DAO. Уровень класса

Реализация на уровне класса подразумевает использование одного коннекта к базе данных для вызова нескольких методов конкретного DAO класса. В этом

случае вершина иерархии DAO в качестве поля может объявлять сам коннект к СУБД или его оболочку, кроме стандартного набора методов, например:

/ # 10 # DAO с полем Connection # AbstractDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Statement;
import by.bsu.action.WrapperConnector;
public abstract class AbstractDAO {
          protected WrapperConnector connector;
          // методы добавления, поиска, замены, удаления
          // методы закрытия коннекта и Statement
          public void close() {
                    connector.closeConnection();
          }
          protected void closeStatement(Statement statement) {
                    connector.closeStatement(statement);
          }
}
```

Реализация конкретного DAO при таком построении взаимодействия никогда в методе не должна закрывать соединение. Соединение закрывается в той части бизнес-логики, откуда выполняются обращения к DAO.

/ # 11 # DAO уровне класса # AbonentDAO.java */*

```
package by.bsu.simpledao;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import by.bsu.action.WrapperConnector;
import by.bsu.subject.Abonent;
public class AbonentDAO extends AbstractDAO {
          public static final String SQL_SELECT_ALL_ABONENTS =
                                                  "SELECT * FROM phonebook";
          public AbonentDAO() {
                    this.connector = new WrapperConnector();
          }
          public List<Abonent> findAll() {
                    List<Abonent> abonents = new ArrayList<>();
                    Statement st = null;
                    try {
                              st = connector.getStatement();
                              ResultSet resultSet =
                                        st.executeQuery(SQL_SELECT_ALL_ABONENTS);
                              while (resultSet.next()) {
                                        Abonent abonent = new Abonent();
                                        abonent.setId(resultSet.getInt("idphonebook"));
                                        abonent.setPhone(resultSet.getInt("phone"));
```

```
                                        abonent.setLastName(resultSet.getString("lastname"));
                                        abonents.add(abonent);
                                }
                        } catch (SQLException e) {
                                System.err.println("SQL exception (request or table failed): " + e);
                        } finally {
                                this.closeStatement(st);
                        }
                        return abonents;
                }
                // другие методы
}
```

Соединение с базой данных инициирует конструктор DAO, либо получает его из пула. В методе остаются возможности по созданию экземпляра **Statement** для выполнения запросов и его закрытию. В данной реализации использовался класс-обертка для соединения, инкапсулирующий процесс создания соединения и упрощающий его использование. Такой подход при организации пула соединений из экземпляров классов-оберток резко затрудняет попадание в пул «диких» соединений, созданных программистом в обход пула.

/ # 12 # класс-оболочка соединения # WrapperConnector.java */*

```
package by.bsu.action;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;
import java.util.MissingResourceException;
import java.util.Properties;
import java.util.ResourceBundle;
public class WrapperConnector {
          private Connection connection;
          public WrapperConnector() {
                    try {
                              ResourceBundle resource =
                                        ResourceBundle.getBundle("resource.database");
                              String url = resource.getString("url");
                              String user = resource.getString("user");
                              String pass = resource.getString("password");
                              Properties prop = new Properties();
                              prop.put("user", user);
                              prop.put("password", pass);
                              connection = DriverManager.getConnection(url, prop);
                    } catch (MissingResourceException e) {
                              System.err.println("properties file is missing " + e);
                    } catch (SQLException e) {
                              System.err.println("not obtained connection " + e);
                    }
```

```
        }
        public Statement getStatement() throws SQLException {
                if (connection != null) {
                        Statement statement = connection.createStatement();
                        if (statement != null) {
                                return statement;
                        }
                }
                throw new SQLException("connection or statement is null");
        }
        public void closeStatement(Statement statement) {
                if (statement != null) {
                        try {
                                statement.close();
                        } catch (SQLException e) {
                                System.err.println("statement is null " + e);
                        }
                }
        }
        public void closeConnection() {
                if (connection != null) {
                        try {
                                connection.close();
                        } catch (SQLException e) {
                                System.err.println(" wrong connection" + e);
                        }
                }
        }
        // другие необходимые делегированные методы интерфейса Connection
}
```

DAO. Уровень логики

На практике чаще всего возникает необходимость при выполнении запроса пользователя обращаться сразу к нескольким ветвям DAO и использовать при этом единственное соединение с БД. В этом случае соединение с БД создается или извлекается из пула до создания экземпляров DAO, а закрывается, соответственно, после выполнения всех обращений к БД.

/ # 13 # DAO с полем Connection без прямой инициализации # AbstractDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import by.bsu.subject.Entity;
```

```
public abstract class AbstractDAO <T extends Entity> {
          protected Connection connection;
          public AbstractDAO(Connection connection) {
                    this.connection = connection;
          }
          public abstract List<T> findAll();
          public abstract T findEntityById(int id);
          public abstract boolean delete(int id);
          public abstract boolean delete(T entity);
          public abstract boolean create(T entity);
          public abstract T update(T entity);
          public void close(Statement st) {
                    try {
                              if (st != null) {
                                        st.close();
                              }
                    } catch (SQLException e) {
                              // лог о невозможности закрытия Statement
                    }
          }
}
```

/ # 14 # примерные реализации DAO # AbonentDAO.java # PaymentDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import by.bsu.subject.Abonent;
public class AbonentDAO extends AbstractDAO <Abonent> {
          public AbonentDAO(Connection connection) {
                    super(connection);
          }
          // реализации методов
}
package by.bsu.simpledao;
import java.sql.Connection;
import java.util.List;
import by.bsu.subject.Payment;
public class PaymentDAO extends AbstractDAO <Payment> {
          public PaymentDAO(Connection connection) {
                    super(connection);
          }
          // реализации методов
}
```

где **Payment** представляет класс-сущность в виде подкласса класса **Entity**.

Схематически применение этого подхода можно увидеть в методе **doLogic()** класса **SomeLogic**.

/ # 15 # использование DAO на уровне логики # SomeLogic.java */*

```
package by.bsu.logic;
import java.sql.Connection;
import java.sql.SQLException;
import by.bsu.pool.ConnectionPool;
import by.bsu.simpledao.AbonentDAO;
import by.bsu.simpledao.PaymentDAO;
import by.bsu.subject.Abonent;
import by.bsu.subject.Payment;
public class SomeLogic {
          public void doLogic(int id) throws SQLException {
                    // 1. создание-получение соединения
                    Connection conn = ConnectionPool.getConnection();
                    // 2. открытие транзакции
                    conn.setAutoCommit(false);
                    // 3. инициализация необходимых экземпляров DAO
                    AbonentDAO abonentDAO = new AbonentDAO(conn);
                    PaymentDAO paymentDAO = new PaymentDAO(conn);
                    // 4. выполнение запросов
                    abonentDAO.findAll();
                    paymentDAO.findEntityById(id);
                    paymentDAO.delete(id);
                    // 5. закрытие транзакции
                    conn.commit();
                    // 6. закрытие-возвращение соединения
                    ConnectionPool.close(conn);
          }
}
```

Задания к главе 12

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 модификацию информации.

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 магазине.
   Для часов необходимо хранить:
   — марку;
   — тип (кварцевые, механические);
   — цену;
   — количество;
   — реквизиты производителя.
   Для производителей необходимо хранить:
   — название;
   — страна.
   - 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 гл. 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 12

Вопрос 12.1.

Какие пакеты содержат интерфейсы и классы JDBC (1)?

1) java.db и javax.db
2) java.sql и javax.sql

3) java.jdbc.sql и javax.jdbc.sql
4) org.sql и org.jdbc
5) java.jdbc и javax.jdbc

Вопрос 12.2.

Даны операторы языка Java:

a) Connection cn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306", "root", "pass");
b) ResultSet rs = st.executeQuery("SELECT * FROM users WHERE id=1");
c) Class.*forName*("org.gjt.mm.mysql.Driver");
d) Statement st = cn.createStatement();
e) System.*out*.println(rs.next());

Расставьте их в правильной последовательности так, чтобы с помощью получившегося кода можно было извлечь данные из БД (1):

1) abdec;
2) cadbe;
3) cabde;
4) ebdac;
5) abced.

Вопрос 12.3.

Какие типы statement-объектов существуют в JDBC (3)?

1) Statement
2) ResultStatement
3) PreparedStatement
4) DriverStatement
5) MetaDataStatement
6) CallableStatement

Вопрос 12.4.

Дана база данных test с таблицей users:

id	name
1	Ivanov

Какая информация добавится в таблицу users после выполнения следующего кода (2)?

```
Connection cn = /* корректное получение соединения */;
Statement st = cn.createStatement();
st.executeUpdate("INSERT INTO users VALUES (2, 'Petrov')");
cn.setAutoCommit(false);
```

```
st.executeUpdate("INSERT INTO users VALUES (3, 'Sidorov')");
cn.setSavepoint("point1");
st.executeUpdate("INSERT INTO users VALUES (4, 'Vasechkin')");
cn.rollback();
st.executeUpdate("INSERT INTO users VALUES (5, 'Blinov')");
cn.commit();
```

1) id=2, name=Petrov
2) id=3, name=Sidorov
3) id=4, name=Vasechkin
4) id=5, name=Blinov

Вопрос 12.5.

Укажите, каким способом можно выполнить хранимую процедуру с помощью JDBC (3):

1) вызвать метод execute() на объекте CallableStatement
2) вызвать метод executeQuery() на объекте CallableStatement
3) вызвать метод executeUpdate() на объекте CallableStatement
4) вызвать метод executeProcedure() на объекте CallableStatement
5) вызвать метод execute() на объекте StoredStatement
6) вызвать метод executeQuery() на объекте StoredStatement
7) вызвать метод executeUpdate() на объекте StoredStatement
8) вызвать метод executeProcedure() на объекте StoredStatement

Глава 13

СЕТЕВЫЕ ПРОГРАММЫ

Если неправильно набрать номер,
никогда не будет гудков «занято».

Загадка Ковака

Поддержка Интернета

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 интернет-приложения, к которым относятся веб-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(s) — 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. IP-адрес существует в двух видах: IPv4 и IPv6. Формат IPv4 представлен 32-битовым числом, обычно записываемым как четыре числа, разделенные

точками, каждое из которых изменяется от **0** до **255**, например **217.21.43.10**. Формат IPv6 представлен 128-битовым числом, обычно записываемым как восемь шестнадцатеричных чисел, разделенных двоеточиями, например **1170:0:0:0:7:771:100A:214B**. 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 Интернету 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 интернет-ресурсу 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 интернет-адрес, необязательный номер порта и путь к каталогу, содержащему файл, например:

```
http://www.oracle.com/download.jsp
```

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, например *имя_страницы#имя_ссылки*.

Определить IP-адрес в приложеннии можно с помощью объекта класса **java.net.InetAddress**. Можно также использовать и специфические классы **Inet4Address** и **Inet6Address**.

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**.

/ # 1 # вывод IP-адреса компьютера и Интернет-ресурса # InetLogic.java*/*

```
package by.bsu.net;
import java.net.InetAddress;
public class InetLogic {
          public static void main(String[] args) {
                    InetAddress currentIP = null;
                    InetAddress bsuIP = null;
                    try {
                              currentIP = InetAddress.getLocalHost();
                              // вывод IP-адреса локального компьютера
                              System.out.println("Мой IP -> " + currentIP.getHostAddress());
                              bsuIP = InetAddress.getByName("www.bsu.by");
                              // вывод IP-адреса БГУ www.bsu.by
                              System.out.println("BSU -> " + bsuIP.getHostAddress());
                    } catch (UnknownHostException e) {
                              // если не удается найти IP
                              System.err.println("адрес недоступен " + e);
                    }
          }
}
```

В результате будет выведено, например:

Мой IP -> 172.17.16.14
BSU -> 217.21.43.10

Метод **getLocalHost()** класса **InetAddress** создает объект **currentIP** и возвращает IP-адрес компьютера.

/ # 2 # присваивание фиктивного имени реальному ресурсу, заданному через IP # UnCheckedHost.java */*

```
package by.bsu.net;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
public class UnCheckedHost {
          public static void main(String[] args) {
                    // задание IP-адреса 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.err.println("адрес недоступен " + e);
                    } catch (IOException e) {
                              System.err.println("ошибка потока " + e);
                    }
          }
}
```

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 Интернете. 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()**.

/ # 3 # запуск страницы из апплета # ShowDocument.java */*

```
package by.bsu.net;
import java.awt.Graphics;
import java.net.MalformedURLException;
import java.net.URL;
import javax.swing.JApplet;
public class ShowDocument extends JApplet {
          private URL bsu = null;
          public String getBaseURL() {
                    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(getBaseURL());
                                this.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 примера апплет следует запускать из браузера, используя следующий HTML-документ:

```
<html>
<body align=center>
<applet code= by.bsu.net.ShowDocument.class></applet>
</body></html>
```

В следующей программе читается содержимое HTML-файла по указанному адресу и выводится в окно консоли.

/ # 4 # чтение документа из Интернета # ReadDocument.java */*

```
package by.bsu.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
public class ReadDocument {
          public static void main(String[] args) {
                    URL bsu = null;
                    String urlName = "http://www.bsu.by";
                    try {
                              bsu = new URL(urlName);
                    } catch (MalformedURLException e) {
                    // некорректно заданы протокол, доменное имя или путь к файлу
                              e.printStackTrace();
                    }
                    if (bsu == null) {
                              throw new RuntimeException();
                    }
                    try (BufferedReader d = new BufferedReader(new InputStreamReader(
```

```
                                        bsu.openStream()))) {
                                String line = "";
                                while ((line = d.readLine()) != null) {
                                        System.out.println(line);
                                }
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
}
```

Для получения более полной информации о ресурсе требуется применять класс **URLConnection**. Для получения экземпляра следует вызвать на экземпляре класса **URL** метод **openConnection()**. Далее при необходимости можно указать значения некоторых свойств. Для установления соединения на полученном экземпляре **URLConnection** вызывается метод **connect()**. При вызове этого метода возможна серьезная блокировка основного потока, поэтому перед попыткой установления соединения следует установить таймаут на соединение и/или на чтение методами **setConnectTimeout(int timeout)** и **setReadTimeout(int timeout)**.

/ # 5 # инфомация об Интернет-ресурсе # ConnectionDemo.java */*

```
package by.bsu.net;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
public class ConnectionDemo {
          public static void main(String[] args) {
                    String urlName = "http://www.oracle.com";
                    int timeout = 10_000_000;
                    URL url;
                    try {
                              url = new URL(urlName);
                              final URLConnection connection = url.openConnection();
                              // установка таймаута на соединение
                              connection.setConnectTimeout(timeout);
                              System.out.println(urlName + " content type: "
                                                  + connection.getContentType());
                              // продолжение извлечения информации из соединения
                    } catch (MalformedURLException e) {
                              e.printStackTrace();
                    } catch (IOException e) {
                              e.printStackTrace();
                    }
          }
}
```

В результате при успешной установке соединения будет выведено:

http://www.oracle.com content type: text/html;charset=ISO-8859-1

Сокетные соединения по протоколу TCP/IP

Сокеты (сетевые разъемы) — это логическое понятие, соответствующее разъемам, 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 socketClient = 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 socketServ = 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, после чего разрывает связь.

/ # 6 # передача клиенту строки # SmallServerSocket.java */*

```
package by.bsu.net;
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket
public class SmallServerSocket {
          public static void main(String[] args) {
                    Socket s = null;
                    PrintStream ps = null;
                    try { // отправка строки клиенту
                              // создание объекта и назначение номера порта
                              ServerSocket server = new ServerSocket(8030);
                              s = server.accept(); // ожидание соединения
                              ps = new PrintStream(s.getOutputStream());
                              // помещение строки "привет!" в буфер
                              ps.println("привет!");
                              // отправка содержимого буфера клиенту
                              ps.flush();
                    } catch (IOException e) {
                              System.err.println("Ошибка соединения потока: " + e);
                    } finally {
                        if (ps != null) {
                              ps.close();
                        }
                        if (s != null) {
                              try { // разрыв соединения
                                        s.close();
                              } catch (IOException e) {
                                        e.printStackTrace();
                              }
                        }
                    }
          }
}
```

/ # 7 # получение клиентом строки # SmallClientSocket.java */*

```
package by.bsu.net;
import java.io.*;
import java.net.*;
public class SmallClientSocket {
          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 message = br.readLine();
                        System.out.println(message);
                } catch (IOException e) {
                        System.err.println("ошибка: " + e);
                } finally {
                    if (br != null) {
                        br.close();
                    }
                        if (socket != null) {
                        try { // разрыв соединения
                                socket.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                    }
                }
        }
}
```

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.

/ # 8 # сервер для множества клиентов # NetServerThread.java */*

```
package by.bsu.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
public class NetServerThread {
          public static void main(String[] args) {
                    try {
                              ServerSocket server = new ServerSocket(8071);
                              System.out.println("initialized");
                              while (true) {
                                        // ожидание клиента
                                        Socket socket = server.accept();
                                        System.out.println(socket.getInetAddress().getHostName()
                                                  + " connected");
                                        /*
                                         * создание отдельного потока для обмена данными
                                         * с соединившимся клиентом
                                         */
                                        ServerThread thread = new ServerThread(socket);
                                        // запуск потока
                                        thread.start();
                              }
                    } catch (IOException e) {
                              System.err.println(e);
                    }
          }
}
class ServerThread extends Thread {
          private PrintStream os; // передача
          private BufferedReader is; // прием
          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.err.println("Disconnect");
```

```
                } finally {
                        disconnect(); // уничтожение потока
                }
        }
        public void disconnect() {
                try {
                        if (os != null) {
                                os.close();
                        }
                        if (is != null) {
                                is.close();
                        }
                        System.out.println(addr.getHostName() + " disconnecting");
                } catch (IOException e) {
                        e.printStackTrace();
                } finally {
                        this.interrupt();
                }
        }
}
```

Сервер передает сообщение клиенту. Для клиентских приложений поддержка многопоточности также необходима. Например, один поток ожидает выполнения операции ввода/вывода, а другие потоки выполняют свои функции.

/ # 9 # получение и передача сообщения клиентом в потоке # NetClient.java */*

```
package by.bsu.net;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
public class NetClient {
        public static void main(String[] args) {
                Socket socket = null;
                BufferedReader br = null;
                try {
                        // установка соединения с сервером
                        socket = new Socket(InetAddress.getLocalHost(), 8071);
                        // или Socket socket = new Socket("ИМЯ_СЕРВЕРА", 8071);
                        PrintStream ps = new PrintStream(socket.getOutputStream());
                        br = new BufferedReader(new InputStreamReader(
                                                        socket.getInputStream()));
                        for (int i = 1; i <= 10; i++) {
                                ps.println("PING");
                                System.out.println(br.readLine());
                                Thread.sleep(1_000);
```

```
                    }
                } catch (UnknownHostException e) {
                    // если не удалось соединиться с сервером
                    System.err.println("адрес недоступен" + e);
                } catch (IOException e) {
                    System.err.println("ошибка I/O потока" + e);
                } catch (InterruptedException e) {
                    System.err.println("ошибка потока выполнения" + e);
                } finally {
                    if (br != null) {
                        try {
                            br.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                    if (socket!= null) {
                        try {
                            socket.close();
                        } catch (IOException e) {
                            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

Протокол UDP (User Datagram Protocol) не устанавливает виртуального соединения и не гарантирует доставки 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.

Конструктор **DatagramPacket(byte[] buf, int length)** 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. Конструктор **DatagramPacket(byte[] buf, int length, InetAddress address, int port)** используется для отправки датаграмм.

Класс **DatagramSocket** может выступать в роли клиента и сервера, т. е. он способен получать и отправлять пакеты. Отправить пакет можно с помощью метода **send(DatagramPacket pac)**, для получения пакета используется метод **receive(DatagramPacket pac)**.

/ # 10 # передача файла по протоколу UDP # UDPSender.java */*

```
package by.bsu.net;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;
public class UDPSender {
          public static void main(String[] args) {
                    String fileName = "audio/toxic.mp3";
                    try (FileInputStream fr = new FileInputStream(new File(fileName))) {
                              byte[] data = new byte[1024];
                              DatagramSocket dSocket = new DatagramSocket();
                              InetAddress address = InetAddress.getLocalHost();
                              DatagramPacket packet;
                              while (fr.read(data) != -1) {
                                        // создание пакета данных
                                  packet = new DatagramPacket(data, data.length, address, 8033);
                                  dSocket.send(packet);// передача пакета
                              }
                              System.out.println("Файл успешно отправлен");
                    } catch (UnknownHostException e) {
                              // неверный адрес получателя
                              e.printStackTrace();
                    } catch (SocketException e) {
                              // возникли ошибки при передаче данных
                              e.printStackTrace();
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    } catch (IOException e) {
                              e.printStackTrace();
                    }
          }
}
```

/ # 11 # прием данных по протоколу UDP # UDPRecipient.java */*

```
package by.bsu.net;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketTimeoutException;
public class UDPRecipient {
          public static void main(String[] args) {
                    File file = new File("UDPtoxic.mp3");
                    System.out.println("Прием данных...");
                    // прием файла
                    acceptFile(file, 8033, 1024);
          }
          private static void acceptFile(File file, int port, int pacSize)  {
                    byte data[] = new byte[pacSize];
                    DatagramPacket pac = new DatagramPacket(data, data.length);
                    try (FileOutputStream os = new FileOutputStream(file)) {
                              DatagramSocket s = new DatagramSocket(port);
                              /* установка времени ожидания: если в течение 60 секунд
                              не принято ни одного пакета, прием данных заканчивается */
                              s.setSoTimeout(60_000);
                              while (true) {
                                        s.receive(pac);
                                        os.write(data);
                                        os.flush();
                              }
                    } catch (SocketTimeoutException e) {
                              // если время ожидания вышло
                       System.err.println("Истекло время ожидания, прием данных закончен");
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    } catch (IOException e) {
                              e.printStackTrace();
                    }
          }
}
```

Задания к главе 13

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 гл. 4 на базе сокетных соединений разработать сетевой вариант системы. Для каждого пользователя должно быть создано клиентское приложение, соединяющееся с сервером.

Тестовые задания к главе 13

Вопрос 13.1.

Выберите правильные утверждения (2):

1) протокол TCP не гарантирует доставку пакетов;
2) протокол UDP не гарантирует доставку пакетов;
3) протоколы TCP и UDP используются в сервисах, где важна гарантированная доставка сообщений;
4) протоколы TCP и UDP используются в сервисах, где важна быстрая доставка сообщений;

5) протокол TCP используется в сервисах, где важна гарантированная доставка сообщений, протокол UDP используется в сервисах, где важна быстрая доставка сообщений;
6) протокол UDP используется в сервисах, где важна гарантированная доставка сообщений, протокол TCP используется в сервисах, где важна быстрая доставка сообщений.

Вопрос 13.2.

IP сервиса google.ru 173.194.35.184. Укажите, с помощью каких методов класса java.net.InetAddress можно это выяснить (2):
1) getHostAddress()
2) getCanonicalHostName()
3) getAddress()
4) getHostName()
5) getLocalHost()

Вопрос 13.3.

Дан код:

```
try {
        ServerSocket server = new ServerSocket(1234);
        Socket sock = server.accept();
} catch (UnknownHostException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
}
```

Укажите, скольких клиентов может обслужить запущенный таким образом сервер (1):
1) ни одного, произойдет ошибка компиляции;
2) сколько угодно;
3) сколько угодно, но не больше, чем разрешено подключений на порт 1234;
4) одного клиента.

Вопрос 13.4.

Укажите, какая исключительная ситуация возникает при выполнении оператора Socket s = **new** Socket("15.12.14.16", 3456); в случае, если соединение с сервером не установлено (1):
1) MalformedURLException
2) UnknownHostException
3) UnknownURLException

4) ConnectException
5) UnknownPortException

Вопрос 13.5.

Дан код:

```
public class Quest {
public static void main(String[] args) {
new Thread() {
        public void run() {
                MyServer.main();
}}.start();
new Thread() {
        public void run() {
                try {
                        sleep(1000);
                        MyClient.main();
                } catch (InterruptedException e) {
                        e.printStackTrace();
}}}.start();}
}
class MyServer {
public static void main() {
try {
 ServerSocket server = ServerSocketFactory.getDefault().createServerSocket(12345);
 Socket socket = server.accept();
 Thread.sleep(1000);
 server.close();
 socket.close();
} catch (IOException e) {
        System.out.println("IOException on server.");
} catch (InterruptedException e) {
        System.out.println("InterruptedException on server.");
} } }
class MyClient {
public static void main() {
try {
        Socket s = SocketFactory.getDefault().createSocket("localhost",12345);
        System.out.print(s.isConnected()); Thread.sleep(5000);
        System.out.println(s.isConnected());
} catch (IOException e) {
        System.out.println("IOException on client.");
```

```
} catch (InterruptedException e){
        System.out.println("InterruptedException on client.");
} } }
```

Какая строка выведется на консоль в результате запуска этого кода, если исключительных ситуаций при его работе не было (1)?

1) truetrue
2) truefalse
3) falsetrue
4) falsefalse
5) нет правильного

Контрольная работа к части 2

Вариант Е

ФИ_____ « __»гр.

0. Записать точное регулярное выражение, соответствующее строке вида: *ФамилияИО_XX*, где *X* — цифра. Например: *ИвановАБ_89*, *ПетровскийТА_01*. Обязательно использовать хотя бы один класс предопределенных символов.
1. В пакете com.bsu создать подкласс Point2D записанного ниже класса

```
package com;
public class Point 1D{
        protected int x;
        Point1D(int x1) {x=x1;}
}
```

2. Создать класс NewString, реализующий интерфейс и наследующий абстрактный класс

```
interface IString {
        void hello(); }
abstract class AbstractString{
        protected abstract void hello();
}
```

3. Создать экземпляр анонимного класса на основе класса AbstractHope

```
abstract class AbstractHope {
        AbstractHope (int counter){}
        abstract boolean sell();
}
```

4. Сколько отличающихся друг от друга потоков будет запущено в цикле?

```
for(int i=0, j=10; i<=j; i++,--j) {
        if(i<j-2) {
          new Thread(new Thread()).start(); //в этой строке ошибок нет
        }}
```

5. Создать экземпляр класса Inner

```
class Owner {
  static int b = 3;
  public class Inner {
        static int c = 1;
}}
```

6. Сколько объектов будет создано после выполнения кода? Ответ пояснить.

```
String str[] = new String[2];
```

7. Реализовать метод hashCode для класса (сигнатура метода должна быть точной)

```
class Point2D{
        int x,y;
}
```

Глава 14

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 хранящиеся в этом формате данные и быстро производить базовые операции над этими данными.

XML является упрощенным 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 — текстовое представление информации 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 названия книги, его можно создать:

```
<title>Java SE 8</title>
<title book="Java SE 8"/>
<title-book>Java SE 8</title-book>
```

Каждый документ начинается декларацией — строкой, указывающей как минимум версию стандарта XML. В качестве других атрибутов могут быть указаны кодировка символов и внешние связи.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

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 все, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы.

Все атрибуты тегов должны быть заключены либо в одинарные, либо в двойные кавычки:

```
<book date-of-issue="04/11/2011" title='Java Industrial'/>
```

В отличие от этого HTML разрешает записывать значение атрибута без кавычек.

Например:

```
<FORM method=POST action=index.jsp>
```

Далее представлены примеры неправильной орфографии и использования тегов:

```
<?xml version="1.0"?>
<book>
          <title>JAXP</title>
</book>
<book value="JavaFX"/>
```

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>.

```
<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.

Любой открывающий тег должен иметь закрывающий.

```
<book>
          <system-exit>Zukov
</book>
```

Если тег не имеет содержимого, можно использовать конструкцию вида **<system-exit/>**. В HTML есть возможность не закрывать теги, и браузер определяет стили по открывающемуся тегу.

Наименования тегов чувствительны к регистру (case-sensitive), т. е., например, теги **<author>**, **<Author>**, **<AUTHOR>** будут совершенно разными. При работе с XML-тегом вида **<system-exit>Zukov</System-Exit>** программа-анализатор просто не найдет завершающий тег и выдаст ошибку. Язык HTML нетребователен к регистру.

Пусть существует XML-документ **students.xml** с данными о студентах:

 # 1 # описание студентов # students.xml

```
<?xml version="1.0" encoding="UTF-8"?>
        <students>
                <student login="MitarAlex7" faculty="mmf">
                        <name>Mitar Alex</name>
                        <telephone>2456474</telephone>
                        <address>
                                <country>Belarus</country>
                                <city>Minsk</city>
                                <street>Kalinovsky 45</street>
                        </address>
                </student>
                <student login="Pashkin5" faculty="mmf">
                        <name>Pashkin Alex</name>
                        <telephone>3453789</telephone>
                        <address>
                                <country>Belarus</country>
                                <city>Brest</city>
                                <street>Knorina 56</street>
                        </address>
                </student>
        </students>
```

Документ обладает древовидной структурой, следовательно, в базе данных по этому описанию требовалось бы создать две таблицы.

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 заключена между **<?** и **?>** и начинаться с идентификатора, называемого **target** (цель).

Например:

```
<?xml-stylesheet type="text/xsl" href="student.xsl"?>
```

Эта инструкция по обработке сообщает браузеру, что для данного документа необходимо применить стилевую таблицу (stylesheet) **student.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 --> -->
```

не внутри значений атрибутов:

```
<book country="BLR<!-- Неправильный комментарий -->"/>
```

не внутри тегов:

```
<book <!-- Неправильный комментарий -->/>
```

Указатели

Текстовые блоки XML-документа не могут содержать символы, 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>
```

Корректность

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, определенных 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

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>
```

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 способна 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 «+» (1 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:

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 'Oracle'>
<sender>&company;</sender>
```

Программа-анализатор, которая будет обрабатывать файл, автоматически подставит значение Oracle 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:

 # 2 # dtd для документа students.xml # students.dtd

```
<?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)>
```

Одна из причин отказа от DTD-описаний — его представление в виде документа, не являющегося по определению XML-документом.

Схема XSD

Схема XSD представляет собой более строгое, чем DTD, руководство по созданию и валидации XML-документа. XSD-схема, в отличие от DTD, является XML-документом, и поэтому она отличается гибкостью при использовании в приложениях, при задании правил документа, а также для дальнейшего расширения новой функциональностью. В отличие от DTD схема содержит большое количество базовых типов (44 типа) и имеет поддержку пространств имен (**namespace**). С помощью схемы XSD можно также проверить документ на корректность, а именно валидность.

Схема XSD первой строкой содержит XML-декларацию. Любая схема своим корневым элементом должна содержать элемент **schema**.

В схеме нужно описать все элементы: их тип, количество повторений, дочерние элементы. Сам элемент создается элементом **element**, который может включать следующие атрибуты:

name — определяет имя элемента;

type — указывает тип элемента;

ref — ссылается на определение элемента, находящегося в другом месте;

minOccurs и **maxOccurs** — количество повторений этого элемента (по умолчанию принимает значение 1), чтобы указать, что количество элементов не ограничено, в атрибуте **maxOccurs** необходимо задать **unbounded**.

```
<element name="telephone" type="positiveInteger" />
```

или

```
<element name="student"
          type="tns:Student"
          minOccurs="1"
          maxOccurs="unbounded" />
```

Если стандартных типов не хватает для полноты описания элемента, то 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** (уникальный идентификатор), **gYear** (год) 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, расположенный 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, задаваемый регулярным выражением;

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="(\\w){8, 20}"/>
          </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 внутри описываемого сложного типа должны располагаться в определенной последовательности, используются элементы **sequence**, **all**, **choice**. 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="positiveInteger"/>
                <element name="address" type="tns:Address"/>
        </sequence>
        <attribute name="login" type="tns:Login" use="required"/>
        <attribute name="faculty" type="string" use="optional"/>
</complexType>
```

Для задания произвольного порядка следования элементов в XML используется такой тег, как **<all>**, который допускает любой порядок.

```
<element name="employee">
        <complexType>
                <all>
                        <element name="phone" type="positiveInteger"/>
                        <element name="salary" type="decimal"/>
                </all>
        </complexType>
</element>
```

Элемент **<choice>** указывает, что в XML может присутствовать *только* один из перечисленных элементов, в то время как элемент **<sequence>** задает строгий порядок дочерних элементов.

Если набор значений поля или атрибута ограничен некоторым множеством, то для его определения следует использовать элемент **enumeration**. Например, атрибут **faculty** может принимать только значения: **mmf**, **geo**, **ksis**. Тогда вместо элемента

```
<attribute name="faculty" type="string" use="optional" />
```

следует записать

```
<attribute name="faculty">
          <simpleType>
                    <restriction base="string">
                              <enumeration value="mmf"></enumeration>
                              <enumeration value="geo"></enumeration>
                              <enumeration value="ksis"></enumeration>
                    </restriction>
          </simpleType>
</attribute>
```

Для списка студентов XML-схема **students.xsd** может выглядеть следующим образом:

 # 3 # xsd-схема для документа students.xml # students.xsd

```
<?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"
          elementFormDefault="qualified">
          <element name="students">
                    <complexType>
                              <sequence>
                                        <element name="student"
                                                  type="tns:Student"
                                                  minOccurs="2"
                                                  maxOccurs="unbounded" />
                              </sequence>
                    </complexType>
          </element>
          <complexType name="Student">
                    <sequence>
                              <element name="name" type="string" />
                              <element name="telephone" type="positiveInteger" />
                              <element name="address" type="tns:Address" />
                    </sequence>
          <attribute name="login" type="tns:Login" use="required" />
          <attribute name="faculty" use="optional" default="mmf">
                    <simpleType>
                              <restriction base="string">
                                        <enumeration value="mmf"></enumeration>
                                        <enumeration value="geo"></enumeration>
                                        <enumeration value="ksis"></enumeration>
                              </restriction>
                    </simpleType>
          </attribute>
          </complexType>
          <simpleType name="Login">
                    <restriction base="ID">
                              <pattern value="([a-zA-Z])[a-zA-Z0-9]{7,19}"/>
```

```
                </restriction>
        </simpleType>
        <complexType name="Address">
                <sequence>
                        <element name="country" type="string" />
                        <element name="city" type="string" />
                        <element name="street" type="string" />
                </sequence>
        </complexType>
</schema>
```

Для объявления атрибутов в элементах, которые могут содержать только текст, используются элементы **simpleContent** и **extension**, с помощью которых базовый тип элемента расширяется атрибутом(ами).

```
<element name="Teacher">
        <complexType>
                <simpleContent>
                        <extension base="string">
                                <attribute name="faculty" type="string"/>
                        </extension>
                </simpleContent>
        </complexType>
</element>
```

Для расширения/ограничения ранее объявленных сложных типов используется элемент **complexContent**. Пусть имеется некоторый тип **PersonType**, содержащий элементы **name**, **telephone** и **address**. Типы **Student** и **Abiturient** не только содержат те же элементы, что и **PersonType**, но и добавляют к ним свои. Тип **Student** добавляет к тегу **student** атрибуты **login** и **faculty**, а тип **Abiturient** добавляет элемент **average-mark**. Тип **PersonType** выглядит следующим образом:

```
<complexType name="PersonType">
        <sequence>
                <element name="name" type="string" />
                <element name="telephone" type="positiveInteger" />
                <element name="address" type="tns:Address" />
        </sequence>
</complexType>
```

Типы **Student** и **Abiturient**, его расширяющие, представлены в виде:

```
<complexType name="Student">
        <complexContent>
                <extension base="tns:PersonType">
                        <attribute name="login" type="tns:Login" use="required" />
                        <attribute name="faculty" use="optional" default="mmf">
                                <simpleType>
                                        <restriction base="string">
                                        <enumeration value="mmf"></enumeration>
```

```
                                        <enumeration value="geo"></enumeration>
                                        <enumeration value="ksis"></enumeration>
                                        </restriction>
                                </simpleType>
                        </attribute>
                </extension>
        </complexContent>
</complexType>
<complexType name="Abiturient">
        <complexContent>
                <extension base="tns:PersonType">
                        <sequence>
                                <element name="average-mark" type="double" />
                        </sequence>
                </extension>
        </complexContent>
</complexType>
```

Соответствие типов и тегов записывается в виде:

```
<element name="person" type="tns:PersonType" abstract="true"></element>
<element name="student" type="tns:Student" substitutionGroup="tns:person"></element>
<element name="abiturient" type="tns:Abiturient"  substitutionGroup="tns:person"></element>
```

Корневой элемент будет ссылаться только на абстрактный элемент **person**.

```
<element name="students">
        <complexType>
                <sequence>
                        <element ref="tns:person" minOccurs="2" maxOccurs="unbounded" />
                </sequence>
        </complexType>
</element>
```

Документ, соответствующий описанным выше правилам выглядит следующим образом:

 # 4 # документ с описанием студентов и абитуриентов # students_ext.xml

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.example.com/students"
        xsi:schemaLocation="http://www.example.com/students person.xsd">
        <abiturient>
                <name>Petkevich</name>
                <telephone>2787474</telephone>
                <address>
                        <country>Belarus</country>
                        <city>Minsk</city>
                        <street>Slobodskaja 7</street>
                </address>
                <average-mark>9.0</average-mark>
        </abiturient>
```

```
        <student login="MitarAlex7" faculty="mmf">
                <name>Mitar Alex</name>
                <telephone>2456474</telephone>
                <address>
                        <country>Belarus</country>
                        <city>Minsk</city>
                        <street>Kalinovsky 45</street>
                </address>
        </student>
        <student login="Pashkin5" faculty="mmf">
                <name>Pashkin Alex</name>
                <telephone>3453789</telephone>
                <address>
                        <country>Belarus</country>
                        <city>Brest</city>
                        <street>Knorina 56</street>
                </address>
        </student>
</students>
```

В приведенном документе 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 имен распространяется на элемент, где он объявлен, и на все дочерние элементы.

Тогда для проверки документа экземпляру-парсеру следует дать указание использовать DTD или схему XSD. В XML-документ для валидации с помощью схемы следует добавить вместо корневого элемента **<students>** элемент **<ns:students>** вида:

```
<ns:students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:ns="http://www.example.com/students"
        xsi:schemaLocation="http://www.example.com/students students.xsd">
```

или

```
<students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://www.example.com/Students"
          xsi:schemaLocation="http://www.example.com/students students.xsd">
```

Следующий пример выполняет проверку документа на валидность, то есть соответствие схеме, средствами языка Java при парсинге документа.

/ # 5 # проверка корректности документа XML с XSD # ValidatorSAX.java */*

```
package by.bsu.valid;
import java.io.File;
import java.io.IOException;
import javax.xml.XMLConstants;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import org.xml.sax.SAXException;
public class ValidatorSAX {
          public static void main(String[ ] args) {
                    String filename = "data/students.xml";
                    String schemaname = "data/students.xsd";
                    String logname = "logs/log.txt";
                    Schema schema = null;
                    String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
                    SchemaFactory factory = SchemaFactory.newInstance(language);
                    try {
                              // установка проверки с использованием XSD
                              schema = factory.newSchema(new File(schemaname));
                              SAXParserFactory spf = SAXParserFactory.newInstance();
                              spf.setSchema(schema);
                              // создание объекта-парсера
                              SAXParser parser = spf.newSAXParser();
                              // установка обработчика ошибок и запуск
                              parser.parse(filename, new StudentErrorHandler(logname));
                              System.out.println(filename + " is valid");
                    } catch (ParserConfigurationException e) {
                              System.err.println(filename + " config error: " + e.getMessage());
                    } catch (SAXException e) {
                              System.err.println(filename + " SAX error: " + e.getMessage());
                    } catch (IOException e) {
                              System.err.println("I/O error: " + e.getMessage());
                    }
          }
}
```

Для запуска этого примера используются только стандартные библиотеки JDK.

Класс обработчика ошибок может выглядеть следующим образом:

/ # 6 # обработчик ошибок # StudentErrorHandler.java */*

```
package by.bsu.valid;
import java.io.IOException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
public class StudentErrorHandler extends DefaultHandler {
                    // создание регистратора ошибок для пакета by.bsu.valid
          private Logger logger = Logger.getLogger("by.bsu.valid");
          public StudentErrorHandler(String log) throws IOException {
                    // 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();
          }
}
```

При проверке XML-документа разумно все ошибки фиксировать, в частности, с помощью логгера Log4j, библиотеку которого log4j-[*version*].jar следует подключить к проекту.

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 'MitarAlex7'.
ERROR - 14 : 41 - cvc-attribute.3: The value 'MitarAlex7' of attribute 'login' on element 'student' is not valid with respect to its type, 'login'.

Если допустить синтаксическую ошибку в XML-документе, например, удалить закрывающую угловую скобку в элементе **telephone**, будет выведено сообщение о фатальной ошибке:

FATAL - 7 : 26 - Element type "telephone2456474" must be followed by either attribute specifications, ">" or "/>".

Можно также провести проверку документа на соответствие XSD с применением возможностей специального класса **Validator**.

/ # 7 # проверка корректности документа XML # ValidatorSAX.java */*

```
package by.bsu.valid;
import java.io.*;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
public class ValidatorSAXXSD {
          public static void main(String[ ] args) {
                    String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
                    String fileName = "data/students.xml";
                    String schemaName = "data/students.xsd";
                    SchemaFactory factory = SchemaFactory.newInstance(language);
                    File schemaLocation = new File(schemaName);
                    try {
                              // создание схемы
                              Schema schema = factory.newSchema(schemaLocation);
                              // создание валидатора на основе схемы
                              Validator validator = schema.newValidator();
                              // проверка документа
                              Source source = new StreamSource(fileName);
                              validator.validate(source);
                              System.out.println(filename + " is valid.");
                    } catch (SAXException e) {
                              System.err.print("validation "+ filename + " is not valid because "
                              + e.getMessage());
                    } catch (IOException e) {
                              System.err.print(filename + " is not valid because "
                              + e.getMessage());
                    }
          }
}
```

Экземпляр класса **Validator** может использовать класс-обработчик ошибок с сохранением информации о них в файле **log.txt**. В этом случае в код примера следует вставить следующий фрагмент, осуществляющий инициализацию и установку объекта **StudentErrorHandler** для экземпляра **Validator**.

```
          StudentErrorHandler sh = new StudentErrorHandler("logs/log.txt");
          validator.setErrorHandler(sh);
          validator.validate(source);
          System.out.println(filename + " validating is ended.");
```

Информация о некорректном содержимом XML-файла в этом случае сохраняется в log-файле и на консоль выводиться не будет.

JAXB. Маршаллизация и демаршаллизация

Начиная с версии Java 6, включены продвинутые механизмы извлечения/сохранения данных при взаимодействии с XML.

Маршаллизация — механизм преобразования данных из java-объектов в конкретное хранилище, будь то документ XML, база данных или простой текстовый файл.

Демаршаллизация — обратный процесс преобразования данных из внешних источников в структуру хранения, поддерживаемую виртуальной машиной. Проблемой остается организация взаимно однозначного соответствия информации в источнике, например, XML-документе, и экземпляре типа данных, принимающем эту информацию.

Соединение этих двух процессов должно корректно определять импорт-экспорт или круговорот информации без потерь и искажений на всех этапах. Информация, взятая из XML-файла и транслированная в объект java, при обратном преобразовании должна быть возвращена в идентичном виде.

Следующий пример на основе экземпляра класса **Students,** содержащего, в свою очередь, список экземпляров класса **Student**, создает структуру документа XML и сохраняет в ней информацию из объекта. Классы **Students** и **Student** разработаны и аннотированы так, чтобы структура создаваемого на их основе XML-документа соответствовала документу **students.xml**, приведенному в листинге #1.

/ # 8 # компонент JavaBean # Student.java */*

```
package by.bsu.xmlstudents;
import javax.xml.bind.annotation.*;
import javax.xml.bind.annotation.adapters.*;
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Student", propOrder = {
    "name",
    "telephone",
    "address"
}) // задание последовательности элементов XML
public class Student {
          @XmlAttribute(required = true)
@XmlJavaTypeAdapter(CollapsedStringAdapter.class)
          @XmlID
          private String login;
          @XmlElement(required = true)
          private String name;
```

```
@XmlAttribute(required = false)
private String faculty;
@XmlElement(required = true)
private int telephone;
@XmlElement(required = true)
private Address address = new Address();
public Student() { } // необходим для маршаллизации/демаршалиизации XML
public Student(String login, String name, String faculty, int telephone, Address address) {
          this.login = login;
          this.name = name;
          this.faculty = faculty;
          this.telephone = telephone;
          this.address = 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 int getTelephone() {
          return telephone;
}
public void setTelephone(int telephone) {
          this.telephone = telephone;
}
public Address getAddress() {
          return address;
}
public void setAddress(Address address) {
          this.address = address;
}
public String toString() {
          return "\nLogin: " + login + "\nName: " + name + "\nTelephone: " + telephone
                    + "\nFaculty: " + faculty + address.toString();
}
@XmlRootElement
@XmlType(name = " address ", propOrder = {
```

```
                                "city",
                                "country",
                                "street"
        })
        public static class Address { // внутренний класс
                private String country;
                private String city;
                private String street;
                public Address() {// необходим для маршаллизации/демаршалиизации XML
                }
                public Address(String country, String city, String street) {
                        this.country = country;
                        this.city = city;
                        this.street = 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;
                }
                public String toString() {
                        return "\nAddress:" + "\n\tCountry: " + country
                        + "\n\tCity: " + city + "\n\tStreet: " + street + "\n";
                }
        }
}
```

Структура класса **Students** предназначена для хранения экземпляров класса **Student**. Класс **Student** в дальнейшем будет использоваться для создания списков и множеств объектов при разработке парсеров на основе информации, извлеченной из XML-документа.

/ # 9 # компонент для хранения списка студентов # Students.java */*

```
package by.bsu.xmlstudents;
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
```

```
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class Students {
          @XmlElement(name="student")
          private ArrayList<Student> list = new ArrayList<Student>();
          public Students() {
                    super();
          }
          public void setList(ArrayList<Student> list) {
                    this.list = list;
          }
          public boolean add(Student st) {
                    return list.add(st);
          }
          @Override
          public String toString() {
                    return "Students [list=" + list + "]";
          }
}
```

Процесс маршаллизации состоит из создания JAXB контекста на основе класса **Students**, создания на его основе экземпляра типа **Marshaller** и сохранения информации в файл.

/ # 10 # создание XML-документа на основе экземпляра класса # MarshalMain.java */*

```
package by.bsu.jaxb;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import by.bsu.xmlstudents.Student;
import by.bsu.xmlstudents.Students;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
public class MarshalMain {
          public static void main(String[] args) {
                    try {
                      JAXBContext context = JAXBContext.newInstance(Students.class);
                      Marshaller m = context.createMarshaller();
                      Students st = new Students() { // анонимный класс
                        {
                        // добавление первого студента
                        Student.Address addr = new Student.Address("BLR", "Minsk", "Skoriny 4");
                        Student s = new Student("gochette", "Klimenko", "mmf", 2095306, addr);
                        this.add(s);
                        // добавление второго студента
                        addr = new Student.Address("BLR", "Polotesk", "Simeona P. 23");
                        s = new Student("ivette", "Teran", "mmf", 2345386, addr);
                        this.add(s);
                        }
```

```
                    };
                    m.marshal(st, new FileOutputStream("data/studs_marsh.xml"));
                    m.marshal(st, System.out); // копия на консоль
                    System.out.println("XML-файл создан");
                } catch (FileNotFoundException e) {
                    System.out.println("XML-файл не может быть создан: " + e);
                } catch (JAXBException e) {
                    System.out.println("JAXB-контекст ошибочен " + e);
                }
        }
}
```

В результате компиляции и запуска программы будет создан XML-документ

 # 11 # сохраненная информация # studs_marsh.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<students>
        <student login="gochette" faculty="mmf">
                <name>Klimenko</name>
                <telephone>2095306</telephone>
                <address>
                        <city>Minsk</city>
                        <country>BLR</country>
                        <street>Skoriny 4</street>
                </address>
        </student>
        <student login="ivette" faculty="mmf">
                <name>Teran</name>
                <telephone>2345386</telephone>
                <address>
                        <city>Polotesk</city>
                        <country>BLR</country>
                        <street>Simeona P. 23</street>
                </address>
        </student>
</students>
```

Процедура демаршаллизации аналогична маршаллизации с той разницей, что в итоге будет получен корректно созданный экземпляр класса **Students**.

/ # 12 # создание экземпляра класса на основе XML-документа # UnMarshalMain.java */*

```
package by.bsu.jaxb;
import java.io.FileNotFoundException;
import java.io.FileReader;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import by.bsu.xmlstudents.Students;
```

```
public class UnMarshalMain {
          public static void main(String[ ] args) {
                    try {
                              JAXBContext jc = JAXBContext.newInstance(Students.class);
                              Unmarshaller u = jc.createUnmarshaller();
                              FileReader reader = new FileReader("data/studs_marsh.xml");
                              Students students = (Students) u.unmarshal(reader);
                              System.out.println(students);
                    } catch (JAXBException e) {
                              e.printStackTrace();
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    }
          }
}
```

Вывод на консоль показывает идентичность информации после восстановления экземпляра.

Students [list=[
Login: gochette
Name: Klimenko
Telephone: 2095306
Faculty: mmf
Address:
Country: BLR
City: Minsk
Street: Skoriny 4
,
Login: ivette
Name: Teran
Telephone: 2345386
Faculty: mmf
Address:
Country: BLR
City: Polotesk
Street: Simeona P. 23
]]

JAXB. Генерация классов

Возможен инжиниринг классов на языке Java на основе XML-схемы. Ниже приведена схема **person.xsd** с описанием типов-классов **PersonType** и его расширений-подклассов **Student** и **Abiturient**, а также типа-класса **Students**, который может содержать список студентов и абитуриентов.

 # 13 # схема с описанием иерархии # person.xsd

```
<?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" elementFormDefault="qualified">
          <element name="person" type="tns:PersonType" abstract="true"></element>
          <element name="student" type="tns:Student" substitutionGroup="tns:person"></element>
          <element name="abiturient" type="tns:Abiturient"
                    substitutionGroup="tns:person"></element>
          <element name="students">
                    <complexType>
                              <sequence>
                                        <element ref="tns:person" minOccurs="2"
                                                                  maxOccurs="unbounded" />
                              </sequence>
                    </complexType>
          </element>
          <complexType name="PersonType">
                    <sequence>
                              <element name="name" type="string" />
                              <element name="telephone" type="positiveInteger" />
                              <element name="address" type="tns:Address" />
                    </sequence>
          </complexType>
          <complexType name="Student">
                    <complexContent>
                              <extension base="tns:PersonType">
                                        <attribute name="login" type="tns:Login" use="required" />
                                        <attribute name="faculty" use="optional" default="mmf">
                                                  <simpleType>
                                                            <restriction base="string">
                                                  <enumeration value="mmf"></enumeration>
                                                  <enumeration value="geo"></enumeration>
                                                  <enumeration value="ksis"></enumeration>
                                                            </restriction>
                                                  </simpleType>
                                        </attribute>
                              </extension>
                    </complexContent>
          </complexType>
          <complexType name="Abiturient">
                    <complexContent>
                              <extension base="tns:PersonType">
                                        <sequence>
                                                  <element name="average-mark" type="double" />
                                        </sequence>
                              </extension>
                    </complexContent>
```

```
        </complexType>
        <simpleType name="Login">
                <restriction base="ID">
                        <pattern value="([a-zA-Z])[a-zA-Z0-9]{7,19}" />
                </restriction>
        </simpleType>
        <complexType name="Address">
                <sequence>
                        <element name="country" type="string" />
                        <element name="city" type="string" />
                        <element name="street" type="string" />
                </sequence>
        </complexType>
</schema>
```

Запуск процесса генерации осуществляется с помощью командной строки:

```
xjc.exe person.xsd
```

В результате будет сгенерирован пакет **com.example.students**, содержащий следующие классы-сущности:

/ # 14 # исходный код классов, сгенерированный на основе XSD # PersonType.java # Abiturient.java # Student.java # Address.java # Students.java */*

```
package com.example.students;
import java.math.BigInteger;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlSchemaType;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "PersonType", propOrder = {
    "name",
    "telephone",
    "address"
})
@XmlSeeAlso({
    Student.class,
    Abiturient.class
})
public class PersonType {
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    @XmlSchemaType(name = "positiveInteger")
    protected BigInteger telephone;
    @XmlElement(required = true)
    protected Address address;
```

```
    public String getName() {
        return name;
    }
    public void setName(String value) {
        this.name = value;
    }
    public BigInteger getTelephone() {
        return telephone;
    }
    public void setTelephone(BigInteger value) {
        this.telephone = value;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address value) {
        this.address = value;
    }
}
package com.example.students;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Abiturient", propOrder = { "averageMark" })
public class Abiturient extends PersonType {
          @XmlElement(name = "average-mark")
          protected double averageMark;
          public double getAverageMark() {
                    return averageMark;
          }
          public void setAverageMark(double value) {
                    this.averageMark = value;
          }
}
package com.example.students;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlID;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.adapters.CollapsedStringAdapter;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Student")
public class Student extends PersonType {
    @XmlAttribute(name = "login", required = true)
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
    @XmlID
```

```
    protected String login;
    @XmlAttribute(name = "faculty")
    protected String faculty;
    public String getLogin() {
        return login;
    }
    public void setLogin(String value) {
        this.login = value;
    }
    public String getFaculty() {
        if (faculty == null) {
            return "mmf";
        } else {
            return faculty;
        }
    }
    public void setFaculty(String value) {
        this.faculty = value;
    }
}
package com.example.students;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Address", propOrder = {
    "country",
    "city",
    "street"
})
public class Address {
    @XmlElement(required = true)
    protected String country;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String street;
    public String getCountry() {
        return country;
    }
    public void setCountry(String value) {
        this.country = value;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String value) {
        this.city = value;
    }
    public String getStreet() {
```

```
        return street;
    }
    public void setStreet(String value) {
        this.street = value;
    }
}
package com.example.students;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElementRef;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "person"
})
@XmlRootElement(name = "students")
public class Students {
    @XmlElementRef(name = "person", namespace = "http://www.example.com/students", type =
JAXBElement.class)
    protected List<JAXBElement<? extends PersonType>> person;
    public List<JAXBElement<? extends PersonType>> getPerson() {
        if (person == null) {
            person = new ArrayList<JAXBElement<? extends PersonType>>();
        }
        return this.person;
    }
}
```

Также сгенерирован класс-фабрика для создания экземпляров перечисленных классов:

/ # 15 # исходный код классов, сгенерированный на основе XSD # ObjectFactory.java */*

```
package com.example.students;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;
@XmlRegistry
public class ObjectFactory {
    private final static QName _Person_QNAME =
                    new QName("http://www.example.com/students", "person");
    private final static QName _Student_QNAME =
                    new QName("http://www.example.com/students", "student");
    private final static QName _Abiturient_QNAME =
                    new QName("http://www.example.com/students", "abiturient");
```

```
    public ObjectFactory() {
    }
    public Students createStudents() {
        return new Students();
    }
    public PersonType createPersonType() {
        return new PersonType();
    }
    public Student createStudent() {
        return new Student();
    }
    public Abiturient createAbiturient() {
        return new Abiturient();
    }
    public Address createAddress() {
        return new Address();
    }
    @XmlElementDecl(namespace = "http://www.example.com/students", name = "person")
    public JAXBElement<PersonType> createPerson(PersonType value) {
        return new JAXBElement<PersonType>(_Person_QNAME, PersonType.class, null, value);
    }
    @XmlElementDecl(namespace = "http://www.example.com/students", name = "student",
                            substitutionHeadNamespace = "http://www.example.com/students",
                            substitutionHeadName = "person")
    public JAXBElement<Student> createStudent(Student value) {
        return new JAXBElement<Student>(_Student_QNAME, Student.class, null, value);
    }
    @XmlElementDecl(namespace = "http://www.example.com/students", name = "abiturient",
                            substitutionHeadNamespace = "http://www.example.com/students",
                            substitutionHeadName = "person")
    public JAXBElement<Abiturient> createAbiturient(Abiturient value) {
        return new JAXBElement<Abiturient>(_Abiturient_QNAME, Abiturient.class, null, value);
    }
}
```

Демаршаллизацию информации из XML-документа для более высокого качества следует производить с применением XSD-схемы:

/ # 16 # создание экземпляра класса из XML с привлечением XSD # UnMarshalWithXSD.java */*

```
package by.bsu.jaxb;
import java.io.File;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import org.xml.sax.SAXException;
```

```
import com.example.students.Students;
public class UnMarshalWithXSD {
          public static void main(String[ ] args) {
                    JAXBContext jc = null;
                    try {
                              jc = JAXBContext.newInstance("com.example.students");
                              Unmarshaller um = jc.createUnmarshaller();
                              String schemaName = "dat/person.xsd";
                              SchemaFactory factory =
                    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
                              File schemaLocation = new File(schemaName);
                              // создание схемы и передача ее демарашаллизатору
                              Schema schema = factory.newSchema(schemaLocation);
                              um.setSchema(schema);
                              Students st =
                                    (Students) um.unmarshal(new File("data/students_ext.xml"));
                              System.out.println(st);
                    } catch (JAXBException e) {
                              e.printStackTrace();
                    } catch (SAXException e) {
                              e.printStackTrace();
                    }
          }
}
```

JAXP

JAXP — Java API for XML Processing. XML-документ 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 проекте. Поскольку XML представляет собой универсальный формат для передачи данных, существуют универсальные средства его обработки — XML-анализаторы (парсеры).

Парсер — это библиотека, которая читает XML-документ, а затем предоставляет набор методов для обработки информации из этого документа.

Валидирующие и невалидирующие анализаторы

Как было уже упомянуто, существуют 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 XSD.

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.

Никакой связи между видом анализатора и видом XML-документа нет. Валидирующий анализатор может разобрать XML-документ, для которого нет XSD, и, наоборот, невалидирующий анализатор может разобрать XML-документ, для которого есть XSD. При этом он просто не будет учитывать описание структуры документа.

Древовидная и псевдособытийная 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 квазисобытия, — это SAX-анализаторы.

Анализаторы, которые ждут команды от приложения для перехода к следующему элементу XML — StAX-анализаторы.

В первом случае анализатор строит в памяти объект, представляющий собой дерево из элементов, соответствующее XML-документу. Далее вся работа ведется именно с этим объектом-деревом.

Во втором случае анализатор работает следующим образом: при чтении/анализе 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, анализатор будет генерировать событие о том, что он встретил начало документа либо его конец, начало элемента либо его конец, символьную информацию внутри элемента и т. д.

Анализатор StAX работает подобно итератору, 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-документа 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.

Псевдособытийная модель

Как уже отмечалось, SAX-анализатор не строит дерево элементов по содержимому XML-файла. Вместо этого анализатор читает файл и генерирует квазисобытие при нахождении элемента, атрибута или текста. 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-документе.

Однако нужно учитывать, для каких целей используются данные из XML-файла. Очевидно, что нет смысла строить дерево объектов, содержащее десятки тысяч элементов в памяти, если все, что необходимо, — просто посчитать точное количество элементов в файле.

SAX-анализаторы

SAX API определяет ряд интерфейсов, используемых при разборе документа. Чаще других используется **org.xml.sax.ContentHandler** и некоторые объявленные в нем методы:

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). Если этой информации достаточно много, то метод может быть вызван более одного раза.

Для обработки предупреждений и ошибок, возникающих при разборе XML-документа, применяется интерфейс **org.xml.sax.ErrorHandler**, содержащий методы:

```
warning(SAXParseException e),
error(SAXParseException e),
fatalError(SAXParseException e).
```

В пакете **org.xml.sax** в SAX2 API содержатся также интерфейсы **DTDHandler**, **DocumentHandler** и **EntityResolver**, которые необходимо реализовать для обработки интересующего 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**, **DocumentHandler**) или наследует класс **org.xml.sax.helpers.DefaultHandler**, и реализовать методы, отвечающие за обработку интересующих частей документа или ошибок.
2. Используя SAX2 API, поддерживаемое всеми SAX-парсерами, создать **org.xml.sax.XMLReader**, например:

```
XMLReader reader = XMLReaderFactory.createXMLReader();
```

   или

```
XMLReader reader =
          XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
```

   Для библиотеки **xercesImpl.jar**, которую можно загрузить по адресу **http://xerces.apache.org/xerces2-j/**.
3. Передать в **XMLReader** объект класса, созданного на шаге 1 с помощью соответствующих методов: **setContentHandler()**, **setErrorHandler()**, **setDTDHandler()**, **setEntityResolver()**.
4. Вызвать метод **parse(String filename)** класса **XMLReader**, которому в качестве параметров передать путь (URI) к анализируемому документу либо **InputSource**.

Следующий пример в результате парсинга выводит на консоль содержимое XML-документа.

/ # 17 # чтение и вывод XML-документа # SimpleStudentHandler.java */*

```
package by.bsy.saxsimple;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Attributes;
public class SimpleStudentHandler extends DefaultHandler {
          @Override
          public void startDocument() {
                    System.out.println("Parsing started");
          }
          @Override
          public void startElement(String uri, String localName, String qName, Attributes attrs) {
                    String s = localName;
                    // получение и вывод информации об атрибутах элемента
                    for (int i = 0; i < attrs.getLength(); i++) {
                              s += " " + attrs.getLocalName(i) + "=" + attrs.getValue(i);
```

```
                }
                System.out.print(s.trim());
        }
        @Override
        public void characters(char[ ] ch, int start, int length) {
                System.out.print(new String(ch, start, length));
        }
        @Override
        public void endElement(String uri, String localName, String qName) {
                System.out.print(localName);
        }
        @Override
        public void endDocument() {
                System.out.println("\nParsing ended");
        }
}
```

Где **uri** — уникальное название **namespace**, **localName** — имя элемента без префикса, задаваемого именем атрибута **xmlns**, например:

```
xmlns:ns=http://www.example.com/Students
```

то есть без **ns**, **qName** — полное имя элемента с префиксом, **attrs** — список атрибутов.

/ # 18 # создание и запуск простейшего парсера # SAXSimpleMain.java*/*

```
package by.bsy.saxsimple;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.SAXException;
import java.io.IOException;
public class SAXSimpleMain {
        public static void main(String[ ] args) {
                try {
                        // создание SAX-анализатора
                        XMLReader reader = XMLReaderFactory.createXMLReader();
                        SimpleStudentHandler handler = new SimpleStudentHandler();
                        reader.setContentHandler(handler);
                        reader.parse("data/students.xml");
                } catch (SAXException e) {
                        System.err.print("ошибка SAX парсера " + e);
                } catch (IOException e) {
                        System.err.print("ошибка I/O потока " + e);
                }
        }
}
```

В результате в консоль будет выведено (в XML-документе должна быть ссылка на XSD):

```
Parsing started
students xsi:schemaLocation=http://www.example.org/Students.xsd students.xsd
   student login=MitarAlex7 faculty=mmf
      nameMitar Alexname
      telephone2456474telephone
      address
         countryBelaruscountry
         cityMinskcity
         streetKalinovsky 45street
      address
   student
   student login=Pashkin5 faculty=mmf
      namePashkin Alexname
      telephone3453789telephone
      address
         countryBelaruscountry
         cityBrestcity
         streetKnorina 56street
      address
   student
students
Parsing ended
```

В следующем приложении производятся разбор документа **students.xml** и инициализация на его основе коллекции объектов класса **Student**.

/ # 19 # формирование коллекции объектов на основании разбора XML-документа # StudentsSAXBuilder.java */*

```
package by.bsu.parsing;
import by.bsu.xmlstudents.Student;
import java.io.IOException;
import java.util.Set;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
public class StudentsSAXBuilder {
          private Set<Student> students;
          private StudentHandler sh;
          private XMLReader reader;
          public StudentsSAXBuilder() {
                    // создание SAX-анализатора
                    sh = new StudentHandler();
                    try {
                              // создание объекта-обработчика
                              reader = XMLReaderFactory.createXMLReader();
```

```
                    reader.setContentHandler(sh);
                } catch (SAXException e) {
                    System.err.print("ошибка SAX парсера: " + e);
                }
        }
        public Set<Student> getStudents() {
                return students;
        }
        public void buildSetStudents(String fileName) {
                try {
                    // разбор XML-документа
                    reader.parse(fileName);
                } catch (SAXException e) {
                    System.err.print("ошибка SAX парсера: " + e);
                } catch (IOException e) {
                    System.err.print("ошибка I/O потока: " + e);
                }
                students = sh.getStudents();
        }
}
package by.bsu.parsing;
public enum StudentEnum {
    STUDENTS("students"),
    LOGIN("login"),
    FACULTY("faculty"),
    STUDENT("student"),
    NAME("name"),
    TELEPHONE("telephone"),
    COUNTRY("country"),
    CITY("city"),
    STREET("street"),
    ADDRESS("address");
    private String value;
    private StudentEnum(String value) {
        this.value = value;
    }
    public String getValue() {
        return value;
    }
}
package by.bsu.xmlstudents;
import java.util.EnumSet;
import java.util.HashSet;
import java.util.Set;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
public class StudentHandler extends DefaultHandler {
        private Set<Student> students;
```

```
private Student current = null;
private StudentEnum currentEnum = null;
private EnumSet<StudentEnum> withText;
public StudentHandler() {
          students = new HashSet<Student>();
          withText = EnumSet.range(StudentEnum.NAME, StudentEnum.STREET);
}
public Set<Student> getStudents() {
          return students;
}
public void startElement(String uri, String localName, String qName, Attributes attrs) {
          if ("student".equals(localName)) {
                    current = new Student();
                    current.setLogin(attrs.getValue(0));
                    if (attrs.getLength() == 2) {
                              current.setFaculty(attrs.getValue(1));
                    }
          } else {
                   StudentEnum temp = StudentEnum.valueOf(localName.toUpperCase());
                    if (withText.contains(temp)) {
                              currentEnum = temp;
                    }
          }
}
public void endElement(String uri, String localName, String qName) {
          if ("student".equals(localName)) {
                    students.add(current);
          }
}
public void characters(char[] ch, int start, int length) {
          String s = new String(ch, start, length).trim();
          if (currentEnum != null) {
                switch (currentEnum) {
                    case NAME:
                              current.setName(s);
                              break;
                    case TELEPHONE:
                              current.setTelephone(Integer.parseInt(s));
                              break;
                    case STREET:
                              current.getAddress().setStreet(s);
                              break;
                    case CITY:
                              current.getAddress().setCity(s);
                              break;
                    case COUNTRY:
                              current.getAddress().setCountry(s);
                              break;
                    default:
```

```
                                    throw new EnumConstantNotPresentException(
                                    currentEnum.getDeclaringClass(), currentEnum.name());
                            }
                        }
                        currentEnum = null;
                }
        }
```

/ # 20 # создание и запуск парсера # */*

```
StudentsSAXBuilder saxBuilder = new StudentsSAXBuilder();
saxBuilder.buildSetStudents("data/students.xml");
System.out.println(saxBuilder.getStudents());
```

В результате на консоль будет выведено следующее множество описаний объектов:

[
Login: MitarAlex7
Name: Mitar Alex
Telephone: 2456474
Faculty: mmf
Address:
Country: Belarus
City: Minsk
Street: Kalinovsky 45
,
Login: Pashkin5
Name: Pashkin Alex
Telephone: 3453789
Faculty: mmf
Address:
Country: Belarus
City: Brest
Street: Knorina 56
]

Древовидная модель

Анализатор DOM представляет собой некоторый общий интерфейс для работы со структурой документа. При разработке DOM-анализаторов различными вендорами предполагалась возможность ковариантности кода, однако при совместном использовании библиотек с аналогичными классами следует следить за совместимостью и корректностью взаимодействия.

DOM строит дерево, которое представляет содержимое XML-документа и определяет набор классов, представляющих каждый элемент в XML-документе (элементы, атрибуты, сущности, текст и т. д.).

В пакете **org.w3c.dom** можно найти интерфейсы, представляющие указанные объекты. 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: Xerces и JAXP, который входит в JDK.

Существуют также библиотеки, предлагающие свои структуры объектов XML с API для доступа к ним.

DOM JAXP

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, замечательная особенность которого — использование части стандартных возможностей 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. Этот 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;

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 элементом XML-документа и его содержимым. 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;

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;

boolean isId() — проверяет атрибут на тип ID.

Ниже приведен стандартный способ разбора документа **students.xml** с использованием DOM-анализатора и инициализация на его основе множества объектов.

/ # 21 # создание обьектов на основе экземпляра Document # StudentsDOMBuilder.java */*

```
package by.bsu.xmlstudents;
import java.io.IOException;
import java.util.HashSet;
import java.util.Set;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
```

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;
public class StudentsDOMBuilder {
private Set<Student> students;
          private DocumentBuilder docBuilder;
          public StudentsDOMBuilder() {
               this.students = new HashSet<Student>();
               // создание DOM-анализатора
               DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
               try {
                              docBuilder = factory.newDocumentBuilder();
               } catch (ParserConfigurationException e) {
                              System.err.println("Ошибка конфигурации парсера: " + e);
               }
          }
          public Set<Student> getStudents() {
          return students;
          }
          public void buildSetStudents(String fileName) {
               Document doc = null;
               try {
                              // parsing XML-документа и создание древовидной структуры
                              doc = docBuilder.parse(fileName);
                              Element root = doc.getDocumentElement();
               // получение списка дочерних элементов <student>
               NodeList studentsList = root.getElementsByTagName("student");
                              for (int i = 0; i < studentsList.getLength(); i++) {
                                        Element studentElement = (Element) studentsList.item(i);
                                        Student student = buildStudent(studentElement);
                                        students.add(student);
                              }
               } catch (IOException e) {
                              System.err.println("File error or I/O error: " + e);
               } catch (SAXException e) {
                              System.err.println("Parsing failure: " + e);
               }
          }
          private Student buildStudent(Element studentElement) {
               Student student = new Student();
               // заполнение объекта student
               student.setFaculty(studentElement.getAttribute("faculty")); // проверка на null
               student.setName(getElementTextContent(studentElement, "name"));
               Integer tel = Integer.parseInt(getElementTextContent(studentElement,
                                        "telephone"));
               student.setTelephone(tel);
               Student.Address address = student.getAddress();
               // заполнение объекта address
               Element adressElement = (Element) studentElement.getElementsByTagName(
```

```
                                        "address").item(0);
            address.setCountry(getElementTextContent(adressElement, "country"));
            address.setCity(getElementTextContent(adressElement, "city"));
            address.setStreet(getElementTextContent(adressElement, "street"));
            student.setLogin(studentElement.getAttribute("login"));
            return student;
        }
        // получение текстового содержимого тега
        private static String getElementTextContent(Element element, String elementName) {
            NodeList nList = element.getElementsByTagName(elementName);
            Node node = nList.item(0);
            String text = node.getTextContent();
            return text;
        }
}
```

/ # 22 # создание и запуск парсера # */*

```
StudentsDOMBuilder domBuilder = new StudentsDOMBuilder();
domBuilder.buildSetStudents("data/students.xml");
System.out.println(domBuilder.getStudents());
```

Создание XML–документа

Документы можно не только читать, но также модифицировать и создавать совершенно новые. Для этого необходимо создать объекты классов **Document**, **Element**, добавить к последнему атрибуты и текстовое содержимое, после чего присоединить их к объекту, который в дереве XML-документа находится выше.

Следующий пример демонстрирует создание XML-документа и запись его в файл. Для записи XML-документа используется класс **Transformer**.

/ # 23 # создание и запись документа # CreateDocument.java */*

```
package by.bsu.transform;
import java.io.FileWriter;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
public class CreateDocument {
```

```
        public static void main(String[] args) {
                DocumentBuilderFactory documentBuilderFactory =
                                                DocumentBuilderFactory.newInstance();
                DocumentBuilder documentBuilder = null;
                try {
                        documentBuilder =
                                documentBuilderFactory.newDocumentBuilder();
                } catch (ParserConfigurationException e) {
                        e.printStackTrace();
                }
                Document document = documentBuilder.newDocument();
                String root = "book";
                Element rootElement = document.createElement(root);
                document.appendChild(rootElement);
                for (int i = 0; i < 1; i++) {
                        String elementName = "name";
                        Element emName = document.createElement(elementName);
                        String name = "Technique Java";
                        emName.appendChild(document.createTextNode(name));

                        String elementAuthor = "author";
                        Element emAuthor = document.createElement(elementAuthor);
                        String author = "Blinov";
                        emAuthor.appendChild(document.createTextNode(author));
                        emAuthor.setAttribute("id", "3");
                        rootElement.appendChild(emName);
                        rootElement.appendChild(emAuthor);
                }
                TransformerFactory transformerFactory = TransformerFactory.newInstance();
                try {
                   Transformer transformer = transformerFactory.newTransformer();
                   DOMSource source = new DOMSource(document);
                   StreamResult result = new StreamResult(new FileWriter("data/book.xml"));
                   transformer.transform(source, result);
                } catch (TransformerConfigurationException e) {
                   e.printStackTrace();
                } catch (TransformerException e) {
                   e.printStackTrace();
                } catch (IOException e) {
                   e.printStackTrace();
                }
        }
}
```

В результате будет создан документ **book.xml** следующего содержания:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<book>
        <name>Technique Java</name>
        <author id="3">Blinov</author>
</book>
```

StAX

StAX (Streaming API for XML), который еще называют pull-парсером, включен в JSDK, начиная с версии Java 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 указывает 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 и расположены в пакете **javax.xml.stream**. Для чтения XML требуется получить ссылку на экземпляр **XMLStreamReader**:

```
StringReader input = new StringReader(fileName); // из пакета java.io
```

или

```
InputStream input = new FileInputStream(new File(fileName));
```

далее

```
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader reader = inputFactory.createXMLStreamReader(input);
```

после чего по экземпляру **XMLStreamReader** можно организовать навигацию аналогично интерфейсу **Iterator**, используя методы **hasNext()** и **next()**:

boolean hasNext() — показывает, есть ли еще элементы;

int next() — переходит к следующей вершине-константе XML, извлекая тип текущей.

При попытке вызове на константе несоответствующего ей метода генерируется исключительная ситуация **IllegalStateException**.

Чаще всего данные извлекаются с применением методов:

String getLocalName() — возвращает название тега (элемента) для текущей константы;

String getAttributeValue(String namespaceURI, String localName) — возвращает значение атрибута по имени;

String getAttributeValue(ine index) — возвращает значение атрибута по номеру позиции;

String getText() — возвращает текст для констант **CHARACTERS, CDATA, COMMENT** и др.

Возможные типы вершин и методы, применимые к ним (см. таблицу ниже).

Типы вершин-констант	Методы
Для всех типов констант	getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(), getLocation(), hasText(), hasName()
START_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()
ATTRIBUTE	next(), nextTag() getAttributeXXX(), isAttributeSpecified(),
NAMESPACE	next(), nextTag() getNamespaceXXX()
END_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()
CHARACTERS	next(), getTextXXX(), nextTag()
CDATA	next(), getTextXXX(), nextTag()
COMMENT	next(), getTextXXX(), nextTag()
SPACE	next(), getTextXXX(), nextTag()
START_DOCUMENT	next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT	close()
PROCESSING_INSTRUCTION	next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE	next(), getLocalName(), getText(), nextTag()

Организация процесса разбора документа XML с помощью StAX приведена в следующем примере:

/ # 24 # реализация разбора XML-документа на основе StAX # StudentsStAXBuilder.java */*

```
package by.bsu.xmlstudents;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
public class StudentsStAXBuilder {
          private HashSet<Student> students = new HashSet<>();
          private XMLInputFactory inputFactory;
          public StudentsStAXBuilder() {
```

```
            inputFactory = XMLInputFactory.newInstance();
    }
    public Set<Student> getStudents() {
            return students;
    }
    public void buildSetStudents(String fileName) {
            FileInputStream inputStream = null;
            XMLStreamReader reader = null;
            String name;
            try {
               inputStream = new FileInputStream(new File(fileName));
               reader = inputFactory.createXMLStreamReader(inputStream);
               // StAX parsing
               while (reader.hasNext()) {
                 int type = reader.next();
                 if (type == XMLStreamConstants.START_ELEMENT) {
                      name = reader.getLocalName();
                   if (StudentEnum.valueOf(name.toUpperCase()) == StudentEnum.STUDENT) {
                                                 Student st = buildStudent(reader);
                                                 students.add(st);
                   }
                 }
               }
            } catch (XMLStreamException ex) {
                      System.err.println("StAX parsing error! " + ex.getMessage());
            } catch (FileNotFoundException ex) {
                      System.err.println("File " + fileName + " not found! " + ex);
            } finally {
                      try {
                                if (inputStream != null) {
                                          inputStream.close();
                                }
                      } catch (IOException e) {
                          System.err.println("Impossible close file "+fileName+" : "+e);
                      }
            }

    }
    private Student buildStudent(XMLStreamReader reader) throws XMLStreamException {
            Student st = new Student();
            st.setLogin(reader.getAttributeValue(null, StudentEnum.LOGIN.getValue()));
            st.setFaculty(reader.getAttributeValue(null,
                              StudentEnum.FACULTY.getValue())); // проверить на null
            String name;
            while (reader.hasNext()) {
                      int type = reader.next();
                      switch (type) {
                         case XMLStreamConstants.START_ELEMENT:
                                name = reader.getLocalName();
                                switch (StudentEnum.valueOf(name.toUpperCase())) {
```

```
                                case NAME:
                                        st.setName(getXMLText(reader));
                                        break;
                                case TELEPHONE:
                                        name = getXMLText(reader);
                                        st.setTelephone(Integer.parseInt(name));
                                        break;
                                case ADDRESS:
                                        st.setAddress(getXMLAddress(reader));
                                        break;
                                }
                                break;
                        case XMLStreamConstants.END_ELEMENT:
                                name = reader.getLocalName();
                                if (StudentEnum.valueOf(name.toUpperCase()) ==
                                        StudentEnum.STUDENT) {
                                        return st;
                                }
                                break;
                        }

                }
                throw new XMLStreamException("Unknown element in tag Student");
        }
        private Student.Address getXMLAddress(XMLStreamReader reader)
                                                                throws XMLStreamException {
                Student.Address address = new Student.Address();
                int type;
                String name;
                while (reader.hasNext()) {
                        type = reader.next();
                        switch (type) {
                                case XMLStreamConstants.START_ELEMENT:
                                        name = reader.getLocalName();
                                        switch (StudentEnum.valueOf(name.toUpperCase())) {
                                                case COUNTRY:
                                                        address.setCountry(getXMLText(reader));
                                                        break;
                                                case CITY:
                                                        address.setCity(getXMLText(reader));
                                                        break;
                                                case STREET:
                                                        address.setStreet(getXMLText(reader));
                                                        break;
                                        }
                                        break;
                                case XMLStreamConstants.END_ELEMENT:
                                        name = reader.getLocalName();
                                        if (StudentEnum.valueOf(name.toUpperCase()) ==
                                                StudentEnum.ADDRESS) {
                                                return address;
```

```
                                        }
                                        break;
                                }
                        }
                        throw new XMLStreamException("Unknown element in tag Address");
                }
                private String getXMLText(XMLStreamReader reader) throws XMLStreamException {
                        String text = null;
                        if (reader.hasNext()) {
                                reader.next();
                                text = reader.getText();
                        }
                        return text;
                }
}
```

Для запуска приложения разбора документа с помощью StAX ниже приведен достаточно простой код, аналогичный коду запуска SAX и DOM парсеров.

/ # 25 # создание и запуск StAX-парсера # */*

```
StudentsStAXBuilder staxBuilder = new StudentsStAXBuilder();
staxBuilder.buildSetStudents("data/students.xml");
System.out.println(staxBuilder.getStudents());
```

Обработку документов с помощью всех трех парсеров можно объединить в одно приложение с использованием шаблонов проектирования **Factory Method** и **Builder**. Для этого будет построена иерархия классов-builder-ов во главе с абстрактным классом **AbstractStudentsBuilder** в виде:

/ # 26 # вершина иерархии builder-ов # AbstractStudentsBuilder.java */*

```
package by.bsu.xmlstudents;
import java.util.HashSet;
import java.util.Set;
public abstract class AbstractStudentsBuilder {
        // protected так как к нему часто обращаются из подкласса
        protected Set<Student> students;
        public AbstractStudentsBuilder() {
                students = new HashSet<Student>();
        }
        public AbstractStudentsBuilder(Set<Student> students) {
                this.students = students;
        }
        public Set<Student> getStudents() {
                return students;
        }
        abstract public void buildSetStudents(String fileName);
}
```

Приведенные выше классы разбора XML-документов с помощью SAX, DOM, StAX следует адаптировать вследствие определения для них абстрактного класса и передачи ему атрибута **Set<Student> students** и метода **getStudents()**.

/ # 27 # реализации конкретных bulder-ов # StudentsSAXBuilder.java # StudentsDOMBuilder.java # StudentsStAXBuilder.java */*

```
public class StudentsSAXBuilder extends AbstractStudentsBuilder {
          private StudentHandler sh;
          private XMLReader reader;
          public StudentsSAXBuilder() {
                    // more code
          }
          public StudentsSAXBuilder (Set<Student> students) {
                    super(students);
                    // more code
          }
          @Override
          public void buildSetStudents(String fileName) {
                    // more code
          }
}
public class StudentsDOMBuilder extends AbstractStudentsBuilder {
          private DocumentBuilder docBuilder;
          public StudentsDOMBuilder() {
                    // more code
          }
          public StudentsDOMBuilder (Set<Student> students) {
                    super(students);
                    // more code
          }
          @Override
          public void buildSetStudents(String fileName) {
                    // more code
          }
}
public class StudentsStAXBuilder extends AbstractStudentsBuilder {
          private XMLInputFactory inputFactory;
          public StudentsStAXBuilder() {
                    // more code
          }
          public StudentsStAXBuilder (Set<Student> students) {
                    super(students);
                    // more code
          }
          @Override
          public void buildSetStudents(String fileName) {
                    // more code
          }
}
```

Шаблон **Factory Method** предназначен для создания объектов, находящихся в иерархической зависимости. В данной ситуации класс, реализующий шаблон, будет производить экземпляры подклассов абстрактного класса **AbstractStudentsBuilder**. Принятие решения о конкретном типе будет производиться на основании передаваемого в factory-метод строкового значения с именем желаемого парсера.

/ # 28 # фабрика для создания конкретных bulder-ов # StudentBuilderFactory.java */*

```
public class StudentBuilderFactory {
          private enum TypeParser {
                    SAX, STAX, DOM
          }
          public AbstractStudentsBuilder createStudentBuilder(String typeParser) {
                    TypeParser type = TypeParser.valueOf(typeParser.toUpperCase());
                    switch (type) {
                        case DOM:
                              return new StudentsDOMBuilder();
                        case STAX:
                              return new StudentsStAXBuilder();
                        case SAX:
                              return new StudentsSAXBuilder();
                        default:
                              throw new EnumConstantNotPresentException(
                                                  type.getDeclaringClass(), type.name());
                    }
          }
}
```

Запускать приложение из метода **main()** стало возможным с помощью кода:

/ # 29 # запуск приложения с выбором парсеров */*

```
StudentBuilderFactory sFactory = new StudentBuilderFactory();
AbstractStudentsBuilder builder = sFactory.createStudentBuilder("stax");
builder.buildSetStudents("data/students.xml");
System.out.println(builder.getStudents());
```

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 собой 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 текстовый документ другого типа.

Пусть требуется построить 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 проблемы. В следующем коде приведен файл таблицы стилей **students.xsl**, решающий поставленную задачу.

 # 30 # документ-стиль для преобразования # students.xsl

```
<?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 "{}"

 # 31 # стиль с ATV # students.xsl

```
<?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.

/ # 32 # трансформация XML # SimpleTransform.java */*

```
package by.bsu.transform;
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("data/students.xsl"));
         // установка исходного XML-документа и конечного XML-файла
         transformer.transform(new StreamSource("data/students.xml"),
                                                    new StreamResult("newstudents.xml"));
         System.out.println("Transform " + fileName + " complete");
         } catch(TransformerException e) {
               System.err.println("Impossible transform file " + fileName + " : " + ex);
         }
     }
}
```

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>Pashkin 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 способ (правила) трансформации одного документа в другой.

Какие же элементы используются в данном листинге?

```
<xsl:output method="xml" indent="yes"/>
```

Инструкция говорит о том, что конечный документ, который получится после преобразования, будет 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 **student**, содержащему атрибут **login** со значением **base**:

```
<xsl:template match="student[@login='base']">
```

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 узлов-листьев. На каждом шаге процессор пытается применить к данному элементу некоторый шаблон преобразования; если в таблице стилей для текущего просматриваемого элемента есть нужный 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 14

1. Создать файл XML и соответствующую ему схему XSD.
2. При разработке XSD использовать простые и комплексные типы, перечисления, шаблоны и предельные значения.
3. Сгенерировать класс, соответствующий данному описанию.
4. Создать приложение для разбора XML-документа и инициализации коллекции объектов информацией из XML-файла. Для разбора использовать SAX, DOM и StAX парсеры. Для сортировки объектов использовать интерфейс Comparator.
5. Произвести проверку XML-документа с привлечением XSD.
6. Определить метод, производящий преобразование разработанного XML-документа в документ, указанный в каждом 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) — предпочтительные условия произрастания: температура (в градусах), освещение (светолюбиво либо нет), полив (мл в неделю);
   — Multiplying — размножение: листьями, черенками либо семенами.
   Корневой элемент назвать Flower.
   С помощью XSL преобразовать XML-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.
   С помощью XSL преобразовать XML-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, поминутная), плата за подключение к тарифу (0–n рублей).
   Корневой элемент назвать Tariff.
   С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать тарифы по абонентской плате.
4. **Лекарственные препараты**.
   Лекарственные препараты имеют следующие характеристики:
   — Name — наименование препарата;
   — Pharm — фирма-производитель;
   — Group — группа препаратов, к которым относится лекарство (антибиотики, болеутоляющие, витамины и т. п.);
   — Analogs (может быть несколько) — содержит наименование аналога;
   — Versions — варианты исполнения (консистенция/вид: таблетки, капсулы, порошок, капли и т. п.). Для каждого варианта исполнения может быть несколько производителей лекарственных препаратов со следующими характеристиками:
     • Certificate — свидетельство о регистрации препарата (номер, даты выдачи/истечения действия, регистрирующая организация);
     • Package — упаковка (тип упаковки, количество в упаковке, цена за упаковку);
     • Dosage — дозировка препарата, периодичность приема;
   Корневой элемент назвать Medicine.
   С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать лекарства по цене.
5. **Компьютеры**.
   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.

С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать Critical.

6. **Электроинструменты**.
Электроинструменты можно структурировать по следующей схеме:
— Model — название модели;
— Handy — одно- или двуручное;
— Origin — страна производства;
— TC (должно быть несколько) — технические характеристики: энергопотребление (низкое, среднее, высокое), производительность (в единицах в час), возможность автономного функционирования и т. д.;
— Material — материал изготовления.
Корневой элемент назвать PowerTools или Power.

С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать страну производства.

7. **Столовые приборы**.
Столовые приборы можно структурировать по следующей схеме:
— Type — тип (нож, вилка, ложка и т. д.);
— Origin — страна производства;
— Visual (должно быть несколько) — визуальные характеристики: лезвие, зубец (длина лезвия, зубца [10– n см], ширина лезвия [10–n мм]), материал (лезвие [сталь, чугун, медь и т. д.]), рукоять (деревянная [если да, то указать тип дерева], пластик, металл);
— Value — коллекционный либо нет.
Корневой элемент назвать FlatWare.

С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по длине лезвия, зубца, объему.

8. **Самолеты**.
Самолеты можно описать по следующей схеме:
— Model — название модели;
— Origin — страна производства;
— Chars (должно быть несколько) — характеристики, могут быть следующими: тип (пассажирский, грузовой, почтовый, пожарный, сельскохозяйственный), количество мест для экипажа, характеристики (грузоподъемность, число пассажиров), наличие радара;
— Parameters — длина (в метрах), ширина (в метрах), высота (в метрах);
— Price — цена (в талерах).
Корневой элемент назвать Plane.

С помощью XSL преобразовать XML-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 г), жиры (в г) и углеводы (в г);
   — Production — предприятие-изготовитель.
   Корневой элемент назвать Candy.
   С помощью XSL преобразовать XML-файл в формат HTML, при выводе отсортировать по месту изготовления.
10. **Пиво**.
   — Name — название;
   — Type — тип пива (темное, светлое, лагер, живое);
   — Al — алкогольное либо безалкогольное;
   — Manufacturer — фирма-производитель;
   — Ingredients (должно быть несколько) — ингредиенты: вода, солод, хмель, сахар и т. д.;
   — Chars (должно быть несколько) — характеристики: количество оборотов (если алкогольное), прозрачность (в процентах), фильтрованное либо нет, пищевая ценность (ккал), способ разлива (объем и материал емкостей).
   Корневой элемент назвать Beer.
   С помощью XSL преобразовать XML-файл в формат XML, при выводе корневым элементом сделать производителя.
11. **Периодические издания**.
   — Title — название;
   — Type — тип (газета, журнал, буклет);
   — Monthly — переодичность выхода;
   — Chars (должно быть несколько) — характеристики: цветность (да либо нет), объем (n страниц), глянцевое (да [только для журналов и буклетов] либо нет [для газет]), подписной индекс (только для газет и журналов).
   Корневой элемент назвать Paper.
   С помощью XSL преобразовать XML-файл в формат plain text, при выводе организовать подачу информации в удобном для прочтения виде.
12. **Туристические путевки.**
   Туристические путевки, предлагаемые агентством, имеют следующие характеристики:
   — Type — тип (выходного дня, экскурсионная, отдых, паломничество и т. д.);
   — Country — страна, выбранная для путешествия;
   — Number days/nights — количество дней и ночей;

— Transport — вид перевозки туристов (авиа, ж/д, авто, лайнер);
— Hotel characteristic (должно быть несколько) — количество звезд, включено ли питание и какое (HB, BB, Al), какой номер (1-, 2-, 3-местные), есть ли телевизор, кондиционер и т. д.;
— Cost — стоимость путевки (сколько и что включено).

Корневой элемент назвать Tourist voucher.

С помощью XSL преобразовать XML-файл в формат HTML, с выводом информации в табличном виде.

13. **Старые открытки**.
— Thema — тема изображения (городской пейзаж, природа, люди, религия, спорт, архитектура...);
— Type — тип (поздравительная, рекламная, обычная). Была ли отправлена;
— Country — страна производства;
— Year — год издания;
— Author — имя автора/ов (если известен);
— Valuable — историческая, коллекционная или тематическая ценность.

Корневой элемент назвать Old Card.

С помощью XSL преобразовать XML-файл в формат PDF с выводом информации в отдельную страницу для каждого концерта.

14. **Банковские вклады**.
— Name — название банка;
— Country — страна регистрации;
— Type — тип вклада (до востребования, срочный, расчетный, накопительный, сберегательный, металлический);
— Depositor — имя вкладчика;
— Account id — номер счета;
— Amount on deposit — сумма вклада;
— Profitability — годовой процент;
— Time constraints — срок вклада.

Корневой элемент назвать Bank.

С помощью XSL преобразовать XML-файл в формат PDF с выводом информации в табличном виде.

Тестовые задания к главе 14

Вопрос 14.1.

Укажите параметры, которые можно использовать при объявлении документа xml (3):

1) errorPage
2) version

3) schema
4) encoding
5) charset
6) standalone

Вопрос 14.2.

Укажите вид документа XML, который соответствует следующему описанию: xml-документ следует синтаксическим правилам XML и правилам, определенным в его DTD или в XSD (1):
1) неправильный xml-документ;
2) правильный xml-документ;
3) неправильно-форматированный xml-документ.

Вопрос 14.3.

Укажите классы, которые входят в состав пакета javax.xml.parsers (4):
1) XMLInputFactory
2) XMLStreamReader
3) DocumentBulider
4) DocumentBuilderFactory
5) Document
6) SAXParser
7) SAXParserFactory

Вопрос 14.4.

Укажите утилиту java-платформы, с помощью которой возможно обратное создание на основе XML-схемы классов (1):
1) java;
2) rmic;
3) apt;
4) xjc.

Вопрос 14.5.

Укажите, какие события обрабатывает SAX-парсер (1):
1) событие начало документа;
2) событие конец документа;
3) событие текстовые данные;
4) событие начало элемента;
5) событие конец элемента;
6) все перечисленные события можно обработать.

Часть 3

Технологии разработки web-приложений

В третьей части даны основы программирования распределенных информационных систем с применением сервлетов и JSP, а также основные принципы создания собственных библиотек тегов.

Глава 15

СЕРВЛЕТЫ

«Пользователь» — слово, используемое компьютерщиками-профессионалами вместо слова «идиот».

Дейв Барри

Приложение Java, запускаемое и выполняемое в контейнере сервера приложений. Клиент общается с таким приложением посредством веб-браузера. Никаких дополнительных приложений на стороне клиента устанавливать не требуется. Сервлеты поддерживаются виртуальной машиной JVM, что предотвращает утечки памяти и обеспечивает функционирование garbage collection. Каждому клиенту сервлет выделяет независимый поток выполнения. Клиент посылает приложению HTTP-запрос, сервлет генерирует ответ и возвращает его клиенту в виде html-документа.

Сервлет:

— компонент приложений Java Enterprise Edition;
— загружается веб-сервером в контейнер;
— выполняется на стороне сервера;
— обрабатывает клиентские запросы;
— динамически генерирует ответы на запросы;
— находится в состоянии ожидания, если запросы отсутствуют;
— принимает запросы от других сервлетов (Servlet chaining);
— поддерживает соединения с ресурсами.

Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP. Технология сервлетов является оболочкой протокола HTTP и поддерживает его как транспорт передачи данных от клиента серверу и обратно. Контейнер сервлетов поддерживает также протокол HTTPS (HTTP и SSL) для защищаемых запросов.

Сервлеты в промышленном программировании используются для:

— приема входящих данных от клиента;
— взаимодействия с бизнес-логикой системы;
— динамической генерации ответа клиенту.

Все сервлеты реализуют общий интерфейс **Servlet** из пакета **javax.servlet**. Для обработки HTTP-запросов в web можно воспользоваться в качестве базового класса абстрактным классом **HttpServlet** из пакета **javax.servlet.http**.

Жизненный цикл сервлета начинается с его инициализации и загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый клиентский запрос. Сервлет готов к обслуживанию любого числа запросов. Завершение существования происходит при выгрузке его из контейнера.

Первым вызывается метод **init()**. Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе размещается код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом **service(HttpServletRequest request, HttpServletResponse response)** сервлета, вызываемый контейнером, а все параметры запроса упаковываются в экземпляр **request** интерфейса **HttpServletRequest**, передаваемый в сервлет. Еще одним параметром этого метода является экземпляр **response** интерфейса **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.

При выгрузке приложения из контейнера, то есть по окончании жизненного цикла сервлета, вызывается метод **destroy()**, в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

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 серверами приложений WebSphere, Weblogic, JBoss, Oracle OC4J Server, Glassfish, Geronimo, Apache Tomcat, а также контейнерами сервлетов tomcat, jetty, grizzly и являются частью платформы JEE.

Сервлеты реализуют интерфейс **Servlet**, в котором кроме рассмотренных выше методов **service()**, **init()**, **destroy()** предусмотрена реализация метода **ServletConfig getServletConfig()** — он возвращает объект, содержащий параметры конфигурации сервлета и дает доступ к среде выполнения.

При разработке сервлетов в качестве суперкласса в большинстве случаев используется не интерфейс **Servlet**, а абстрактный класс **HttpServlet**, отвечающий за обработку запросов HTTP.

Метод **service()** класса **HttpServlet** служит диспетчером для других методов, каждый из которых обрабатывает методы доступа к соответствующим ресурсам. В спецификации HTTP определены методы: **GET**, **HEAD**, **POST**, **PUT**, **DELETE**, **OPTIONS** и **TRACE**. Наиболее часто употребляются методы **GET** и **POST**, передающие на сервер запросы, а также параметры для их выполнения.

Метод **GET** (method="GET") используется для запроса содержимого указанного ресурса, изображения или гипертекстового документа. Вместе с запросом могут передаваться дополнительные параметры как часть URI, значения могут

выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер. Этот метод является основным методом взаимодействия браузера клиента и веб-сервера.

Метод **POST** используется для передачи пользовательских данных в содержимом HTTP-запроса на сервер. Пользовательские данные упакованы в тело запроса согласно полю заголовка **Content-Type** и/или включены в URI запроса. При использовании метода **POST** под URI подразумевается ресурс, который будет обрабатывать запрос.

Метод **PUT** схож с методом **POST** за тем исключением, что здесь URI подразумевает ресурс, который будет создан или сохранен на сервере в результате выполнения **PUT**-запроса.

Метод **DELETE** предназначен для удаления целевого ресурса.

Оба эти действия на некоторых серверах могут запрещаться из-за угрозы внутренней безопасности.

Метод **HEAD** предполагает возврат сервером такого же ответа, как и при использовании **GET**, но без тела ответа. Метод обычно используется для того, чтобы проверить существование ресурса либо узнать, изменился ли запрашиваемый ресурс с момента последнего обращения.

Метод **OPTIONS** должен возвращать информацию о возможностях веб-сервера или параметрах соединения для конкретного ресурса.

Метод **TRACE** возвращает клиенту запрос в том виде, в каком он пришел на сервер — используется для отладки, определяя заголовки, добавляемые промежуточными серверами, а также для тестирования настроек соединения.

Методы **HEAD**, **OPTIONS** и **TRACE**, как правило, реализуются сервером веб-приложения прозрачно для программиста и не требуют какой-либо специальной обработки. Поэтому обычно при разработке веб-приложений они не рассматриваются.

Методы **GET** и **HEAD**, по принятым соглашениям, не должны иметь другого назначения, кроме как передача информации клиенту — они должны быть «безопасными» («safe»). То есть в результате их выполнения на сервере не должно происходить никаких «побочных эффектов», которые повлияют на состояние ресурсов сервера — таких, как создание, изменение, удаление данных. Также подразумевается, что запросы с «безопасными» методами могут быть выполнены браузером клиента в автоматическом режиме, без специального разрешения пользователя. Так как ссылки через элемент <a> в коде HTML по умолчанию подразумевают использование браузером **GET**-запросов, то довольно часто можно встретить нарушение «безопасности» **GET**-запросов из-за использования элемента <a> для выполнения каких-либо действий, изменяющих состояние сервера, например, ссылки вида:

```
<a href="http://example.com/do?action=delete">Delete database</a>
```

Далеко не всегда такое нарушение влечет негативные последствия, однако так как инфраструктура глобальных сетей предполагает соблюдение соглашения

о «безопасности» **GET**-запросов, то при взаимодействии со следующими факторами могут возникать негативные последствия.

1. **Кэширующие прокси.** Так как **GET**-запрос (в отличие от **POST**) может быть кэширован промежуточным кэширующим прокси-сервером, то при обработке запроса прокси может не выполнить запрос к целевому серверу, а сразу вернуть результат. Таким образом, если запрос подразумевал какое-то действие над данными на сервере, то это действие не будет выполнено.
2. **Предзагрузка.** Некоторые браузеры для ускорения навигации выполняют загрузку страниц и других ресурсов по ссылкам еще до того, как пользователь выполнил переход по этим ссылкам. В этом случае при нарушении «безопасности» браузер выполнит запрос, который не был инициирован пользователем. Например, если на странице расположены ссылки, удаляющие какие-то объекты через **GET**-запросы, то при открытии этой страницы браузер может вызвать удаление всех этих объектов.
3. **Поисковые системы и другие клиенты, действующие в автоматическом режиме.** Разнообразные автоматические клиенты не могут отличить запросы, которые выполняют какие-либо действия от запросов, которые просто возвращают информацию. Поэтому они обычно ориентируются по методу запроса: **GET**-запросы выполняются и используются для дальнейшего анализа, а **POST**-запросы игнорируются. При нарушении «безопасности» **GET**-запросов возникает ситуация, аналогичная предзагрузке: выполняются те действия, которые при «нормальном» взаимодействии с приложением не должны были бы выполняться.
4. **Нарушение авторизации**. Если выполнение действия предполагает наличие определенных полномочий, то использование **GET**-запроса для этого предоставляет злоумышленнику возможность воспользоваться привилегиями другого пользователя (например, путем размещения ссылки с нужным **GET**-запросом в <img> элементе на произвольном сайте, который может посетить пользователь, обладающий нужными полномочиями с идентификацией через cookies) и тем самым выполнить неавторизованное действие.

Поэтому важно понимать различия между методами **GET** и **POST** и использовать **GET** только там, где выполняется только непосредственная передача данных без выполнения активных действий.

Методы **GET**, **HEAD**, **PUT** и **DELETE** должны быть идемпотентными («idempotence») в том смысле, что повторное (более одного раза) выполнение запросов с помощью этих методов будет иметь тот же эффект, что и однократное выполнение.

По умолчанию параметры передаются в запрос в формате:

```
?имя1=значение&имя2=значение
```

Однако форматы упаковки параметров могут быть самые разные, например, в случае передачи файлов с использованием формы

```
enctype="multipart/form-data"
```

Также часто вместо формата представления параметров вида

```
/article.jsp?id=1234&page=4
```

может быть использована так называемая Friendly URL форма, более понятная для человека, не являющегося программистом, и может быть представлена подобным образом:

```
/article/1234/page/4
```

либо

```
/article_1234/page_4
```

Однако такая форма представления параметров обычно требует дополнительной работы со стороны программиста.

В задачу метода **service()** класса **HttpServlet** входит анализ полученного через запрос метода доступа к ресурсам и вызов метода со сходным именем, где перед именем добавляется префикс **do**: **doGet()**, **doPost()**, **doHead()**, **doPut()**, **doDelete()**, **doOptions()** и **doTrace()**. Разработчик должен переопределить нужный метод, разместив в нем функциональную логику.

Механизмами протокола HTTP не предусмотрено сохранение информации о своем клиенте. Однако веб-приложение может требовать информации о клиенте, особенно такое, реализация которого предусматривает аутентификацию клиента без необходимости ее подтверждения при смене страниц. Кроме поддержки распределенной сессии, о которой разговор пойдет в главе 21, существуют и примитивные механизмы получения сведений о клиенте:

— файлы cookie — текстовые файлы, ассоциированные с приложением и сохраняемые на компьютере клиента.
— добавление в строку GET-запроса дополнительных параметров.
— скрытые HTML-теги вида

```
<input type="hidden" name ="userType" value ="admin">
```

которые необходимо добавлять в каждую страницу приложения для сохранения целостности ее архитектуры.

В следующем примере приведен готовый к выполнению, но не к практическому применению, шаблон сервлета:

// # 1 # простейший сервлет # FirstServlet.java

```
package by.bsu.first.servlet;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import javax.servlet.annotation.WebServlet;
@WebServlet("/FirstServlettest")
public class FirstServlet extends HttpServlet {
          public FirstServlet() {
                    super();
          }
          public void init() throws ServletException {
          }
          protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
                    response.setContentType("text/html");
                    response.getWriter().print("This is " + this.getClass().getName()
                                                            + ", using the GET method");
          }
          protected void doPost(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
                    response.setContentType("text/html");
                    response.getWriter()("This is " + this.getClass().getName()
                                                            + ", using the POST method");
          }
          public void destroy() {
                    super.destroy(); // Just puts "destroy" string in log
          }
}
```

С версии Servlet API 3.0 регистрация сервлетов в файле-дескрипторе **web.xml** необязательна. Конфигурация определяется с помощью аннотации **@WebServlet** и в расширенном виде с указанием имени сервлета в контексте:

```
@WebServlet(name = "FirstServletname", urlPatterns = {"/FirstServlettest"})
public class FirstServlet extends HttpServlet {
}
```

Аннотация **@WebServlet** может содержать и другие методы-члены, о чем будет сказано ниже. Также аннотированию подлежат методы реализации функциональности сервлета.

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 росту объема кода сервлета, который на определенном этапе становится неконтролируемым и реализует вследствие этого антишаблон «Волшебный сервлет». Приведенный выше сервлет имеет признаки антишаблона, так как содержит метод **print()** для формирования кода HTML. Сервлет должен использоваться только для контроля реализации бизнес-логики приложения и обязан быть отделен как от непосредственного формирования текста ответа на запрос, так и от данных, необходимых для этого. Обычно для формирования ответа на запрос применяются возможности JSP, JSPX, JSF и XHTML. Признаки наличия антишаблонов все же будут встречаться ниже, но это отступление сделано только с точки зрения компактности примеров.

Сервлет является компонентом веб-приложения, который будет называться **FirstProject** и размещаться в папке **/WEB-INF/classes** проекта.

Запуск контейнера сервлетов и размещение проекта

Здесь и далее применяется веб-сервер 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 **8087**.

Ниже приведены необходимые действия по запуску сервлета из предыдущего примера с помощью сервера Tomcat, который установлен в каталоге **/Apache Software Foundation/Tomcat[***версия***]**. В этом же каталоге размещаются следующие подкаталоги:

/bin — содержит файлы запуска монитора и контейнера сервлетов вида **tomcat[***номер***].exe**, **tomcat[***номер***]w.exe** и некоторые необходимые для этого библиотеки;

/lib — содержит библиотеки служебных классов, в частности Servlet API и JSP API; используемые внешние библиотеки (если они есть), упакованные в JAR-файлы;

/conf — содержит конфигурационные файлы, в частности, конфигурационный файл сервера **server.xml**;

/logs — в этот каталог записываются log-файлы, инициированные сервером;

/webapps — в этот каталог помещаются папки с веб-приложениями, содержащие в свою очередь сервлеты и другие компоненты конкретного приложения.

В каталог **/webapps** необходимо поместить папку **/FirstProject** с вложенным в нее сервлетом **FirstServlet**. Кроме того, папка **/FirstProject** должна содержать каталог **/WEB-INF**, в котором помещаются подкаталоги:

/classes — содержит class-файл сервлета **by.bsu.first.servlet.FirstServlet**;

/src — содержит исходный файл сервлета **FirstServlet.java** (опционально);

а также **web.xml** — дескриптор доставки (развертывания) приложения располагается в каталоге **/WEB-INF**.

В файле **web.xml** для версий Servlet API, не превышающих 2.5, 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 путь к сервлету в браузере не будет совпадать с истинным именем класса сервлета. Например:

 # 2 # простейший дескрипторный файл # web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns=http://java.sun.com/xml/ns/javaee
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID" version="2.5">
<display-name>FirstProject</display-name>
<servlet>
          <display-name>FirstServletdisplay</display-name>
          <servlet-name>FirstServletname</servlet-name>
          <servlet-class>by.bsu.first.servlet.FirstServlet</servlet-class>
</servlet>
<servlet-mapping>
          <servlet-name>FirstServletname</servlet-name>
          <url-pattern>/FirstServlettest</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>
</web-app>
```

Здесь указано имя сервлета **FirstServletname**, путь к откомпилированному классу сервлета **FirstServlet.class**, а также URL-имя сервлета **FirstServlettest**, по которому происходит его вызов.

Для версии сервлетов 3.0 теги мэппинга не нужны, так как эта информация определяется аннотацией.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">
  <display-name>FirstProject</display-name>
  <session-config>
          <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
          <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Таким образом, требуется выполнить действия:

1) компиляцию сервлета с указанием в **–cp** пути к архиву **servlet-api.jar**;
2) полученный файл класса **FirstServlet.class** поместить в каталог **/FirstProject/WEB-INF/classes/by/bsu/first/servlet**;
3) в каталог **/FirstProject/WEB-INF** поместить файл конфигурации **web.xml**;
4) в каталог **/FirstProject** поместить файл стартовой JSP-страницы **index.jsp**;
5) разместить папку **/FirstProject** в каталог **/webapps** сервера Tomcat;
6) стартовать Apache Tomcat;
7) запустить браузер и ввести адрес

```
http://localhost:8080/FirstProject/FirstServlettest
```

   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="FirstServlettest">
        <input type="submit" value="Execute">
</form>
```

Для запуска проекта в браузере набирается строка

```
http://localhost:8080/FirstProject/index.jsp
```

или

```
http://localhost:8080/FirstProject/
```

Сервлет будет вызван из JSP-страницы по URL-имени **FirstServlettest**, и в результате в браузер будет выведено:

Рис. 15.1. *Вывод сервлета после вызова метода doGet()*

Простая JSP-страница

Технология 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.

JSP-код состоит из специальных тегов и выражений, которые указывают контейнеру соответствие между тегами и java-кодом для генерации сервлета или его части. Таким образом поддерживается документ, который одновременно содержит и статическую страницу, и теги Java, управляющие страницей. Статические части HTML-страниц посылаются в виде строк в метод **write()**. Динамические части включаются прямо в код сервлета. С момента формирования ответа сервера страница ведет себя как обычная HTML-страница с ассоциированным сервлетом.

Чтобы создать простейшую JSP, достаточно взять HTML-страницу и заменить расширение **html** на **jsp**. Только в этом случае для запуска страницы необходим специальный application server с контейнером сервлетов и особое размещение самой страницы.

Взаимодействие сервлета и JSP

Страницы JSP и сервлеты никогда не следует использовать 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.

При необходимости расширения функциональности системы *не следует* создавать дополнительные сервлеты. Сервлет в приложении должен быть *один*. При создании нового учебного приложения во избежание путаницы всегда следует создавать новый проект.

Для демонстрации взаимодействия JSP-страниц и сервлета будет решена задача определения времени между загрузкой страницы в браузер и нажатием кнопки на этой же странице.

Страница JSP с последующим вызовом другой JSP-страницы, отображающей результаты выполнения запроса.

 # 3 # страница JSP с вызовом сервлета # index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html><head><title>JSP Timing</title>
    </head>
    <body>
        <h5>Счетчик времени от запуска приложения до нажатия кнопки</h5>
```

```
        <jsp:useBean id="calendar" class="java.util.GregorianCalendar"/>
        <form name="Simple" action="timeaction" method="POST">
            <input type="hidden" name="time" value="${calendar.timeInMillis}"/>
            <input type="submit" name="button" value="Посчитать время"/>
        </form>
</body></html>
```

В результате запуска стартовой страницы проекта в браузер будет выведено:

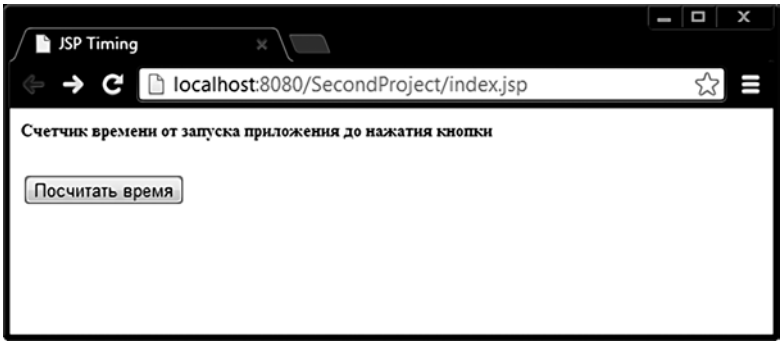

Рис. 15.2. *Стартовая страница приложения*

Кодировка для символов кириллицы задана с помощью директивы **page**. Action-тег **useBean** используется для создания объекта класса **GregorianCalendar** в области видимости JSP. Сервлет **TimeServlet** вызывается методом **POST**. Ему передается с переменной **calendar** информация о времени выполнения страницы.

// # 4 # простой контроллер # ServletTiming.java

```
package by.bsu.second.servlet;
import java.io.IOException;
import java.util.GregorianCalendar;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.annotation.WebServlet;
@WebServlet("/timeaction")
public class TimeServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
        processRequest(request, response);
    }
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
```

```
        processRequest(request, response);
    }
  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
          GregorianCalendar gc = new GregorianCalendar();
          String timeJsp = request.getParameter("time");
          float delta = ((float)(gc.getTimeInMillis() - Long.parseLong(timeJsp)))/1_000;
        request.setAttribute("res", delta);
        request.getRequestDispatcher("/jsp/result.jsp").forward(request, response);
    }
}
```

Обмен информацией между JSP и сервлетом чаще всего осуществляется с помощью атрибутов объектов **HttpServletRequest**, **HttpSession**, **HttpServletContext**. Вызов **result.jsp** из сервлета в данном случае производится методом **forward()** интерфейса **RequestDispatcher**.

Для вызова JSP по относительному пути применяется метод **forward()**, для обращения к JSP по абсолютному пути используется метод **sendRedirect()**. Отличие этих методов состоит в том, что с методом **forward()** передается уже существующий объект запроса **request**, а при вызове метода **sendRedirect()** формируется новый запрос. Информацию в последнем случае следует передавать с другими объектами. К тому же метод **forward()** срабатывает быстрее.

 # 5 # страница, вызванная сервлетом # result.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html><head><title>Result Page</title></head><body>
         <p>Кнопка нажата через ${res} сек </p>
</body></html>
```

После обращения к сервлету и выполнения им действий по созданию атрибута объекта **request** будет вызвана страница **result.jsp**, интерпретирующая результат в виде:

Рис. 15.3. *Вывод информации страницей result.jsp*

В данном коде используется Expression Language (EL) для доступа к атрибутам запроса в виде **${calendar.timeInMillis}** и **${res}**.

Кроме сокращенной нотации вызова методов с префиксом **get** по умолчанию можно использовать полную форму вызова метода с помощью EL, а именно: **${calendar.getTimeInMillis()}** или **${calendar.isLeapYear(2013)}**. Методы, которые не возвращают значение, вызывать нельзя.

Интерфейс ServletContext

Каждому приложению ставится в соответствие единственный экземпляр, ассоциированный с контейнером сервлетов, реализующий **ServletContext**. Контекст выполнения сервлета предоставляет средства для общения с application-сервером и позволяет получать информацию о среде выполнения, а также использовать ресурсы совместно с другими объектами приложения. В частности, можно добавить/извлечь/удалить атрибуты, извлечь параметры контекста или записать информацию в log-файл. Получить ссылку на объект **ServletContext** можно вызовом метода **getServletContext()** на экземпляре сервлета.

Ряд методов предназначен для управления атрибутами, с помощью которых передается информация между различными компонентами приложения (JSP, сервлетами):

void setAttribute(String name, Object value) — добавляет атрибут;

Object getAttribute(String name) — получает значение атрибута по имени;

Enumeration getAttributeNames() — получает список имен атрибутов;

void removeAttribute(String name) — удаляет атрибут из контекста.

Другими словами, используя объект, реализующий **ServletContext**, можно связывать объекты с экземплярами сервлета, сессии и запроса.

Приложению могут быть заданы параметры инициализации (параметры контекста), получить доступ к которым можно с помощью методов:

String getInitParameter(String paramName) — извлечение параметра контекста по имени;

Enumeration getInitParameterNames() — извлечение набора имен параметров контекста, например, из кода метода сервлета

```
String admin = this.getServletContext().getInitParameter("administrator");
```

Для получения значения, необходимо, чтобы дескрипторный файл приложения **web.xml** содержал тег **<context-param>** со значением и именем параметра в виде:

```
<context-param>
        <param-name>administrator</param-name>
        <param-value>blinov(a)gmail(dot)com</param-value>
</context-param>
```

Экземпляр **ServletContext** может применяться для обмена информацией между сервлетами приложения. Но в большей части промышленных приложений

применяется односервлетная модель, поэтому использовать объект контекста следует только для хранения информации о конфигурации приложения или общей информации для всех клиентов.

Следующие методы позволяют получить из контекста сервлета базовую информацию:

String getRealPath(String filename) — определение истинного маршрута файла относительно каталога, в котором сервер хранит документы;

String getServerInfo() — возвращает имя и номер версии контейнера сервлетов;

String getContextPath() — возвращает имя веб-приложения, находящегося в папке контейнера сервлетов, для Tomcat это будет папка **webapps**;

String getServletContextName() — возвращает имя веб-приложения, заданное в дескрипторе приложения в теге **<display-name>**;

void log(String msg) — запись лога с сообщением **msg**;

void log(String msg, Throwable t) — запись лога с текстовым сообщением **msg** и сообщением, извлекаемым из исключения.

Интерфейс ServletConfig

Параметры инициализации сервлета содержатся в экземпляре **ServletConfig**. Получить доступ к нему можно методом **getServletConfig()**, вызываемым на экземпляре сервлета.

Методы интерфейса:

Enumeration getInitParameterNames() — получение списка имен параметров инициализации сервлета;

String getInitParameter(String name) — определение значения конкретного параметра по его имени;

String getServletName() — определение имени сервлета, с которым связан текущий объект **ServletConfig**;

String getServletContext() — получение экземпляра контекста.

Чтобы задать параметры инициализации сервлета **EMailServlet**, который будет рассмотрен позже, необходимо в тег <**servlet**> дескрипторного файла **web.xml** вложить тег <**init-param**> с описанием имени и значения параметра в виде:

```
<servlet>
  <servlet-name>EMailServlet</servlet-name>
  <servlet-class>by.bsu.mail.servlet.EMailServlet</servlet-class>
        <init-param>
            <param-name>mail.smtps.host</param-name>
            <param-value>smtp.gmail.com</param-value>
        </init-param>
        <init-param>
            <param-name>mail.smtp.port</param-name>
```

```
            <param-value>465</param-value>
        </init-param>
        <init-param>
            <param-name>smtps.auth.user</param-name>
            <param-value>blinov@gmail.com</param-value>
        </init-param>
        <init-param>
            <param-name>smtps.auth.pass</param-name>
            <param-value>real_password</param-value>
        </init-param>
        <init-param>
            <param-name>mail.transport.protocol</param-name>
            <param-value>smtps</param-value>
        </init-param>
</servlet>
```

или для версии 3.0:

```
@WebServlet(
          urlPatterns = { "/mailservlet" },
          initParams = {
                    @WebInitParam(name = "mail.smtps.host", value = "smtp.gmail.com"),
                    @WebInitParam(name = "mail.smtp.port", value = "465")
                    @WebInitParam(name = "smtps.auth.user", value = "blinov@gmail.com")
                    @WebInitParam(name = "smtps.auth.pass", value = "real_password")
                    @WebInitParam(name = "mail.transport.protocol", value = "smtps")
                    })
public class EMailServlet extends HttpServlet { }
```

Тогда для доступа к параметрам инициализации сервлета и их дальнейшего использования в приложении, например, в коде метода **init()** сервлета следует применить:

```
          ServletConfig sc = this.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);
                       //...
          }
```

Интерфейс HttpServletRequest

Информация от клиента веб-приложения отправляется серверу в виде объекта запроса типа **HttpServletRequest**. Данный интерфейс является производным от интерфейса **ServletRequest**. Используя методы интерфейса **ServletRequest**,

можно получить: информацию о сервлете, доступ к сессии и параметрам запроса, управление атрибутами и кодировками, а также детали протокола HTTP.

При обращении к серверу с запросом от клиента 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**;

String getCharacterEncoding() — определение символьной кодировки запроса;

void setCharacterEncoding(String code) — задание 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;

String getQueryString() — извлечение строки HTTP-запроса;

String getMethod() — определение имени метода доступа к ресурсам, на основе которого построен 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.

Непосредственно в интерфейсе **HttpServletRequest** объявлен ряд методов, позволяющих связывать внешние ресурсы с запросом:

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;

void removeAttribute(String name) — удаление атрибута по имени;

Cookie[] getCookies() — извлечение массива cookie, полученного с запросом. Файл cookie — маленький символьный файл, сохраняемый приложением на стороне клиента;

Ниже рассматривается применение некоторых методов интерфейса **HttpServletRequest** для получения данных о запросе, посылаемом методом **GET** и генерации ответа клиенту. Вызов методов в сервлете **FirstServlet** проекта **FirstProject** на объекте **request**

```
PrintWriter out= response.getWriter();
out.println(request.getMethod());
out.println(request.getRequestURL());
out.println(request.getProtocol());
out.println(request.getRemoteAddr());
out.println(request.getContextPath());
out.println(request.getScheme());
```

и передача результатов в браузер с помощью потока **PrintWriter** дадут следующие результаты:

Рис. 15.4. *Извлечение информации из запроса*

Из запроса также можно извлечь информацию о заголовке запроса, которая оказывается достаточно исчерпывающей:

```
Enumeration< String > e = request.getHeaderNames();
          while (e.hasMoreElements()) {
                    String name = e.nextElement();
                    String value = request.getHeader(name);
                    out.println(name + " = " + value);
          }
```

Результат выполнения фрагмента кода изображен на рис. 15.5.

Последний пример следует попробовать вызывать на разных компьютерах с различными конфигурациями, используя несколько браузеров, чтобы оценить выводимую информацию.

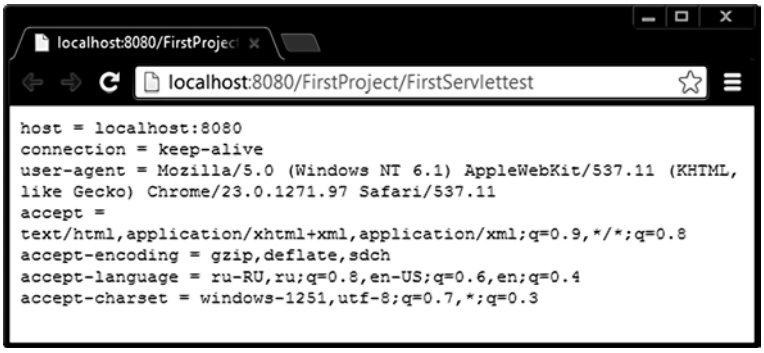

Рис. 15.5. *Извлечение информации из заголовка запроса*

Таким образом, при обращении клиента по адресу URL контейнер сервлета перехватывает запрос и вызывает метод **doGet()** или **doPost()**. Эти методы вызываются после конфигурации и создания объектов, реализующих интерфейсы **HttpServletRequest**, **HttpServletResponse**. Задача методов **doGet()** и **doPost()** — взаимодействие с HTTP-запросом клиента и создание HTTP-ответа, основанного на данных запроса и результатах взаимодействия с бизнес-логикой.

Интерфейс HttpServletResponse

Генерируемые сервлетами данные пересылаются серверу-контейнеру с помощью объектов, реализующих интерфейс **ServletResponse**, а сервер, в свою очередь, формирует и пересылает ответ клиенту, инициировавшему запрос.

В интерфейсе **HttpServletResponse**, наследующем интерфейс **ServletResponse**, есть несколько важных методов:

void setContentType(String type) — установка MIME-типа генерируемых документов;

void addCookie(Cookie cookies) — добавление файла cookie к объекту ответа для пересылки на клиентский компьютер;

void sendError(int sc, String msg) — сообщение о возникших ошибках, где **sc** — код ошибки, **msg** — текстовое сообщение;

void sendRedirect(String location) — переход по указанному адресу, без передачи экземпляра **request**.

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.

Атрибуты и параметры

Какие объекты и каким образом могут получать, передавать, предоставлять информацию? Ответ представлен в таблице:

	Атрибут	**Параметр**
Носитель	page (PageContext) request(HttpServletRequest) session (HttpSession) application (ServletContext)	context (инициализация) servlet (инициализация) request (запрос)
Метод установки	setAttribute(String name, Object value)	добавить параметры можно только к request
Метод извлечения	getAttribute(String name)	getInitParameter(String name) getParameter(String name)
Возвращаемый тип	Object	String и String[]

Рис. 15.6. *Таблица носителей информации*

Чаще всего для передачи информации между элементами системы используются объекты классов **HttpServletRequest** и **HttpSession**.

Многопоточность в сервлете

Контейнер сервлетов для каждого клиента создает независимый поток выполнения. Вероятна ситуация, когда два клиента одновременно обратятся к общему для всех потоков ресурсу. Метод **service()** и, соответственно, **doGet()**, **doPost()** должны быть реализованы с учетом процессов многопоточности. Любой доступ к разделяемым ресурсам, которыми могут быть файлы, объекты, необходимо защитить ключевым словом **synchronized**. Ниже приведен пример посимвольного формирования строки сервлетом с паузой в 500 миллисекунд между добавлениями символов к этой строке, что позволяет другим клиентам, вызвавшим сервлет, успеть вклиниться в процесс вывода при отсутствии синхронизации.

/ # 6 # доступ к синхронизированным ресурсам # ServletSynchronization.java */*

```
package by.bsu.multi.servlet;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.servlet.annotation.WebServlet;
@WebServlet("/ServletSynchronization")
public class ServletSynchronization extends HttpServlet {
  // синхронизируемый объект
  private final StringBuilder locked = new StringBuilder();
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
                                                    throws ServletException, IOException {
        Writer out = res.getWriter();
        out.write(makeString());
        out.flush();
  }
  private String makeString() {
  // 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) {
            // пустой
      }
      String result = locked.toString();
      locked.delete(0, SYNCHRO.length());
      return result;
    } // ending synchronized block
  }
}
```

Результаты работы сервлета при наличии и отсутствии синхронизации в браузере Google Chrome представлены на рисунках.

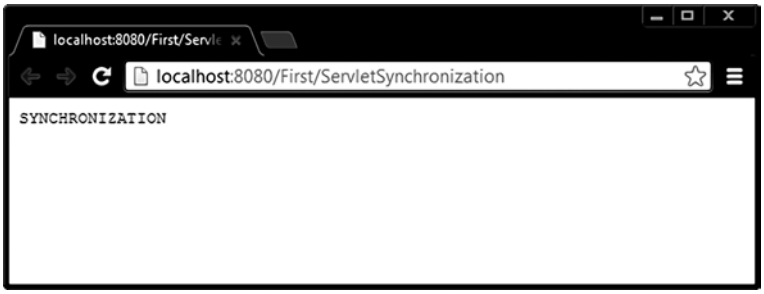

Рис. 15.7. *Результат работы сервлета с блоком синхронизации*

Если закомментировать блок синхронизации, результат будет таким, как показано на рис. 15.8.

Получить такой результат возможно при вызове данного приложения физически разными клиентами с разных компьютеров. Браузер Google Chrome организует последовательный доступ от одного клиента к одному приложению, и проблем с одновременным доступом к общему объекту не возникает.

Можно синхронизировать и весь сервлет целиком, но причиной, почему это никогда и никем не делается, является возможность нахождения критической секции вне основного пути выполнения программы.

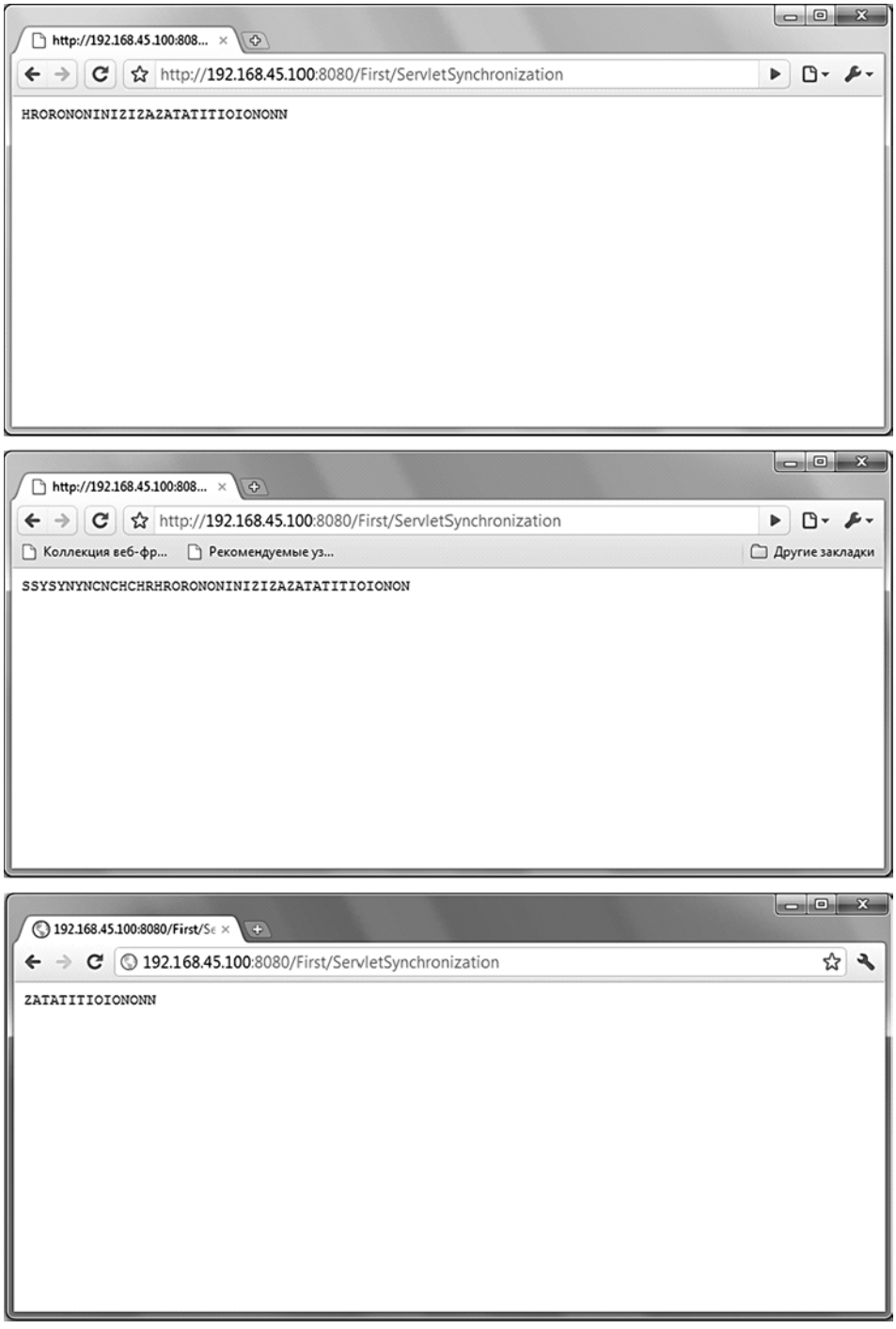

Рис. 15.8. *Результат работы сервлета без синхронизации*

Многопоточность и электронная почта

Распределенное приложение может быть эффективным только в случае, если оно способно принимать информацию от физически удаленных клиентов и производить ее обработку максимально быстро. Увеличить скорость работы веб-приложения позволяет применение многопоточных режимов функционирования частей приложения.

Рассылка электронной почты, в том числе и автоматическая, является стандартным процессом при использовании веб-приложений. Собственный почтовый сервер создать легко, только необходимо указать адрес почтового сервера, который будет использован в качестве транспорта.

При построении простейшего почтового сервера требуются интерфейсы API JavaMail. Библиотека JavaMail содержит классы, позволяющие моделировать систему электронной почты. Класс **javax.mail.Session** представляет сеанс почтовой связи, класс **javax.mail.Message** — почтовое сообщение, класс **javax.mail.internet.InternetAddress** — адрес электронной почты.

Необходимую библиотеку, содержащую, в частности, архив **mail.jar**, следует загрузить по адресу oracle.com/technetwork/java/javamail/. Далее следует добавить этот файл в каталог jar-файлов серверу приложений (**lib** для Tomcat).

В случае автономного запуска без использования промежуточного сервиса для успешной работы почтового приложения бывает необходимо запустить почтовую программу James, являющуюся также одним из проектов **apache.org**.

Ниже приведена страница JSP, содержащая форму для заполнения основных полей: «**Кому**» — «**Send to**», «**Тема сообщения**» — «**Subject**», «**Текст письма**» — «**Body**»

 # 7 # страница создания электронного письма # index.jsp

```
<%@ page language="java" contentType= "text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
          <head>
                    <title>JavaMail</title>
          </head>
          <body>
          <form action="MailServlet" method="POST">
                    <table>
                              <tr>
                                        <td>Send to:</td>
                                        <td><input type="text" name="to"/></td>
                              </tr>
                              <tr>
                                        <td>Subject:</td>
                                        <td><input type="text" name="subject"/></td>
                              </tr>
                    </table>
```

```
                    <hr/>
                    <textarea type="text" name="body" rows="5" cols="45">Message text
                    </textarea>
                    <br/><br/>
                    <input type="submit" value="Send message!"/>
          </form>
</body>
</html>
```

Рис. 15.9. *Формирование запроса на отправку письма*

Параллельные процессы по отправке письма и предложению пользователю в это же самое время создать новое письмо организуются с применением потока в следующем сервлете.

// # 8 # сервлет почтового приложения # MailServlet.java

```
package by.bsu.mail.servlet;
import java.io.IOException;
import java.util.Properties;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.mail.action.MailThread;
import javax.servlet.annotation.WebServlet;
@WebServlet("/MailServlet")
public class MailServlet extends HttpServlet {
          @Override
          protected void doPost (HttpServletRequest request,
                    HttpServletResponse response) throws ServletException, IOException {
                    // запрос параметров почтового сервера из mail.properties
                    Properties properties = new Properties();
```

```
                ServletContext context = getServletContext();
                String filename = context.getInitParameter("mail");
                // загрузка параметров почтового сервера в объект свойств
                properties.load(context.getResourceAsStream(filename));
                MailThread mailOperator =
                        new MailThread(request.getParameter("to"),
                                request.getParameter("subject"),
                                request.getParameter("body"),
                                properties);
                // запуск процесса отправки письма в отдельном потоке
                mailOperator.start();
                // переход на страницу с предложением о создании нового письма
                request.getRequestDispatcher("/send.jsp").forward(request, response);
        }
}
```

Конфигурация почтового сервера размещена в файле **/config/mail.properties**.

```
#####################################
######### Session properties #########
#####################################
mail.smtp.host=smtp.gmail.com
mail.smtp.port=465
mail.user.name=REAL~MAIL~NAME@gmail.com
mail.user.password=REAL~PASSWORD
```

 # 9 # страница перехода к созданию нового сообщения # send.jsp

```
<html>
        <head><title>Send Result</title></head>
        <body>
                <p>Sending in progress...</p>
                <a href="index.jsp">Return to new mail page</a>
        </body>
</html>
```

В результате в браузер будет выведено:

Рис. 15.10. *Формирование запроса на отправку письма*

Здесь **Return to new mail page** 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.

Недостатком такого применения многопоточности будет отсутствие информации об успешном отправлении письма или возникших при этом ошибках и исключениях. В данной ситуации такую информацию можно только сохранить в логах или выдавать на специальных страницах. Поэтому современные почтовые онлайн системы и не используют многопоточность в таком виде, так как пришлось бы делать достаточно сложный интерфейс по сообщению пользователю об ошибках при отправке почты, что сделало бы работу в системе более сложной и противоречивой.

// # 10 # поток, отправляющий почтовые сообщения # MailThread.java

```
package by.bsu.mail.action;
import java.util.Properties;
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 by.bsu.mail.creator.SessionCreator;
public class MailThread extends Thread {
          private MimeMessage message;
          private String sendToEmail;
          private String mailSubject;
          private String mailText;
          private Properties properties;
          public MailThread(String sendToEmail,
                              String mailSubject, String mailText, Properties properties) {
                    this.sendToEmail = sendToEmail;
                    this.mailSubject = mailSubject;
                    this.mailText = mailText;
                    this.properties = properties;
          }
          private void init() {
                    // объект почтовой сессии
                    Session mailSession = (new SessionCreator(properties)).createSession();
                    mailSession.setDebug(true);
                    // создание объекта почтового сообщения
                    message = new MimeMessage(mailSession);
                    try {
                              // загрузка параметров в объект почтового сообщения
                              message.setSubject(mailSubject);
                              message.setContent(mailText, "text/html");
```

```
                    message.setRecipient(Message.RecipientType.TO,
                                                    new InternetAddress(sendToEmail));
                } catch (AddressException e) {
                    System.err.print("Некорректный адрес:" + sendToEmail + "  " + e);
                    // in log file
                } catch (MessagingException e) {
                    System.err.print("Ошибка формирования сообщения" + e);
                    // in log file
                }
        }
        public void run() {
                init();
                try {
                    // отправка почтового сообщения
                    Transport.send(message);
                } catch (MessagingException e) {
                    System.err.print("Ошибка при отправлении сообщения" + e);
                    // in log file
                }
        }
}
```

*// # 11 # конфигурирование почтовой сессии # SessionCreator.java */*

```
package by.bsu.mail.creator;
import java.util.Properties;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
public class SessionCreator {
        private String smtpHost;
        private String smtpPort;
        private String userName;
        private String userPassword;
        private Properties sessionProperties;
        public SessionCreator(Properties configProperties) {
                smtpHost = configProperties.getProperty("mail.smtp.host");
                smtpPort = configProperties.getProperty("mail.smtp.port");
                userName = configProperties.getProperty("mail.user.name");
                userPassword = configProperties.getProperty("mail.user.password");
                // загрузка параметров почтового сервера в свойства почтовой сессии
                sessionProperties = new Properties();
                sessionProperties.setProperty("mail.transport.protocol", "smtp");
                sessionProperties.setProperty("mail.host", smtpHost);
                sessionProperties.put("mail.smtp.auth", "true");
                sessionProperties.put("mail.smtp.port", smtpPort);
                sessionProperties.put("mail.smtp.socketFactory.port", smtpPort);
                sessionProperties.put("mail.smtp.socketFactory.class",
                                                "javax.net.ssl.SSLSocketFactory");
                sessionProperties.put("mail.smtp.socketFactory.fallback", "false");
                sessionProperties.setProperty("mail.smtp.quitwait", "false");
```

```
        }
        public Session createSession() {
                return Session.getDefaultInstance(sessionProperties,
                        new javax.mail.Authenticator() {
                          protected PasswordAuthentication getPasswordAuthentication() {
                            return new PasswordAuthentication(userName, userPassword);
                          }
                        });
        }
}
```

Параметры контекста сервлета в файле **web.xml** для данного приложения представлены в виде:

```
<context-param>
        <param-name>mail</param-name>
        <param-value>/WEB-INF/classes/config/mail.properties</param-value>
</context-param>
```

Задания к главе 15

Вариант А

Создать сервлет и взаимодействующие с ним классы 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. вывод информации 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 0º С; г) три самых теплых дня;

9. игра с сервером в «21»;
10. реализация адаптивного текста 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 пробельный символ;

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 15

Вопрос 15.1.

Параметры конфигурации сервлета были заданы в файле-дескрипторе развертывания сервлета. Указать, с помощью методов какого интерфейса можно прочитать эти параметры в сервлете (1):

1) Servlet
2) ServletRequest
3) ServletResponse
4) ServletConfig
5) ServletContext

Вопрос 15.2.

Выбрать вариант URL, определяемый в адресной строке браузера и отправляющий http-запрос сервлету методом POST (1):

1) http://localhost:8080/Quests/Quest?method=post
2) POST http://localhost:8080/Quests/Quest

3) [POST]http://localhost:8080/Quests/Quest
4) http://localhost:8080/Quests/Quest[post]
5) ничего из вышеперечисленного

Вопрос 15.3.

Выбрать метод, который за жизненный цикл сервлета может быть вызван более одного раза (1):

1) init()
2) service()
3) destroy()
4) конструктор сервлета

Вопрос 15.4.

Дан код:

```
public class Quest extends HttpServlet {
        protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
                String str1 = request.getParameter("param1");
                String str2 = request.getParameter("param2");
                response.getWriter().println(str1 + " " + str2);
        }}
```

Указать вариант записи URL, в результате запроса которого в сервлет инициализирует переменную str1 в значение one, а переменную str2 в значение two:

1) http://localhost:8080/Tests3-18/Quest?param1=\"one\"&param2=\"two\"
2) http://localhost:8080/Tests3-18/Quest?param1="one"&param2="two"
3) http://localhost:8080/Tests3-18/Quest?param1=one&param2=two
4) http://localhost:8080/Tests3-18/Quest?param1=one+param2=two
5) http://localhost:8080/Tests3-18/Quest?param1=one?param2=two
6) http://localhost:8080/Tests3-18/Quest?param1="one"?param2="two"

Глава 16

JAVA SERVER PAGE

В стране непуганых дизайнеров существует страшное поверье: если дизайнер увидит код java на странице jsp, то он сойдет с ума и ему не поможет даже «кока-кола».

Цитата из javatutor.net

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems (в настоящее время поглощена компанией Oracle), 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 содержания и вида ответа, страницы JSP выполняют функцию отображения результатов работы приложения в виде текстовых документов типа HTML, XML, WML и некоторых других. JSP поддерживают как JavaScript, так и HTML-теги. JavaScript обычно используется, чтобы добавить функциональные возможности на уровне HTML-страницы.

Принято разделять динамическое и статическое содержимое JSP.

Динамические ресурсы. Результаты их деятельности изменяются во время выполнения приложения. Обычно представлены в виде выражений Expression Language, библиотек тегов и тегов разработчика.

Статические ресурсы. Не изменяются сами 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 (JSP Engine) и в большинстве случаев в доступе к уровню данных.

Рекомендуется разделить и разрабатывать параллельно две части приложения: часть, состоящая только из динамических ресурсов, и часть, состоящая только из статических ресурсов.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

— *разделение динамического и статического содержания.* Возможность разделить логику приложения и дизайн веб-страницы снижает сложность разработки веб-приложений и упрощает их поддержку;

— *независимость от платформы.* Технологии Java не зависят от платформы, следовательно, JSP могут выполняться практически на любом веб-сервере. Разрабатывать JSP можно на любой платформе;
— *многократное использование компонентов.* Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание веб-приложений;
— *теги.* Спецификация JSP содержит библиотеку стандартных тегов JSTL, позволяет разработчику создавать собственные теги, кроме того, теги обеспечивают возможность использования JavaBean и обращение к классам бизнес-логики.

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.

В спецификации JSP 1.2 были объявлены только пять основных тегов:

<%@ *директива* **%>** — используется для установки параметров серверной страницы JSP;

<%! *объявление* **%>** — (*нежелателен в современном программировании*) содержит поля и методы, которые вызываются в expression-блоке и становятся полями и методами генерируемого сервлета. Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптлетах и выражениях;

<% *скриптлет* **%>** — (*нежелателен*) 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* **%>** — (*нежелателен*) содержит 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.

Ниже приведен простой пример «вживления» java-кода JSP-страницу.

 # 1 # jsp-страница с нежелательными тегами # old_style.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html><head><title>JSP-страница</title></head><body>
<%!       private int count = 0;
          String version = new String("1.2");
```

```
          private String getName() {return "Old Style";} %>
<% 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); %>
</body></html>
```

Такая страница трудна для понимания, несмотря на элементарность действий. В конце прошлого тысячелетия существовала техника создания страницы на основе одного скриплета, в который вставлялся большой фрагмент кода, что в корне противоречило концепции JSP. В современных приложениях скриплеты, выражения и объявления не применяются вовсе из-за нарушения «теговой» структуры страницы.

Жизненный цикл

Процессы, выполняемые с файлом JSP при *первом* вызове:

— браузер делает запрос к странице JSP;
— JSP-engine анализирует содержание файла JSP и создает сервлет с кодом, основанным на исходном тексте файла JSP, при этом engine транслирует статическое содержимое в методы вывода и помещает его в метод **_jspService()**. Полученный сервлет будет ответственен за генерацию статических элементов, определенных во время разработки. Динамические элементы транслируются в java-код;
— код сервлета компилируется в файл ***.class**. и загружается в контейнер. В итоге сервлет на основе JSP установлен и готов к работе;
— выполняется метод **init()** сервлета;
— вызывается метод **_jspService()**, и сервлет логически исполняется, формируя экземпляр **response**;
— комбинация статического HTML и графики вместе с результатами исполнения динамических элементов, определенных в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа **HttpServletResponse**.

Следующие обращения к файлу JSP просто вызовут метод **_jspService()** сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен или сервлет не будет выгружен из контейнера. Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности, в его метод **_jspService()**.

Если рассмотреть преобразование в сервлет простейшей страницы JSP, отправляющей в браузер приветствие:

 # 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:

Hello, Bender

а код сервлета на базе JSP будет выглядеть следующим образом:

// # 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);
    }
  }
}
```

Следует обратить внимание на объявление в методе **_jspService()** целого ряда локальных переменных. Ко всем этим переменным возможен доступ прямо из текста JSP по имени в выражениях Expression Language.

Неявные объекты в expression language

JSP expression language определяет свое множество неявных объектов, пересекающееся с множеством неявных объектов для JSP. Страница JSP с помощью скриптлетов всегда имеет доступ ко всем локальным переменным и параметрам метода **_jspService()** сервлета, создаваемым jsp-engine на основе этой же страницы. Наиболее очевидные:

— **pageContext** — определяет контекст JSP-страницы и предоставляет доступ к другим неявным объектам. Объект класса **javax.servlet.jsp.PageContext**. Область видимости в пределах страницы;
— **servletContext —** определяет контекст для сервлета JSP страницы и любого веб-компонента, содержащегося в этом приложении;
— **param** — содержит все параметры, переданные странице в виде параметров формы или параметров адресной строки;
— **paramValues** — содержит список всех значений параметров, переданных на страницу;
— **initParam** — содержит конфигурационные параметры страницы, заданные в файле **web.xml**;
— **cookie** — содержит список переменных, переданных с файлом cookie;
— **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 запроса;

— **response** — представляет собой 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;
— **config** — содержит параметры конфигурации сервлета и является экземпляром класса **javax.servlet.ServletConfig**. 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;
— **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;
— **page** — явная ссылка **this** для текущего экземпляра данной страницы является объектом **java.lang.Object**. Область видимости в пределах страницы;
— **exception** — представляет исключение одного из подклассов **java.lang.Throwable**, которое передается странице, помеченной как error page. Область видимости в пределах страницы ошибок.

К неявным объектам возможно и обращение по имени, не допускающее никаких сокращений.

Существуют также объекты, которые обеспечивают доступ к различным областям видимости, а именно: **pageScope**, **requestScope**, **sessionScope**, **applicationScope**.

 # 4 # неявные объекты в EL # el_instance.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>EL for pageContext</title></head>
<body>
Путь к контексту : ${ pageContext.request.contextPath } <br/>
Имя хоста : ${ header["host"] }<br/>
Тип и кодировка контента : ${pageContext.response.contentType}<br/>
Кодировка ответа : ${pageContext.response.characterEncoding}<br/>
ID сессии : ${pageContext.request.session.id}<br/>
Время создания сессии в мсек : ${pageContext.request.session.creationTime}<br/>
Время последнего доступа к сессии : ${pageContext.request.session.lastAccessedTime}<br/>
Имя сервлета : ${pageContext.servletConfig.servletName}
</body></html>
```

Рис. 16.1. *Обращение к неявным объектам*

Стандартные элементы action

Большинство стандартных тегов, содержащих символ %, в современном программировании не применяются. Используются action-теги версии JSP 2.0. Они позволяют создавать правильные JSP — документы. В тоже время декларации, выражения и скриптлеты остаются под негласным запретом из-за явного внедрения java-кода в JSP.

Action-теги:

— **<jsp:include>** — позволяет включать статические и динамические ресурсы в контекст генерируемой страницы при запросе вида

```
<jsp:include page="относительный URL" flush="true"/>
```

— Включаемая страница не может объявлять собственные заголовки, определяющие тип или кодировку;
— **<jsp:declaration>** — (*нежелателен*) объявление, аналогичен тегу **<%! %>**;
— **<jsp:scriptlet>** — (*нежелателен*) скриптлет, аналогичен тегу **<% %>**;
— **<jsp:expression>** — (*нежелателен*) выражение, аналогичен тегу **<%= %>**;
— **<jsp:text>** — вывод текста;
— **<jsp:useBean>** — объявление экземпляра компонента Java Bean или класса, обладающего **public**-конструктором по умолчанию. Если экземпляр с указанным идентификатором не существует, то он будет создан в области видимости **scope** со значением **page** (страница), **request** (запрос), **session** (сессия) или **application** (приложение). Объявляется, как правило, с атрибутами **id** (имя объекта), **class** (полное имя класса), **type** (по умолчанию **class**).

```
<jsp:useBean id="ob" scope="request" class="by.bsu.test.Message" />
```

что идентично java-коду:

```
by.bsu.test.Message ob = new by.bsu.test.Message();
```

Создан объект **ob** класса **Message**, и в дальнейшем через этот объект можно вызывать доступные методы класса. По умолчанию область видимости устанавливается в значение **page**. Специфика компонентов JavaBean в том, что если компонент имеет поле **text**, экземпляр компонента имеет параметр **text**, а метод, устанавливающий значение, должен называться **setText(String txt)**, возвращающий значение — **getText()**.

// # 5 # простой Bean-class # Message.java

```
package by.bsu.test;
public class Message {
          private Integer id;
          private String text = "";
          public Message() {
          }
          public Message(Integer id, String text) {
                    this.id = id;
                    this.text = text;
          }
          public void setText(String text) {
                    this.text = text;
          }
          public String getText() {
                    return text;
          }
          public Integer getId() {
                    return id;
          }
          public void setId(Integer id) {
                    this.id = id;
          }
          @Override
          public String toString() {
                    return "Message [id=" + id + ", text=" + text + "]";
          }
}
```

В теге **useBean** можно создавать только объекты классов, у которых определен конструктор по умолчанию;

— **<jsp:setProperty>** — позволяет устанавливать значения полей указанного в атрибуте **text** 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="text" value="hello" />
```

что идентично java-коду:

```
ob.setText("hello");
```

или с автоматическим преобразованием строки в число

```
<jsp:setProperty name="ob" property="id" value="717" />
```

идентично:

```
ob.setId(717);
```

что возможно только для интегральных и логических (boolean) типов;

— **<jsp:getProperty>** — извлекает значения поля указанного объекта, преобразует его в строку и отправляет в неявный объект **out**:

```
<jsp:getProperty name="ob" property="text" />
```

— **<jsp:forward>** — позволяет передать запрос другой странице или сервлету:

```
<jsp:forward page="относительный URL"/>
```

— **<jsp:plugin>** — замещается тегом **<OBJECT>** или **<EMBED>** в зависимости от типа браузера, в котором будет выполняться подключаемый апплет или Java Bean;

— **<jsp:params>** — группирует параметры внутри тега **jsp:plugin**;

— **<jsp:param>** — добавляет параметры в объект запроса, например, в элементах **forward**, **include**, **plugin**;

— **<jsp:fallback>** — указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься. Используется внутри элемента **plugin**.

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 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 и библиотек тегов.

JSP-документ

Современные тенденции оформления клиентской части стремятся к XML-формату, например, в виде **xhtml**. Такой формат менее подвержен грамматическим ошибкам разработчика, так как осуществляется проверка на well-formed.

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 **page** для обычной JSP:

```
<%@ page contentType="text/html"%>
```

для JSP-документа:

```
<jsp:directive.page contentType="text/html" />
```

Если к директиве **page** добавить атрибут **session="false"**, то атрибуты сессии будут недоступны.

Директива **include** для обычной JSP:

```
<%@ include file="header.jspf"%>
```

для JSP-документа:

```
<jsp:include file="header.jspf" />
```

Директива **taglib** для обычной JSP:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

для JSP-документа:

```
<jsp:root xmlns:c="http://java.sun.com/jsp/jstl/core"/>
```

Выше были представлены директива и action-тег **include**. При включении статических ресурсов в страницу оба механизма работают идентично. При включении динамических ресурсов (другой JSP или JSP-фрагмента) директива **include** сразу конвертирует их непосредственно в создаваемый сервлет и включает в процесс исполнения, таким образом, исходная и включаемая страница могут пользоваться одним и тем же пространством имен. Применение action-тега **include** формирует вид запрашиваемой страницы уже после выполнения сгенерированного сервлета и статически включает в его объект **response**.

Таким образом, для включения динамического содержимого в JSP-страницу каждый раз, когда та получает запрос, используется директива **include**. В этом случае включаемые сегменты имеют доступ к объектам **page**, **request**, **session** и **application** исходной страницы и ко всем атрибутам, которые имеют эти

объекты. Если использовать action-тег **jsp:include**, то изменения включаемого сегмента отразятся только после изменения исходной страницы (контейнер JSP перекомпилирует исходную страницу).

Для иллюстрации отличий директивы и action-тега **include** следует объявить некоторую jsp с обращением к переменной с именем **calendar**, которая на данной странице не объявлялась, а именно:

 # 6 # страница для вставки в другую страницу # time.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
<head><title>Fragment</title></head>
<body>
          <hr/>
             Время в миллисекундах: ${calendar.timeInMillis}
          <hr/>
</body>
</html>
```

Теперь в **index.jsp** будут использованы и директива, и action-тег.

 # 7 # страница, в которую вставляется контент другой страницы # index.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
<head><title>Index</title></head>
<body>
<jsp:useBean id="calendar" scope="page" class="java.util.GregorianCalendar"/>
          Directive
          <%@ include file="time.jsp"%>
<br/>
          Action-tag
          <jsp:include page="time.jsp"></jsp:include>
</body>
</html>
```

В результате имеется: если страница вставлена с помощью директивы **include**, то ее объявления получают доступ к переменным в области видимости основной страницы.

Другой способ импорта статического содержимого представляет тег **<c:import>** из библиотеки Java Standard Tag Library (JSTL).

Expression Language

В JSP API введено понятие Expression Language (EL). EL используется для упрощения доступа к данным (атрибутам, параметрам и т. п.), хранящимся в различных областях видимости: **page**, **request**, **session, application** и вычисления простых выражений.

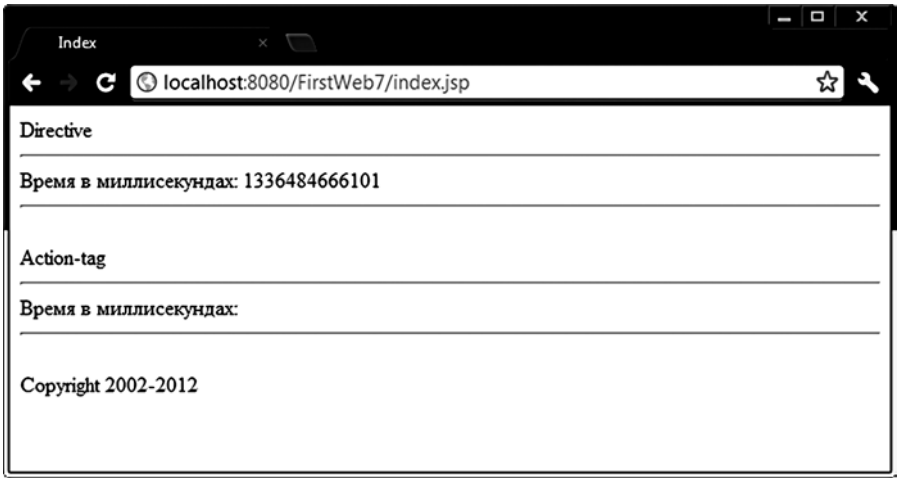

Рис. 16.2. *Отличия директивы и action-тега include*

EL вызывается при помощи конструкции **${***имя_атрибута***}**.

Начиная с версии спецификации JSP 2.0, EL является частью JSP и поддерживается без всяких сторонних библиотек.

EL-идентификатор ссылается на переменную, возвращаемую вызовом вида **pageContext.findAttribute(***имя_атрибута***)**. В общем случае переменная может быть сохранена в любой области видимости: **page** (**PageContext**), **request** (**HttpServletRequest**), **session** (**HttpSession**), **application** (**ServletContext**). Перечислены в порядке возрастания длительности существования. Если переменная не найдена, возвращается **null**. Значение **null** в поток вывода не передается. Также возможен доступ к параметрам запроса через предопределенные объекты **param** и **paramValues**, а также к заголовкам запроса через **requestHeaders** и некоторым другим, как уже было сказано.

Данные, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой коллекции — такие как, **List**, **Set**, **Map**, массивы и др. EL предоставляет доступ к этим объектам при помощи операторов **«.»** и **«[]»**. Способ применения этих операторов зависит от типа объекта. К элементу массива или списка доступ производится обычной индексацией с оператором **«[]»**. К элементам типа **Map**: либо указанием ключа после оператора «точка», либо обращением к ключу также по имени, но в операторе **«[]»**.

Обращение к экземпляру класса производится с неявным вызовом метода **toString()** в виде:

```
${ ob }
```

Обращение к значению поля экземпляра класса производится с неявным вызовом метода **get***Имя***()** в виде:

```
${ ob.id }
```

то есть вызван будет метод **getId()**.

 # 8 # вызов метода через bean-экземпляр # el_object.jsp

```
<%@ page contentType="text/html; charset=UTF-8"/>
<html>
<head><title>EL for object</title></head>
<body>
          <jsp:useBean id="ob" scope="page" class="by.bsu.test.Message" />
          <jsp:setProperty name="ob" property="text" value="Bender" />
          ${ ob.text } in Rio
</body></html>
```

С помощью оператора «точка» можно вызывать другие методы класса, к которому принадлежит объект. В этом случае сигнатура метода должна быть представлена полностью без каких-либо сокращений.

Если JSP вызывается из сервлета, передача объекта в качестве значения атрибута запроса осуществляется в виде:

```
Message obj = new Message();
obj.setText("Bender");
request.setAttribute("ob", obj);
```

В результате запуска этой страницы в браузер будет выведено:

Bender in Rio

Создать объект-список и получить доступ к нему, его элементам и полям элементов возможно с помощью следующего кода:

 # 9 # доступ к элементам списка # el_list.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>List</title></head>
<body>
          Элемент списка: ${ lst[0] } <br/>
          Значение поля элемента списка: ${ lst[0].text } <br/>
          Весь список: ${ lst }
</body>
</html>
```

При вызове **el_list.jsp** из сервлета, что значительно более естественно, чем создание объекта в самой странице, передача объекта осуществляется с помощью экземпляра запроса в виде:

```
MessageList<Message> list = new MessageList<Message>();
request.setAttribute("lst", list);
```

где класс **MessageList** представляет собой реализацию списка в виде наследования или агрегирования (объявление в качестве поля класса):

// # 10 # список, наследующий коллекцию # MessageList.java

```
package by.bsu.test;
import java.util.ArrayList;
public class MessageList extends ArrayList<Message>{
          { // логический блок вставлен для упрощения создания коллекции
            // в реальном коде его быть не должно!
                    this.add(new Message(721, "Hello"));
                    this.add(new Message(315, "Bye"));
          }
          public MessageList() { }
          @Override
          public String toString() {
                    String str = "";
                    for(Message msg: this) {
                              str += msg;
                    }
                    return str;
          }
}
```

В браузер будет выведено:

Рис. 16.3. *Использование EL для доступа к элементам списка*

Создать объект-карту типа **Map** и получить доступ к нему, его элементам и полям элементов возможно с помощью следующего кода:

 # 11 # доступ к элементам карты отображения # el_map.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head>
<title>Map</title>
</head>
<body>
```

```
        Доступ по прямому значению ключа: ${ map.Blinov } <br/>
        Доступ по ключу: ${ map["Romanchik"] } <br/>
        Доступ к полю ключа: ${ map["Romanchik"].id } <br/>
        Вся карта: ${ map }
</body></html>
```

Если **el_map.jsp** вызывается из сервлета, добавление объекта к запросу производится в виде:

```
MessageMap<String, Message> messMap = new MessageMap<String, Message> ();
request.setAttribute("map", messMap);
```

где класс **MessageMap** представляет примитивнейшую реализацию карты отображения:

// # 12 # карта отображений, наследующая HashMap # MessageMap.java

```
package by.bsu.test;
import java.util.HashMap;
public class MessageMap extends HashMap<String, Message>{
        { // упрощение
                put("Blinov", new Message(1, "Work"));
                put("Romanchik", new Message(865, "Holiday"));
        }
}
```

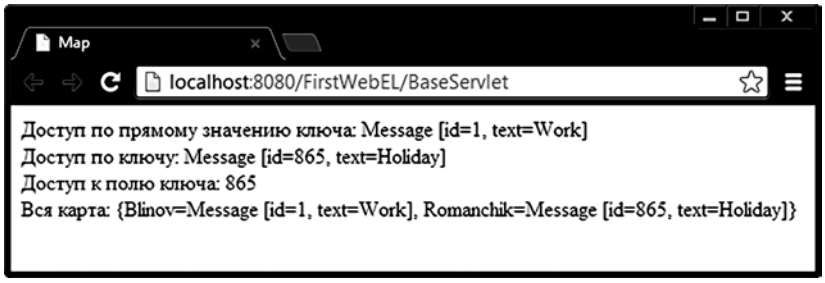

Рис. 16.4. *EL для доступа к значениям карты отображения*

Чтобы организовать обработку коллекции через цикл, а не посредством прямого индексирования, следует применить тег **<c:forEach>** из библиотеки JSTL, о нем будет рассказано в главе 17.

Параметры запроса, определенные на одной странице, доступны на другой странице при прямом переходе с первой страницы на вторую.

 # 13 # определение и передача параметров с запросом # el_param.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>Parameters</title></head>
<body>
```

```
<form action="el_param_browse.jsp">
          First Name: <input type="text" name="firstname"><br/>
          e-mail 1  : <input type="text" name="mail"><br/>
          e-mail 2  : <input type="text" name="mail"><br/>
          <input type="submit" name="submit" value="submit">
</form>
</body></html>
```

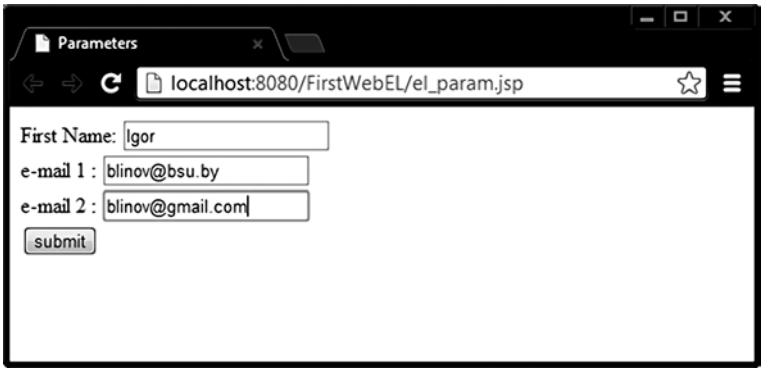

Рис. 16.5. *Определение и ввод параметров запроса*

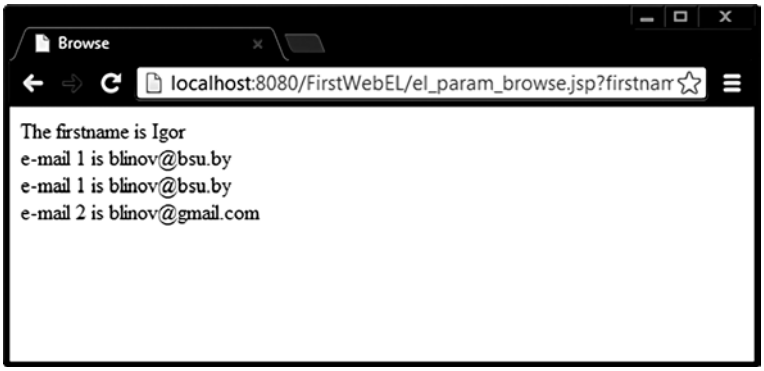

Рис. 16.6. *Извлечение и отображение параметров запроса*

EL с помощью неявных объектов **param** и **paramValues** получает доступ к параметрам, определенным на вызывающей странице:

 # 14 # доступ к параметрам запроса, поступившим из другой страницы
 # el_param_browse.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>EL for parameter</title></head>
<body>
```

```
        The firstname is ${ param.firstname }<br/>
        e-mail 1   is ${ param.mail }<br/>
        e-mail 1   is ${ paramValues.mail[0] }<br/>
        e-mail 2   is ${ paramValues.mail[1] }
</body></html>
```

Операторы в EL поддерживают наиболее часто используемые манипуляции данными.

Типы EL операторов

Стандартные операторы отношения:

== (или **eq**), **!=** (или **ne**), < (или **lt**), > (или **gt**), <= (или **le**), >= (или **ge**).
Арифметические операторы: +, -, *, / (или **div**), **%** (или **mod**).
Логические операторы: **&&** (или **and**), || (или **or**), **!** (или **not**).

Применение некоторых из операторов приведено в следующем коде, где для наглядности после знака равно приведен потенциальный результат, ожидаемый в окне браузера:

 # 15 # операции в EL # el_operation.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>EL operation</title></head>
<body>
        ${ 1 > (7/3) } = false
        ${ 7.0 >= 5 } = true
        ${ 100.0 == 100.000000000000001 } = true
        ${ 'Z' < 'a' } = true
        ${ 'dog' gt 'doc' } = true
        ${ 7.0E3 + 1.4 } = 7001.4
        ${ 17 mod 7 } = 3
</body></html>
```

Оператор **empty** используется для проверки значения переменной на **null**, или «пустое значение». Термин «пустое значение» зависит от типа проверяемого объекта. Это понятие включает нулевую длину для строки или нулевой размер для коллекции или массива.

 # 16 # проверка обьекта на null и значения его поля на пустое значение # el_empty.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>operator empty</title></head>
<body>
```

```
<jsp:useBean id="ob" scope="page" class="by.bsu.test.Message" />
        Message is not valid: ${not empty ob and empty ob.text }
</body>
</html>
```

В браузер будет выведено:

Message is not valid: true

Если в JSP отсутствует объявление или доступ к объекту **ob** типа **Message**, то при попытке обращения **${ob}** в результате ничего выведено не будет. Поток вывода игнорирует значения типа **null**. Более того, при обращении к полю т.е get-методу несуществующего объекта вида **${ob.text}** генерации исключения не произойдет. Если же объект существует в какой-либо области видимости, а обращение происходит к несуществующему полю, то будет сгенерировано исключение.

Обработка ошибок

При выполнении веб-приложений, как и любых других, могут возникать ошибки и исключительные ситуации. Возникает вопрос об их обработке. Если исключение в java-коде не обрабатывается, то оно попадает в контейнер сервлетов и получает код 500 с генерацией сообщения вида «500 Internal Server Error». Возникает также при вызове сервлетом метода **sendError(500)** на объекте **HttpServletResponse**. Такие действия производятся, как правило, для исключений времени выполнения. Исключения, генерируемые веб-приложением и не перехватываемые фильтром, сервлетом или JSP. При отсутствии запрашиваемой страницы генерируется ошибка с кодом 404. При запрете доступа — код 403. При отсутствии ответа сервера в течение определенного времени — код 504. При передаче слишком длинной строки запроса — код 414.

Для обработки исключений в зависимости от типа в приложении могут использоваться обычные JSP-страницы, специальные JSP для обработки ошибок или HTML-страницы. Для настройки соответствия ошибок и обработчиков используется элемент **error-page** файла **web.xml**.

Например, для обработки ошибки по коду:

```
<error-page>
        <error-code>404</error-code>
        <location>/errors/error404.jsp</location>
</error-page>
```

или для обработки страницей ошибок конкретного типа необработанного ранее исключения

```
<error-page>
        <exception-type>javax.el.PropertyNotFoundException</exception-type>
        <location>/errors/error_runtime.jsp</location>
</error-page>
```

В элементе **error-code** указывается код ошибки, в элементе **exception-type** — тип исключения.

Чтобы страница распознавалась приложением как «error page», следует установить значение атрибута **isErrorPage** в **true**:

```
<%@ page isErrorPage="true" %>
```

Ниже приведен пример с генерацией кода ошибки 500. При запуске страницы в коде соответствующего ей сервлета генерируется исключение **javax.el.PropertyNotFoundException**, и управление передается странице **error_runtime.jsp** в соответствии с тегом **<error-page>** в **web.xml**.

 # 17 # генерация ошибки # generation.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>Генерация исключения</title></head>
<body>
          <jsp:useBean id="ob" class="java.lang.String"/>
          ${ob.tostring}
</body></html>
```

Генерация исключения произойдет из-за обращения к несуществующему свойству экземпляра класса **String**.

Страница, вызываемая при ошибках, может иметь статический вид, но при необходимости сообщает о типе и месте возникшего исключения в понятной для клиента приложения форме. Информация о коде и типе ошибки, а также о месте ее возникновения, извлекается из неявного объекта **pageContext**, точнее, из его поля **errorData**, являющегося экземпляром класса **javax.servlet.jsp.ErrorData**.

 # 18 # страница ошибок # error_runtime.jsp

```
<%@ page isErrorPage="true" contentType="text/html; charset=UTF-8"
                                                    pageEncoding="UTF-8"%>
<html><title>Error Page</title>
          <body>
                    Request from ${pageContext.errorData.requestURI} is failed
                    <br/>
                    Servlet name: ${pageContext.errorData.servletName}
                    <br/>
                    Status code: ${pageContext.errorData.statusCode}
                    <br/>
                    Exception: ${pageContext.exception}
                    <br/>
                    Message from exception: ${pageContext.exception.message}
          </body>
</html>
```

В результате запуска страницы **index.jsp** в браузер будет выведена информация об ошибке в виде:

Request from /FirstWebEL/generation.jsp is failed
Servlet name: jsp
Status code: 500
Exception: javax.el.PropertyNotFoundException:
Property 'tostring' not found on type java.lang.String
Message from exception: Property 'tostring' not found on type java.lang.String

Для указания конкретной error page, обрабатывающей ошибки, которые возникают при выполнении только этой страницы, можно использовать директиву **page** с атрибутом **errorPage**:

<%@ page errorPage="*path***" %>**, где **path** — это путь к странице-обработчику ошибок из обычной исполняемой страницы, например:

```
<%@ page errorPage="/errors/error_runtime.jsp" %>
```

Взаимодействие JSP — сервлет — JSP

В большинстве приложений используются не сервлеты или JSP по отдельности, а их архитектурное взаимодействие. Страница JSP представляет вид для результатов выполнения запроса клиента, а сервлет отвечает за вызов классов бизнес-логики и передачу результатов выполнения бизнес-логики в соответствующую JSP и ее 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.

Применение шаблона на практике приводит к тому, что трехуровневая базовая модель распадается на многоуровневую. Число и назначение слоев может существенно отличаться в зависимости от разработанного архитектурного решения.

В предлагаемом ниже подходе добавляются: уровень служб, обрабатывающих объект запроса, уровень логики, имплементирующий конкретные бизнес-процессы приложения и уровень организации взаимодействия с базой данных.

Реализацию достаточно простой, но эффективной технологии построения распределенного приложения можно рассмотреть на примере решения задачи проверки логина

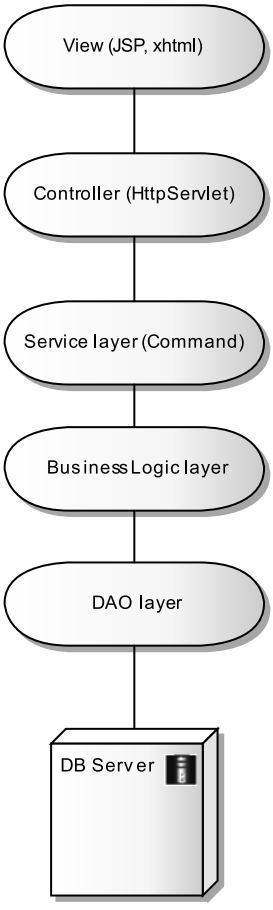

Рис. 16.7. *Вариант реализации шаблона MVC*

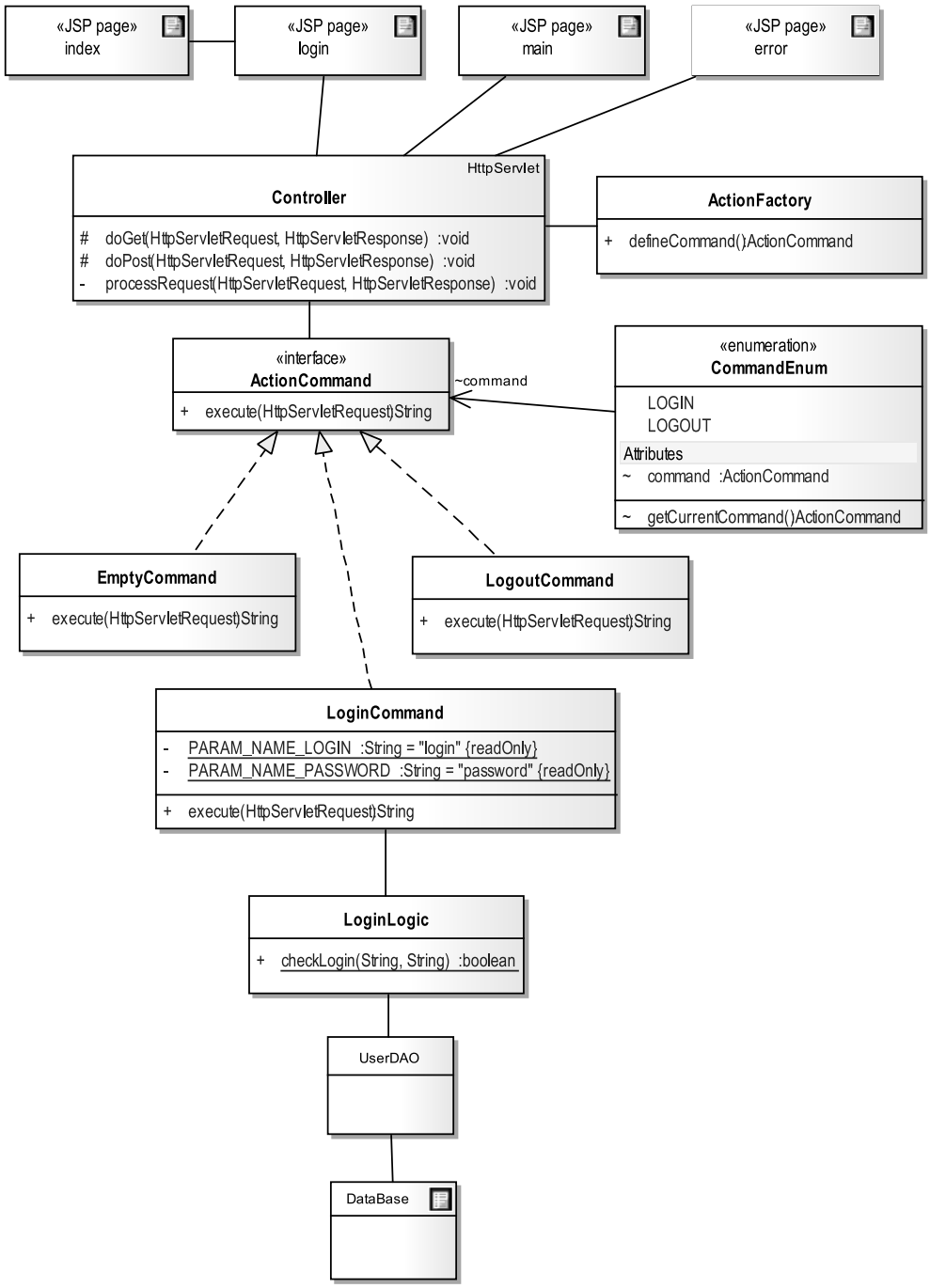

Рис. 16.8. *Диаграмма базового «движка» приложения*

и пароля пользователя с выводом приветствия в случае положительного результата. Схематично организацию данного приложения можно представить в виде рис. 16.8.

Вход в приложение осуществляется обычно через индексную страницу, в данном случае **index.jsp**.

 # 19 # переход на страницу login # index.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<html>
<head><title>Index</title></head>
<body>
          <jsp:forward page="/jsp/login.jsp"/>
</body></html>
```

«Индексная» страница является формальной и выполняет **forward** запроса на первую активную страницу приложения. Следующая страница **login.jsp** содержит форму для ввода логина и пароля для аутентификации в системе:

 # 20 # форма ввода информации и вызов контроллера # /jsp/login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>Login</title></head>
<body>
<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/>
          ${errorLoginPassMessage}
                    <br/>
          ${wrongAction}
                    <br/>
          ${nullPage}
                    <br/>
<input type="submit" value="Log in"/>
</form><hr/>
</body></html>
```

Страница содержит скрытое поле **hidden**, которое должно содержать указание на действие, связанное с функционалом этой страницы. В данном случае — запуск процесса аутентификации пользователя. В атрибут **errorLoginPassMessage** будет передаваться сообщение после неудачной попытки ввода логина-пароля. В атрибут **wrongAction** сообщение о несуществующей команде.

Демонстрационное приложение содержит страницу **main.jsp**, которая отображается пользователю в случае успешной аутентификации в приложении:

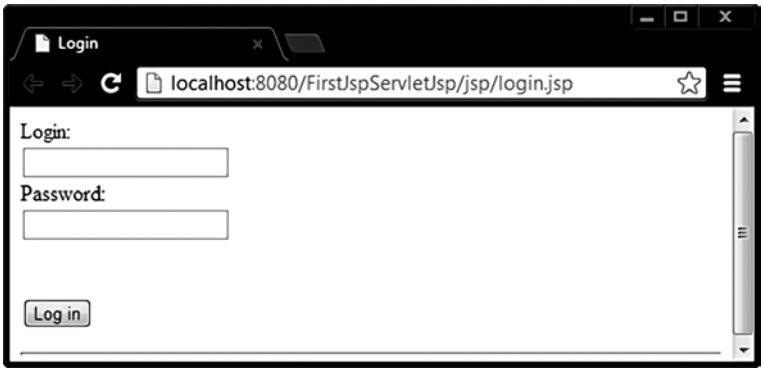

Рис. 16.9. *Страница аутентификации*

 # 21 # основная страница при успешном прохождении аутентификации # /jsp/main.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>Welcome</title></head>
<body>
          <h3>Welcome</h3>
          <hr/>
                    ${user}, hello!
          <hr/>
          <a href="controller?command=logout">Logout</a>
</body></html>
```

Страница **main.jsp** также объявляет команду, для передачи параметра с командой здесь использована строка с запросом типа **GET**, но это лишь демонстрация возможностей. В реальном примере следует передавать команду в скрытом поле и только по методу **POST**. Метод **doGet()** в сервлете оставить пустым.

Рис. 16.10. *Основная страница*

Если пароль введен неправильно, то атрибут **errorLoginPassMessage** получит соответствующее значение и будет выведено:

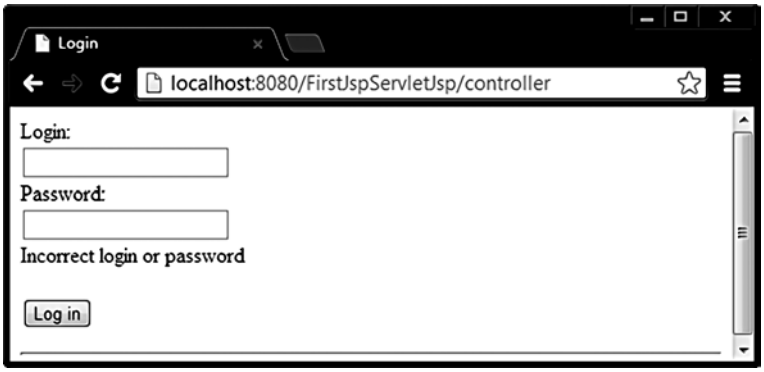

Рис. 16.11. *Вывод сообщения при ошибке аутентификации*

Страница **error.jsp**, последняя из созданных, загружается пользователю в случае возникновения ошибок:

 # 22 # страница вывода ошибок # /error/error.jsp

```
<%@ page isErrorPage="true" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>Error Page</title></head>
<body>
                    Request from ${pageContext.errorData.requestURI} is failed
                    <br/>
                    Servlet name or type: ${pageContext.errorData.servletName}
                    <br/>
                    Status code: ${pageContext.errorData.statusCode}
                    <br/>
                    Exception: ${pageContext.errorData.throwable}
</body></html>
```

В эту страницу следует добавить возможность перехода на какую-либо область приложения. Такие страницы должны быть в любом приложении, в данном случае она также может содержать команду, позволяющую пользователю продолжить взаимодействие с системой.

Код сервлета-контроллера **Controller**, который не будет изменяться при добавлении новых страниц.

/ # 23 # контроллер запросов # Controller.java */*

```
package by.bsu.example.servlet;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
```

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.example.command.ActionCommand;
import by.bsu.example.command.factory.ActionFactory;
import by.bsu.example.resource.ConfigurationManager;
import by.bsu.example.resource.MessageManager;
@WebServlet("/controller")
public class Controller extends HttpServlet {
        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;
                // определение команды, пришедшей из JSP
                ActionFactory client = new ActionFactory();
                ActionCommand command = client.defineCommand(request);
                /*
                 * вызов реализованного метода execute() и передача параметров
                 * классу-обработчику конкретной команды
                 */
                page = command.execute(request);
                // метод возвращает страницу ответа
                // page = null; // поэксперементировать!
                if (page != null) {
                        RequestDispatcher dispatcher =
                                getServletContext().getRequestDispatcher(page);
                        // вызов страницы ответа на запрос
                        dispatcher.forward(request, response);
                } else {
                        // установка страницы с сообщением об ошибке
                        page = ConfigurationManager.getProperty("path.page.index");
                        request.getSession().setAttribute("nullPage",
                        MessageManager.getProperty("message.nullpage"));
                        response.sendRedirect(request.getContextPath() + page);
                }
        }
}
```

Для определения типа и создания экземпляра-команды используется элемент реализации шаблона **Factory Method** класс **ActionFactory**, метод которого извлекает из запроса значение параметра **command** и на его основе извлекает

соответствующий запросу объект-команду. На самом деле экземпляр **request** не должен свободно передаваться в бизнес-логику приложения во избежание некорректной модификации, поэтому следует предварительно извлекать необходимую информацию из экземпляра-запроса и сохранять ее в объекте класса, специально для этого предназначенном, а именно:

/ # 24 # класс для хранения контента запроса # SessionRequestContent.java */*

```
public class SessionRequestContent {
          private HashMap<String, Object> requestAttributes;
          private HashMap<String, String[]> requestParameters;
          private HashMap<String, Object> sessionAttributes;
          // конструкторы
          // метод извлечения информации из запроса
          public void extractValues(HttpServletRequest request) {
                    // реализация
          }
          // метод добавления в запрос данных для передачи в jsp
          public void insertAttributes(HttpServletRequest request) {
                    // реализация
          }
          // some methods
}
```

По окончании выполнения команд бизнес-логики, полученные данные следует передать в **request** для передачи клиенту. В такой ситуации метод **defineCommand()** в качестве параметра должен будет объявлять класс **SessionRequestContent**, а не **HttpServletRequest**.

/ # 25 # класс получения обьекта-команды # ActionFactory.java */*

```
package by.bsu.example.command.factory;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.command.ActionCommand;
import by.bsu.example.command.EmptyCommand;
import by.bsu.example.resource.MessageManager;
public class ActionFactory {
          public ActionCommand defineCommand(HttpServletRequest request) {
                    ActionCommand current = new EmptyCommand();
                    // извлечение имени команды из запроса
                    String action = request.getParameter("command");
                    if (action == null || action.isEmpty()) {
                              // если команда не задана в текущем запросе
                              return current;
                    }
                    // получение объекта, соответствующего команде
                    try {
                              CommandEnum currentEnum =
                                        CommandEnum.valueOf(action.toUpperCase());
```

```
                    current = currentEnum.getCurrentCommand();
                } catch (IllegalArgumentException e) {
                    request.setAttribute("wrongAction", action
                            + MessageManager.getProperty("message.wrongaction"));
                }
                return current;
        }
}
```

При большом числе клиентов приложения решение определять все команды в перечислении не повлечет за собой создание большого количества объектов-команд, так как все клиенты будут пользоваться общими экземплярами команд.

/ # 26 # «хранилище» команд # CommandEnum.java */*

```
package by.bsu.example.command.client;
import by.bsu.example.command.ActionCommand;
import by.bsu.example.command.LoginCommand;
import by.bsu.example.command.LogoutCommand;
public enum CommandEnum {
        LOGIN {
                {
                    this.command = new LoginCommand();
                }
        },
        LOGOUT {
                {
                    this.command = new LogoutCommand();
                }
        };
        ActionCommand command;
        public ActionCommand getCurrentCommand() {
                return command;
        }
}
```

Выполнение всех команд будет осуществляться при помощи простой реализации шаблона **Command**. Интерфейс **ActionCommand** определяет одно действие **execute()**, реализации интерфейса определяют в имплементированном методе бизнес-логику выполнения соответствующей команды.

/ # 27 # интерфейс, определяющий контракт и его реализации # ActionCommand.java */*

```
package by.bsu.example.command;
import javax.servlet.http.HttpServletRequest;
public interface ActionCommand {
        String execute(HttpServletRequest request);
}
```

/ # 28 # бизнес-логика команды на аутентификацию # LoginCommand.java */*

```
package by.bsu.example.command;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.logic.LoginLogic;
import by.bsu.example.resource.ConfigurationManager;
import by.bsu.example.resource.MessageManager;
public class LoginCommand implements ActionCommand {
          private static final String PARAM_NAME_LOGIN = "login";
          private static final String PARAM_NAME_PASSWORD = "password";
          @Override
          public String execute(HttpServletRequest request) {
                    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.getProperty("path.page.main");
                    } else {
                              request.setAttribute("errorLoginPassMessage",
                                        MessageManager.getProperty("message.loginerror"));
                              page = ConfigurationManager.getProperty("path.page.login");
                    }
                    return page;
          }
}
```

Ниже приведен код класса бизнес-логики **LoginLogic**, который должен выполнять проверку правильности введенных логина и пароля с помощью запроса в БД, но для демонстрации простой логики работы приложения используется «заглушка» — логин и пароль хранятся в константах класса, чего никогда не следует делать в реальном проекте:

/ # 29 # псевдоаутентификация # LoginLogic.java */*

```
package by.bsu.example.logic;
public class LoginLogic {
          private final static String ADMIN_LOGIN = "admin";
          private final static String ADMIN_PASS = "Qwe123";
          public static boolean checkLogin(String enterLogin, String enterPass) {
                    return ADMIN_LOGIN.equals(enterLogin) &&
                                                  ADMIN_PASS.equals(enterPass);
          }
}
```

Команда **EmptyCommand** будет выполнена, если к сервлету произошло обращение без команды. В дальнейшем такое действие будет лишним, так как

будут показаны возможности блокировки запроса, не соответствующего требованиям логики приложения.

/ # 30 # пустая команда # EmptyCommand.java */*

```
package by.bsu.example.command;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.resource.ConfigurationManager;
public class EmptyCommand implements ActionCommand {
          @Override
          public String execute(HttpServletRequest request) {
                    /* в случае ошибки или прямого обращения к контроллеру
                     * переадресация на страницу ввода логина */
                    String page = ConfigurationManager.getProperty("path.page.login");
                    return page;
          }
}
```

/ # 31 # команда выхода из системы и уничтожения сессии # LogoutCommand.java */*

```
package by.bsu.example.commands;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.resource.ConfigurationManager;
public class LogoutCommand implements ActionCommand {
          @Override
          public String execute(HttpServletRequest request) {
                    String page = ConfigurationManager.getProperty("path.page.index");
                    // уничтожение сессии
                    request.getSession().invalidate();
                    return page;
          }
}
```

Служебные классы, извлекающие из properties-файлов необходимую для функционирования приложения информацию:

/ # 32 # сообщения системы # MessageManager.java */*

```
package by.bsu.example.resource;
import java.util.ResourceBundle;
public class MessageManager {
          private final static ResourceBundle resourceBundle =
                              ResourceBundle.getBundle("resources.messages");
          // класс извлекает информацию из файла messages.properties
          private MessageManager() { }
          public static String getProperty(String key) {
                    return resourceBundle.getString(key);
          }
}
```

/ # 33 # конфигурация системы # ConfigurationManager.java */*

```
package by.bsu.example.resource;
import java.util.ResourceBundle;
public class ConfigurationManager {
          private final static ResourceBundle resourceBundle =
                                   ResourceBundle.getBundle("resources.config");
          // класс извлекает информацию из файла config.properties
          private ConfigurationManager() { }
          public static String getProperty(String key) {
                    return resourceBundle.getString(key);
          }
}
```

Далее приведено содержимое файла *config.properties:*

```
##############################
## Application configuration ##
##############################
path.page.index=/index.jsp
path.page.login=/jsp/login.jsp
path.page.main=/jsp/main.jsp
path.page.error=/jsp/error/error.jsp
##############################
```

Далее приведено содержимое файла *messages.properties:*

```
##############################
##         Messages         ##
##############################
message.loginerror = Incorrect login or password.
message.nullpage = Page not found. Business logic error.
message.wrongaction = : command not found or wrong!
```

И последнее, что надо сделать в приложении, — настроить дескриптор приложения в файле **web.xml**:

 # 33 # дескриптор приложения # web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">
          <display-name> FirstJspServletJsp </display-name>
          <welcome-file-list>
                    <welcome-file>index.jsp</welcome-file>
          </welcome-file-list>
          <error-page>
                    <error-code>404</error-code>
```

```
            <location>/jsp/error/error.jsp</location>
        </error-page>
        <error-page>
            <exception-type>java.lang.RuntimeException</exception-type>
            <location>/jsp/error/error.jsp</location>
        </error-page>
</web-app>
```

В общем случае генерация исключения в приложении и обращение по несуществующему адресу относятся к различным типам, поэтому следует создавать отдельные error page. Данное приложение очень простое, поэтому вторая страница ошибок явно лишняя.

Запуск приложения может производиться из командной строки браузера вида:

```
http://localhost:8080/FirstJspServletJsp/index.jsp
```

или просто

```
http://localhost:8080/FirstJspServletJsp/
```

Во втором случае также будет произведено обращение к **index.jsp**, так как этот адрес указан в элементе **<welcome-file-list>** дескриптора приложения.

Структура и имена папок в веб-приложении должны иметь общий вид, как на рис. 16.12.

Рис. 16.12. *Стандартная структура каталогов веб-приложения*

Пул соединений

При большом количестве клиентов, работающих с приложением, к его базе данных выполняется большое количество запросов. Установление соединения с базой данных является дорогостоящей по требуемым ресурсам операцией. Эффективный способ решения данной проблемы — 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 десятков).

Ниже представлена существующая на настоящий момент стратегия конфигурации пула средствами Tomcat. В **web.xml** добавляется тег описания ресурса **<resource-ref>** вида

```
<description>MySQL</description>
<resource-ref>
        <description>MySQL DB Connection Pool</description>
        <res-ref-name>jdbc/testphones</res-ref-name>
```

```
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
</resource-ref>
```

где указано имя ресурса **jdbc/testphones**, запрашиваемого через JNDI; тип объекта ресурса **javax.sql**.**DataSource**; определение контейнеру сервлета ответственности за авторизацию доступа к базе данных. Контейнер своими средствами будет поддерживать пул соединений с БД.

Далее необходимо сконфигурировать параметры пула соединений для Apache Tomcat в файле **context.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/FirstJspServletJsp">
   <Resource
          name="jdbc/testphones"
          author="Container"
          type="javax.sql.DataSource"
          maxActive="32"
          maxIdle="8"
          maxWait="10000"
          username="root"
          password="pass"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://localhost:3306/testphones"
   />
</Context>
```

где:

name — имя ресурса, совпадающее по значению с именем, объявленным в **<res-ref-name>** дескриптора **web.xml**;

указание на тип интерфейса **type** также должно совпадать со значением в **<res-type>**;

username — имя владельца базы данных;

password — пароль владельца;

driverClassName — класс драйвера JDBC;

url — адрес физического расположения БД;

maxActive — размер пула, то есть максимальное число активных соединений, при превышении — ожидание освободившегося;

maxIdle — максимальное число свободных объектов-соединений;

maxWait — время (миллисекунды) ожидания предоставления объекта-соединения из пула соединений при невозможности немедленной выдачи соединения, например, если весь пул выбран. По истечении этого времени при непредоставлении соединения генерируется исключение.

Непосредственно в коде приложения найти и запустить данный пул соединений можно с помощью JNDI (Java Naming Directory Interface).

Класс **javax.naming.InitialContext** как часть JNDI API обеспечивает взаимодействие с каталогом именованных объектов, а именно: хранение в некоторой

области необходимых классов объектов и предоставление к ним именованного доступа по запросу приложения (не только к БД, но и вообще к любому доступному через JNDI). В каталоге ресурсов можно сопоставить объект источника данных **javax.sql.DataSource** с некоторым именем, предварительно создав объект **DataSource**.

Разделяемый доступ к источнику данных можно организовать, например, путем объявления статической переменной типа **DataSource** из пакета **javax.sql**, однако в JEE принято использовать для этих целей каталог или имя контекста приложения. Источник данных типа **DataSource** — это компонент, предоставляющий соединение с приложением СУБД.

Доступ к внешнему ресурсу предоставляется с помощью метода **lookup()** по его, ресурса, имени. Методу **lookup()** передается имя, всегда начинающееся с имени корневого контекста.

```
Context envCtx = (Context) (new InitialContext().lookup("java:comp/env"));
DataSource ds = (DataSource) envCtx.lookup("jdbc/testphones");
```

Соединение извлекается из пула очень просто:

```
Connection cn = ds.getConnection();
```

После выполнения запроса соединение завершается, и его объект должен быть возвращен обратно в пул вызовом:

```
cn.close();
```

Производители СУБД для облегчения создания пула соединений определяют собственный класс на основе интерфейса **DataSource**. Подобные классы есть и у MySQL и у Oracle. В этом случае пул соединений может быть создан с помощью класса **oracle.jdbc.pool.OracleDataSource**:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@server:1521/testphones ");
ods.setUser("root");
ods.setPassword("pass");
ods.setConnectionCacheName(CACHE_NAME);
Properties cacheProps = new Properties();
cacheProps.setProperty("MinLimit", "0");
cacheProps.setProperty("MaxLimit", "4");
cacheProps.setProperty("InitialLimit", "1");
cacheProps.setProperty("ConnectionWaitTimeout", "5");
cacheProps.setProperty("ValidateConnection", "true");
ods.setConnectionProperties(cacheProps);
Connection cn = ods.getConnection();
```

Драйвер загружается непосредственно при создании экземпляра **OracleDataSource**.

Задания к главе 16

Вариант А

Построить веб-систему, поддерживающую заданную функциональность:

— на основе сущностей предметной области создать классы, их описывающие;
— классы и методы должны иметь названия отражающие их функциональность, и быть грамотно структурированы по пакетам;
— оформление кода должно соответствовать Java Code Convention;
— информацию о предметной области хранить в БД, для доступа использовать API JDBC с использованием пула соединений, стандартного или разработанного самостоятельно. В качестве СУБД рекомендуется MySQL или Derby;
— приложение должно поддерживать работу с кириллицей (быть многоязычной), в том числе и при хранении информации в БД;
— архитектура приложения должна соответствовать шаблону Model-View-Controller;
— при реализации алгоритмов бизнес-логики использовать шаблоны GoF: Factory Method, Command, Builder, Strategy, State, Observer etc.
— используя сервлеты и JSP, реализовать функции, предложенные в постановке конкретной задачи;
— в страницах JSP применять библиотеку JSTL;
— при разработке бизнес-логики использовать сессии и фильтры;
— использовать Log4j;
— код должен содержать комментарии.

1. Система **Факультатив**. Существует перечень **Курсов,** за каждым из которых закреплен один **Преподаватель**. **Студент** записывается на один или несколько **Курсов**. По окончании обучения **Преподаватель** выставляет **Студенту** и добавляет отзыв.
2. Система **Платежи**. **Клиент** имеет одну или несколько **Кредитных Карт,** каждая из которых соответствует некоторому **Счету** в системе платежей. **Клиент** может при помощи **Счета** сделать **Платеж**, заблокировать **Счет** и пополнить **Счет**. **Администратор** снимает блокировку.
3. Система **Приемная комиссия**. **Абитуриент** регистрируется на один из **Факультетов** с фиксированным планом набора, вводит баллы по соответствующим **Предметам и аттестату**. Результаты **Администратором** регистрируются в **Ведомости**. Система подсчитывает сумму баллов и определяет **Абитуриентов**, зачисленных в учебное заведение.
4. Система **Библиотека**. **Читатель** имеет возможность осуществлять поиск и заказ **Книг** в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. **Книга** может присутствовать в **Библиотеке** в одном или нескольких экземплярах.

5. Система **Больница**. **Врач** определяет диагноз, делает назначение **Пациенту** (процедуры, лекарства, операции). Назначение может выполнить**ь Медсестра (**процедуры, лекарств**а)** или **Врач** (любое назначение). **Пациент** может быть выписан из **Больницы**, при этом фиксируется окончательный диагноз.
6. Система **Турагентство**. **Заказчик** выбирает и оплачивает **Тур (отдых, экскурсия, шоппинг)**. **Турагент** определяет тур как «горящий», размеры скидок постоянным клиентам.
7. Система **Телефонная станция**. **Администратор** осуществляет подключение **Абонентов**. **Абонент** может выбрать одну или несколько из предоставляемых **Услуг**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**. **Администратор** может просмотреть список неоплаченных **Счетов** и заблокировать **Абонента**.
8. Система **Автобаза**. **Диспетчер** распределяет **Заявки** на **Рейсы** между **Водителями,** за каждым из которых закреплен свой **Автомобиль**. На **Рейс** может быть назначен **Автомобиль**, находящийся в исправном состоянии и характеристики которого соответствуют **Заявке. Водитель** делает отметку о выполнении **Рейса** и состоянии **Автомобиля**.
9. Система **Интернет-магазин**. **Администратор** осуществляет ведение каталога **Товаров**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** может занести неплательщиков в «черный список».
10. Система **Авиакомпания**. **Авиакомпания** имеет список рейсов. **Диспетчер** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. **Администратор** управляет списком рейсов.
11. Система **LowCost-Авиакомпания**. **Клиент** заказывает и оплачивает **Билет** на **Рейс** с учетом наличия/отсутствия багажа и права первоочередной регистрации и посадки (Цена **Билета** может быть ниже стоимости провоза багажа). С приближением даты **Рейса** или наполнением самолета цена на **Билет** может повышаться.
12. Система **Периодические издания**. **Администратор** осуществляет ведение каталога периодических **Изданий**. **Читатель** может оформить **Подписку**, предварительно выбрав периодические **Издания** из списка. Система подсчитывает стоимость и регистрирует **Платеж**.
13. Система **Заказ гостиницы**. **Клиент** заполняет **Заявку**, указывая количество мест в номере, класс апартаментов и время пребывания. **Администратор** просматривает поступившую **Заявку**, выделяет наиболее подходящий из доступных **Номеров**, после чего система выставляет **Счет Клиенту**.
14. 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**.
15. Система **Прокат автомобилей**. **Клиент** выбирает **Автомобиль** из списка доступных. Заполняет форму **Заказа**, указывая паспортные данные, срок

аренды. **Клиент** оплачивает **Заказ**. **Администратор** регистрирует возврат автомобиля. В случае повреждения **Автомобиля Администратор** вносит информацию и выставляет счет за ремонт. **Администратор** может отклонить **Заявку**, указав причины отказа.

16. Система **Скачки. Клиент** делает **Ставки** разных видов на **Забеги**. **Букмекер** устанавливает уровень выигрыша. **Администратор** фиксирует результаты **Забегов**.
17. Система **Тестирование**. **Тьютор** создает **Тест** из нескольких **Вопросов** закрытого типа (выбор одного или более вариантов из N предложенных) по определенному **Предмету**. **Студент** просматривает список доступных **Тестов,** отвечает на **Вопросы**.
18. Система **Ресторан**. **Клиент** осуществляет заказ из **Меню**. **Администратор** подтверждает **Заказ** и отправляет его на кухню для исполнения. **Администратор** выставляет **Счет**. **Клиент** производит его оплату.
19. Система **Кофе-машина**. **Пользователь** обладает **Счетом**. **Кофе-машина** содержит набор **Напитков**, с заданным числом порций и дополнительных **Ингредиентов**. **Пользователь** может купить один или несколько **Напитков**. **Администратор Кофе-машины** осуществляет ее наполнение.
20. Система **Парк**. **Владелец** парка дает указания **Леснику** о высадке (лечении, художественной обработке, уничтожении) **Растений**. **Лесник** отчитывается о выполнении. **Владелец** просматривает результаты и подтверждает исполнение.
21. Система **Команда разработчиков**. **Заказчик** представляет **Техническое Задание (ТЗ)**, в котором перечислен перечень **Работ** с указанием квалификации и количества требуемых специалистов. **Менеджер** рассматривает **ТЗ** и оформляет **Проект**, назначая на него незанятых **Разработчиков** требуемой квалификации, после чего рассчитывается стоимость **Проекта** и **Заказчику** выставляется **Счет**. **Разработчик** имеет возможность отметить количество часов, затраченных на работу над проектом.
22. Система **Железнодорожная касса***. **Пассажир** делает **Заявку** на билет до необходимой ему станции назначения, время и дату поездки. Система осуществляет поиск подходящего **Поезда**. **Пассажир** делает выбор **Поезда** и получает **Счет** на оплату. **Администратор** управляет списком зарегистрированных пассажиров.
23. Система **Городской транспорт***. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом или расписанием.

Тестовые задания к главе 16

Вопрос 16.1.

Расставьте фазы жизненного цикла JSP в правильном порядке (1):

a) вызов метода jspInit; b) трансляция; c) вызов метода _jspService; d) компиляция; e) загрузка класса; f) вызов метод jspDestroy; g) создание экземпляра.

1) dbcgaef;
2) dbgeacf;
3) bdegacf;
4) bdcgeaf.

Вопрос 16.2.

Укажите, в чем состоит различие директивы include (<%@ include … %>) и action-тэг include (<jsp:include />) (1):

1) различий нет, это две формы записи, использующие JSP и XML синтаксис соответственно;
2) директива include позволяет вставлять текст или код в процессе трансляции страницы JSP в сервлет, однако, в отличие от действия jsp:include, рассматривает ресурс как статический объект;
3) директива include позволяет вставлять текст или код в процессе трансляции страницы JSP в сервлет, однако, в отличие от действия jsp:include, рассматривает ресурс как динамический объект.

Вопрос 16.3.

Укажите объекты, доступные в коде JSP без их специального объявления или импорта (5):

1) out
2) request
3) response
4) session
5) pageContext
6) cookies

Вопрос 16.4.

Выберите идентификаторы JSP-директив (3):

1) page
2) useBean
3) forward
4) include
5) taglib
6) param

Глава 17

СЕССИИ, СОБЫТИЯ И ФИЛЬТРЫ

Сеанс (сессия)

При посещении клиентом веб-ресурса и выполнении вариантов запросов контекстная информация о клиенте не хранится. В протоколе HTTP нет возможностей для сохранения и изменения информации о предыдущих действиях 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 есть сессия между клиентом и сервером, устанавливаемая на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и веб-сервером в момент получения первого запроса к веб-приложению. Каждый клиент устанавливает с сервером свой собственный сеанс, который сохраняется до окончания работы с приложением.

Сеанс используется для обеспечения хранения данных при последовательном выполнении нескольких запросов различных веб-страниц 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 явный доступ к экземпляру сеанса/сессии пользователя веб-приложения, используются методы **getSession(boolean create)** или **getSession()** интерфейса **HttpServletRequest**. Экземпляр запроса возвращает ссылку на объект сессии. Метод не создает сессию, а только дает доступ

с помощью запроса к экземпляру сессии **HttpSession**, соответствующему данному пользователю.

Если для метода **getSession(boolean create)** 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/создает новый сеанс:

```
HttpSession session = request.getSession(true);
```

после чего начинается сбор информации о клиенте.

Если метод **getSession(boolean param)** вызывать с параметром **false**, то ссылка на активный сеанс будет возвращена, если он существует, если же сеанс уничтожен или не был активирован, то метод возвратит **null**. Метод **getSession()** без параметров работает так же, как и **getSession(true)**.

Сессия содержит информацию о дате и времени создания последнего обращения к сессии, которая может быть извлечена с помощью методов **getCreationTime()** и **getLastAccessedTime()**.

Метод **String getId()** возвращает уникальный идентификатор, который получает каждый сеанс при создании. Метод **isNew()** возвращает **false** для уже существующего сеанса и **true** — для нового сеанса. Задать время (в секундах) бездействия сессии, по истечении которого экземпляр сессии будет уничтожен, можно методом **setMaxInactiveInterval(long sec)**.

Чтобы сохранить значения переменной в текущем сеансе, используется метод **setAttribute(String name, Object value)** класса **HttpSession**, прочесть — **getAttribute(String name)**, удалить — **removeAttribute(String name)**. Список имен всех переменных, сохраненных в текущем сеансе, можно получить, используя метод **Enumeration getAttributeNames()**, работающий так же, как и соответствующий метод интерфейса **HttpServletRequest**.

Принудительно завершить сеанс можно методом **invalidate()**. В результате для сеанса будут уничтожены все связи с используемыми объектами, и данные, сохраненные в старом сеансе, будут потеряны для клиента и всех приложений.

/ # 1 # добавление информации в сессию # SessionControlServlet.java */*

```
package by.bsu.control.listener;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
import javax.servlet.annotation.WebServlet;
@WebServlet("/sessionservlet")
public class SessionControlServlet extends HttpServlet {
          protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
```

```
                    HttpSession session = request.getSession(true);
                    if (session.getAttribute("role") == null) {
                              session.setAttribute("role", "moderator");
                    }
                    /* количество запросов, которые были сделаны
                    к данному сервлету текущим пользователем
                    в рамках текущей пользовательской сессии */
                    Integer counter = (Integer) session.getAttribute("counter");
                    if (counter == null) {
                              session.setAttribute("counter", 1);
                    } else {
          /* увеличивает счетчик обращений к текущему сервлету и кладет его в сессию */
                              counter++;
                              session.setAttribute("counter", counter);
                    }
          request.setAttribute("lifecycle", "CONTROL request LIFECYCLE");
          request.getRequestDispatcher("/jsp/sessionattr.jsp").forward(request, response);
          }
}
```

Задача страницы **sessionattr.jsp** — отразить информацию о роли пользователя и счетчике обращений.

 # 2 # информация о сессии # /jsp/sessionattr.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"   pageEncoding="UTF-8"%>
<html><head><title>Session Attribute</title></head>
<body>
          Role: ${role}
<br /><hr />
          Counter: ${counter}
<br /><hr />
          MaxInactiveInterval: ${pageContext.session.maxInactiveInterval}<br/>
          ID session: ${pageContext.session.id}<br/>
          Lifecycle: ${lifecycle}<br/>
<a href="index.jsp">Back to index.jsp</a>
</body></html>
```

В результате в браузер будет выведено:

В качестве данных сеанса выступают: счетчик обращений — объект типа **Integer** и роль пользователя. В ответ на пользовательский запрос сервлет **SessionServlet** возвращает страницу **sessionattr.jsp**, на которой отображаются все атрибуты сессии, а также идентификационный номер сессии и время инвалидации сессии. Время жизни сессии при отсутствии активности пользователя задано 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.

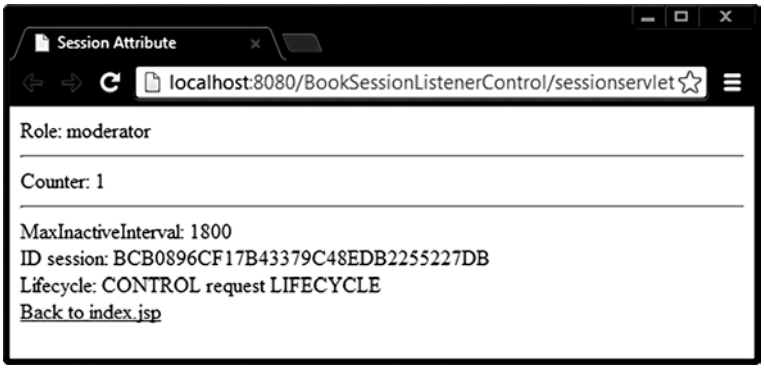

Рис. 17.1. *Информация о сессии*

 # 3 # стартовая страница # index.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>Index Page</title></head>
<body>
          Lifecycle: ${lifecycle}
<a href= "sessionservlet">Session Servlet</a>
</body></html>
```

В следующем примере произведено моделирование процесса ликвидации сессии при отсутствии активности за определенный промежуток времени. Реализация корректно работает только в случае, если приложением пользуется один клиент. Для нескольких клиентов алгоритм будем немного сложнее.

/ # 4 # инвалидация сессии по времени # TimeoutServlet.java */*

```
package by.bsu.timeoutsession;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
public class TimeoutServlet extends HttpServlet {
          protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                                  throws ServletException, IOException {
                    HttpSession session = null;
                    if (SessionLocator.flag) {
                              // "создание" сессии и установка времени инвалидации
                              session = req.getSession();
                              int timeLive = 10; // десять секунд!
                              session.setMaxInactiveInterval(timeLive);
                              SessionLocator.flag = false;
```

```
                    } else {
                    // если сессия не существует, то ссылка на нее не будет получена
                              session = req.getSession(false);
                              if (session == null) {
                                        SessionLocator.flag = true;
                              }
                    }
          req.setAttribute("messages", SessionLocator.addMessage(session));
          req.getRequestDispatcher("/jsp/time.jsp").forward(req, resp);
          }
}
```

/ # 5 # формирование сообщений для передачи в jsp # SessionLocator.java */*

```
package by.bsu.timeoutsession;
import java.util.ArrayList;
import java.util.Date;
import javax.servlet.http.HttpSession;
public class SessionLocator {
private final static String BR = "<br/><hr/>";
          public static boolean flag = true;
          public static ArrayList<String> addMessage(HttpSession session) {
                    ArrayList<String> messages = new ArrayList<String>();
                    if (session != null) { // если сессия существует
                              messages.add("Creation Time : "
                                                  + new Date(session.getCreationTime()) + BR);
                              messages.add("Session id : " + session.getId() + BR);
                              messages.add("Session alive!" + BR);
                    } else { // если сессии уже не существует
                              messages.add("Session disabled!" + BR);
                    }
                    return messages;
          }
}
```

 # 6 # информация о состоянии сессии # /jsp/time.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head><title>Session Control</title></head>
<body>
          ${messages}
<br/>
<a href ="timeoutservlet">Back to Servlet</a>
</body></html>
```

При первом запуске в браузер будет выведено (см. рис. 17.2).

Если повторить запрос к сервлету менее чем через 10 секунд, вывод будет идентично повторен. Если же запрос повторить более чем через десять секунд,

Рис. 17.2. *Жизненный цикл сессии. Сессия «alive»*

сессия будет автоматически уничтожена, и в браузер будет выведено следующее сообщение:

Рис. 17.3. *Жизненный цикл сессии. Сессия «not alive»*

Следующее обращение к сервлету приведет к созданию новой сессии.

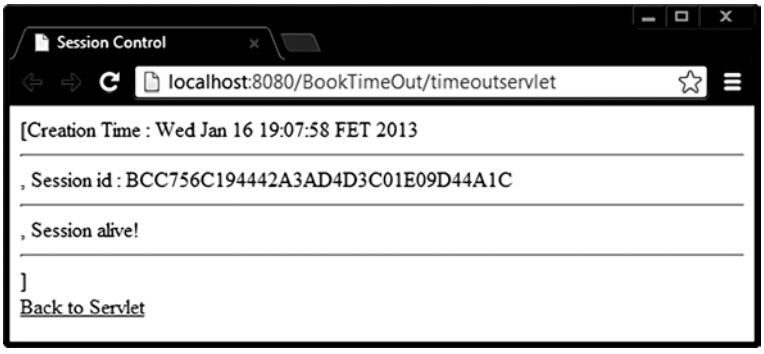

Рис. 17.4. *Жизненный цикл сессии. Сессия вновь «alive»*

Файлы Cookie

Для хранения информации на компьютере клиента используются возможности класса **javax.servlet.http.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 веб-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 веб-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("model", "Canon D7000");
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 «Canon D7000».

Экземпляр **Cookie** имеет целый ряд параметров: путь, домен, номер версии, время жизни, комментарий. Одним из ключевых является срок жизни в секундах от момента первой отправки клиенту, определить который следует методом **setMaxAge(long sec)**. Если параметр не указан, то cookie существует только до момента прерывания контакта клиента с приложением.

В предложенном примере cookie добавляются в сервлете, в сервлете же инициируется процесс их извлечения и передачи информации, содержащейся в файле страницы **maincookie.jsp**.

 # 7 # информация из cookie # /jsp/maincookie.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>from Cookie </title></head>
<body>
          ${messages}
<a href= "cookiecontroller">Messages from Cookie</a>
</body></html>
```

При старте приложения **${messages}** ничего не выведет, так как атрибут **messages** не существует. Сервлет **CookieController** добавляет cookie к объекту-ответу и пытается извлечь cookie из объекта-запроса. При первом запуске извлекать нечего, так как запрос пришел без cookie.

/ # 8 # создание и чтение cookie # CookieController.java */*

```
package by.bsu.servlet.cookie;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import by.bsu.servlet.action;
public class CookieController extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                          throws ServletException, IOException {
          CookieAction.setCookie(response); // добавление cookie
          // извлечение cookie и добавление информации к request
          request.setAttribute("messages", CookieAction.addToRequest(request));
          request.getRequestDispatcher("/jsp/maincookie.jsp").forward(request, response);
          }
}
```

/ # 9 # формирование и извлечение cookie # CookieAction.java */*

```
package by.bsu.servlet.action;
import java.util.ArrayList;
import java.util.Date;
import javax.servlet.http.HttpSession;
public class CookieAction {
private static int number = 1;
          public static void setCookie(HttpServletResponse resp) {
                    String name = "JamesBond";
                    String role = "00" + number++;
                    Cookie c = new Cookie(name, role);
                    c.setMaxAge(60 * 60); // время жизни файла cookie
                    resp.addCookie(c); // добавление cookie к объекту-ответу
                    value = resp.getLocale().toString();
                    Cookie loc = new Cookie("locale", value);
```

```
                resp.addCookie(loc);
        }
        public static ArrayList<String> addToRequest(HttpServletRequest request) {
                ArrayList<String> messages = new ArrayList<>();
                Cookie[ ] cookies = request.getCookies();
                if (cookies != null) {
                        messages.add("Number cookies : " + cookies.length);
                        for (int i = 0; i < cookies.length; i++) {
                                Cookie c = cookies[i];
                                messages.add(c.getName() + " = " + c.getValue());
                        } // end for
                } // end if
                return messages;
        }
}
```

При первом запуске приложения cookie, естественно, переданы не будут, так как они еще не созданы, и браузер еще не получал ни одного ответа.

При повторном обращении к сервлету cookie уже будет передан с запросом и в браузер будет выведено:

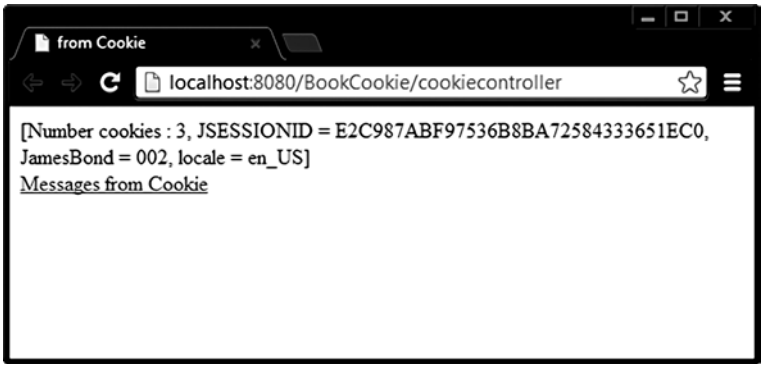

Рис. 17.5. *Информация из файла cookie*

Файл cookie, как это видно из указанного примера, может быть перезаписан при генерации каждого нового объекта-ответа.

Обработка событий

Изменение состояния некоторых объектов приложения представляют интерес и в чисто информационных целях, и в целях реализации реакции на происходящее. Например, можно отслеживать время входа и выхода из системы пользователей, действия, выполняемые пользователем. Осуществлять инициализацию ресурсов в момент запуска системы, создания пользовательской сессии, и просто служить источником дополнительной информации.

Интерфейсы и их методы
ServletContextListener **contextInitialized(ServletContextEvent e)** **contextDestroyed(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)**

Рис. 17.6. *Интерфейсы событий и сигнатуры их методов*

Существует несколько интерфейсов, которые позволяют следить за событиями, связанными с сеансом, контекстом и запросом сервлета, генерируемыми во время жизненного цикла веб-приложения:

— **javax.servlet.ServletContextListener** — обрабатывает события создания/удаления контекста сервлета;

— **javax.servlet.ServletContextAttributeListener** — обрабатывает события создания/удаления/модификации атрибутов контекста сервлета;

— **javax.servlet.http.HttpSessionListener** — обрабатывает события создания/удаления HTTP-сессии;

— **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/деактивацией HTTP-сессии;

— **javax.servlet.ServletRequestListener** — обрабатывает события создания/удаления запроса;
— **javax.servlet.ServletRequestAttributeListener** — обрабатывает события создания/удаления/модификации атрибутов запроса сервлета.

Регистрация блока прослушивания производится в дескрипторном файле приложения или аннотированием класса, реализующего интерфейс **Listener**.

Демонстрация обработки событий изменения состояния атрибутов сессии рассматривается на примере #1 данной главы. Обрабатываться будут события добавления атрибута **role**, а также добавления и изменения атрибута для счетчика обращений к сервлету **counter**. Для этого необходимо создать класс, реализующий интерфейс **HttpSessionAttributeListener** и реализовать необходимые для выполнения поставленной задачи методы.

/ # 10 # обработка событий добавления и изменения атрибута сессии # SessionListener.java */*

```
package by.bsu.control.listener;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
@WebListener
public class SessionListenerImpl implements HttpSessionAttributeListener {
    public void attributeRemoved(HttpSessionBindingEvent ev) {
    }
    public void attributeAdded(HttpSessionBindingEvent ev) {
          // запись в log-файл или иные действия
          System.out.println("add: " + ev.getClass().getSimpleName() + " : "+ ev.getName()
                                                                    + " : " + ev.getValue());
    }
    public void attributeReplaced(HttpSessionBindingEvent ev) {
          // запись в log-файл или иные действия
          System.out.println("replace: " + ev.getClass().getSimpleName() + " : " + ev.getName()
                                                                      + " : " + ev.getValue());
    }
}
```

Экземпляр класса события **HttpSessionBindingEvent** дает доступ к самой сессии методом **getSession()**.

Зарегистрировать обработку событий можно также в элементе **<listener>** дескрипторного файла **web.xml**:

```
<listener>
          <listener-class>by.bsu.control.listener.SessionListenerImpl</listener-class>
</listener>
```

В результате запуска сервлета **SessionControlServlet** и двух обращений к нему в консоль будет выведено:

add: HttpSessionBindingEvent : role : moderator
add: HttpSessionBindingEvent : counter : 1

replace: HttpSessionBindingEvent : counter : 1
replace: HttpSessionBindingEvent : counter : 2

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 уничтожения.

/ # 11 # обработка событий создания и уничтожения запроса к сервлету # SimpleRequestListener.java */*

```
package by.bsu.control.listener;
import javax.servlet.ServletContext;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpServletRequest;
@WebListener
public class SimpleRequestListener implements ServletRequestListener {
          public void requestInitialized(ServletRequestEvent ev) {
                    // будет использован для получения информации о запросе
                    HttpServletRequest req = (HttpServletRequest) ev.getServletRequest();
                    String uri = "Request Initialized for " + req.getRequestURI();

                    String id = "Request Initialized with ID="
                                        + req.getRequestedSessionId();
                    System.out.println(uri + "\n" + id);
                    ServletContext context = ev.getServletContext();
                    // счетчик числа созданных запросов
                    Integer reqCount = (Integer)req.getSession().getAttribute("counter");
                    if(reqCount == null) {
                              reqCount = 0;
                    }
                    context.log(uri + "\n" +id + ", Request Counter =" + reqCount);
          }
          public void requestDestroyed(ServletRequestEvent ev) {
                    System.out.println("Request Destroyed: "
                                        + ev.getServletRequest().getAttribute("lifecycle"));
          }
}
```

С помощью экземпляра **ServletRequestEvent**, как видно из примера, можно получить доступ к экземпляру запроса **HttpServletRequest**, а через него и к сессии пользователя, и к контексту приложения.

После вызова страницы **index.jsp** будет создан новый **request** с нулевым значением атрибута **lifecycle:**

Request Initialized for /BookSessionListenerControl/index.jsp
Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3
Request Destroyed: null

При переходе к сервлету атрибут будет добавлен:

Request Initialized for /BookSessionListenerControl/sessionservlet
Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3
Request Destroyed: CONTROL request LIFECYCLE

После обратного перехода к **index.jsp** объект **request**, а вместе с ним атрибут **lifecycle**, будет уничтожен:

Request Initialized for /BookSessionListenerControl/index.jsp
Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3
Request Destroyed: null

/ # 12 # обработка событий инициализации контекста приложения # SimpleContextListener.java */*

```
package by.bsu.control.listener;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
@WebListener
public class SimpleContextListener implements ServletContextListener {
    public void contextInitialized(ServletContextEvent ev) {
          ServletContext context = ev.getServletContext();
          String mailFeedback = context.getInitParameter("feedback");
          context.log("Context Initialized with parameter: " + mailFeedback);
          System.out.println("Context Initialized with parameter: " + mailFeedback);
    }
    public void contextDestroyed(ServletContextEvent ev) {
    }
}
```

Параметры инициализации представлены в **web.xml** в виде:

```
<context-param>
          <param-name>feedback</param-name>
          <param-value>blinov@gmail.com</param-value>
</context-param>
```

Так как событие инициализации контекста приложения производится только один раз за его жизненный цикл, то за все время работы приложения в log-файл только один раз будут выведены записи:

Dec 31, 2012 11:57:05 PM org.apache.catalina.core.ApplicationContext log
INFO: Context Initialized with parameter: blinov@gmail.com

Фильтры

Реализация интерфейса **Filter** позволяет создать объект, который перехватывает запрос, может трансформировать заголовок и содержимое запроса клиента. Фильтры не создают запрос или ответ, а только модифицируют их. 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 или jsp 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 и/или jsp.

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 **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(ServletRequest request, ServletResponse response, FilterChain chain)** 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.

Фильтр может модифицировать ответ сервера клиенту. Одним из распространенных приемов использования фильтра является модификация кодировки запроса и/или ответа. Когда страница посылает в сервлет запрос клиента, используется

установленная в запросе кодировка. В настоящее время некоторыми браузерами кодировка устанавливается по умолчанию и не зависит от кодировки страницы, обращающейся к серверу, и к тому же вообще не передается серверу приложений. Для корректной интерпретации передаваемых с запросом, например, символов кириллицы, необходимо перехватить запрос и заменить его кодировку.

 # 13 # обращение к кодировке запроса, переданной браузером # index.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>Encoding Filter</title></head>
<body>
          Кодировка запроса: ${ pageContext.request.characterEncoding }
</body></html>
```

Если фильтр не подключать, то кодировка запроса не будет определена, потому что браузер всегда передает значение **null** для кодировки запроса.

Реализация интерфейса **Filter** для поставленной задачи, изменяющей кодировку запроса и ответа на кодировку, заданную параметром фильтра.

// # 14 # фильтр, корректирующий кодировку запроса и ответа # EncodingFilter.java

```
package by.bsu.barrier.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;
@WebFilter(urlPatterns = { "/*" },
          initParams = {
          @WebInitParam(name = "encoding", value = "UTF-8", description = "Encoding Param") })
public class EncodingFilter implements Filter {
          private String code;
          public void init(FilterConfig fConfig) throws ServletException {
                    code = fConfig.getInitParameter("encoding");
          }
          public void doFilter(ServletRequest request, ServletResponse response,
                              FilterChain chain) throws IOException, ServletException {
                    String codeRequest = request.getCharacterEncoding();
                    // установка кодировки из параметров фильтра, если не установлена
                    if (code != null && !code.equalsIgnoreCase(codeRequest)) {
                              request.setCharacterEncoding(code);
                              response.setCharacterEncoding(code);
                    }
```

```
                chain.doFilter(request, response);
        }
        public void destroy() {
                code = null;
        }
}
```

Параметры инициализации задаются внутри объявления аннотации **@WebFilter** аннотацией **@WebInitParam**. Аналогичная конфигурация для предыдущей версии определяется в дескрипторе **web.xml**:

```
<filter>
        <filter-name>encodingfilter</filter-name>
        <filter-class>by.bsu.sample.filter.EncodingFilter</filter-class>
        <init-param>
                <param-name>encoding</param-name>
                <param-value>UTF-8</param-value>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>encodingfilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

Для использования фильтра для всех сервлетов и JSP используется шаблон "**/***", только для JSP — "***.jsp**", для каталога — "**/jsp/admin/***", для конкретной jsp — "**/index.jsp**", для сервлета — "**/controller**".

Фильтры применяются для обеспечения безопасности приложения, а именно, легко решают задачу запрещения несанкционированного прямого обращения к JSP. Следствием установки шаблона **urlPatterns** в значение "**/jsp/***" будет вызов фильтра при любом прямом обращении из строки браузера. Фильтр в итоге перенаправит вызов на указанную страницу.

/ # 15 # фильтр, перенаправляющий все прямые обращения к страницам на стартовую страницу # PageRedirectSecurityFilter.java */*

```
package by.bsu.barrier.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebFilter( urlPatterns = { "/jsp/*" },
```

```
        initParams = { @WebInitParam(name = "INDEX_PATH", value = "/index.jsp") })
public class PageRedirectSecurityFilter implements Filter {
        private String indexPath;
        public void init(FilterConfig fConfig) throws ServletException {
                indexPath = fConfig.getInitParameter("INDEX_PATH");
        }
        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
                HttpServletRequest httpRequest = (HttpServletRequest) request;
                HttpServletResponse httpResponse = (HttpServletResponse) response;
                // переход на заданную страницу
                httpResponse.sendRedirect(httpRequest.getContextPath() + indexPath);
                chain.doFilter(request, response);
        }
        public void destroy() {
        }
}
```

Фильтр также успешно перехватывает обращения к сервлету. Например, обращение к сервлету из любого источника перехватывается следующим фильтром. Фильтр проверяет тип пользователя, извлекая его значение из сессии. Если тип не задан, то фильтр присваивает ему значение **GUEST** и перенаправляет клиента на страницу **guest.jsp**.

/ # 16 # фильтр, перенаправляющий всех новых пользователей на гостевую страницу # ServletSecurityFilter.java */*

```
package by.bsu.barrier.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import by.bsu.barrier.client.ClientType;
@WebFilter(urlPatterns = { "/controller" }, servletNames = { "MainServlet" })
public class ServletSecurityFilter implements Filter {
        public void destroy() {
        }
        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
                HttpServletRequest req = (HttpServletRequest) request;
                HttpServletResponse resp = (HttpServletResponse) response;
                HttpSession session = req.getSession();
```

```
                ClientType type = (ClientType) session.getAttribute("userType");
                if (type == null) {
                        type = ClientType.GUEST;
                        session.setAttribute("userType", type);
                        RequestDispatcher dispatcher = request.getServletContext()
                                                .getRequestDispatcher("/jsp/guest.jsp");
                        dispatcher.forward(req, resp);
                        return;
                }
                // pass the request along the filter chain
                chain.doFilter(request, response);
        }
        public void init(FilterConfig fConfig) throws ServletException {
        }
}
```

/ # 17 # перечисление с типами пользователей некоторой системы # ClientType.java */*

```
package by.bsu.barrier.client;
public enum ClientType {
        GUEST, USER, ADMINISTRATOR
}
```

Фильтры в общем случае не перехватывают запросы, перенаправленные с помощью методов **forward()** и **include()** экземпляра **RequestDispatcher**. Элемент **dispatcherTypes** позволяет указать фильтру способ получить управление: непосредственно от сервлета или клиента значением **DispatcherType.REQUEST**, в результате перенаправления с использованием метода **forward()** значением **DispatcherType.FORWARD** или метода **include()** значением **DispatcherType.INCLUDE**) и некоторых других:

```
@WebFilter(dispatcherTypes = {
                DispatcherType.REQUEST,
                DispatcherType.FORWARD,
                DispatcherType.INCLUDE
                }, urlPatterns = { "/jsp/*" })
```

Конфигурировать доступ можно любым необходимым набором значений.

Задания к главе 17

Вариант А

Для всех заданий использовать авторизованный вход в приложение. Параметры авторизации, дату входа в приложение и время работы сохранять в сессии.

1. В тексте, хранящемся в файле, определить длину содержащейся в нем максимальной серии символов, отличных от букв. Все такие серии символов с найденной длиной сохранить в cookie.

2. В файле хранится текст. Для каждого из слов, которые вводятся в текстовые поля HTML-документа, вывести в файл cookie, сколько раз они встречаются в тексте.
3. В файле 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: ключевые слова используются только для обозначения операторов, 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.
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.
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, *.kz 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 17

Вопрос 17.1.

Выберите корректные утверждения (2):

1) сессия может использоваться разными сервлетами для доступа к одному клиенту;
2) сессия может быть завершена только автоматически;
3) сессия не может быть повторно завершена;
4) если промежуток активного времени для сессии установить в ноль или отрицательное число, то сессию невозможно будет создать.

Вопрос 17.2.

Дан код:

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
HttpSession session = request.getSession(false);
if (session == null) {
        response.getWriter().println("no session");
        session = request.getSession(true);
} else {
        // i - статическое поле сервлета; static int i = 1 при объявлении
        response.getWriter().println("session " + i);
        i++;
    }
}
```

Какая информация будет результатом выполнения трехкратного запроса к указанному методу doGet, если у клиента запрещен прием cookie-файлов (1)?

1) no session;
2) session 1;
3) session 2;
4) session 3;
5) ничего из вышеперечисленного.

Вопрос 17.3.

Дан код:

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
// i - статическое поле сервлета; static int i = 1 при объявлении
Cookie[] cook = request.getCookies();
if (cook != null) {
        Cookie cook2 = new Cookie(i + "", i + "");
        i++;
        cook2.setMaxAge(10);
        response.addCookie(cook2);
        response.getWriter().println("cookies = " + cook.length);
} else {
        Cookie cook2 = new Cookie(0 + "", 0 + "");
        cook2.setMaxAge(10);
        response.addCookie(cook2);
        response.getWriter().println("one cookie");
        }
}
```

Укажите, сколько cookie-записей уйдет в ответе клиенту (response), если один клиент осуществит вызов метода doGet 5 раз в течение 10 секунд (1):

1) 5;
2) 4;
3) 3;
4) 2;
5) 1;
6) 0;
7) произойдет ошибка компиляции.

Вопрос 17.4.

Укажите типы событий, которые обрабатывают listener-объекты уровня сессии, определяющие жизненный цикл сессии (3):

1) session creation
2) addition of session attributes

3) removal of session attributes
4) replacement of session attributes
5) session invalidation
6) session timeout

Вопрос 17.5.

Выберите корректные утверждения (2):

1) фильтры позволяют создавать объекты, которые могут изменять заголовок запроса;
2) фильтры не позволяют создавать объекты, которые могут изменять заголовок ответа;
3) фильтры могут создать новый запрос и ответ;
4) фильтр можно использовать для фильтрации более одного сервлета.

Глава 18

JSP STANDARD TAG LIBRARY

Повторное использование кода — обычная техника программирования. В большинстве случаев такой код инкапсулируется в методе, при создании JSP — в стандартом теге. Наиболее стандартные и востребованные решения были определены и собраны в библиотеку JSTL.

JSP-страницы, включающие 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.

Библиотека тегов JSTL состоит из пяти групп тегов: основные теги — **core**, теги форматирования — **formatting**, теги для работы с SQL — **sql**, теги для обработки XML — **xml**, функции-теги для обработки строк — **functions**.

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 (*не рекомендуется*);
— обработку XML;
— форматирование и разбор строк посредством библиотеки функций.

Библиотека	Число тегов	Описание
core	14	*Основные*: **if**/**then** выражения и конструкции множественного выбора; вывод; создание и удаление контекстных переменных; управление свойствами JavaBeans компонентов; перехват исключений; **forEach** для итерирования коллекций; создание URL и импортирование их содержимого.
formatting	12	*Интернационализация и форматирование*: установка локализации; локализация текста и структуры сообщений; форматирование и анализ чисел, процентов, денежных единиц и дат.

Библиотека	Число тегов	Описание
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.
fn	16	*Строки*: обработка объектов типа String и определение размера коллекций.

Библиотеку JSTL версии 1.2.1 (**javax.servlet.jsp.jstl-1.2.1.jar** и **javax.servlet.jsp.jstl-api-1.2.1.jar**) или более позднюю версию можно загрузить с сайта apache.org. Библиотеки следует разместить в каталоге **/lib** проекта или в соответствующем каталоге контейнера сервлетов. При указании значения параметра **xmlns** элемента **root** (для JSP-страницы значение параметра директивы **taglib uri=""**) необходимо быть внимательным, так как при неправильном указании адреса JSP-страница не сможет получить доступ к тегам JSTL. Проверить корректность значения 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:

 # 1 # простой вывод # first.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
          <c:out value="Welcome to JSTL"/>
</body></html>
```

Тег **<c:out/>** отправляет значение параметра **value** в поток **JspWriter**.

JSTL core

Библиотека **core** содержит в себе наиболее часто используемые решения.

Теги общего назначения:

<c:out /> — вычисляет и выводит значение выражения;

<c:set /> — создает и 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 выполняется, 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, массива, итерируемого объекта;

<c:forTokens /> — выполняет тело тега для каждой лексемы в строке.

Теги обработки URL:

<c:redirect/> — перенаправляет запрос на указанный адрес **URL**;

<c:import/> — добавляет на JSP содержимое указанного веб-ресурса;

<c:url/> — формирует адрес с учетом 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**. Существует аналогия между тегом **<c:set>** и тегом **<jsp:useBean>**. Оба создают и помещают экземпляры в заданную область видимости. Но **<jsp:useBean>** только непосредственно создает экземпляр конкретного типа, а **<c:set>**, создав ссылку, позволяет извлекать значение, например, из параметров запроса, сессии и т. д.

 # 2 # демонстрация работы тегов c:set, c:if # jstl_set_if.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<c:set var="user" value="guest" scope="page"/>
<c:if test="${ not empty user and user eq 'guest' }">
          User is Guest
</c:if>
<br/>
</body></html>
```

В браузер будет выведено:

User is Guest

Тег **<c:if>** вычисляет значение атрибута **test**. Если результатом будет истина, как в приведенном выше примере, то тело тега выполняется, если ложь — игнорируется. Где понятию «ложь» соответствует любое значение, отличное от **true**.

Сохранить результаты вычисления в атрибуте **test** можно в другом атрибуте **var** с возможностью следующего использования в контексте страницы.

 # 3 # использование значения атрибута test # jstl_if.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>Core: if</title></head>
```

```
<body>
<c:if test="${ 5 < 7 }" var = "firstOperation" scope = "page">
          first if : is TRUE
</c:if>
<br/>
<c:out value="First Operation variable is : ${ firstOperation }"/>
<br/>
<c:if test="${ firstOperation } + some bt" var = "secondOperation">
          second if : is TRUE
</c:if>
<br/>
<c:out value="Second Operation variable is : ${ secondOperation }"/>
</body></html>
```

В результате в браузер будет выведено:

first if : is TRUE
First Operation variable is : true
Second Operation variable is : false

Тег **<c:set>** может присваивать значение и без помощи атрибута **value**, просто объявляя его в теле тега:

```
<c:set var="user" scope="page">
          guest
</c:set>
```

Такая запись необходима в ситуации, когда размещение значения в атрибут **var** невозможно: например, при обязательном наличии в значении двойных или одинарных кавычек.

```
<c:set var="users" scope="page">
          "guest" 'client' "admin"
</c:set>
<c:out value="${users}"/>
```

Тег **<c:remove>** удаляет переменную из области видимости и при попытке доступа к ней после выполнения тега результатом будет **null**.

 # 4 # удаление атрибута # remove.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<c:set var="user" scope="page">
          "guest" 'client' "admin"
</c:set>
          User is set= <c:out value="${user}" />
<c:remove var="user"/>
<br/>
          User after remove= <c:out value="${user}"/>
</body></html>
```

После срабатывания тега **<c:remove>** атрибут **user** будет удален из области видимости и при попытке его поиска будет возвращено значение **null**, в браузере не отображаемое.

Автоматическое приведение типов и перехват исключений

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 **Double**, то значение идентификатора будет приведено к типу **Double**.

 # 5 # приведение типов # type.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<c:set var="number" value="7.1" scope="session"/>
<c:if test="${ number < 9.0 }">
          <c:out value ="Number ${ number }"/> is smaller than (9.0)
</c:if>
</body></html>
```

Параметр из переменной **number** передается в виде строки **String**, и преобразованное к числу значение будет корректным при использовании переменной. Причем совпадение по типу должно быть точным. Если ожидается тип **Long**, а передается строка в виде **Double**, приведение типов будет некорректным, что приведет к генерации исключения, как в предлагаемом коде

```
<c:set var="number" value="7.1" scope="session"/>
<c:if test="${ number < 9 }">
         <c:out value ="Number ${ number }"/> is smaller than (9)
</c:if>
```

где число **7.1** не может быть приведено к ожидаемому типу **Long**. В итоге страница выполнена не будет и нормальная работа остановлена генерацией ошибки с кодом 500.

Для предотвращения такой ситуации и нормального завершения выполнения страницы следует использовать тег **<c:catch>**.

 # 6 # перехват исключения # jstl_catch.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"   pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<title>Core: catch</title>
```

```
</head>
<body>
          <c:set var="number" value="7.1" scope="session" />
          <c:catch var="formatException">
                    <c:if test="${ number < 9 }">
                              <c:out value="Number ${ number }" /> is smaller than (9)
                    </c:if>
          </c:catch>
          <br/>
          <c:if test="${not empty formatException }">
                    Wrong format number: <c:out value="${ number }" /> isn't Long
                    <br /><hr />
                    Generated exception
                    <hr />
                    ${ formatException }
          </c:if>
</body></html>
```

В итоге разработчик будет предупрежден о неправильных данных и сможет принять меры по недопущению некорректной информации в указанный параметр.

Тег **<c:catch>** введен для избирательной обработки одного или нескольких исключений, генерация которых не должна приводить к переходу на заданные страницы обработки ошибок. Страница, на которой произошло исключение, будет корректно исполнена до конца.

Выражение **<c:if test="${not empty formatException}">** позволяет определить, происходила ошибка или нет. В случае возникновения исключения переменная в атрибуте **var** тега **</c:catch>** инициализируется экземпляром **javax.el.ELException**, доступ к которому может быть получен в контексте страницы, как это и сделано в приведенном примере.

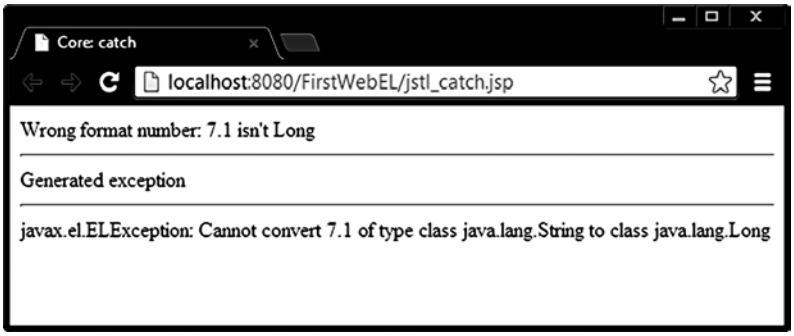

Рис. 18.1. *Генерация и перехват исключения*

Исключающие условия <c:choose>

Операции по управлению условным переходом существенно расширяются благодаря тегу **<c:choose>**. Выполнение производится только одного из всего набора действий тега после определения истинности условия. Все остальные выражения, даже истинные, игнорируются, после чего управление передается тегу, следующему после тега **</c:choose>**. Если ни одно из действий **<c:when>** не выполнено, то есть **test** не принял значение **true**, то управление переходит к единственному оператору **<c:otherwise>**, который может и отсутствовать.

 # 7 # множественный условный переход # choose_when.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>Core: choose</title></head>
<body>
<c:set var="number" value="50"/>
          <c:choose>
                              <c:when test="${ number > 10 }" >
                                        <c:out value="число ${ number } больше 10"/>
                              </c:when>
                              <c:when test="${ number > 40 }" >
                                        <c:out value="число ${ number } больше 40"/>
                              </c:when>
                              <c:when test="${ number > 60 }" >
                                        <c:out value="число ${ number } больше 60"/>
                              </c:when>
                              <c:otherwise>
                                        <c:out value="число ${ number } меньше 10"/>
                              </c:otherwise>
          </c:choose>
</body></html>
```

Если необходимо повторить функциональность if-then-else, следует использовать в теге **<c:choose>** только один **<c:when>** вместе с **<c:otherwise>**. Для организации конструкции, подобной оператору **switch**, в условиях следует использовать точные совпадения значений или непересекающиеся интервалы значений, например:

```
<c:choose>
                    <c:when test="${ number > 10 and number <= 40 }" >
                              больше 10 и меньше 40
                    </c:when>
                    <c:when test="${ number > 40 and number <= 60 }" >
                              больше 40 и меньше 60
                    </c:when>
                    <c:when test="${ number > 60 }" >
```

```
                    больше 60
                </c:when>
                <c:otherwise>
                    меньше 10
                </c:otherwise>
</c:choose>
```

Итераторы <c:forEach> и <c:forTokens>

Возможности EL предоставляют способ доступа к элементам массивов, списков, множеств, карт отображений, свойств и других итерируемых объектов. Однако часто возникает задача вывести всю коллекцию или ее часть, которая может оказаться значительной по размеру. К тому же, в следующем запросе эта коллекция может иметь уже другой размер. Для корректной работы с такими объектами используется тег **<c:forEach>**.

Пусть к экземпляру запроса в методе **doGet()** сервлета **BaseServlet** добавлен атрибут **lst**, представляющий список:

```
MessageList list = new MessageList();
request.setAttribute("lst", list);
request.getRequestDispatcher("jstl_foreach.jsp").forward(request,response);
```

где экземпляр класса **MessageList** предварительно заполнен значениями:

```
public class MessageList extends ArrayList<Message> {
        { // логический блок вставлен для упрощения создания коллекции
          // в реальном коде его быть не должно!
                this.add(new Message(72, "Hello"));
                this.add(new Message(31, "Bye"));
                this.add(new Message(11, "Good Bye"));
                this.add(new Message(37, "Wow"));
                this.add(new Message(92, "Hey"));
                this.add(new Message(77, "Hi"));
                this.add(new Message(71, "Ok"));
        }
}
```

Тег **<c:forEach>** имеет атрибуты **items** для размещения коллекции и **var** для доступа к элементу коллекции на каждой итерации. Необязательный атрибут **varStatus** содержит номер итерации, доступ к которому осуществляется методом **getCount()**:

 # 8 # демонстрация работы тегов c:forEach # jstl_foreach.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>Core: forEach</title></head>
```

```
<body>
<table>
    <c:forEach var="elem" items="${lst}" varStatus="status">
          <tr>
                    <td><c:out value="${ elem }" /></td>
                    <td><c:out value="${ elem.id }" /></td>
                    <td><c:out value="${ elem.text }" /></td>
                    <td><c:out value="${ status.count }" /></td>
          </tr>
    </c:forEach>
</table>
</body></html>
```

В результате в браузер будет выведено (см. рис. 18.2).

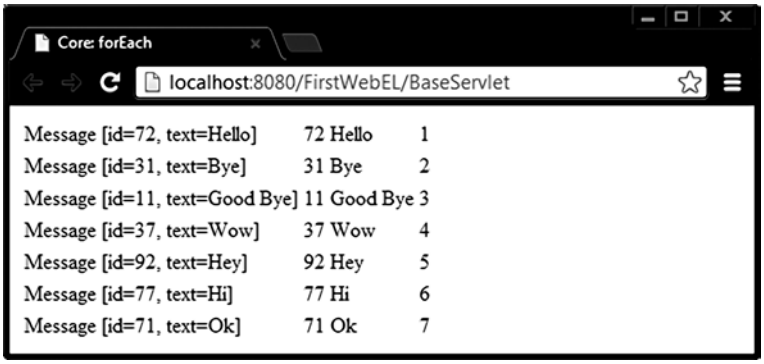

Рис. 18.2. *Обработка списка в теге foreach*

Цикл может начинаться не с первого элемента коллекции и заканчиваться не последним. Атрибуты **begin** (должен принимать значение >=0) и **end** (должен быть >= **begin**) обеспечивают ограниченное действие цикла. Тег **<c:forEach>** в виде

```
<c:forEach var="elem" items="${lst}" varStatus="status" begin="2" end="4">
```

Изменит вывод предыдущего примера на следующий (см. рис. 18.3).

Также можно изменять размер шага движения по коллекции, то есть выводить, например, через один, два или более элементов. Такое использование позволяет организовать атрибут **step**.

```
<c:forEach var="elem" items="${ lst }" varStatus="status" step="2">
```

Вывод исходного примера изменится в очередной раз (см. рис. 18.4).

Атрибут **varStatus** в каждом из случаев будет содержать относительную нумерацию в порядке доступа к элементам в цикле вне зависимости от их порядка в самой коллекции.

Если необходимо разбить на части объект типа **String** с использованием разделителей, применяется тег **<c:forTokens>**. Кроме атрибутов **var** и **items**

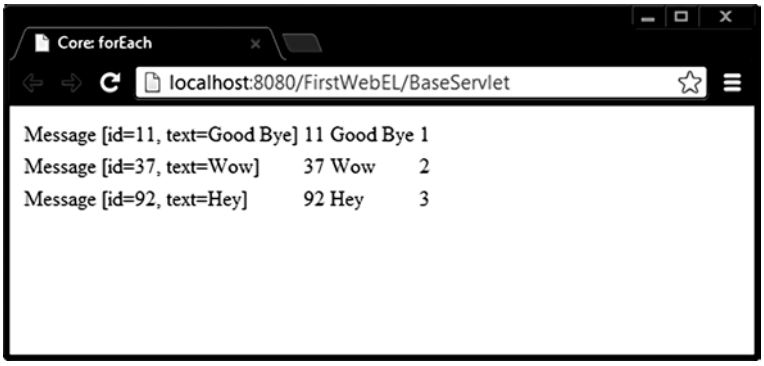

Рис. 18.3. *Обработка части списка циклом foreach*

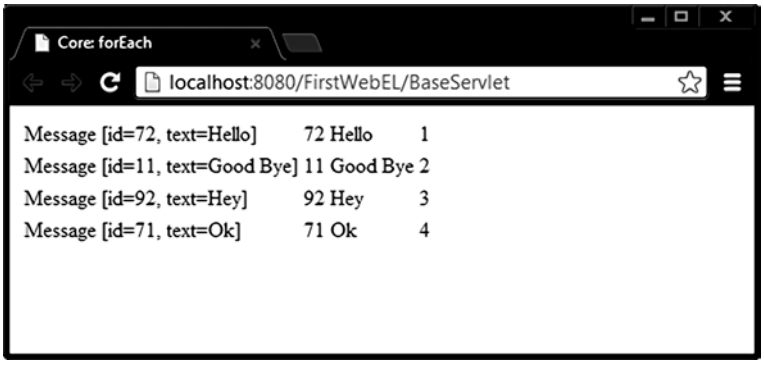

Рис. 18.4. *Обработка списка циклом foreach через один элемент*

с аналогичными свойствами здесь применяется атрибут **delims** для определения списка символов-разделителей, отбрасываемых при выделении подстрок. При его отсутствии строка не разбивается.

 # 9 # разбиение строки по шаблону # token.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>Core: forTokens</title></head>
<body>
<c:set var="str" value="45, 76 - 32 : 77 . 91" />
    <c:forTokens var="token" items="${ str }" delims=".,-:)">
          <c:out value="${ token }" /><br/>
    </c:forTokens>
</body></html>
```

Результатом выполнения будет вывод в браузер столбца чисел:

45
76
32
77
91

Регулярные выражения для определения разделителей не применяются.

Включение ресурсов

В реальных проектах JSP-страницы часто состоят из набора статических и динамических элементов, находящихся в разных местах контекста и вне его. Заголовок всех страниц приложения (header) и нижняя часть (footer) могут быть одинаковыми. Чтобы избежать плохой практики «copy-paste», следует выделить повторяющиеся части в отдельные страницы или фрагменты страниц и импортировать по необходимости.

Тег **<c:import>** позволяет включать содержимое ресурса, указанного в единственном обязательном атрибуте **url**. Адрес может быть внешним или внутренним, относительным или абсолютным по отношению к данному контексту.

Примеры использования:

```
• <c:import url="\WEB-INF\jspf\footer.jspf" charEncoding="utf-8"/ >
• <c:import url="\jspf\header.jspf" charEncoding="utf-8" >
      <param name="title" value="import header info"/>
  </c:import>
• <c:import url="ftp://somestore.com/project/sample.war"/>
• <c:import url="\js\calendar.js" />
```

Тег **<c:import>** получает доступ к источнику, чтение информации из которого происходит непосредственно без буферизации. Контент включается построчно в исходную JSP. По сравнению с action-тегом <**jsp:include**> и директивой <%@**include %>** тег **<c:import>** обеспечивает более совершенное включение динамических ресурсов.

Пусть существует некоторое приложение, состоящее из четырех простых страниц: **index.jsp**, **admin.jsp**, **header.jsp** и **footer.jsp**. Все страницы размещены в папках так, как указано на следующем рисунке.

Стартовая страница импортирует общие страницы **header.jsp** и **footer.jsp** и содержит простой переход на страницу **admin.jsp**, которая, в свою очередь также импортирует указанные страницы.

Рис. 18.5.
Структура проекта с jsp, разложенными по папкам

 # 10 # стартовая страница с двумя тегами c:import # index.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>
<head><title>Index page</title></head>
<body>
        <c:import url="jsp\admin\fragment\header.jsp" />
<br/>
<a href="jsp\admin\admin.jsp" >GoTo admin page</a>
        <c:import url="jsp\common\footer.jsp" />
</body></html>
```

Если путь к файлу начинается из корневой директории проекта, то перед первой папкой не следует ставить символ «\» текущего контекста.

Страницы **header.jsp** и **footer.jsp** содержат выражение для извлечения из области видимости запроса значения **admin**. Это значение никто, ни в одной из приведенных страниц не добавлял. Значению в итоге будет присвоен **null**, которое поток вывода игнорирует.

 # 11 # импортируемая страница заголовка # header.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
<head><title>Header</title></head>
<body>
<hr/>
        Hello, ${requestScope.admin}
<hr/>
</body></html>
```

 # 12 # импортируемая страница с нижней частью # footer.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
<head><title>Footer</title></head>
<body>
<hr/>
        Copyright, 2001-2013, ${requestScope.admin}
</body></html>
```

Результат выполнения страницы **index.jsp** (см. рис. 18.6).

Страница **admin.jsp** содержит тег **<c:set>** для добавления атрибута **admin** в область видимости запроса. При импорте страницы получат доступ к этому значению и осуществят его вывод.

 # 13 # страница админа с двумя тегами c:import # admin.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>
```

```
<head><title>Admin page</title></head>
<body>
<c:set var="admin" value="Blinov" scope="request"/>
<c:import url="fragment\header.jsp" />
  <form action="${ pageContext.request.contextPath }/index.jsp">
          Content admin page<br/>
          <input type="submit" value="to Index">
  </form>
<c:import url="..\common\footer.jsp" />
</body></html>
```

После добавления атрибута будет получено (рис. 18.7.).

При возврате на **index.jsp** будет создан новый экземпляр **request**, поэтому на этой странице переменная **admin** существовать не будет.

Кодировка включаемой страницы по умолчанию всегда имеет значение ISO-8859-1. Если включаемая страница содержит кириллические символы, то кодировку следует указывать явно при импорте:

```
<c:import url="fragment\header.jsp" charEncoding="utf-8" />
```

Рис. 18.6. *Запуск страницы index.jsp до добавления атрибутов*

Рис. 18.7. *Страница admin.jsp с добавленным атрибутом*

Динамические адреса и перенаправление

Запрос можно перенаправить на другую страницу, применив тег **<c:redirect>**. Тег не является часто используемым, так как навигация между страницами при следовании принципам шаблона MVC обычно сильно ограничена.

 # 14 # редирект на другую страницу с передачей параметров # index_redirect.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<html>
<head><title>Index with redirect</title></head>
<body>
<c:redirect url="jsp/admin/destination.jsp">
          <c:param name="firstname" value="Ostap"/>
          <c:param name="lastname" value="Bender"/>
</c:redirect>
</body></html>
```

При запуске данной страницы будет сразу же осуществлен переход на страницу **destination.jsp**, и пользователь приложения увидит только результаты ее работы, а именно:

 # 15 # целевая страница с выводом переданных параметров # destination.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>
<head><title>Destination page</title></head>
<body>
<c:forEach var="ps" items="${param}">
          <c:out value="${ps.key} : ${ps.value}"/><br/>
</c:forEach>
</body></html>
```

В результате в браузер будут выведены имена переданных параметров и их значения:

lastname: Bender
firstname: Ostap

Тег **<c:param>** позволяет объявлять и передавать с запросом параметры, которые доступны на целевой странице и передаются со строкой запроса в виде:

```
http://localhost:8080/FirstProject/jsp/admin/destination.jsp?firstname=Ostap&lastname=Bender
```

Тег **<c:url>** позволяет формировать переменную с адресом, значение которого может извлекаться из контекста или формироваться динамически:

 # 16 # динамическое формирование адресов # index_url.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>
<head><title>Index page</title></head>
<body>
          <c:url value="jsp/admin/destination.jsp" var="newUrl" />
          <a href='<c:out value="${ newUrl }"/>'>go</a>
</body></html>
```

JSTL formatting

Библиотека содержит теги форматирования и интернационализации для разработки локализованных приложений. При грамотном использовании страницы вообще не будут содержать текст, а вся информация (надписи на кнопках и ссылках, заголовки, сообщения и пр.) будет извлекаться из файлов свойств.

```
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

— для страницы JSP;

```
<jsp:root version="1.2" xmlns:fmt= "http://java.sun.com/jsp/jstl/fmt">
```

— для JSP-документа.

Теги интернационализации:

<fmt:setLocale/> — устанавливает региональные установки для страницы на основе объекта класса **Locale**;

<fmt:setBundle/>, **<fmt:bundle/>** — устанавливают объект **ResourceBundle**, используемый на странице. В зависимости от установленного значения локали выбирается файл ресурсов (как правило, properties), соответствующий указанному языку, стране или региону;

<fmt:message/> — извлекает локализованное сообщение из ресурса, определенного тегами **<fmt:setBundle/>** или **<fmt:bundle/>**;

<fmt:requestEncoding /> — с атрибутом **value="utf-8"** устанавливает кодировку входящего запроса.

Теги форматирования дат и чисел:

<fmt:timeZone/>, **<fmt:setTimeZone/>** — устанавливает часовой пояс, используемый для форматирования;

<fmt:formatNumber/>, **<fmt:formatDate/>** — форматирует числа/даты с учетом 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 **formatdate.jsp** выводит на экран текущую дату и время с учетом установленного объекта класса **Locale**.

 # 17 # вывод даты и времени # formatdate.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html><head><title>Формат даты</title></head>
<body>
<jsp:useBean id="now" class="java.util.Date" />
<fmt:setLocale value="en-EN"/>
          Вывод даты в формате English<br/>
Сегодня: <fmt:formatDate value="${now}" /><br/>
<fmt:setLocale value="ru-RU"/>
          Вывод даты в формате Russian<br/>
Сегодня: <fmt:formatDate value="${now}" /><br/>
          Стиль времени:
(short): <fmt:formatDate value="${now}" type="time" timeStyle="short" /><br/>
(medium):<fmt:formatDate value="${now}" type="time" timeStyle="medium" /><br/>
(long): <fmt:formatDate value="${now}" type="time" timeStyle="Long" /><br/>
</body></html>
```

Результатом работы будет вывод в браузер (см. рис. 18.8.).

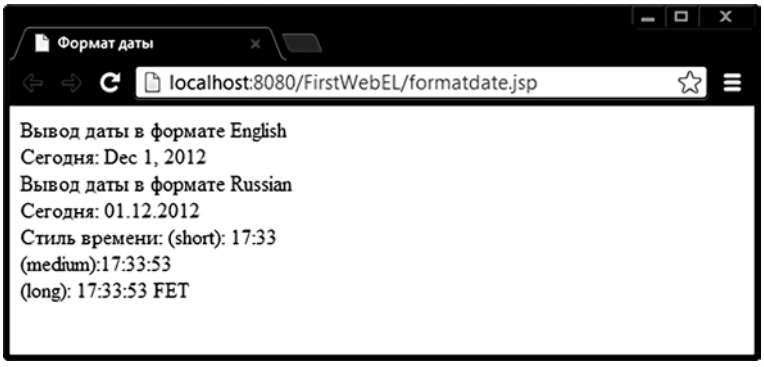

Рис. 18.8. *Вывод даты и времени в разных форматах*

В следующем примере реализован еще один способ вывода времени и даты

 # 18 # полный вывод даты и времени # timezone.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<html>
<head><title>timezone</title></head>
<body>
<jsp:useBean id="now" class="java.util.Date" />
```

```
          Вывод даты и времени с помощью тега<br/> fmt:formatDate
          и установки TimeZone
<br/>
<fmt:setLocale value="ru-RU"/>
<fmt:timeZone value="GMT+4:00">
          Locale: RU :
          <fmt:formatDate value="${now}" type="both" dateStyle="medium" timeStyle="long"/><br/>
</fmt:timeZone>
<fmt:setLocale value="pl-PL"/>
<fmt:timeZone value="GMT+1:00">
          Locale: PL :
<fmt:formatDate value="${now}" type="both" dateStyle="full" timeStyle="medium"/><br/>
</fmt:timeZone>
</body></html>
```

Результат работы страницы смотреть на рис. 18.9.

Рис. 18.9. *Вывод даты и времени при задании часовых поясов*

Страница **formatnumber.jsp** выводит числа в соответствии с региональными установками.

 # 19 # форматы числа # formatnumber.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html><head><title>Формат числа</title></head>
<body>    <c:set var="currentNumber" value="118000"/>
          <c:out value="Вывод формата числа : ${currentNumber}"/> <br/>
Формат (по умолчанию) : <fmt:formatNumber value="${currentNumber}" /><br/>
Процентный формат : <fmt:formatNumber value="${currentNumber}" type="percent"/><br/>
          <fmt:setLocale value="be-BY"/>
Белорусские рубли : <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
          <fmt:setLocale value="pl-PL"/>
Польская валюта :
```

```
        <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
Французская валюта :
        <fmt:setLocale value="fr-FR"/>
        <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
</body></html>
```

Результат работы страницы смотреть на рис. 18.10.

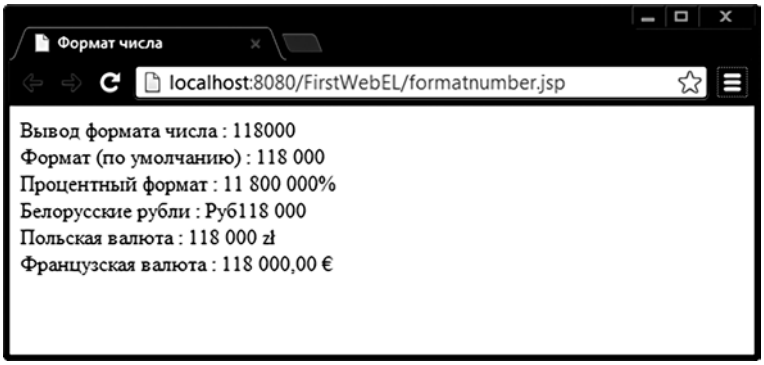

Рис. 18.10. *Обычный, денежный и процентный форматы числа*

Теги **<fmt:setLocale>** и **<fmt:setBundle>** позволяют задать локализацию страницы в целом и исключить из текста страниц конкретные текстовые сообщения, заменяя их ссылками на имена ключей файла ресурсов с расширением **properties**.

 # 20 # локализация страницы с использованием fmt:setBundle # setbundle.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<fmt:setLocale value="en_US" scope="session" />
<fmt:setBundle basename="resource.pagecontent" var="rb" />
<html><head>
<title><fmt:message key="label.title" bundle="${ rb }" /></title>
</head>
<body>
<fmt:message key="label.welcome" bundle="${ rb }" />
<hr/>
<fmt:message key="footer.copyright" bundle="${ rb }" />
</body></html>
```

Результат работы страницы смотреть на рис. 18.11. Атрибуты **var** и **bundle** можно синхронно опустить.

Где файлы **pagecontent_en_us.properties** и **pagecontent_ru_ru.properties** содержат информацию о значении ключей в соответствии с заданным для этих файлов языком и страной:

Рис. 18.11. *Локализованная страница*

 # 21 # файлы ресурсов # pagecontent_en_us.properties # pagecontent_ru_ru.properties

```
# Locale en_US
label.title=Example
label.welcome=Welcome!
footer.copyright=Copyright 2001-2013, Blinov
```

а также:

```
# Locale ru_RU
label.title=Пример
label.welcome=Добро пожаловать!
footer.copyright=Все права защищены 2001-2013, Блинов
```

Тег **<fmt:setBundle>** задает возможность работы с конкретным файлом ресурсов для всей области видимости страницы.

Тег **<fmt:bundle>** задает возможность работы в своей области видимости с конкретным префиксом, например **label.**, из своей области видимости. То есть информация из ключей, начинающихся с других префиксов, будет недоступна

 # 22 # локализация страницы с использованием fmt:bundle # bundle.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<fmt:setLocale value="ru_RU" scope="session" />
<fmt:bundle basename="resource.pagecontent" prefix = "label." >
          <html><head>
                    <title><fmt:message key="title" /></title>
          </head>
          <body>
                    <fmt:message key="welcome" />
          </body></html>
</fmt:bundle>
```

Атрибут **prefix** можно опустить, тогда ключи нужно указывать полностью. В результате в браузер будет выведено сообщение:

Добро пожаловать!

Можно попытаться получить доступ к информации с другим ключом:

 # 23 # локализация страницы с использованием fmt:bundle # bundle.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<fmt:setLocale value="ru_RU" scope="session" />
<fmt:bundle basename="resource.pagecontent" prefix = "label." >
          <html><head>
                    <title><fmt:message key="title" /></title>
          </head>
          <body>
                    <fmt:message key="welcome" />
                    <br/>
                    <fmt:message key="footer.copyright" />
          </body></html>
</fmt:bundle>
```

В результате в браузер уже будет получено:

Добро пожаловать!
??? label. footer.copyright???

При попытке по ключу **footer.copyright** тег **<fmt:message>** пытается найти соответствующее значение с определенным для него префиксом **label** и, не найдя его в файле ресурсов, выдает сообщение **??? label. footer.copyright???**

JSTL sql

Используется для выполнения запросов SQL непосредственно из JSP и обработки результатов запроса в JSP.

<%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
— для обычной страницы JSP;

<jsp:root version="1.2" xmlns:sql="http://java.sun.com/jsp/jstl/sql">
— 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> — устанавливает data source (пула соединений) 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 запрос на вставку/удаление/изменение 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 импорта, парсинга и обработки данных из XML-документов в документе JSPX.

Подключение библиотеки для XML формата JSP:

```
<jsp:root version="1.2" xmlns:x= "http://java.sun.com/jsp/jstl/xml">
```

Список тегов:

<x:set> — XML-версия тега **<c:set>;**

<x:out> — XML-версия тега **<c:out>;**

<x:forEach> — XML-версия тега **<c:forEach>**;

<x:if> — XML-версия тега **<c:if>**;

<x:choose> — XML-версия тега **<c:choose>**;

<x:when> — XML-версия тега **<c:when>;**

<x:otherwise> — XML-версия тега **<c:otherwise>**;

<x:parse> — разбор XML-документа;

<x:transform> — трансформация XML-документа с применением XSLT-преобразования;

<x:param> — XML-версия тега **<c:param>**, определяющая параметры для другого тега **<x:transform>**.

XML-документ может быть импортирован из внешнего файла или может быть размещен непосредственно в коде JSP, что практикуется крайне редко.

Тег **<x:parse var="doc">** выполняет разбор документа, представленного в виде тела тега, и помещает построенный объект в переменную **doc**, объявленную в атрибуте **var** тега **parse**.

В XML-документе представлена информация о нескольких студентах, причем данные об одном студенте находятся в теге **<student>**. Вывод информации о всех студентах производится тегом

```
<x:forEach select="$doc/students/student" var="stud">
```

где элемент **<student>** со всем его содержимым ассоциируется с переменной **stud** после выбора из объекта **doc** с помощью простого оператора **$doc/students/student**. Далее на каждой итерации цикла из объекта **stud** выбираются элементы **<x:out select="$stud/name"/>** и атрибуты **<x:out select="$stud/@login"/>**, которые выводятся в браузер тегом **<x:out>**.

 # 24 # разбор внедренного документа # xml_inside.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" xmlns:x="http://java.sun.com/jsp/jstl/xml"
version="2.0">
<jsp:directive.page contentType="text/html; charset=UTF-8"/>
```

```
<html>
  <head><title>XML inside</title></head>
  <body>
      <!--using tag x:parse-->
      <x:parse var="doc">
          <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>
       </x:parse>
      <!--using tags x:forEach and x:out-->
      <x:forEach select="$doc/students/student" var="stud">
        Name:
         <x:out select="$stud/name"/><br/>
        Login:
         <x:out select="$stud/@login" /><br/>
        Faculty:
         <x:out select="$stud/@faculty" /><br/>
        Country:
         <x:out select="$stud/address/country" /><br/>
        City:
         <x:out select="$stud/address/city" /><br/>
        Street:
         <x:out select="$stud/address/street" /><br/>
        Telephone:
        <x:out select="$stud/telephone" /><br/>
        <hr/><br/>
      </x:forEach>
</body></html>
</jsp:root>
```

Результатом будет вывод в браузер (см. рис. 18.12.).

Размещение XML-информации непосредственно в документе JSPX снижает гибкость приложения, поэтому разумным решением будет импортировать XML-документ, размещенный вне кода страницы.

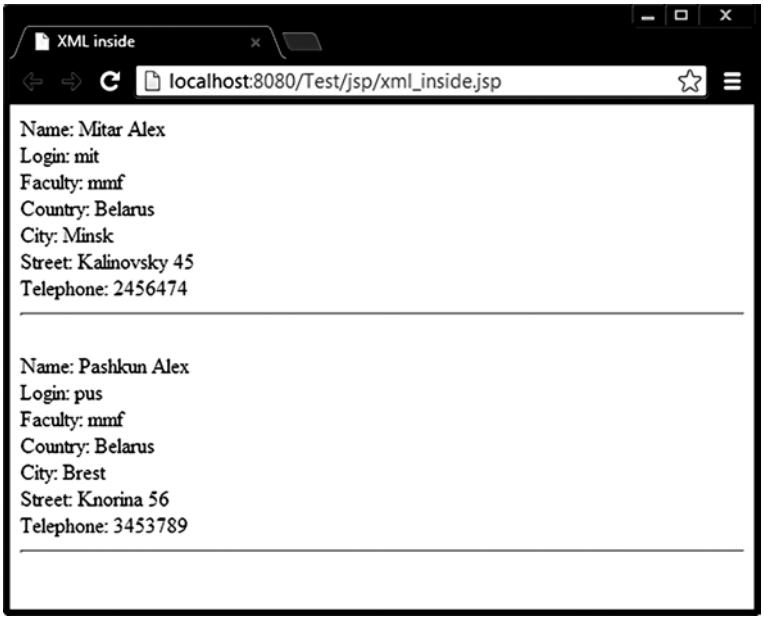

Рис. 18.12. *Разбор XML-информации*

 # 25 # извлечение информации из внешнего документа # xml_outside.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:x="http://java.sun.com/jsp/jstl/xml"
          xmlns:c="http://java.sun.com/jsp/jstl/core" version="2.0">
          <jsp:directive.page contentType="text/html; charset=UTF-8" />
          <html><head><title>XML outside</title></head><body>
          <x:parse var="doc">
                    <c:import url="../xml/students.xml" />
          </x:parse>
          <!--using tag x:set-->
          Student name:
          <x:set var="studentName" select="$doc/students/student[1]/name" />
          <x:out select="$studentName/." />
</body></html>
</jsp:root>
```

Результатом будет вывод в браузер строки:

Student name: Mitar Alex

Тег выбор по условию **<x:if>** работает аналогично тегу из библиотеки **core**, только использует свой синтаксис поиска соответствия в экземпляре документа.

 # 26 # поиск информации # xml_if.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" xmlns:x="http://java.sun.com/jsp/jstl/xml"
xmlns:c="http://java.sun.com/jsp/jstl/core" version="2.0">
<jsp:directive.page contentType="text/html; charset=UTF-8"/>
<html><head><title>XML x:if</title></head><body>
      <x:parse var="doc">
          <c:import url="../xml/students.xml"/>
      </x:parse>
     <!--using x:if tag-->
     Find students from Minsk:
     <x:forEach select="$doc/students/student" var="stud">
          <x:if select="$stud/address/city = 'Minsk' ">
             <x:out select="$stud/name" /><br/>
          </x:if>
    </x:forEach>
</body></html>
</jsp:root>
```

Результатом будет вывод в браузер информации о результатах поиска:

Find students from Minsk: Mitar Alex

Тег **<x:choose>** также аналогичен тегу из базовой библиотеки:

 # 27 # множественный выбор # xml_choose.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" xmlns:x="http://java.sun.com/jsp/jstl/xml"
xmlns:c="http://java.sun.com/jsp/jstl/core" version="2.0">
<jsp:directive.page contentType="text/html; charset=UTF-8"/>
<html><head><title>XML choose</title></head><body>
<c:import url="../xml/students.xml" var="studXML"/>
<x:parse var="doc" doc="${studXML}"/>
<!--using tags x:choose -->
     <x:forEach select="$doc/students/student" var="stud">
        <x:out select="$stud/name" />:
        <x:choose>
           <x:when select="$stud/address/city = 'Minsk'">
             This student from Minsk
           </x:when>
           <x:when select="$stud/address/city = 'Brest'">
             This student from Brest
           </x:when>
           <x:otherwise>
             This student not from Minsk and Brest
           </x:otherwise>
        </x:choose>
        <br/>
     </x:forEach>
</body></html>
</jsp:root>
```

Рис. 18.13. *Выбор информации по нескольким условиям*

Результатом будет вывод в браузер (см. рис. 18.13.).

Преобразование XML-документа в другую форму документа выполняется тегом **<x:transform>**. Для трансформации используется XSL-таблица. В приведенном примере XML-документ преобразуется в HTML-форму с таблицей, в которую выбирается часть информации из XML.

 # 28 # XSL-трансформация # xml_transform.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" xmlns:x="http://java.sun.com/jsp/jstl/xml"
xmlns:c="http://java.sun.com/jsp/jstl/core" version="2.0">
<jsp:directive.page contentType="text/html; charset=UTF-8"/>
<html><head><title>XML transform</title></head>
  <body>
    <!--using tag x:transform-->
   <c:import url="../xml/students.xml" var="studXML"/>
   <c:import url="../xml/studentsXSL.xsl" var="studXSL"/>
   <x:transform doc="${studXML}" xslt="${studXSL}"/>
  </body></html>
</jsp:root>
```

 # 29 # правило трансформации # studentsXSL.xsl

```
< xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="/">
    <html><body>
          <table border="0" bgcolor="white">
                   <tr>
                             <td>Name</td>
                             <td>City</td>
                   </tr>
          <xsl:for-each select="students/student">
                    <tr>
                             <td>
                                       <xsl:value-of select="name" />
                             </td>
```

```
                        <td>
                                <xsl:value-of select="address/city"/>
                        </td>
                    </tr>
                </xsl:for-each>
                </table>
            </body></html>
            </xsl:template>
</xsl:stylesheet>
```

Результатом будет вывод таблицы в браузер (см. рис. 18.14.).

Рис. 18.14. *XSL трансформация XML-документа*

JSTL functions

Теги из этой библиотеки выполняют роль, подобную смыслу библиотеки **fmt**, только не для чисел и дат, а для строк. Теги библиотеки используют префикс **fn** и во многом копируют известные методы класса **String** и не составляют особой сложности в применении.

Подключение библиотеки осуществляется директивой **taglib**:

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

Список тегов:

${fn:length(*аргумент*)} — подсчитывает число элементов в коллекции или длину строки;

${fn:toUpperCase(String str)}** и **${fn:toLowerCase(String str)} — изменяет регистр строки;

${fn:substring(String str, int from, int to)} — извлекает подстроку;

${fn:substringAfter(String str, String after)}** и **${fn:substringBefore(String str, String before)} — извлекает подстроку до или после указанной во втором аргументе;

${fn:trim(String str)} — обрезает все пробелы по краям строки;

${fn:replace(String str, String str1, String str2)} — заменяет все вхождения строки **str1** на строку **str2**;

${fn:split(String str, String delim)} — разбивает строку на подстроки, используя **delim**, как разделитель;

${fn:join(массив, String delim)} — соединяет массив в строку, вставляя между элементами подстроку **delim**;

${fn:escapeXml(String xmlString)} — сохраняет в обрабатываемой строке теги;

${fn:indexOf(String str, String searchString)} — возвращает индекс первого вхождения строки **searchString**;

<c:if test="${fn:startsWith(String str, String part)}"> — возвращает истину, если строка начинается с подстроки **part**;

<c:if test="${fn:endsWith(String str, String part)}"> — возвращает истину, если строка заканчивается на подстроку **part**;

<c:if test="${fn:contains(String name, String searchString)}"> — возвращает истину, если строка содержит подстроку **searchString**;

<c:if test="${fn:containsIgnoreCase(String name, String searchString))}"> — возвращает истину, если строка содержит подстроку **searchString**, без учета регистра.

Задания к главе 18

Вариант А

1. Реализовать вывод списка, элементами которого являются другие списки вида List<List<Order>>.
2. Реализовать вывод карты, значения которой представлены множествами вида Map<Order, Set<Item>>.
3. Реализовать вывод параметризованного списка вида List<List<T extends Collection>>.
4. Реализовать вывод очереди, элементами которой являются множества вида Queue<Set<Order>>.
5. Реализовать вывод очереди PriorityQueue, элементы представлены в виде Map.
6. Реализовать возможность интернационализации страницы путем выбора языка отображения с помощью кнопок.
7. Реализовать возможность изменения числовых и финансовых данных страницы с учетом страны и языка.
8. Реализовать возможность изменения дат и часовых поясов с учетом региональных установок компьютера.
9. Решить задачи 6-8, используя чтение информации из файлов XML.

Тестовые задания к главе 18

Вопрос 18.1.

Укажите, какая директива используется для подключения библиотеки тегов jstl (1):
1) jstl
2) page
3) taglib
4) include

Вопрос 18.2.

На jsp-странице используется тег <c:out value="{4*5+12}" />Укажите, каким будет результат обработки этого тега (1):
1) 32;
2) 20+12;
3) 4*5+12;
4) {4*5+12};
5) ничего из вышеперечисленного.

Вопрос 18.3.

Укажите библиотеку jstl, применение тегов которой позволяет использовать на jsp-страницы файлы свойств (*.properties) (1):
1) Core tags;
2) Formatting tags;
3) SQL tags;
4) XML tags;
5) JSTL Functions.

Вопрос 18.4.

Укажите слово-оператор, применение которого в EL-выражении приведет к необходимости определения значения типа «строго более чем» (1):
1) eq
2) ne
3) lt
4) gt
5) le
6) ge

Вопрос 18.5.

Дан java-оператор: obj.getArrayValue(0)+obj.getArrayValue(1);, где getArrayValue — метод, возвращающий i-й элемент инкапсулированного массива. Укажите правильное EL-выражение, соответствующее этому оператору (1):

1) ${obj.getArrayValue[0]+obj. getArrayValue[1]}
2) ${obj.arrayValue[0]+obj.arrayValue[1]}
3) ${obj.arrayValue(0)+obj.arrayValue(1)}
4) ${obj. getArrayValue(0)+obj. getArrayValue(1)}

Глава 19

ПОЛЬЗОВАТЕЛЬСКИЕ ТЕГИ

Начиная с версии JSP 1.1 у разработчиков появилась возможность определения собственных тегов. Это значительно упростило жизнь веб-дизайнерам, которым привычнее использовать теги, а не код на языке Java. Если один и тот же блок кода используется на разных страницах, то он — явный кандидат для переноса кода в пользовательский тег. Фактически последний представляет собой перенос java-кода из страницы JSP во вполне обычный java-класс, что можно считать продолжением идеи о необходимости отделения логики от представления: страница JSP должна содержать как можно меньше логики.

Для создания пользовательских тегов необходимо определить класс обработчика тега, определяющий его поведение, а также дескрипторный файл библиотеки тегов Tag Library Descriptor (файл **.tld**), в которой описываются один или несколько тегов, устанавливающих соответствия между именами XML-элементов и реализацией тегов.

При определении нового тега создается класс, который должен реализовывать интерфейс **javax.servlet.jsp.tagext.Tag**. Обычно создается класс, который наследует один из классов **TagSupport** или **BodyTagSupport** (подкласс класса **TagSupport**). Класс **BodyTagSupport** обладает большими возможностями по обработке тела тега по сравнению с классом **TagSupport**. 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.

Первый тег

Самым простым тегом представляется тег, не имеющий ни атрибутов, ни тела, то есть не получающий никакой информации непосредственно из страницы. Для создания тега без атрибутов или тела достаточно переопределить метод **doStartTag()**, определяющий код, который вызывается во время запроса, если обнаруживается начальный элемент тега. Если в определении тега отсутствует тело, метод **doStartTag()** должен возвратить константу **SKIP_BODY**, дающую указание системе игнорировать любое содержимое между начальными и конечными элементами создаваемого тега. Значение **EVAL_BODY_INCLUDE** следует использовать при необходимости включения в поток вывода тела тега.

Метод **int doAfterBody()** — вызывается после обработки тела. Если вернуть в нем константу **EVAL_BODY_AGAIN**, то метод будет вызван еще раз после вывода в поток тела тега. Возвращение **SKIP_BODY** приведет к вызову метода **doEndTag()**.

Метод **int doEndTag()** вызывается один раз, когда отработаны все остальные методы. По умолчанию возвращает значение **EVAL_PAGE**, разрешающее дальнейшую отработку страницы. Значение **SKIP_PAGE** позволит прекратить дальнейшую обработку страницы.

Класс **TagSupport** содержит поля **id** и **pageContext**. Первый определяет уникальный идентификатор экземпляра тега. Второй представляет экземпляр **pageContext** класса **PageContext**, обладающий доступом ко всей области имен, ассоциированной с контекстом страницы. Как известно, с помощью методов класса **PageContext** можно получить доступ к:

ServletRequest getRequest() — экземпляру запроса;
ServletResponce getResponse() — экземпляру ответа;
ServletContext getServletContext() — экземпляру контекста сервлета;
ServletConfig getServletConfig() — экземпляру конфигурации сервлета;
HttpSession getSession() — экземпляру сессии пользователя;
JspWriter getOut() — потоку вывода;
ErrorData getErrorData() — экземпляру с информацией об ошибках.

Тег получает доступ практически ко всему содержимому контейнера.

Кроме этого возможно:

— с помощью метода **forward(String relativeUrlPath)** сделать перенаправление на другую страницу или сервлет;
— с помощью метода **include(String relativeUrlPath)** включить в поток выполнения текущие ресурсы **ServletRequest** или **ServletResponse**, определяемые относительным адресом.

В качестве примера можно привести следующий класс тега, с помощью которого клиенту отправляется информация о текущем времени и локали.

/ # 1 # простейший тег без тела и атрибутов # InfoTimeTag.java */*

```
package by.bsu.tag.custom;
import java.io.IOException;
import java.util.GregorianCalendar;
import java.util.Locale;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
@SuppressWarnings("serial")
public class InfoTimeTag extends TagSupport {
          @Override
          public int doStartTag() throws JspException {
                    GregorianCalendar gc = new GregorianCalendar();
                    String time = "<hr/>Time : <b> " + gc.getTime() + " </b><hr/>";
                    String locale = "Locale : <b> " + Locale.getDefault() + " </b><hr/> ";
```

```
                                        try {
                                                JspWriter out = pageContext.getOut();
                                                out.write(time + locale);
                                        } catch (IOException e) {
                                                throw new JspException(e.getMessage());
                                        }
                        return SKIP_BODY;
                }
                @Override
                public int doEndTag() throws JspException {
                        return EVAL_PAGE;
                }
}
```

Чтобы сгенерировать вывод, следует использовать метод **write()** класса **JspWriter**, который выводит на страницу содержимое параметра. Метод **getOut()** этого класса возвращает ссылку на поток **JspWriter**, с помощью которой осуществляется вывод.

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 библиотеки пользовательских тегов должен иметь 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, обрабатывающий тег. Для однозначной идентификации используется полное имя класса, например: **by.bsu.tag.custom.InfoTimeTag**. 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. Идентификатор представляет собой имя, которое используется для доступа к библиотеке из JSP в директиве **taglib**, а также для ее регистрации в **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 **short-name**;

— **tag-class** — полное имя класса-обработчика тега;
— **info** — краткое описание тега;
— **body-content** — тип тела тега: **empty** — пустое тело; **scriptless** — тело состоит из всего того, что может находиться в JSP-файле, за исключением скриптлетов; **tagdependent** — тело интерпретируется классом, реализующим данный тег и передается ему без изменений. Как правило, это текст, написанный на другом языке: встроенный оператор SQL, регулярное выражение и проч.

Вся эта информация помещается в файл **custom.tld**, который для JSP версии 2.1 имеет вид:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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">
          <tlib-version>1.0</tlib-version>
          <short-name>ctg</short-name>
          <uri>customtags</uri>
          <tag>
                    <name>info-time</name>
                    <tag-class>by.bsu.tag.custom.InfoTimeTag</tag-class>
                    <body-content>empty</body-content>
          </tag>
</taglib>
```

Поместить tld-файл в папку **/WEB-INF/tld** проекта. Зарегистрировать или использовать библиотеку пользовательских тегов **custom.tld** для приложения можно тремя способами:

1) зарегистрировать библиотеку в файле **web.xml**, для чего следует создать тег **<jsp-config>** после тега **<welcome-file-list>**:

```
<jsp-config>
<taglib>
          <taglib-uri>customtags</taglib-uri>
          <taglib-location>/WEB-INF/tld/custom.tld</taglib-location>
          </taglib>
</jsp-config>
```

Тогда в странице JSP подключение библиотеки должно выглядеть

```
<%@ taglib prefix="ctg" uri="customtags" %>,
```

и тег **<uri>** в tld-файле содержит идентификатор **customtags**.

2) если библиотека не зарегистрирована в **web.xml**, то для доступа из jsp просто указывается прямой путь к tld-файлу

```
<%@ taglib prefix="ctg" uri="/WEB-INF/tld/custom.tld" %>,
```

что может быть использовано при его размещении не только на сервере приложений, но и по конкретному адресу в интернете.

3) чаще всего библиотеки упаковывают в архивный файл и размещают вместе с другими библиотеками. Тогда доступ обеспечивает

```
<%@ taglib prefix="ctg" uri="/WEB-INF/lib/custom.jar" %>
```

Непосредственное использование в странице простейшего тега может выглядеть следующим образом:

 # 2 # вызов простого тега # info.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="ctg" uri="customtags" %>
<html><head>Time & Locale info</head>
        <body>
                <ctg:info-time/>
        </body>
</html>
```

В результате выполнения тега браузер получит следующую информацию:

Рис. 19.1. *Тег без атрибутов и тела*

Тег с атрибутами

Тег может содержать атрибуты и передавать их значения для обработки в соответствующий ему класс. Для этого при описании тега в файле TLD используются элементы, которые должны объявляться внутри элемента **tag** с помощью элемента **attribute**. Внутри элемента **attribute** между тегами **<attribute>** и **</attribute>** могут находиться следующие элементы:

— **name** — (обязательный элемент) имя атрибута;
— **required** — (обязательный элемент) значение **true** указывает на обязательность наличия данного атрибута при использовании тега. При значении **false** описываемый атрибут может отсутствовать;
— **rtexprvalue** — (необязательный элемент) показывает, может ли значение атрибута быть JSP-выражением вида **${***expression***}** (значение **true**) или оно

должно задаваться строкой данных (значение **false**). По умолчанию устанавливается **false**, поэтому этот элемент обычно опускается, если не требуется задавать значения атрибутов во время запроса;

— **type** — (необязательный элемент) задает тип атрибута во время выполнения. По умолчанию **java.lang.String**.

Соответственно для каждого из атрибутов тега класс, его реализующий, должен содержать, например, поле **value** и setter-метод **setValue(),** если атрибут именован как **value**.

В следующем примере приведен простейший тег с атрибутом **role**, который выводит пользователю сообщение:

// # 3 # тег с атрибутом # HelloTag.java

```
package by.bsu.tag.custom;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
import java.io.IOException;
@SuppressWarnings("serial")
public class HelloTag extends TagSupport {
     private String role;
     public void setRole(String role) {
        this.role = role;
     }
    @Override
     public int doStartTag() throws JspException {
        try {
          String to = null;
          if ("administrator".equalsIgnoreCase(role)) {
                    to = "Hello, " + role;
          } else {
                    to = "Welcome, " + role;
          }
          pageContext.getOut().write("<hr/>" + to + "<hr/>");
        } catch (IOException e) {
                    throw new JspException(e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

В файл **custom.tld** должна быть помещена следующая информация о теге:

```
<tag>
        <name>hello</name>
        <tag-class>by.bsu.tag.custom.HelloTag</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <name>role</name>
```

```
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
</tag>
```

Использовать созданный тег в файле **hellofirst.jsp** можно следующим образом:

 # 4 # вызов тега с передачей ему значения # hellofirst.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="ctg" uri="customtags"%><html>
<head><title>tag: Hello</title></head>
<body>
          <ctg:hello role="${ user }"/>
</body></html>
```

Где атрибут **user** добавлен к запросу командой:

```
request.setAttribute("user", "Administrator");
```

тогда в браузер будет выведено:

Рис. 19.2. *Тег с атрибутом*

Тег с телом

Как и в обычных тегах, между открывающим и закрывающим пользовательскими тегами может находиться тело тега. При описании такого тега в tld-файле элемент **body-content** должен принимать значение **scriptless**.

Ниже приводится класс обработки тега, который получает значения атрибута **rows** (в данном случае методом установки значения для атрибута **rows** будет метод **setRows(Integer rows)** и формирует таблицу с указанным количеством строк, в поля таблицы заносятся значения из тела тега:

// # 5 # тег с телом # RevenueTableTag.java

```
package by.bsu.tag.custom;
import java.io.IOException;
import java.util.Locale;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
@SuppressWarnings("serial")
public class RevenueTableTag extends TagSupport {
          private String head;
          private int rows;
          public void setHead (String head) {
                    this. head = head;
          }
          public void setRows(Integer rows) {
                    this.rows = rows;
          }
          @Override
          public int doStartTag() throws JspTagException {
                    try {
                              JspWriter out = pageContext.getOut();
                              out.write("<table border='1'><colgroup span='2' title='title' />");
                              out.write("<caption>"+ Locale.getDefault().getDisplayCountry()
                                                                        +"</caption>");
                              out.write("<thead><tr><th scope='col'>"+head+"</th></tr></thead>");
                              out.write("<tbody><tr><td>");
                    } catch (IOException e) {
                              throw new JspTagException(e.getMessage());
                    }
                    return EVAL_BODY_INCLUDE;
          }
          @Override
          public int doAfterBody() throws JspTagException {
                    if (rows-- > 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;
                    }
          }
          @Override
          public int doEndTag() throws JspTagException {
                    try {
```

```
                    pageContext.getOut().write("</td></tr></tbody></table>");
                } catch (IOException e) {
                    throw new JspTagException(e.getMessage());
                }
                return EVAL_PAGE;
        }
}
```

В файл **custom.tld** следует вставить информацию о теге в виде:

```
<tag>
        <name>table-revenue</name>
        <tag-class>by.bsu.tag.custom.RevenueTableTag</tag-class>
        <body-content>scriptless</body-content>
        <attribute>
                <name>rows</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
                </attribute>
        <attribute>
                <name>head</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
                </attribute>
</tag>
```

При использовании в файле JSP тег **table-revenue** может вызываться с атрибутами и без них:

 # 6 # применение тега с телом # table.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="ctg" uri="customtags"%>
<html><head><title>tag: custom table</title></head>
<body>
        <ctg:table-revenue rows="${ rw.size }" head="Revenue">
                ${ rw.revenue }
        </ctg:table-revenue >
<br/>
        <ctg:table-revenue>5 rub BulbaComp</ctg:table-revenue >
</body></html>
```

К запросу добавлен объект, который и предоставит информацию для заполнения таблицы:

```
        VendorMap map = new VendorMap();
        request.setAttribute("rw", map);
```

В результате вызова этой JSP из сервлета клиенту будет возвращено (см. рис. 19.3.).

Класс, предоставляющий информацию выглядит следующим образом:

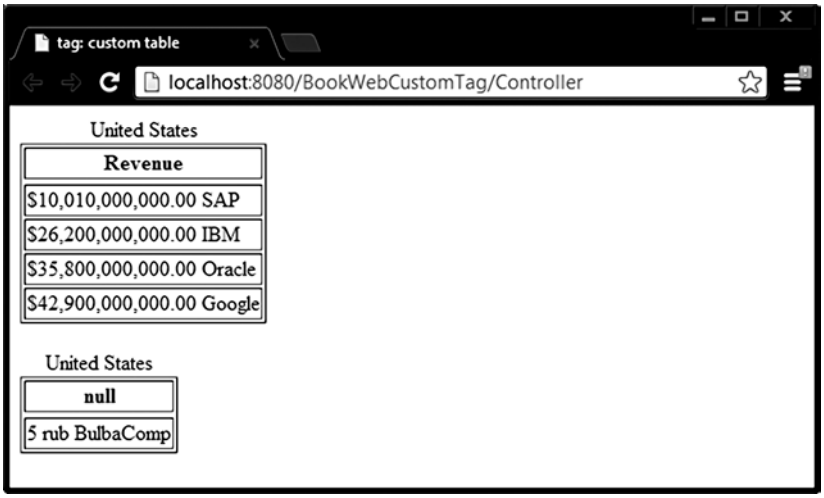

Рис. 19.3. *Выполнение тега с телом*

// # 7 # инфо-класс # VendorMap.java

```
package by.bsu.tag.entity;
import java.text.NumberFormat;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
public class VendorMap {
          private HashMap<String, Double> map = new HashMap<String, Double>();
          {
                    this.map.put("IBM", 26.2E9);
                    this.map.put("Oracle", 35.8E9);
                    this.map.put("SAP", 10.01E9);
                    this.map.put("Google", 42.9E9);
          }
          private Iterator<Map.Entry<String, Double>> it = map.entrySet().iterator();
          public int getSize() {
                    return map.size();
          }
          public String getRevenue() {
                    NumberFormat nf = NumberFormat.getCurrencyInstance();
                    if (it.hasNext()) {
                              Map.Entry<String, Double> m = it.next();
                              return nf.format(m.getValue()) + "  " + m.getKey();
                    } else {
                              return null;
                    }
          }
}
```

Обработка тела тега

При необходимости доступа к информации, содержащейся в теле тега, перед выводом его в поток **JspWriter** при создании тега используется наследование от класса **BodyTagSupport**, реализующего в свою очередь интерфейс **BodyTag**. Кроме методов класса **TagSupport** (суперкласс для **BodyTagSupport**), класса **BodyTagSupport** реализует метод **doInitBody()**, который вызывается один раз перед первой обработкой тела, после вызова метода **doStartTag()**.

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.

Тело тега (информация между открывающим и закрывающим тегами) помещается в экземпляр класса **BodyContent**.

/ # 8 # тег с обработкой тела # MailTag.java */*

```
package by.bsu.tag.custom;
import java.io.IOException;
import java.util.regex.Pattern;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;
@SuppressWarnings("serial")
public class MailTag extends BodyTagSupport {
          private static final String MAIL_PATTERN = "(\\w{6,})@(\\w+\\.)(\\w{2,4})";
          @Override
          public int doAfterBody() throws JspException {
                    BodyContent content = this.getBodyContent();
                    String body = content.getString();
                    String res = null;
                    if (Pattern.matches(MAIL_PATTERN, body)) {
                              res = body.replaceAll("\\.", "(dot)");
                              res = res.replaceFirst("@", "(a)");
                    } else {
                              res = body + " is invalid e-mail";
                    }
                    JspWriter out = content.getEnclosingWriter();
                    try {
                              out.write(res);
                    } catch (IOException e) {
                              throw new JspTagException(e.getMessage());
                    }
                    return SKIP_BODY;
          }
}
```

В файл **custom.tld** следует вставить информацию о теге в виде:

```
<tag>
        <name>safe-mail</name>
        <tag-class>by.bsu.custom.tags.MailTag</tag-class>
        <body-content>scriptless</body-content>
</tag>
```

Для корректной работы в тег **ctg:safe-mail** должна быть передана корректная информация с почтовым адресом:

 # 9 # применение тега с обработкой тела # mail_transform.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"   pageEncoding="UTF-8"%>
<%@ taglib prefix="ctg" uri="customtags"%>
<html>
<head><title>tag: safe mail </title></head>
<body>
        <ctg:safe-mail>blinov@gmail.com</ctg:safe-mail>
<br/>
        <ctg:safe-mail>blinov@gmail..com</ctg:safe-mail>
</body>
</html>
```

В результате запуска этой JSP клиенту будет возвращено:

Рис. 19.4. *Обработка тела*

Функции-теги

Для решения небольших задач в рамках страницы применяются функции-теги. Валидацию, форматирование, преобразование можно выполнять не с помощью классов-тегов, а с привлечением функций-тегов. Однако java-класс создавать необходимо.

Пусть требования к отображению информации на странице таковы, что некоторые значения, передаваемые на страницу, обязательно должны быть не нулевыми или не пустыми. Для решения проблемы можно прибегнуть к функции-тегу.

Метод, отвечающий за функцию-тег, должен быть статическим и всегда иметь возвращаемое значение.

/ # 10 # класс-описание тега функции # AddFunction.java */*

```
package by.bsu.tag.custom;
public class NotNullFunction {
          public static String notNull(Object ob) {
                    String res = null;
                    if (ob == null || ob.toString().isEmpty()) {
                              res = "Attribute or Parameter is null or empty";
                    } else {
                              res = ob.toString();
                    }
                    return res;
          }
}
```

В файл **custom.tld** следует вставить информацию о функции-теге в виде:

```
<function>
          <name>notnull</name>
<function-class>by.bsu.tag.custom.NotNullFunction</function-class>
          <function-signature>String notNull(java.lang.Object)</function-signature>
</function>
```

где элемент **function-class** содержит путь к классу-описанию, а элемент **function-signature** описывает сигнатуру функции с параметрами.

Применение тега

 # 11 # прямое применение функции-тега # function.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"    pageEncoding="UTF-8"%>
 <%@ taglib prefix="ctg" uri="customtags" %>
<html>
<head><title>tag: function</title></head>
<body>
          Не пустой параметр: ${ctg:notnull("Oracle Java")}
<br/>
          Строка нулевой длины: ${ctg:notnull("")}
</body></html>
```

В действительности нет необходимости в прямой передаче строки или объекта в подобную функцию. В качестве аргумента в нее передаются значения атрибутов или параметров. Пусть атрибут передается с экземпляром **request** на страницу в виде:

```
request.setAttribute("user", "Administrator");
request.getRequestDispatcher("/nullfunction.jsp").forward(request, response);
```

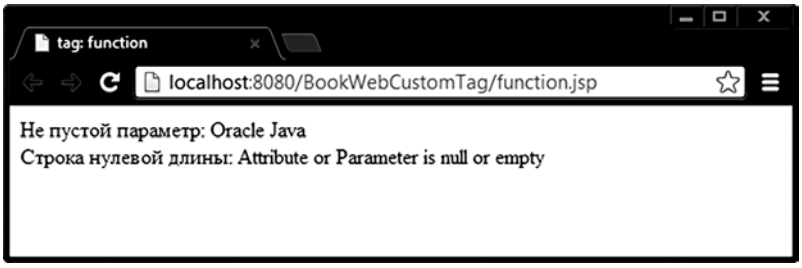

Рис. 19.5. *Функция-тег*

Тег-функция извлекает из запроса значение атрибута и выполняет свою функциональность, используя обращение к атрибуту в виде:

```
${ctg:notnull(user)}
```

что позволяет явно определить и отобразить на странице — пустое или не пустое значение передано. Если же использовать стандартную запись **${user}** при пустом значении атрибута или его отсутствии, то браузер не отобразит ничего.

 # 12 # прямое применение функции-тега # nullfunction.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"    pageEncoding="UTF-8"%>
 <%@ taglib prefix="ctg" uri="customtags" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>tag: null function</title>
</head>
<body>
          User: ${ctg:notnull(user)}
</body>
</html>
```

Элементы action для тегов

Элемент **jsp:attribute** позволяет определить значение атрибута тега в теле XML-элемента, а не через значение атрибута стандартного или пользовательского тега. В случае обычного строкового значения достаточно записать:

```
<ctg:hello role="Bender" />
```

Но если значение атрибута будет таким: **ХК "Динамо"(Рига)
, то в атрибут это значение поместить невозможно, так как атрибут не может содержать теги, да и с двойными кавычками также будут проблемы. Тег **jsp:attribute решает проблему вынесением имени атрибута **role** со сложным

содержимым в значение атрибута **name**, при этом содержимое атрибута становится телом тега.

```
<ctg:hello>
          <jsp:attribute name="role">
                    <b>ХК "Динамо"(Рига)</b><br/>
          </jsp:attribute>
</ctg:hello>
```

Если тег содержит тело, но для него необходимо определять элементы **jsp:attribute**, то само тело тега также нужно указать явно при помощи стандартного действия **jsp:body**:

```
<ctg:table-revenue>
          <jsp:attribute name="rows">
                    ${requestScope.rw.size}
          </jsp:attribute>
          <jsp:body>
                    ${requestScope.rw.revenue}"
          </jsp:body>
</ctg:table-revenue>
```

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="Z2" >*
          <jsp:attribute name="Style">
                    color:red
          </jsp:attribute>
          <jsp:body>
                    Simple Text
          </jsp:body>
</jsp:element>
```

в результате должно быть сгенерировано:

<Z2 Style="color:red">Simple Text</Z2>

Стандартные действия **jsp:doBody** и **jsp:invoke** используются только в специальных файлах с расширением **.tag**. 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.

Задания к главе 19

Вариант А

Создать классы пользовательских тегов, формирующих нужное количество элементов (строк, ячеек и др.) для размещения результатов выполнения запроса.

1. Элемент массива называют локальным максимумом, если у него нет соседа большего, чем он сам. Аналогично определяется локальный минимум. Определить количество локальных максимумов и локальных минимумов в заданном строкой массиве чисел. Массив задает клиент. Возвратить все максимумы и минимумы пользователю.
2. В неубывающей последовательности, заданной клиентом, найти количество различных элементов и количество элементов, меньших, чем заданное число, и вернуть ему результат.
3. Дана числовая последовательность a_1, a_2,…, a_n. Вычислить суммы вида $S_i = a_i + a_{i+1} + \ldots + a_j$ 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 оценки 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, 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 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 БД 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 БД хранятся координаты вершин выпуклых четырехугольников на плоскости. Сформировать ответ клиенту, содержащий координаты всех вершин трапеций, которые можно сформировать из данных точек.
14. В БД 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 абсцисс.
15. В БД хранятся координаты множества точек плоскости 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 прямых.

Вариант В

Для заданий варианта В предыдущей главы применить пользовательские теги для визуализации работы приложения.

Тестовые задания к главе 19

Вопрос 19.1.

Что должен возвращать метод doStartTag(), если в теге отсутствует тело (1)?

1) константу SKIP_BODY
2) исключение SkipBodyException
3) константу EMPTY_BODY
4) исключение EmptyBodyException
5) ничего из вышеперечисленного

Вопрос 19.2.

Какой из методов класса BodyTagSupport может быть вызван несколько раз (1)?

1) doInitBody()
2) doStartTag()
3) doAfterBody()
4) doEndTag()

Вопрос 19.3.

В файле web.xml зарегистрирована пользовательская библиотека тегов:

```
<taglib>
        <taglib-uri>/WEB-INF/tld/mytaglib.tld</taglib-uri>
        <taglib-location>/WEB-INF/tld/mytaglib.tld</taglib-location>
</taglib>
```

Укажите правильный вариант написания директивы, позволяющей использовать пользовательскую библиотеку тегов на jsp-странице (1):

1) <%@ taglib uri="*/tld/mytaglib.tld*" prefix="*mytag*"%>
2) <%@ taglib uri="*/WEB-INF/tld/mytaglib.tld*" prefix="*mytag*"%>
3) <%@ taglib uri="*/WEB-INF/tld/mytaglib*" prefix="*mytag*"%>
4) <%@ taglib uri="*/tld/mytaglib*" prefix="*mytag*"%>

Вопрос 19.4.

Какие значения может принимать тег <body-content> при описании пользовательского тега в tld-файле?

1) empty
2) html
3) jsp
4) no_jsp
5) tagdependent
6) choice

Вопрос 19.5.

Укажите правильный вариант подключения tld-файла при создании jsp-документа:

1) <jsp:root xmlns:jsp="*http://java.sun.com/JSP/Page*" xmlns:mytag="*/WEB-INF/tld/mytaglib.tld*" version="*1.2*">
2) <jsp:taglib xmlns:jsp="*http://java.sun.com/JSP/Page*" xmlns:mytag="*/WEB-INF/tld/mytaglib.tld*" version="*1.2*">
3) <jsp:root xmlns:jsp="*http://java.sun.com/JSP/Page*" xmlns:uri="*/WEB-INF/tld/mytaglib.tld*" xmlms:prefix="*mytag*" version="*1.2*">
4) <jsp:taglib xmlns:jsp="*http://java.sun.com/JSP/Page*" xmlns:uri="*/WEB-INF/tld/mytaglib.tld*" xmlms:prefix="*mytag*" version="*1.2*">
5) нет правильного ответа

Часть 4

Шаблоны проектирования

В четвертой части даны базовые понятия архитектуры взаимодействия классов на основе дизайн паттернов GRASP и GoF.

Глава 20

ШАБЛОНЫ И АНТИШАБЛОНЫ

В коде, написанном мной месяц назад, всегда хочется что-то чуть-чуть поправить. В коде полугодичной давности хочется поменять очень многое, а код, написанный два-три года назад, превращает меня в эмо: хочется заплакать и умереть.

Цитата из статьи на Habrahabr

В задачах проектирования информационных систем, классов, их составляющих, при распределении обязанностей и способов взаимодействия объектов этих классов перед программистом возникает серьезная проблема. 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 (паттерна).

Разработано огромное количество шаблонов, как вариаций первопроходцев от GoF, так и совершенно самостоятельных решений. В этой части книги будут рассмотрены шаблоны GoF и GRASP, а также антишаблоны.

GoF и GRASP шаблоны представляют собой две различные точки зрения на организацию классов. GRASP представляют обобщенный взгляд на организацию самих классов и их взаимодействия вне зависимости от целевой задачи, решаемой этими классами. GoF представляют собой рецепты решения конкретных, и при этом достаточно узких, проблем. Тем не менее, если их использовать совместно, то качество и удобочитаемость кода повысятся.

Шаблоны очень хорошо выражают преимущества объектно-ориентированного подхода, в частности, знаменитых «трех китов»: наследования, полиморфизма и инкапсуляции.

При решении задач программирования с применением шаблонов следует придерживаться следующих рекомендаций:

— изучение шаблонов приносит пользу вне зависимости от того, как часто они используются при программировании;
— не следует торопиться с применением шаблонов при решении новой задачи;
— шаблон в чистом (классическом, из учебника) виде, как правило, неприменим, применимы только вариации;
— при применении шаблона следует начинать с его простейшей реализации, а уж затем вносить изменения по адаптации к конкретной ситуации;
— если после применения шаблона код стал хуже, то шаблон лучше убрать.

Шаблоны GRASP

Наиболее общие принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах GRASP (General Responsibility Assignment Software Patterns). Ниже будут сформулированы некоторые базовые способы и некоторые стандартные решения, придерживаясь которых можно создавать хорошо структурированный и понятный код.

Прежде чем рассмотреть шаблоны GRASP, следует привести список основных шаблонов, составляющих базовые принципы ООП как парадигмы.

Шаблон Expert

Все классы делятся на две большие группы: классы-носители информации, классы, производящие действия. Классов-носителей значительно меньше, но они играют роли отображения сущностей реального мира в системе анимирования «неживых» сущностей и, как правило, должны быть представителями Java Beans.

Классы-носители информации, в общем, не должны выполнять действий по манипулированию значениями своих полей, не считая установки, извлечения и поддержки функциональности по контракту, навязанной классом **Object**.

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 функциональности программной системы.

Простые эксперты определять достаточно просто, например **Student**, **Item**, **Order** и т. д. Однако часто возникает потребность в процессе разработки 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?

/ # 1 # шаблон Expert # Test.java # Quest.java */*

```
package by.bsu.expert;
          // информационный эксперт
public class Test {
          private int testId;
          private String testName;
          private int questNumber;
          private long time;
          // реализация конструкторов и методов
```

```
}
package by.bsu.expert;
          // информационный эксперт
public class Quest {
          private int questId;
          private int testId;
          // реализация конструкторов и методов
}
```

Количество вопросов из теста, на которые дан ответ, есть число созданных объектов класса **Quest**. Такую информацию можно поместить в описание класса **Test**, но в данной ситуации это приведет к загромождению класса. Итак, класс **Test** ответствен за общие характеристики теста. Класс **CurrentStateTest** будет владеть информацией о текущем состоянии теста.

/ # 2 # шаблон Expert # CurrentStateTest.java */*

```
package by.bsu.expert;
public class CurrentStateTest {
          private int testId;
          private int studentId;
          private int currentQuestId;
          private long timeRemain;
          private Queue<Long> listQuestId;
          // конструкторы и методы
}
```

Класс **CurrentStateTest** достаточно серьезно отличается от класса **Test** и откровенно просто воспринимается с первого взгляда.

Этот класс может быть объявлен как внутренний класс класса **Test** и при небольших изменениях соответствовать шаблону State из группы шаблонов GoF.

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 или внутреннего класса 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. Простым и надежным решением будет создание новых информационных экспертов, как и было сделано в приведенном выше примере.

Шаблон Creator

Существует большая вероятность того, что класс станет 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** содержит или получает данные инициализации (has the initializing data), которые будут передаваться объектам класса **A** при его создании;
— класс **B** записывает или активно использует (records or closely uses) экземпляры объектов **A**;
— класс **B** агрегирует (aggregate) объекты **A**;
— класс **B** содержит (contains) объекты **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.

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**, например, агрегирующий экземпляры объектов **Quest**.

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

/ # 3 # шаблон Creator # RequestQuestById.java */*

```
package by.bsu.expert;
public class BuildTest {
          // поля, методы
          public void buildTest(Queue<Quest> q) {
                    q.add(makeQuest(параметры));
                    // реализация
          }
          private Quest makeQuest(параметры) {
                    // реализация
                    return new Quest(параметры);
          }
}
```

Шаблон Creator способствует низкой зависимости между классами (Low Coupling), так как экземпляры класса, которым необходимо содержать ссылку на некоторые объекты, должны создавать эти объекты. Если класс создает некоторый объект самостоятельно, то тем самым он перестает быть зависимым от класса, отвечающего за создание объектов для него.

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.

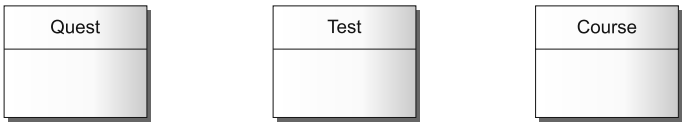

Рис. 20.1. *Классы, которые необходимо связать*

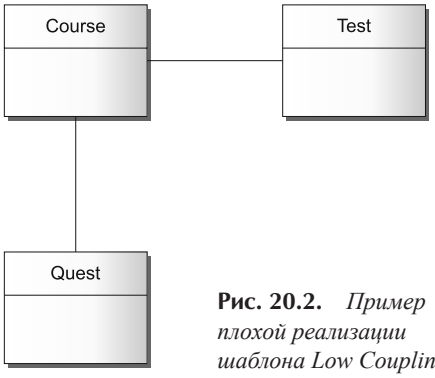

Рис. 20.2. *Пример плохой реализации шаблона Low Coupling*

Учебный курс содержит тесты, которые состоят из вопросов. Пусть требуется создать экземпляр класса **Quest** и добавить его в экземпляр класса **Test**. В предметной области регистрация объекта **Test** выполняется объектом **Course**.

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.

/ # 4 # шаблон Low Coupling # Test.java # Course.java */*

```
package by.bsu.lowcoupling;
public class Course {
          // поля, методы
          public void makeTest(int id) {
                    Test test = new Test(параметры);
                    // реализация
                       while(условие) {
                          Quest quest = new Quest(параметры);
                          // реализация
                          test.addQuest(quest);
                    }
          }
}
package by.bsu.lowcoupling;
public class Test {
          // поля , методы
          public void addQuest(Quest quest){
                    // реализация
          }
}
```

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

Второй вариант распределения обязанностей с устранением класса **Course** от создания объектов вопросов представлен на рисунке 20.3.

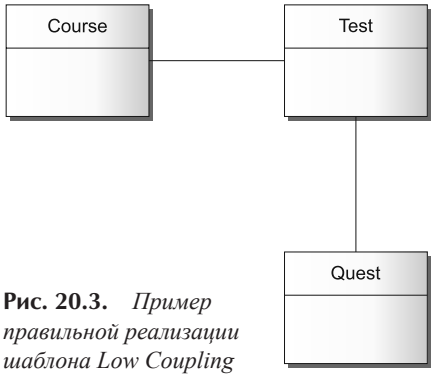

Рис. 20.3. *Пример правильной реализации шаблона Low Coupling*

```
/* # 5 # шаблон Low Coupling # Test.java # Course.java */
package by.bsu.lowcoupling;
public class Course {
              // поля
          public void makeTest() {
                    Test test = new Test(параметры);
                    // реализация
                    test.createTest();
                    // реализация
              }
          }
package by.bsu.lowcoupling;
public class Test {
          // поля
          public void createTest() {
          // реализация
                    while(условие) {
                              Quest quest = new Quest(параметры);
                              // реализация
                    }
              }
          }
```

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 специализированности класса на своих обязанностях. При высоком зацеплении обязанности класса тесно связаны между собой, и класс не выполняет работ непомерных объемов. 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;

/ # 6 # очень слабое зацепление # Initializer.java */*

```
package by.bsu.cohesion;
public class Initializer {
          public void createTCPServer(String port) {
                    // реализация
          }
          public int connectToDB (URL url) {
                    // реализация
          }
```

```
        public void initXMLDocument(String fileName) {
                // реализация
        }
}
```

2) *слабое зацепление*. Единоличное выполнение сложной задачи из одной функциональной области;

/ # 7 # слабое зацепление # NetLogicCreator.java */*

```
package by.bsu.cohesion;
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;

/ # 8 # среднее зацепление # TCPServerHelper.java */*

```
package by.bsu.cohesion;
public class TCPServerHelper {
        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.

```
/* # 9 # высокое зацепление # TCPServerCreator.java # DataTransmission.java #
CodingData.java */

package by.bsu.cohesion;
public class TCPServerCreator {
          public void createTCPServer() {
                    // реализация
          }
}
package by.bsu.cohesion;
public class DataTransmission {
          public void receiveData() {
                    // реализация
          }
          public void sendData() {
                    // реализация
          }
}
package by.bsu.cohesion;
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 легко узнаваемы, 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. Интерфейсов 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.

Антишаблон: Раздутый контроллер (н-р, 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.

Антишаблоны

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» — 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 длинны и сложны. 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 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 всплыть 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 — простоту, 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 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.

Глава 21

ПОРОЖДАЮЩИЕ ШАБЛОНЫ

Шаблоны — это странные и загадочные существа с душевнобольной иерархической системой. Как известно, тушки этих мифических существ часто используют жрецы ООП для своих бессвязных заклинаний.

<Абсурдопедия>

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 возможность 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.

Несмотря на все различия, шаблоны взаимосвязаны, и нередко небольшое изменение ответственности или связи того или иного класса может привести к тому, что применение одного шаблона приводит к замене исходного шаблона на другой, часто даже неродственный.

Порождающие шаблоны предназначаются для организации процесса создания объектов и все до единого соответствуют шаблону Creator из GRASP.

К порождающим шаблонам относятся:

Abstract Factory (Абстрактная Фабрика) — предоставляет интерфейс для создания связанных между собой объектов семейств классов без указания их конкретных реализаций (families of product objects);

Factory Method (Фабричный метод) — определяет интерфейс для создания объектов из иерархического семейства классов на основе передаваемых данных (subclass of object that is instantiated);

Builder (Строитель) — создает объект конкретного класса различными способами (how a composite object gets created);

Singleton (Одиночка) — гарантирует существование только одного или конечного числа экземпляров класса (the sole instance of a class);

Prototype (Прототип) — применяется при создании сложных объектов. На основе прототипа объекты сохраняются и воссоздаются, например, путем копирования (class of object that is instantiated).

Шаблон Factory Method

Необходимо определить механизм создания объектов по заданному признаку для классов, находящихся в иерархической структуре. Классу-фабрике и его подклассам разрешается создавать экземпляры с помощью общего для всех интерфейса, причем именно подклассы решают, какой экземпляр следует создавать. Этот интерфейс может содержаться в абстрактном классе, интерфейсе или даже в обычном классе. Фабричный метод не должен владеть информацией о конкретном типе создаваемого объекта, но знает о классе-вершине иерархии создаваемых объектов. Фабричный метод (**Factory Method**) — основа для функционирования абстрактной фабрики (**Abstract Factory**).

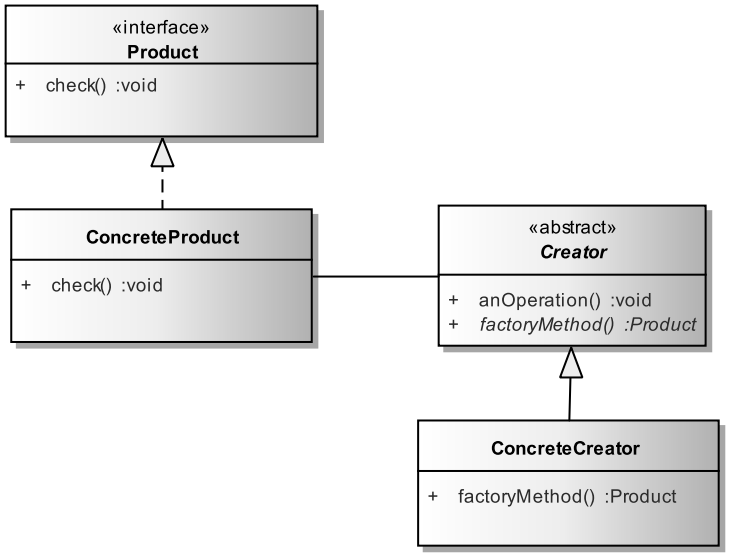

Рис. 21.1. *Базовая реализация шаблона Factory Method*

```
/* # 1 # базовая реализация FactoryMethod # Product.java # ConcreteProduct.java #
 Creator.java # ConcreteCreator.java # FactoryMethodRunner.java */
```

```
package by.bsu.factorymethod;
public interface Product {
    void check();
}
package by.bsu.factorymethod;
public class ConcreteProduct implements Product {
          //поля, конструкторы
    public void check() {
        System.out.println("concrete product");
    }
}
package by.bsu.factorymethod;
public abstract class Creator {
    public abstract Product factoryMethod();
    public void anOperation() {
        System.out.println("operation");
    }
}
package by.bsu.factorymethod;
public class ConcreteCreator extends Creator {
    public Product factoryMethod() {
        // подготовительные действия
        this.anOperation();
        return new ConcreteProduct();
    }
}
package by.bsu.factorymethod;
public class FactoryMethodRunner {
    public static void main(String[ ] args) {
          Creator creator = new ConcreteCreator();
          Product product = creator.factoryMethod();
          product.check();
    }
}
```

Программная реализация может быть представлена в общем виде следующим образом.

```
/* # 2 # реализация шаблона с одним статическим методом # AbstractOrder.java #
 SimpleOrder.java # ExtendedOrder.java # TypeOrder.java # OrederFactory.java */
```

```
package by.bsu.factory;
public abstract class AbstractOrder {
          // поля и методы
          public abstract void perform();
}
```

```
package by.bsu.factory;
public class SimpleOrder extends AbstractOrder {
          // поля и методы
          public void perform() {
                    System.out.println("Simple order");
          }
}
package by.bsu.factory;
public class ExtendedOrder extends AbstractOrder {
          // поля и методы
    public void perform() {
                    System.out.println("Extended order");
    }
}
package by.bsu.factory;
public enum TypeOrder {
          SIMPLE, EXTENDED
          }
package by.bsu.factory;
public class OrderFactory { // простейшая реализация шаблона
    public static AbstractOrder getOrderFromFactory(String type) {
          TypeOrder sign = TypeOrder.valueOf(type.toUpperCase());
          switch(sign) {
                    case SIMPLE: return new SimpleOrder();
                    case EXTENDED: return new ExtendedOrder();
                    default : throw new EnumConstantNotPresentException(
                                                                        TypeOrder.class, sign.name());
          }
    }
    public void anOperation() {
          System.out.println("operation");
    }
}
```

В качестве параметра метода передается некоторое значение, в соответствии с которым будет осуществляться инициализация объекта одного из подклассов абстрактного класса.

Приведенная реализация проста для восприятия, но обладает рядом недостатков. При добавлении новых подклассов в иерархию создаваемых объектов потребуется внесение изменений в сам фабричный метод. При разрастании иерархии будет усложняться логика процесса создания объектов, что повлечет разрастание кода фабричного метода с повышением его уровня связанности.

/ # 3 # создание объектов с помощью шаблона # RunFactoryMethodSimplest.java */*

```
package by.bsu.factory;
public class RunFactoryMethodSimplest {
    public static void main(String args[ ]) {
```

```
        AbstractOrder ob1 = OrderFactory.getOrderFromFactory("simple");
        AbstractOrder ob2 = OrderFactory.getOrderFromFactory("extended");
        ob1.perform();
        ob2.perform();
    }
}
```

На практике иерархия классов для создаваемых экземпляров может быть достаточно разветвленной, да и сам процесс создания одного экземпляра может состоять из большого числа действий. Следующим вариантом применения шаблона будет выделение каждому классу создаваемых экземпляров своего персонального метода. Основной класс шаблона представляет собой класс, который имеет различные методы для создания объектов.

/ # 4 # вариант фабрики с различными методами для создания экземпляров, объявленных в одном классе # OrderFactoryVariant.java */*

```
package by.bsu.factory;
public class OrderFactoryVariant { // стандартная реализация шаблона
          // поля, методы
          public SimpleOrder getSimpleOrder() {
                    return new SimpleOrder();
          }
          public ExtendedOrder getExtendedOrder() {
                    return new ExtendedOrder();
          }
          public void anOperation() {
                    System.out.println("operation");
          }
}
```

Процесс создания экземпляров по-прежнему сосредоточен в одном классе, что делает класс подверженным изменениям при расширении иерархии создаваемых объектов.

Если процессу создания экземпляра одного класса выделять собственный класс, находящийся, в свою очередь, в иерархической зависимости от некоего абстрактного создателя **AbstractOrderFactory**. В этой ситуации шаблон **Factory Method** становится весьма близок простой вариации шаблона **Abstract Factory**, особенно если добавить к реализации класс-клиент, ответственный за действия по инициализации фабричного метода и использованию созданного экземпляра. В базовой реализации фактически такие действия не выделяются в отдельном методе или классе, а производятся по необходимости.

Реализация с клиентом и с использованием метода подставки при создании конкретных фабричных методов.

```
/* # 5 # вариант фабрики с классом-клиентом для различных классов создания
экземпляров # AbstractOrderFactory.java # SimpleOrderFactory.java #
ExtendedOrderFactory.java # Client.java */
```

```
package by.bsu.factory.client;
public abstract class AbstractOrderFactory {
          public abstract AbstractOrder getOrder();
          public void anOperation() {
                    System.out.println("operation");
          }
}
package by.bsu.factory.client;
public class SimpleOrderFactory extends AbstractOrderFactory {
          @Override
          public SimpleOrder getOrder() {
                    return new SimpleOrder();
          }
}
package by.bsu.factory.client;
public class ExtendedOrderFactory extends AbstractOrderFactory {
          @Override
          public ExtendedOrder getOrder() {
                    return new ExtendedOrder();
          }
}
package by.bsu.factory.client;
public class Client {
          // поля, методы
          public void someOperation(AbstractOrderFactory factory) {
                    AbstractOrder order = factory.getOrder();
                    order.perform();
          }
}
```

Инициализация фабрики в классе **Client** может производиться не только в методе, но и через соответствующий конструктор или отдельный метод. Таким же образом может создаваться и использоваться экземпляр реализаций класса **AbstractOrder**.

Один из простейших примеров применения данного шаблона уже был рассмотрен в примере главы «Полиморфизм и наследование».

Недостаток последней реализации состоит в потребности создавать подкласс фабрики для создания экземпляров нового класса в иерархии. Недостаток превращается в достоинство, если код рассматривать с точки зрения структурированности.

```
/* # 6 # параметризованная фабрика # AbstractOrderFactory.java # SimpleOrderFactory.java
# ExtendedOrderFactory.java */
```

```
package by.bsu.factory.generic;
public abstract class AbstractOrderFactory<T extends AbstractOrder> {
```

```
        public abstract T getInstance();
        public void anOperation() {
                System.out.println("operation");
        }
}
package by.bsu.factory.generic;
public class SimpleOrderFactory extends AbstractOrderFactory<SimpleOrder> {
        @Override
        public SimpleOrder getInstance() {
                return new SimpleOrder();
        }
}
package by.bsu.factory.generic;
public class ExtendedOrderFactory extends AbstractOrderFactory<ExtendedOrder> {
        @Override
        public ExtendedOrder getInstance() {
                return new ExtendedOrder();
        }
}
```

Наличие общих методов, контролирующих корректность процесса создания каждого экземпляра класса определяет преимущество шаблона **Factory Method** перед простой инициализацией экземпляра.

Шаблон Abstract Factory

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.

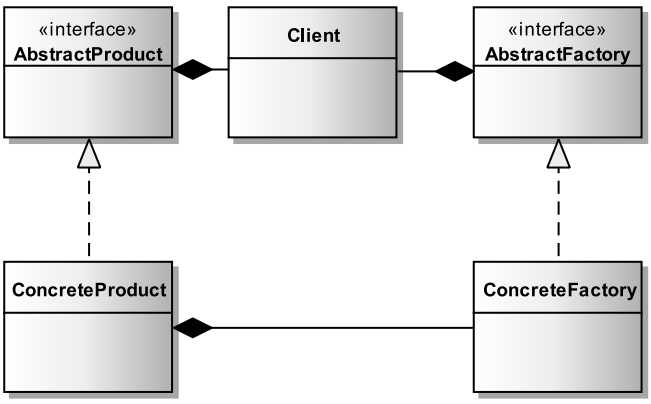

Рис. 21.2. *Базовая реализация шаблона AbstractFactory*

Предложенное решение делает конкретные классы обособленными. Так как абстрактная фабрика реализует процесс создания классов-фабрик и саму процедуру инициализации объектов, то она изолирует приложение от деталей реализации классов. Порождаемые классы должны находиться в иерархической зависимости внутри своего семейства.

Шаблон может применяться также и для создания объектов только одного семейства. Базовая реализация шаблона представляет как раз такой случай.

/ # 7 # базовая реализация Abstract Factory # AbstractProduct.java # ConcreteProduct.java # AbstractFactory.java # ConcreteFactory.java # Client.java # Runner.java*/*

```
package by.bsu.abstractfactory;
public interface AbstractProduct {
          void info();
}
package by.bsu.abstractfactory;
public class ConcreteProduct implements AbstractProduct {
          // поля
          // конструкторы
          // методы
          public void info() {
                    System.out.println("Concrete product");
          }
}
package by.bsu.abstractfactory;
public interface AbstractFactory {
          public AbstractProduct createProduct();
}
package by.bsu.abstractfactory;
public class ConcreteFactory implements AbstractFactory {
          public AbstractProduct createProduct() {
                    System.out.println("Creating concrete product");
                    AbstractProduct product = new ConcreteProduct();
                    return product;
          }
}
package by.bsu.abstractfactory;
public class Client {
          private AbstractFactory abstractFactory;
          private AbstractProduct abstractProduct;
          public void action() {
                    abstractProduct = abstractFactory.createProduct();
                    abstractProduct.info();
          }
          public void setAbstractFactory(AbstractFactory factory) {
                    abstractFactory = factory;
          }
}
```

```
package by.bsu.abstractfactory;
public class Runner {
         public static void main(String[] args) {
                  Client client = new Client();
                  client.setAbstractFactory(new ConcreteFactory());
                  client.action();
         }
}
```

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

Производители объектов реализуют методы по созданию не связанных иерархическими зависимостями объектов. Класс **AbstractMediaFactory** — абстрактная фабрика, а классы **AudioFactory** и **VideoFactory** — конкретные производители объектов, наследуемые от нее. Конкретные фабрики могут создавать объекты-продукты для TCP и UDP протоколов.

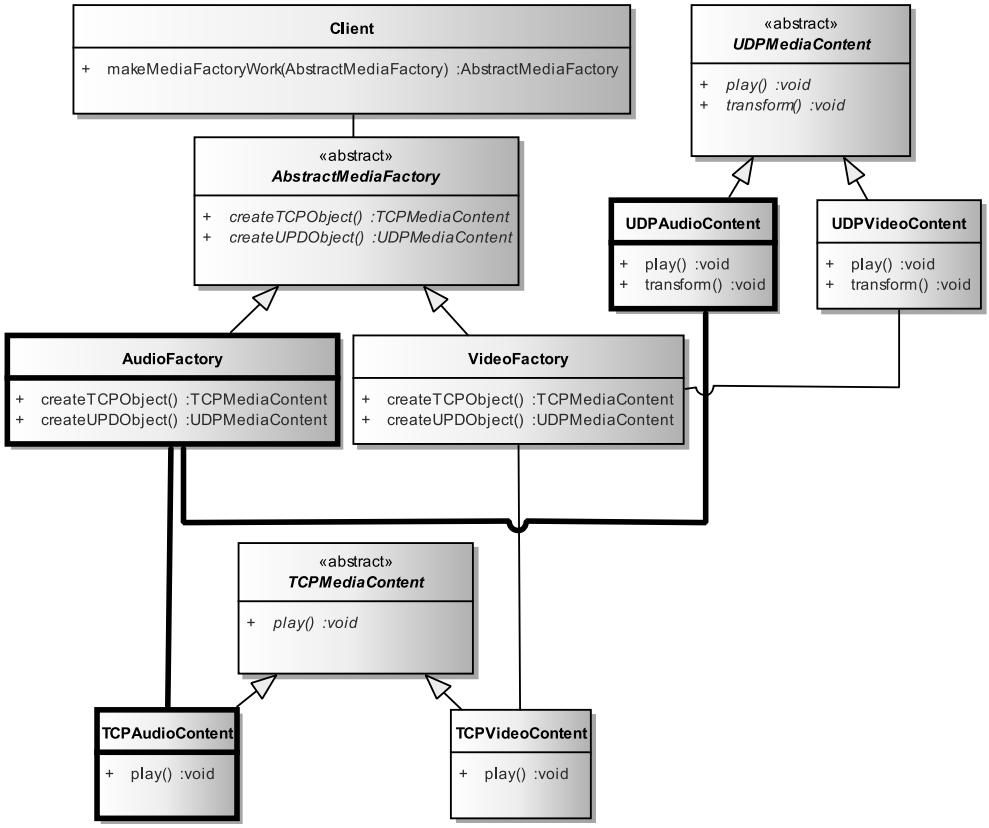

Рис. 21.3. *Пример реализации шаблона AbstractFactory*

```
/* # 8 # классы-фабрики по созданию несвязанных обьектов # AbstractMediaFactory.java
# AudioFactory.java # VideoFactory.java */
```

```
package by.bsu.abstractfactory.media;
public abstract class AbstractMediaFactory {
          public abstract TCPMediaContent createTCPObject();
          public abstract UDPMediaContent createUDPObject();
}
package by.bsu.abstractfactory.media;
public class AudioFactory extends AbstractMediaFactory {
          public TCPMediaContent createTCPObject() {
                    return new TCPAudioContent();
          }
          public UDPMediaContent createUDPObject() {
                    return new UDPAudioContent();
          }
}
package by.bsu.abstractfactory.media;
public class VideoFactory extends AbstractMediaFactory {
          public TCPMediaContent createTCPObject() {
                    return new TCPVideoContent();
          }
          public UDPMediaContent createUDPObject() {
                    return new UDPVideoContent();
          }
}
```

Рассматриваются два вида классов-продуктов: *Имя***AudioContent**, *Имя***VideoContent**. Каждый из них может быть представлен в одном из двух видов: для TCP протокола или для UDP.

```
/* # 9 # классы-продукты # TCPMediaContent.java # TCPAudioContent.java #
TCPVideoContent.java # UDPMediaContent.java # UDPAudioContent.java #
UDPVideoContent.java # Client.java */
```

```
package by.bsu.abstractfactory.media;
public abstract class TCPMediaContent {
          abstract void play();
}
package by.bsu.abstractfactory.media;
public class TCPAudioContent extends TCPMediaContent {
          void play() { // more code
          }
}
package by.bsu.abstractfactory.media;
public class TCPVideoContent extends TCPMediaContent {
          void play() { // more code
          }
}
```

```
package by.bsu.abstractfactory.media;
public abstract class UDPMediaContent {
          abstract void play();
          abstract void transform();
}
package by.bsu.abstractfactory.media;
public class UDPAudioContent extends UDPMediaContent {
          void play() { // more code
          }
          void transform() { // more code
          }
}
package by.bsu.abstractfactory.media;
public class UDPVideoContent extends UDPMediaContent {
          void play() { // more code
          }
          void transform() { // more code
          }
}
package by.bsu.abstractfactory.media;
public class Client {
          private UDPMediaContent contentUDP;
          private TCPMediaContent contentTCP;
          public void makeMediaFactoryWork(AbstractMediaFactory factory){
                    contentUDP = factory.createUDPObject();
                    contentTCP = factory.createTCPObject();
                    // использование созданных объектов
          }
}
```

Признаки использования шаблона **Abstract Factory** при создании семейств объектов:

— необходимо создавать объекты из одного или нескольких семейств;
— семейства имеют иерархическую внутреннюю структуру;
— между классами различных семейств могут прослеживаться логические связи;
— способ создания объектов должен быть скрыт.

Отличительной чертой шаблона **Abstract Factory** является определение типа объекта по внешнему признаку.

Шаблон Builder

Шаблон **Builder** заранее знает тип объекта, в то время, как его основной задачей является определение способа создания объекта на основе значений передаваемых параметров. Необходимо реализовать процесс инициализации сложного объекта или объекта со сложным процессом конструирования, определяя для него только тип и свойства. Детали построения объекта или его частей

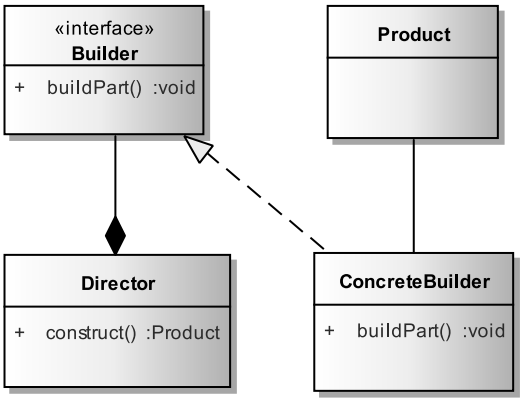

Рис. 21.4. *Базовая реализации шаблона Builder*

остаются скрытыми и определяются в подклассах. Базовая реализация выглядит следующим образом.

/ # 10 # базовая реализация Builder # Product.java # Builder.java # ConcreteBuilder.java # Director.java # Runner.java */*

```
package by.bsu.builder.base;
public class Product {
    private int part1;
    private String part2;
    public void setPart1(int part1) {
        this.part1 = part1;
    }
    public void setPart2(String part2) {
        this.part2 = part2;
    }
}
package by.bsu.builder.base;
public interface Builder {
    Product getResult();
    void buildPart1(int part1);
    void buildPart2(String part2);
}
package by.bsu.builder.base;
public class ConcreteBuilder implements Builder {
    private Product product = new Product();
    public Product getResult () {
        return product;
    }
    public void buildPart1 (int part1) {
          // реализация
        product.setPart1(part1);
    }
```

```
 public void buildPart2 (String part2) {
          // реализация
        product.setPart2(part2);
    }
}
package by.bsu.builder.base;
public class Director {
    private Builder builder; // нет необходимости в этом поле
    public Director (String builderMode) {
          // init builder
    }
public Product construct(String sourceName) {
          // чтение данных
        builder.buildPart1(параметр);
        builder.buildPart2(параметр);
        return builder.getResult();
    }
}
package by.bsu.builder.base;
public class Runner {
    public static void main(String[] args) {
        Director director = new Director("Concrete");
        Product prod = director.construct("sourcePath");
    }
}
```

Хорошим примером реализации шаблона **Builder** служит процесс создания объекта на основе информации, извлекаемой из XML-документа различными парсерами.

Класс **BaseBuilder** определяет абстрактный интерфейс для создания частей объекта класса **User**. Классы **DomBuilder, SAXBuilder** и **StAXBuilder** конструируют и собирают вместе части объекта класса **User**, а также представляют 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 шаблона появляется возможность контролировать пошагово весь процесс создания объекта-продукта.

/ # 11 # создаваемый объект объект # User.java */*

```
package by.bsu.builder;
public class User {
    private String login;
    private String password;
    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;
    }
}
```

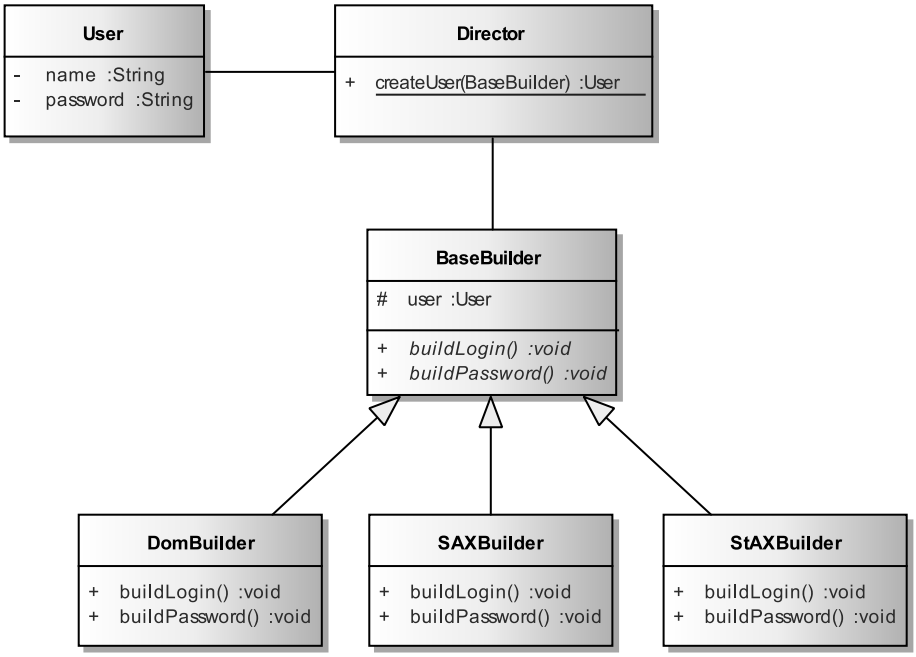

Рис. 21.5. *Пример реализации шаблона Builder*

Класс **BaseBuilder** — абстрактный класс-строитель, объявляющий в качестве поля ссылку на создаваемый объект и абстрактные методы его построения. Классы **DomBuilder, SAXBuilder** и **StAXBuilder** — наследуемые от него классы, реализующие специальные способы создания объекта.

/ # 12 # разные способы построения объекта # BaseBuilder.java # SAXBuilder.java # DomBuilder.java # StAXBuilder.java */*

```
package by.bsu.builder;
public abstract class BaseBuilder {
    protected User user = new User();
    public User getUser() {
        return user;
    }
    public abstract void buildLogin();
    public abstract void buildPassword();
    // public abstract void buildUser(); // возможен как вариант
}
```

```
package by.bsu.builder;
public class DOMBuilder extends BaseBuilder {
    // инициализация парсера DOM
    public void buildLogin() {
          // чтение логина
    }
    public void buildPassword() {
          // чтение пароля
    }
}
package by.bsu.builder;
public class SAXBuilder extends BaseBuilder {
    // инициализация парсера SAX
    public void buildLogin() {
          // чтение логина
    }
    public void buildPassword() {
          // чтение пароля
    }
}
package by.bsu.builder;
public class StAXBuilder extends BaseBuilder {
    // инициализация парсера StAX
    public void buildLogin() {
          // чтение логина
    }
    public void buildPassword() {
          // чтение пароля
    }
}
```

Процесс создания объектов с использованием одного принципа реализован ниже.

/ # 13 # тестирование процесса создания обьекта # Director.java # Runner.java */*

```
package by.bsu.builder;
public class Director {
    public static User createUser(BaseBuilder builder) {
        builder.buildLogin();
        builder.buildPassword();
        return builder.getUser();
    }
}
package by.bsu.builder;
public class Runner {
    public static void main(String args[]) {
        User e1 = Director.createUser(new DomBuilder());
        User e2 = Director.createUser(new SAXBuilder());
    }
}
```

Особенности применения шаблона **Builder**:
— осуществление широкого управления процессами создания объекта;
— процедура создания объекта независима от его внутренней организации.

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. Решение должно подходить для многопоточных приложений при условии отсутствия опасности возникновения исключительных ситуаций в конструкторе.

В настоящее время существует реализация шаблона, инициализация экземпляра которого не требует защиты при использовании в многопоточном режиме.

/ # 14 # современная реализация шаблона «Одиночка» # SingletonEnum.java */*

```
public enum SingletonEnum {
          INSTANCE;
          // поля, методы
}
```

Любая реализация класса **Enum** гарантирует наличие только одного экземпляра каждого своего элемента. При реализации шаблона **Singleton** у перечисления описывается только один элемент.

Возможность возникновения исключения в конструкторе при создании единственного экземпляра налагает определенную ответственность на процесс инициализации экземпляра. Повторный процесс инициализации должен быть осуществим в реализации класса-одиночки.

/ # 15 # реализация шаблона «Одиночка» # Singleton.java */*

```
package by.bsu.singleton;
public class Singleton {
          private final static Singleton INSTANCE = new Singleton();
          private Singleton() { } // private constructor
          public static Singleton getInstance() {
                    return INSTANCE;
          }
}
```

Класс объявляет статический метод **getInstance()**, который инициализирует экземпляр класса при первом вызове, а в последствии позволяет клиентам получать контролируемый доступ к единственному экземпляру. Такая реализация не рекомендует объявлять в классе другие статические методы, так как при вызове любого из них также произойдет инициализация поля **INSTANCE**.

Если разрешить классу иметь подклассы, то такой шаблон позволит уточнять методы через подклассы, а также разрешить появление более чем одного экземпляра. На практике такой подход используется крайне редко.

Объявление статического внутреннего класса, статическое поле которого представляет экземпляр-одиночку, решает проблему сторонних статических методов класса-владельца и обеспечивает ленивую инициализацию. Предложено Биллом Пью (Bill Pugh).

/ # 16 # решение Била Пью для шаблона «Одиночка» # LazyInitImpl.java */*

```
package by.bsu.singleton;
public class LazyInitImpl {
          private LazyInitImpl() {}
          private static class SingletonHolder { // nested class
                    private final static LazyInitImpl INSTANCE = new LazyInitImpl();
          }
          public static LazyInitImpl getInstance() {
                    return SingletonHolder.INSTANCE;
          }
}
```

Единственный экземпляр инициализируется при первом вызове метода **getInstance()**. Проблема с обработкой исключительных ситуаций в конструкторе по прежнему не решена. Так что, если конструктор класса не вызывает опасений создания исключительных ситуаций, то можно использовать этот подход.

Для обеспечения синхронизации без инициализации экземпляра в статическом поле можно использовать класс **ReentrantLock**:

/ # 17 # Lock для шаблона «Одиночка» # LockImpl.java */*

```
package by.bsu.singleton;
import java.util.concurrent.locks.ReentrantLock;
public class LockImpl {
          private static LockImpl instance = null;
          private static ReentrantLock lock = new ReentrantLock();
          private LockImpl() { }
          public static LockImpl getInstance() {
                    lock.lock(); // блокировка
                    try {
                              if (instance == null) {
                                        instance = new LockImpl();
                              }
                    } finally {
                              lock.unlock(); // снятие блокировки
                    }
                    return instance;
          }
}
```

Данное решение несколько более производительно, чем при **synchronized**.

В случае, если шаблон **Singleton** подразумевает ограничение на количество ссылок больше одной, то удобно использовать инициализацию через **Semaphore**.

/ # 18 # Semaphore для шаблона «Одиночка» # SemaphoreImpl.java */*

```
package by.bsu.singleton;
import java.util.ArrayList;
import java.util.concurrent.Semaphore;
public class SemaphoreImpl {
          private static final int MAX_AVAILABLE = 10; // лимит экземпляров списка
          private static Semaphore semaphore= new Semaphore(MAX_AVAILABLE, true);
          private static ArrayList<SemaphoreImpl> instances =
                              new ArrayList<SemaphoreImpl >(MAX_AVAILABLE);
          private SemaphoreImpl() { }
          public static SemaphoreImpl getInstance(int index) throws SingletonException{
                    if (index >= 0 && index < instances.size()) { // доступ к элементу списка
                              return instances.get(index);
                    }
                    if (semaphore.tryAcquire()) { // уменьшение значение семафора на 1
                              SemaphoreImpl tmp = new SemaphoreImpl();
                              instances.add(tmp);
                              return tmp;
                    }
                    throw new SingletonException("Превышен лимит на число экземпляров");
          }
}
```

/ # 19 # класс-исключение # SingletonException.java */*

```
package by.bsu.singleton;
public class SingletonException extends Exception {
          public SingletonException() { }
          public SingletonException(String error) {
                    super(error);
          }
}
```

При первом вызове **getInstance()** с любым значением список еще пуст, в первом блоке **if** условие ложное. Второй блок **if** уменьшает значение семафора с 10 до 9 и создает 0, добавляет в список экземпляр **SemaphoreImpl**. В списке теперь присутствует один экземпляр, и он доступен при вызове **getInstance()** с параметром, равным 0. При следующих вызовах, например, с инкрементным увеличением индекса семафор уменьшает свое значение, а список пополняется недостающими экземплярами. При попытке получить доступ к объекту с индексом вне допустимых значений списка при обнуленном значении семафора будет генерироваться исключение. Приведенное решение — самое медленное, однако его приемущество в настраиваемом количестве ссылок.

При создании единственного экземпляра в режиме многопоточности следует гарантировать невозможность получить не до конца сконструированный объект и при этом не потерять в производительности из-за постоянного контроля ссылки логов синхронизации. В одном из допустимых решений этой проблемы прибегают к **volatile** переменной.

/ # 20 # volatile для шаблона «Одиночка» # VolatileImpl.java */*

```
package by.bsu.singleton;
public class VolatileImpl {
          private static VolatileImpl instance = null;
          private volatile static boolean instanceCreated = false;
          private VolatileImpl() { }
          public static VolatileImpl getInstance() {
                    if (!instanceCreated) {
                              synchronized (VolatileImpl.class) {
                                        // или любой другой тип блокировки
                                        // например, Lock или Semaphore
                                        try {
                                                  if (!instanceCreated) {
                                                            instance = new VolatileImpl();
                                                            instanceCreated = true;
                                                  }
                                        } catch (Exception e) {
                                                  // обработка исключительной ситуации
                                                  // при инициализации
                                        }
                              }
                    }
                    return instance;
          }
}
```

Модификатор **volatile** объявляет, что по адресу ссылки на объект будет храниться адрес переменной в памяти кэша процессора, исключающий возможность получения по ссылке устаревших данных в многопоточном приложении.

Шаблон Prototype

Некоторые шаблоны реализуются и используются в самом языке. Процесс клонирования объектов фактически реализует шаблон **Prototype**. Процесс получения копии объекта всегда проще, чем создание нового. Получение данных для создания новых объектов к тому же может быть затруднено или вообще невозможно.

Пусть имеется эталонный набор объектов, представленный в виде коллекции. Объекты важны и могут понадобиться владельцу в любой момент времени.

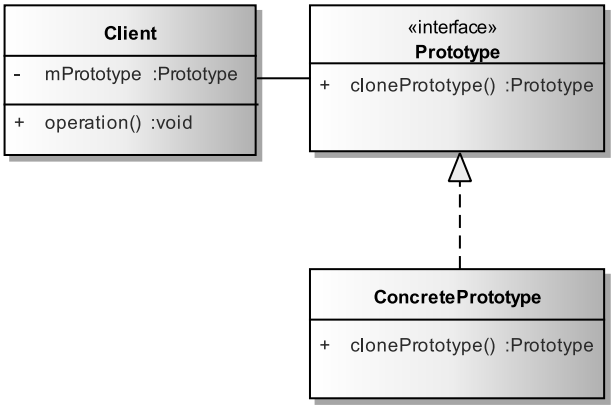

Рис. 21.6. *Базовая реализация шаблона Prototype*

Но эти же объекты могут быть необходимы кому-либо еще. Владелец готов предоставить объекты, причем желательно так, чтобы не нужно было беспокоиться об их целостности. Для обеспечеия безопасности экземпляров в коллекции владельцу объектов достаточно предоставить всем желающим точную копию объекта путем клонирования.

Шаблон **Prototype** позволяет специализировать механизм создания прототипов для заданной абстракции предметной области. Особенно это может быть полезно при применении глубокого клонирования.

Базовая схема шаблона представлена на следующей диаграмме.

/ # 21 # определение интерфейса для классов, поддерживающих прототипирование # Prototype.java */*

```
package by.bsu.prototype.base;
public interface Prototype {
    // методы
    public Prototype cloneProtype();
}
```

/ # 22 # реализация класса с созданием копии своего экземпляра # ConcretePrototype.java */*

```
package by.bsu.prototype.base;
public class ConcretePrototype implements Prototype {
        // поля и полиморфные методы
    public Prototype cloneProtype() {
        // реализация процесса создания объекта-прототипа (клона)
        return object;
    }
}
```

/ # 23 # класс, создающий и использующий копию экземпляра # Client.java */*

```
package by.bsu.prototype.base;
public class Client {
    private Prototype mPrototype;
    public void operation() {
        Prototype instance = mPrototype.clonePrototype();
        // действия с клоном
    }
}
```

Для списка книг и журналов необходимо организовать возможность поиска и клонирования экземпляра по идентификационному номеру, а также клонирование целого списка. Необходимые методы определены в интерфейсе **Client**.

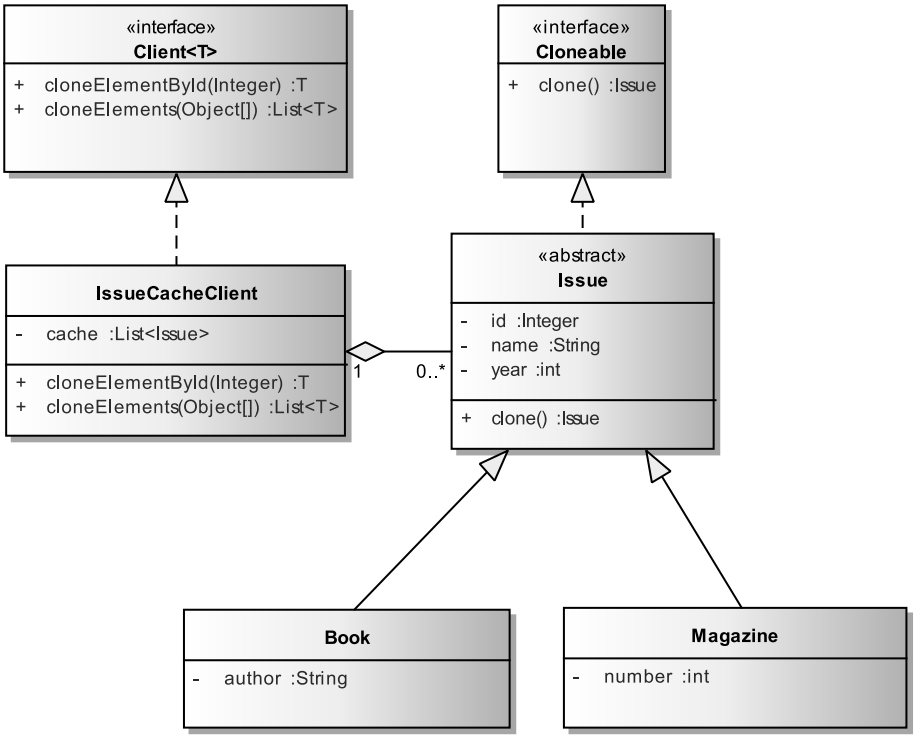

Рис. 21.7. *Реализация Prototype для клонирования элемента коллекции*

/ # 24 # интерфейс для классов для работы с прототипами # Client.java */*

```
package by.bsu.protoype.generic;
import java.util.List;
public interface Client<T> {
          T cloneElementById(Integer id);
```

```
        List<T> cloneElements(Object... param);
        // другие методы
}
```

Класс **IssueCacheClient** представляет конкретные реализации процессов клонирования.

/ # 25 # класс, создающий и использующий копию экземпляра # IssueCacheClient.java */*

```
package by.bsu.protoype.generic;
import java.util.ArrayList;
import java.util.List;
public class IssueCacheClient implements Client<Issue> { // Prototype
        private List<Issue> cache;
        public IssueCachePrototype() {
                cache = new ArrayList<Issue>();
        }
        public IssueCachePrototype(List<Issue> issueList) {
                this.cache = issueList;
        }
        @Override
        public Issue cloneElementById(Integer id) {
                for(Issue issue : cache) {
                        if(issue.getId().equals(id)) {
                                return issue.clone();
                        }
                }
                throw new IllegalArgumentException("illegal ID " + id);
        }
        @Override
        public List<Issue> cloneElements(Object... param) {
                ArrayList<Issue> list = new ArrayList<Issue>();
                // реализация поиска, клонирования и организации новой коллекции
                return list;
        }
}
```

Абстрактный класс **Issue** определяет процесс клонирования для всех своих подклассов, если только они в качестве полей используют неизменяемые типы. В случае использования изменяемых классов в качестве поля подкласса класса **Issue**, метод **clone()** требуется переопределять для этого типа. Во все классы примера # 26 добавить методы **get-**, **set-**, **toString()**.

/ # 26 # иерархия классов, поддерживающих клонирование # Issue.java # Book.java # Magazine.java */*

```
package by.bsu.protoype.generic;
public abstract class Issue implements Cloneable {
        private Integer id;
        private String name;
```

```
        private int year;
        public Issue(Integer id, String name, int year) {
                this.id = id;
                this.name = name;
                this.year = year;
        }
        @Override
        protected Issue clone() {
                Issue cloned = null;
                try {
                        cloned = (Issue) super.clone();
                } catch(CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return cloned;
        }
}
package by.bsu.protoype.generic;
public class Book extends Issue {
        private String author;
        public Book(Integer id, String author, String name, int year) {
                super(id, name, year);
                this.author = author;
        }
}
package by.bsu.protoype.generic;
public class Magazine extends Issue {
        private int number;
        public Magazine(Integer id, int number, String name , int year) {
                super(id, name, year);
                this.number = number;
        }
}
```

/ # 27 # запуск процесса создания прототипов # ProRunner.java */*

```
package by.bsu.protoype.generic;
import java.util.ArrayList;
public class ProRunner {
        public static void main(String[] args) {
                ArrayList<Issue> issueList = new ArrayList<Issue>() {
                  {
                    this.add(new Book(615, "Steve McConnell", "Code Complete", 2012));
                    this.add(new Book(453, "Bruce Eckel", "Thinking in Java", 2006));
                    this.add(new Book(721, "Joshua Bloch", "Effective Java", 2008));
                    this.add(new Magazine(1009, 9, "PC Magazine", 2012));
                  }
                };
                IssueCacheClient cache = new IssueCacheClient(issueList);
                Issue copy = cache.cloneElementById(453);
```

```
                System.out.println(issueList);
                System.out.println(copy);
        }
}
```

Применение шаблона с использованием стандартных возможностей языка фактически представляет декомпозицию стандартного процесса клонирования, рассмотренную при изучении методов класса **Object**.

Задания к главе 21

В любой задаче обязательно создание иерархии классов сущностей создаваемой системы.

1. Паттерн Builder. Разработать модель системы Автомобиль. Написать код приложения, который будет позволять порождать как серийные автомобили, так и автомобили по специальному заказу. Использовать шаблон.
2. Паттерн Builder. Разработать модель системы Музыкальный коллектив. Написать код приложения, позволяющий создавать певческие, танцевальные и смешанные коллективы.
3. Паттерн Builder. Разработать модель системы Комплексный обед. Написать код приложения, позволяющий создавать как стандартные комплексные обеды, так и обеды, в которые включены дополнительные блюда из меню.
4. Паттерн Builder. Имеется текст статьи в формате TXT. Статья состоит из заголовка, фамилий авторов, самого текста статьи и хеш-кода текста статьи. Написать приложение, позволяющее конвертировать документ в формате TXT в документ формата XML, необходимо также проверять корректность хэш-кода статьи.
5. Паттерн Abstract Factory. Написать код приложения, позволяющий универсально записывать данные о совершенном телефонном звонке в базу данных и xml-файл, а также считывать эту информацию. Использовать также шаблон DAO.
6. Паттерн Abstract Factory. Написать код приложения, позволяющий сохранять регистрационные данные пользователя в базе данных. Состав регистрационных данных у каждого пользователя может быть различен. Использовать также шаблон DAO.
7. Паттерн Abstract Factory. Разработать систему Кинопрокат. Пользователь может выбрать определенную киноленту, при заказе киноленты указывается язык звуковой дорожки, который совпадает с языком файла субтитров. Система должна поставлять фильм с требуемыми характеристиками, причем при смене языка звуковой дорожки должен меняться и язык файла субтитров и наоборот.
8. Паттерн Prototype. Существует набор статей в википедии. Реализовать процесс раздачи статей по требованию для изменения, сохраняя исходный вариант для возможного восстановления статьи в исходном виде.
9. Паттерн Factory Method. Фигуры игры «тетрис». Реализовать процесс случайного выбора фигуры из конечного набора фигур. Предусмотреть появление супер-фигур с большим числом клеток, чем обычные.

Глава 22

ШАБЛОНЫ ПОВЕДЕНИЯ

Шаблоны поведения GoF 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 (object that can fulfill a request);

Command (Команда) — используется для определения по некоторому признаку объекта конкретного класса, которому будет передан запрос для обработки (when and how a request is fulfilled);

Iterator (Итератор) — позволяет последовательно обойти все элементы коллекции или другого составного объекта, не зная деталей внутреннего представления данных (how an aggregate's elements are accessed, traversed);

Mediator (Посредник) — позволяет снизить число связей между классами при большом их количестве, выделяя один класс, знающий все о методах других классов (how and which objects interact with each other);

Memento (Хранитель) — сохраняет текущее состояние объекта для дальнейшего восстановления (what private information is stored outside an object, and when);

Observer (Наблюдатель) — позволяет при зависимости между объектами типа «один ко многим» отслеживать изменения объекта (number of objects that depend on another object; how the dependent objects stay up to date);

State (Состояние) — позволяет объекту изменять свое поведение за счет изменения внутреннего объекта состояния (states of an object);

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

Template Method (Шаблонный Метод) — создает родительский класс, использующий несколько методов, реализация которых возложена на производные классы (steps of an algorithm);

Visitor (Посетитель) — представляет операцию в одном или нескольких связанных классах некоторой структуры, которую вызывает специфичный для каждого такого класса метод в другом классе (operations that can be applied to object(s) without changing their class(es));

Interpreter (Интерпретатор) — для определенного способа представления информации определяет правила (grammar and interpretation of a language).

Шаблон Chain of Responsibility

Позвояет принять решение по обработке объекта на том уровне, на котором оно получено, или передаваться дальше по цепочке обработчиков. Применение шаблона связывает объекты составных частей приложения между собой по цепочке для передачи запроса на обработку от более низких, детализированных слоев системы к более высоким базовым.

Пусть существует базовая задача.

Задача. Сдать экзамен

- *Подзадача*. Сдать тест по предмету
- *Подзадача*. Сдать задание (одно или несколько)

Следует организовать делегирование решения одной задачи одному классу, другой — другому и т д.

Инициатор сообщения не будет иметь никакой явной функциональной связи с его обработчиком.

Для решения проблемы в класс вводится поле, идентифицирующее задачу. Остается только реализовать последовательность связей или вызовов между задачами.

Для решения вводится класс или интерфейс **Handler**, определяющий метод, передающий сообщение следующему обработчику. Причем реализация процесса передачи зависит от поставленной задачи.

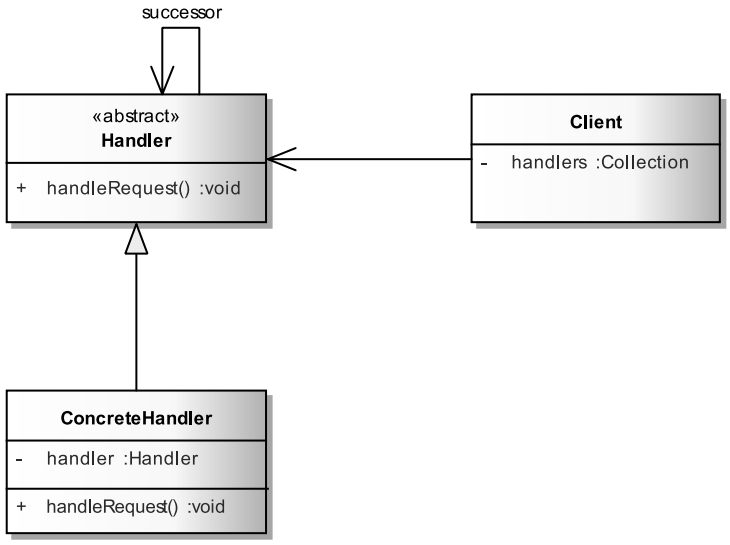

Рис. 22.1. *Базовая диаграмма шаблона Chain of Responsibility*

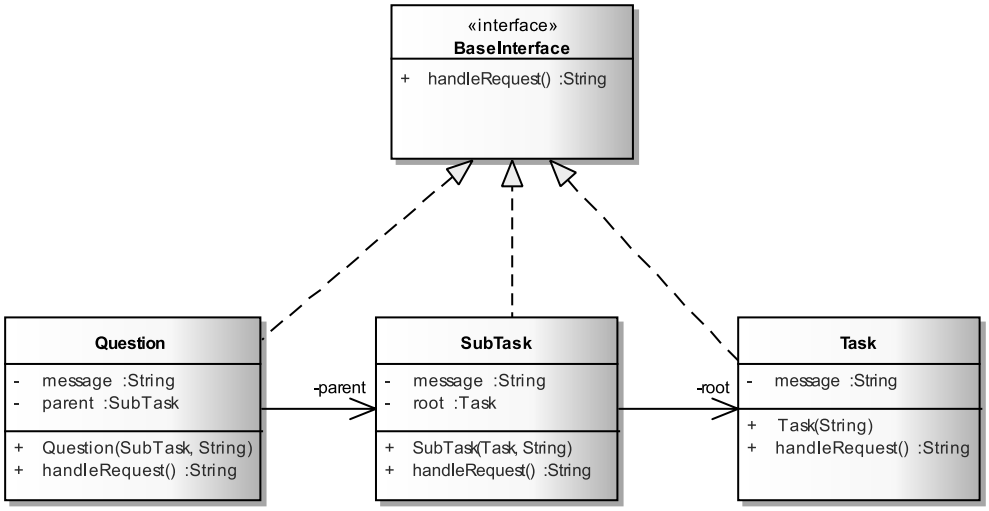

Рис. 22.2. *Реализация шаблона Chain of Responsibility*

В классе **ConcreteHandler**, наследующем класс **Handler**, хранится ссылка на следующий объект **Handler**. Если ссылка нулевая, цепочка обработки заканчивается. Метод **handleRequest()** определяет, как обрабатывать переданное сообщение.

/ # 1 # базовая реализация handlers # BaseInterface.java # Task.java # SubTask.java # Question.java*/*

```
package by.bsu.chainofresp;
public interface BaseInterface {
    public String handleRequest();
}
package by.bsu.chainofresp;
public class Task implements BaseInterface {
    private String message = "";
    public Task(String message) {
        this.message = message;
    }
    public String handleRequest() {
        System.out.println("message in Task: " + message);
        return message;
    }
}
package by.bsu.chainofresp;
public class SubTask implements BaseInterface {
    private String message = "";
    private Task root = null;
    public SubTask(Task root, String message) {
```

```
        this.root = root;
        this.message = message;
    }
    public String handleRequest() {
        System.out.println("message in SubTask: " + message);
        if (root == null) {
            return message;
        } else {
            return root.handleRequest();
        }
    }
}
package by.bsu.chainofresp;
public class Question implements BaseInterface {
          private String message = "";
          private SubTask parent = null;
          public Question(SubTask parent, String message) {
                    this.parent = parent;
                    this.message = message;
          }
          public String handleRequest() {
                    System.out.println("message in Question: " + message);
                    if (parent == null) {
                              return message;
                    } else {
                              return parent.handleRequest();
                    }
          }
}
```

Последовательность передачи сообщений или корректное построение цепочки зависит от пользователя.

/ # 2 # конфигурирование и запуск цепочки # MainChain.java */*

```
package by.bsu.chainofresp;
public class MainChain {
    public static void main(String args[]) {
          // конфигурирование цепочки
        Task root = new Task("Получить зачет");
        SubTask subTask = new SubTask(root, "Написать тест");
        Question question = new Question(subTask, "Сделать лабораторную");
          // запуск
        System.out.println("Message from Question < " + question.handleRequest() +" >");
    }
}
```

В результате выполненная цепочки задач будет получен результат:

message in Question: Сделать лабораторную
message in SubTask: Написать тест

message in Task: Получить зачет
Message from Question < Получить зачет >

Применяются различные стратегии реализации обработчика **handleRequest():**

— динамическая. Каждый класс-обработчик реализует свой процесс обработки с возможностью изменения стратегии пересылки;
— расширяющая. Каждый последующий обработчик добавляет свой функционал к предыдущему;
— дефолтная. Существует класс-обработчик, используемый всеми уровнями при отсутствии явного класса пересылки.

Также существуют различные стратегии пересылки:

— пересылка всех сообщений, кроме обрабатываемых явным образом;
— обработка всех сообщений, кроме пересылаемых явным образом;
— обработка по умолчанию сообщений, для которых не указан обработчик;
— отказ от обработки сообщений, для которых не указан обработчик.

Требуется определить систему обработки аутентификации, авторизации и назначения заданий некоему сотруднику в зависмости от его роли. Последовательность обработки можно конфигурировать необходимым пользователю образом.

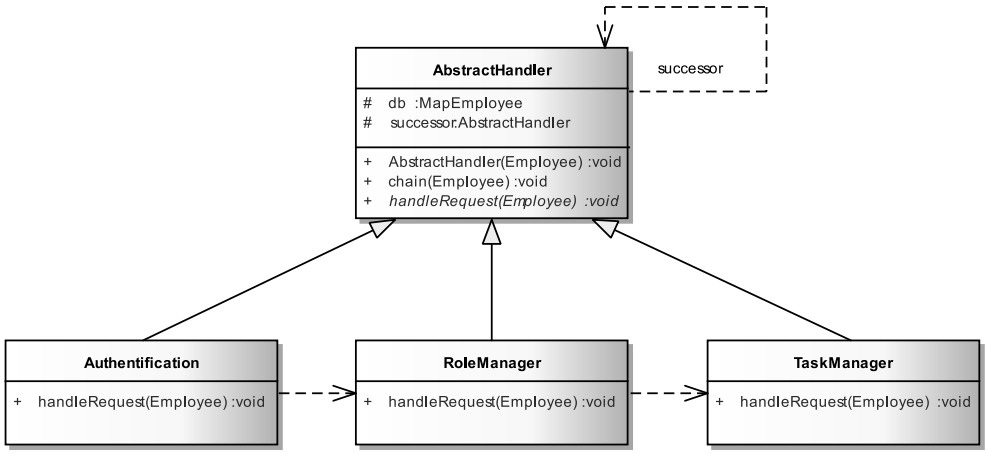

Рис. 22.3. *Реализация шаблона Chain of Responsibility*

/ # 3 # обработчик по умолчанию # AbstractHandler.java */*

```
package by.bsu.chainofresponsibility.filters;
import by.bsu.chainofresponsibility.MapEmployee;
import by.bsu.chainofresponsibility.Employee;
public abstract class AbstractHandler {
          protected MapEmployee db;
          protected AbstractHandler successor = DefaultHandleRequest.getHandleRequest();
          public AbstractHandler(AbstractHandler sucessor) {
```

```
                this.db = new MapEmployee();
                this.successor = sucessor;
        }
        public AbstractHandler() {
                this.db = new MapEmployee();
        }
        public void setSuccessor(AbstractHandler successor) {
                this.successor = successor;
        }
        abstract public void handleRequest(Employee emp);
        public void chain(Employee user) {
                if (db.containsUser(user)) {
                        handleRequest(user);
                        successor.chain(user);
                } else {
                        System.out.println("user don't exist");
                }
        }
        private static class DefaultHandleRequest extends AbstractHandler {
                private static DefaultHandleRequest handler = new DefaultHandleRequest();
                private DefaultHandleRequest() {
                }
                public static DefaultHandleRequest getHandleRequest() {
                        return handler;
                }
                @Override
                public void chain(Employee user) { // always empty
                }
                @Override
                public void handleRequest(Employee user) {
                        // обработчик по умолчанию, если существует
                }
        }
}
```

Если обработчик для экземпляра не определен, то по умолчанию никаких действий в данной реализации произведено не будет.

Для запуска цепочки обработчиков после конфигурирования следует вызвать метод **chain(Employee user)**.

/ # 4 # обработчики сообщений # Authentification.java # RoleManager.java # TaskManager.java */*

```
package by.bsu.chainofresponsibility.filters;
import by.bsu.chainofresponsibility.Employee;
public class Authentification extends AbstractHandler {
        public Authentification() { // more code
        }
        @Override
        public void handleRequest(Employee user) {
```

```
                    if (checkStatus(user)) {
                              // some code here
                    }
          }
          public boolean checkStatus(Employee user) {
                    boolean flag = true;
                    System.out.println(user);
                    System.out.println("check user status");
                    // check user status
                    return flag;
          }
}
package by.bsu.chainofresponsibility.filters;
import by.bsu.chainofresponsibility.Employee;
public class RoleManager extends AbstractHandler {
          public RoleManager() {
          }
          @Override
          public void handleRequest(Employee user) {
                    checkPermission();
          }
          public void checkPermission() {
                    System.out.println("checking role");
                    // checking role
          }
}
package by.bsu.chainofresponsibility.filters;
import by.bsu.chainofresponsibility.Employee;
public class TaskManager extends AbstractHandler {
          public TaskManager() { // more code
          }
          @Override
          public void handleRequest(Employee user) {
                    assignTask();
          }
          public void assignTask() {
                    System.out.println("assign task");
          }
}
```

Сообщение представляет собой экземпляр некоего конкретного класса. Таких классов и экземпляров может быть несколько.

// # 5 # класс, сущность сообщения которого обрабатывается # Employee.java

```
package by.bsu.chainofresponsibility;
public class Employee {
          private int id;
          private String login;
          private String password;
          public Employee(int id, String login, String password) {
```

```
                this.id = id;
                this.login = login;
                this.password = password;
        }
        public String getLogin() {
                return login;
        }
        public String getPassword() {
                return password;
        }
        public int getId() {
                return id;
        }
        @Override
        public int hashCode() {
                int result = id + ((login == null) ? 0 : login.hashCode());
                result = 31 * result + ((password == null) ? 0 : password.hashCode());
                return result;
        }
        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                Employee other = (Employee) obj;
                if (id != other.id)
                        return false;
                if (login == null) {
                        if (other.login != null)
                                return false;
                } else if (!login.equals(other.login))
                        return false;
                if (password == null) {
                        if (other.password != null) {
                                return false;
                        }
                } else if (!password.equals(other.password)) {
                        return false;
                }
                return true;
        }
        @Override
        public String toString() {
            return "Employee [id=" + id + ", login=" + login + ", password=" + password + "]";
        }
}
```

// # 6 # карта обрабатываемых реализаций сущностей # MapEmployee.java

```
package by.bsu.chainofresponsibility;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class MapEmployee {
          private HashMap<Integer, Employee> users = new HashMap<Integer, Employee>();
          public MapEmployee() {
                    users.put(1, new Employee(10, "admin", "passwordAdmin"));
                    users.put(2, new Employee(20, "employee", "passwordEmployee"));
                    users.put(3, new Employee(30, "user", "passwordUser"));
          }
          public HashMap<Integer, Employee> getUsers() {
                    return users;
          }
          public boolean containsUser(Employee emp) {
                    return users.containsValue(emp);
          }
}
```

// # 7 # конфигурирование и запуск # ChainDemo.java

```
package by.bsu.chainofresponsibility;
import by.bsu.chainofresponsibility.filters.Authentification;
import by.bsu.chainofresponsibility.filters.RoleManager;
import by.bsu.chainofresponsibility.filters.TaskManager;
public class ChainDemo {
          static public void main(String[] args) {
                    Employee user = new Employee(30, "user", "passwordUser");
                    // конфигурирование цепи
                    RoleManager rm = new RoleManager();
                    Authentification auth = new Authentification();
                    TaskManager tm = new TaskManager();
                    auth.setSuccessor(rm);
                    rm.setSuccessor(tm);
                    System.out.println("----chain--start----");
                    auth.chain(user);
          }
}
```

----chain--start----
Employee [id=30, login=user, password=passwordUser]
check user status
checking role
assign task

Конфигурирование цепочки, выполняемое явным образом, затрудняет процесс использования шаблона. В технологиях, использующих цепочки, конфигурирование обычно не осуществляется вызовом методов или конструкторов с передачей им сообщений. Хорошим примером является конфигурирование фильтров в JEE. Изначально процесс конфигурирования был реализован с использованием xml-mapping. В поздних версиях был введен процесс конфигурации с помощью аннотаций.

Шаблон Command

При работе с приложением пользователь выполняет различные операции, в ответ система всегда должна знать, где находятся данные для ее выполнения и какие действия следует выполнить. Все данные, необходимые для выполнения операции можно объединить в один объект, который и будет определять действие, или, по иному, команду.

При передаче в бизнес-логику системы запроса на действие в произвольном виде, запрос таким образом преобразуется в объект-команду, метод которого может быть вызван, а сам объект может быть сохранен и/или 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.

Основной интерфейс объекта-команды определяется в абстрактном классе **AbstractCommand** или в интерфейсе **ICommand** и в общем случае представлен одним методом **execute()**. Подклассы определяют конкретного исполнителя запроса, методы **execute()** которых обращаются к требуемой операции бизнес-логики.

Кроме команд участником шаблона может быть **Invoker**, класс вызова исполняющего метода команды. Класс **Receiver**, экземпляру которого и предназначен объект-команда. Методы класса **Receiver** выполняет переданный с командой запрос. Для определения типа команды и инициализации подходящего объекта испоьзуется класс **Client**.

Шаблон **Command** используется при:

— разделении источника запроса и его исполнителя;
— необходимости построения последовательности команд, порядок в которй определяет сам пользователь в зависимости от своего желания или результатов выполнения предыдущей команды;
— необходимости отмены предыдущей команды;
— выполнении операций с транзакциями.

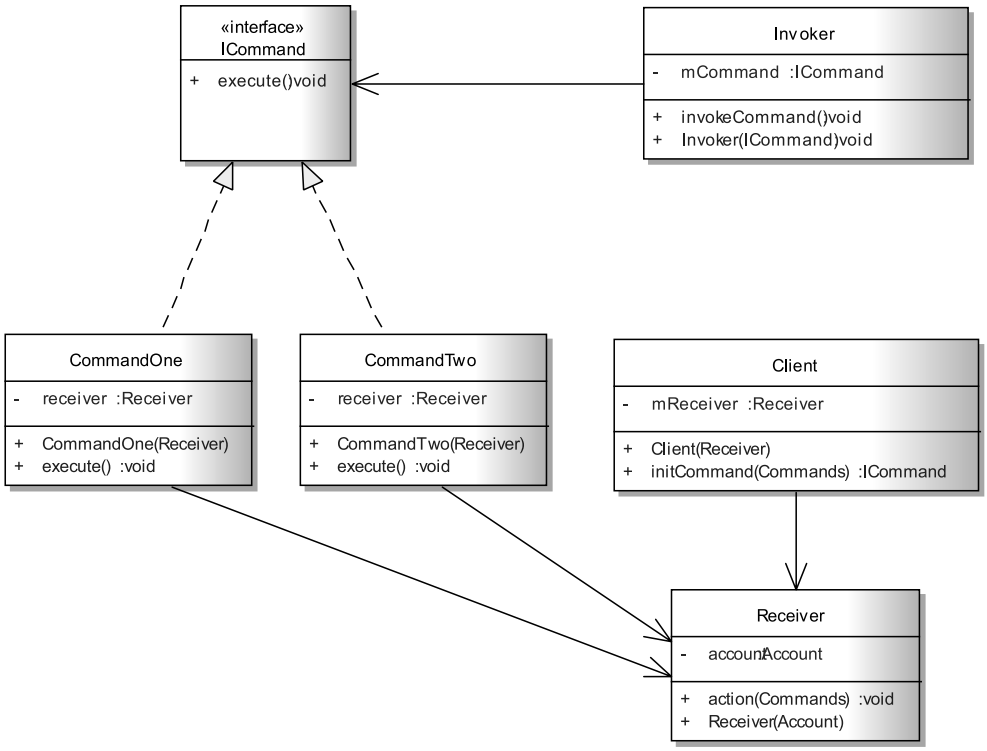

Рис. 22.4. *Базовая реализация шаблона Command*

/ # 8 # интерфейс и реализации команды # ICommand.java # TypeCommand.java # CommandOne.java # CommandTwo.java */*

```
package by.bsu.command.base;
public interface ICommand {
          void execute();
}
package by.bsu.command.base;
public enum TypeCommand {
          ONE, TWO
}
package by.bsu.command.base;
public class CommandOne implements ICommand {
          private Receiver receiver;
          public CommandOne(Receiver reciever) {
                    this.receiver = reciever;
          }
          public void execute() {
                    System.out.println("Determine connection between receiver and action");
                    receiver.action(TypeCommand.ONE);
```

```
        }
}
package by.bsu.command.base;
public class CommandTwo implements ICommand {
        private Receiver receiver;
        public CommandTwo(Receiver reciever) {
                this.receiver = reciever;
        }
        public void execute() {
                System.out.println("Determine connection between receiver and action");
                receiver.action(TypeCommand.TWO);
        }
}
```

/ # 9 # вызов команды # Invoker.java */*

```
package by.bsu.command.base;
public class Invoker {
        private ICommand mCommand;
        public Invoker(ICommand command) {
                mCommand = command;
        }
        public void invokeCommand() {
                System.out.println("Refer to command for execution");
                mCommand.execute();
        }
}
```

/ # 10 # исполнитель команды # Receiver.java */*

```
package by.bsu.command.base;
public class Receiver {
        public void action(TypeCommand cmd) {
                switch(cmd){
                case ONE:
                    System.out.println("Know the information to complete request One");
                    break;
                case TWO:
                    System.out.println("Know the information to complete request Two");
                    break;
                }
        }
}
```

/ # 11 # создатель команды, на основании переданных данных # Client.java */*

```
package by.bsu.command.base;
public class Client {
        private Receiver mReceiver;
        public Client(Receiver receiver) {
```

```
                mReceiver = receiver;
        }
        public ICommand initCommand(TypeCommand cmd) {
                ICommand command = null;
                switch(cmd) {
                   case ONE:
                       System.out.println("Creating command One and set up its receiver");
                       command = new CommandOne(mReceiver);
                       break;
                   case TWO:
                       System.out.println("Creating command Two and set up its receiver");
                       command = new CommandTwo(mReceiver);
                       break;
                }
                return command;
        }
}
```

/ # 12 # демонстрация работы шаблона Command # RunnerBase.java */*

```
package by.bsu.command;
public class RunnerBase {
        public static void main(String[] args) {
                Receiver receiver = new Receiver();
                Client client = new Client(receiver);
                ICommand commandOne = client.initCommand(TypeCommand.ONE);
                Invoker invokerOne = new Invoker(commandOne);
                invokerOne.invokeCommand();
                ICommand commandTwo = client.initCommand(TypeCommand.TWO);
                Invoker invokerTwo = new Invoker(commandTwo);
                invokerTwo.invokeCommand();
        }
}
```

Со счетом в банке можно выполнять различные операции. Число операций при развитии системы может увеличиваться. Необходимо разработать поведение, позволяющее добавлять новые команды, не изменяя способ создания и вызова команд, то есть ядро системы.

/ # 13 # интерфейс и реализация команды # ICommand.java # CommandTypes.java # CreditingCommand.java # WithdrawingCommand.java # BlockingCommand.java */*

```
package by.bsu.command;
public interface ICommand {
        void execute();
}
package by.bsu.command;
public enum CommandTypes {
        CREDITING, WITHDRAWING, BLOCKING;
}
```

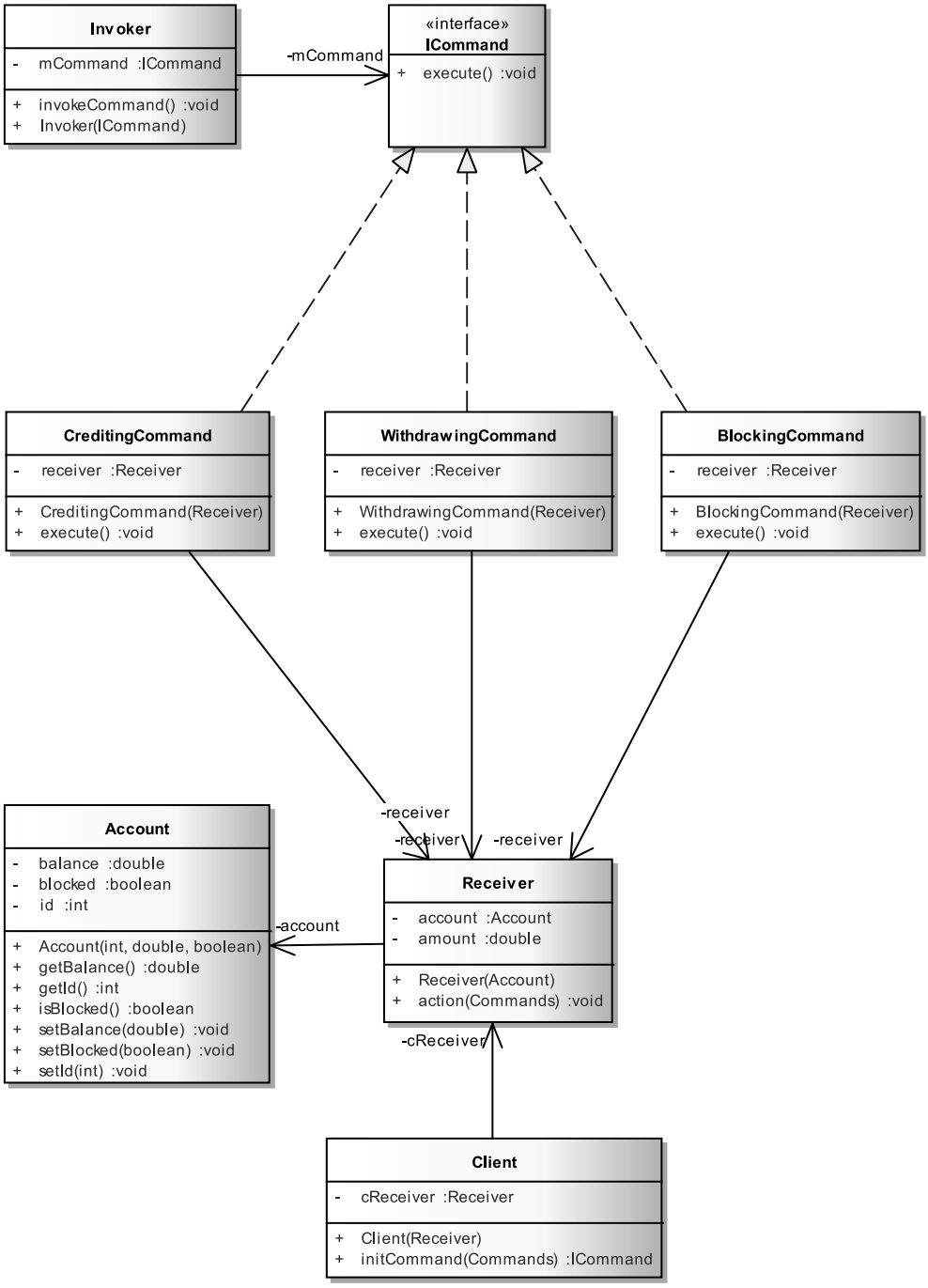

Рис. 22.5. *Реализация шаблона Command*

```
package by.bsu.command;
public class CreditingCommand implements ICommand {
          private Receiver receiver;
          public CreditingCommand(Receiver receiver) {
                    this.receiver = receiver;
          }
          public void execute() {
                    receiver.action(CommandTypes.CREDITING);
          }
}
package by.bsu.command;
public class WithdrawingCommand implements ICommand {
          private Receiver receiver;
          public WithdrawingCommand(Receiver receiver){
                    this.receiver = receiver;
          }
          public void execute() {
                    receiver.action(CommandTypes.WITHDRAWING);
          }
}
package by.bsu.command;
public class BlockingCommand implements ICommand {
          private Receiver receiver;
          public BlockingCommand(Receiver receiver) {
                    this.receiver = receiver;
          }
          public void execute() {
                    receiver.action(CommandTypes.BLOCKING);
          }
}
```

Интерфейс **ICommand** определяет абстракцию для обработки объектов-команд. Элементами перечисления **CommandTypes** являются команды, выполняемые над банковским счетом. Классы **CreditingCommand, WithdrawingCommand, BlockingCommand** реализуют объект-команды. Класс **Receiver** располагает информацией о способах выполнения операций. Реализации этого класса могут быть независимыми друг от друга, что чаще всего и происходит в реальных системах. Класс **Client** создает для каждой операции свой экземпляр **receiver**. Класс **Invoker** непосредственно вызывает команду для выполнения.

/ # 14 # исполнитель команды, обладающий информацией для ее выполнения # Receiver.java */*

```
package by.bsu.command;
public class Receiver {
          private Account account;
          private double amount;
          private static final double INTEREST_RATE = 9.5; // должно быть получено извне
          public Receiver(Account account) {
```

```
                this.account = account;
        }
        public void action(CommandTypes cmd) {
                switch (cmd) {
                case CREDITING: // реализация операции CREDITING
                        if (account.isBlocked()) {
                            System.out.println("Sorry, the account #" + account.getId()
                                    + " is blocked! You can't credit charges to it");
                        } else {
                            double balance = account.getBalance();
                            balance *= INTEREST_RATE * 0.01;
                            account.setBalance(balance);
                            System.out.println("Crediting is performed with "
                               + INTEREST_RATE + " % interest rate to the account #"
                               + account.getId());
                        }
                        break;
                case WITHDRAWING: // реализация операции WITHDRAWING
                        if (account.isBlocked()) {
                            System.out.println("Sorry, the account#" + account.getId()
                                    + " is blocked!" + " You can't withdraw money");
                        } else {
                            double balance = account.getBalance();
                            balance -= amount;
                            account.setBalance(balance);
                            System.out.println(amount
                               + " is withdrawed from the account #" + account.getId());
                        }
                        break;
                case BLOCKING: // реализация операции BLOCKING
                        if (account.isBlocked()) {
                            account.setBlocked(false);
                            System.out.println("The account #" + account.getId()
                                            + " is unblocked");
                        } else {
                            account.setBlocked(true);
                            System.out.println("The account #" + account.getId()
                                            + " is blocked");
                        }
                        break;
                }
        }
}
```

/ # 15 # создатель команды, инкапсулирующий в cReceiver информацию для ее выполнения # Client.java */*

```
package by.bsu.command;
public class Client {
        private Receiver cReceiver;
```

```
        public Client(Receiver receiver) {
                cReceiver = receiver;
        }
        public ICommand initCommand(CommandTypes cmd) {
                ICommand command = null;
                switch(cmd) {
                case CREDITING:
                   System.out.println("Creating command CREDITING & set up its receiver");
                   command = new CreditingCommand(cReceiver);
                        break;
                case WITHDRAWING:
                   System.out.println("Creating command WITHDRAWING & set up its receiver");
                   command = new WithdrawingCommand(cReceiver);
                        break;
                case BLOCKING:
                   System.out.println("Creating command BLOCKING & set up its receiver");
                   command = new BlockingCommand(cReceiver);
                        break;
                }
                return command;
        }
}
```

/ # 16 # реализация класса Invoker, который передает команду для выполнения, ничего о ней не зная # Invoker.java */*

```
package by.bsu.command;
public class Invoker {
        private ICommand mCommand;
        public Invoker(ICommand command) {
                mCommand = command;
        }
        public void invokeCommand() {
                System.out.println("Refer to command for execution");
                mCommand.execute();
        }
}
```

/ # 17 # бизнес-сущность # Account.java */*

```
package by.bsu.command;
public class Account {
        private int id;
        private double balance;
        private boolean blocked;
        public Account(int id, double balance, boolean blocked) {
                this.id = id;
                this.balance = balance;
                this.blocked = blocked;
        }
```

```
        public double getBalance() {
                return balance;
        }
        public void setBalance(double balance) {
                this.balance = balance;
        }
        public Boolean isBlocked() {
                return blocked;
        }
        public void setBlocked(boolean blocked) {
                this.blocked = blocked;
        }
        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
}
```

/ # 18 # инициализация и вызов команд # BankExample.java */*

```
package by.bsu.command;
public class BankExample {
        public static void main(String[] args) {
        Account account = new Account(210012, 1100, false);

        Receiver receiver = new Receiver(account);
        Client client = new Client(receiver);

        ICommand commandCrediting = client.initCommand(CommandTypes.CREDITING);
        Invoker invokerCrediting = new Invoker(commandCrediting);
        invokerCrediting.invokeCommand();

        ICommandcommandWithdrawing = client.initCommand(CommandTypes.WITHDRAWING);
        Invoker invokerWithdrawing = new Invoker(commandWithdrawing);
        invokerWithdrawing.invokeCommand();

        ICommand commandBlocking = client.initCommand(CommandTypes.BLOCKING);
        Invoker invokerBlocking = new Invoker(commandBlocking);
        invokerBlocking.invokeCommand();
        }
}
```

Объект-команда получен прямой инициализацией на основе переданного параметра. На практике данный объект создается или извлекается из коллекции на основе признака вызываемой команды. В этом случае появляется возможность изменять реакцию приложения на запрос команды простой заменой объекта-управления.

Шаблон Iterator

Шаблон определяет способ перебора или последовательного доступа к элементам составного экземпляра класса без раскрытия его структуры. Определяется в стандартных классах коллекций Java для отделения алгоритма работы с коллекцией от ее содержимого. Сам итерируемый класс может и не быть коллекцией, но, как правило, в этом случае он должен агрегировать объект-коллекцию.

Позволяет самостоятельно определить способ и параметры перебора. Способов перебора также может быть несколько: восходящий, нисходящий, с заданием шага и проч.

Итерируемый объект может быть передан в цикл **for(:)**.

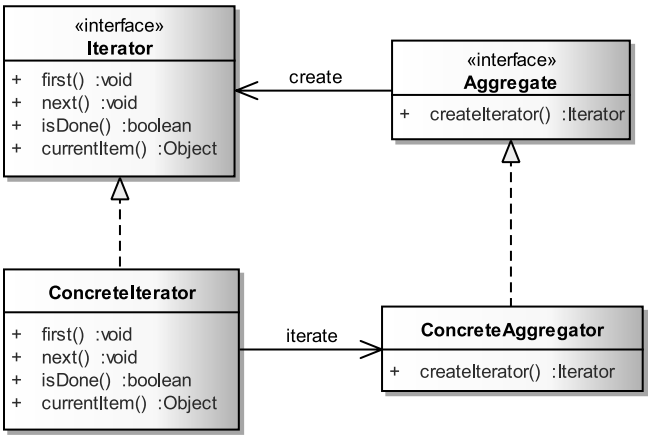

Рис. 22.6. *Базовая реализация шаблона Iterator*

Реализация шаблона должна представить выполнение операций:

— доступе к стартовому объекту извлечения, метод **first()**;
— перемещение к следующему элементу, **next()**;
— извлечение текущего элемента, **currentItem()**;
— проверка наличия следующего элемента, **isDone()**.

Интерфейс **Iterator** объединяет перечисленные выше методы, а класс **ConcreteIterator** предоставляет реализацию этих методов.

Интерфейс **Aggregate** определяет метод, создающий конкретный экземпляр **Iterator**. Класс **ConcreteAggregate** реализует метод извлечения итератора и, как правило, содержит итерируемую коллекцию или другой объект, нуждающийся в переборе.

/ # 19 # Iterator и его реализация # Iterator.java # ConcreteIterator.java */*

```
package by.bsu.iterator.base;
public interface Iterator { // аналог java.util.Iterator
         void first(); // устанавливает итератор в начальную позицию
```

```
        boolean isDone(); // аналог hasNext()
        void next(); // изменяет позицию итератора, аналог next()
        Object currentItem(); // извлекает объект в текущей позиции, аналог next()
}
package by.bsu.iterator.base;
public class ConcreteIterator implements Iterator {
        private ConcreteAggregate concreteAggregate;
        public ConcreteIterator(ConcreteAggregate aggregate) {
                concreteAggregate = aggregate;
        }
        public void first() {
                // First element in concreteAggregate
        }
        public void next() {
                // Next element in concreteAggregate
        }
        public boolean isDone() {
                // Check whether the end is reached
        }
        public Object currentItem() {
                // Return current item in concreteAggregate
        }
}
```

/ # 20 # интерфейс Aggregate и его реализация # Aggregate.java # ConcreteAggregate.java */*

```
package by.bsu.iterator.base;
// аналог java.lang.Iterable
public interface Aggregate {
        public Iterator createIterator();
}
package by.bsu.iterator.base;
public class ConcreteAggregate implements Aggregate {
        public Iterator createIterator() {
                System.out.println("Creating concrete iterator for concrete aggregate.");
                Iterator iterator = new ConcreteIterator(this);
                return iterator;
        }
}
```

/ # 21 # использование итератора # RunnerIteratorDemo.java */*

```
package by.bsu.iterator.base;
public class RunnerIteratorDemo {
        public static void main(String[] args) {
                Aggregate aggregate = new ConcreteAggregate();
                Iterator iterator = aggregate.createIterator();
                iterator.first();
```

```
                while (!iterator.isDone()) {
                        Object data = iterator.currentItem();
                        iterator.next();
                }
        }
}
```

В стандартных библиотеках Java используется отличный от стандартного интерфейс реализации шаблона. Функциональность методов next() и currentItem() объединена в одном методе **next()**, метод **isDone()** имеет другое имя — **hasNext()**. Функциональность метода **first()** передана методу создания итератора.

Ниже приведена реализация итератора для класса **StudentSession**, инкапсулирующего в виде карты **HashMap** результаты студенческой сессии. Шаблон Iterator представлен в классическом виде, но с использованием методов итератора, предоставленного стандартными библиотеками языка Java.

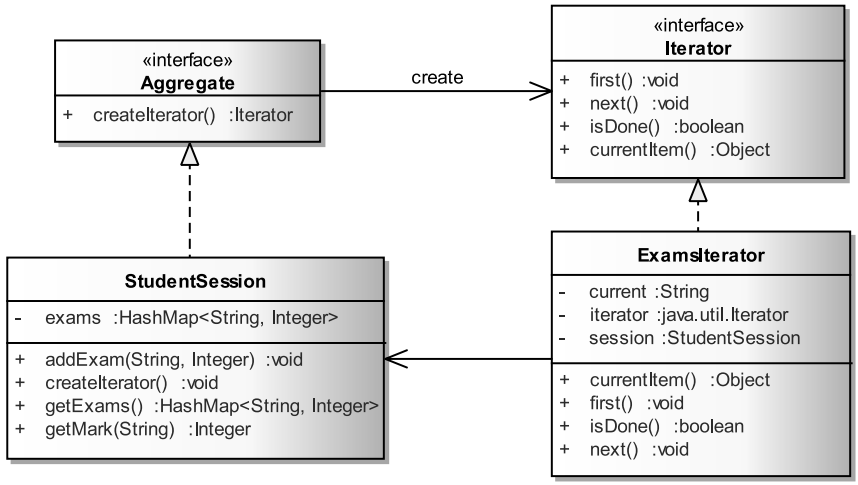

Рис. 22.7. *Пример реализации шаблона Iterator*

/ # 22 # интерфейс Iterator и его реализация # Customterator.java # ExamsIterator.java */*

```
package by.bsu.iterator;
public interface CustomIterator {
        void first();
        void next();
        boolean isDone();
        Object currentItem();
}
package by.bsu.iterator;
import java.util.NoSuchElementException;
public class ExamsIterator implements CustomIterator {
        private StudentSession session;
```

```
        private String current;
        private java.util.Iterator<String> iterator;
        private boolean done;
        public ExamsIterator(StudentSession session) {
                this.session = session;
        }
        public Object currentItem() {
                return current;
        }
        public void first() {
                iterator = session.getExams().keySet().iterator();
                next();
        }
        public boolean isDone() {
                return done;
        }
        public void next() {
                if (iterator.hasNext()) {
                        current = iterator.next();
                } else {
                        done = true;
                }
        }
}
```

/ # 23 # интерфейс Aggregate и его реализация # Aggregate.java # StudentSession.java */*

```
package by.bsu.iterator;
public interface Aggregate {
        CustomIterator createIterator();
}
package by.bsu.iterator;
import java.util.HashMap;
public class StudentSession implements Aggregate {
        private HashMap<String, Integer> exams = new HashMap<String, Integer>();
        public CustomIterator createIterator() {
                CustomIterator iterator = new ExamsIterator(this);
                iterator.first();
                return iterator;
        }
        public void addExam(String name, Integer mark) {
                exams.put(name, mark);
        }
        public Integer getMark(String name) {
                return exams.get(name);
        }
        public HashMap<String, Integer> getExams() {
                return exams;
        }
}
```

/ # 24 # демонстрация работы итератора # IteratorExample.java */*

```
package by.bsu.iterator;
public class IteratorExample {
          public static void main(String[ ] args) {
                    StudentSession session = new StudentSession();
                    session.addExam("MA", 9);
                    session.addExam("TFCV", 10);
                    session.addExam("DS", 8);
                    System.out.println("The list of exams: ");
                    CustomIterator iterator = session.createIterator();
                    while (!iterator.isDone()) {
                              System.out.println(iterator.currentItem());
                              iterator.next();
                    }
          }
}
```

Однако такая реализация шаблона **Iterator** не позволит использовать для обхода объекта **StudentSession** цикл **for(:)**. Для придания идентичности пользовательской реализации шаблона итерируемый класс должен реализовать интерфейс **java.lang.Iterable** и предоставить реализацию метода **iterator()** в виде

/ # 25 # реализация, совместимая с JavaSE # StudentSession.java */*

```
public class StudentSession implements Iterable<String> {
          // some code here
          // реализация итератора
          @Override
          public Iterator<Item> iterator() {
                    return listItems.iterator();
          }
}
```

Класс **ConcreteIterator** может быть не только внешним, но и внутренним. Внешний итератор позволяет обходить элементы объекта внешним пользователям, в то время как внутренний итератор обходит элементы по требованию извне. Реализован он также может быть и как внешний класс, и как внутренний. Итератор может работать со статической копией объекта, предоставленной ему при создании, а может работать с коллекцией в реальном состоянии. Последний вариант предрасположен к динамическим ошибкам, поэтому к его реализации нужно подходить аккуратно.

Итераторы могут использовать сложные способы обхода, например, при обходе дерева, составленного по шаблону **Composite**.

Шаблон Mediator

По мере развития все системы становятся более сложными. Управлять передачей запросов становится все сложнее. Право управлять взаимодействием множества объектов можно передать некоторому основному объекту. Связи между классами системы ослабевают, восприятие и поддержка системы становятся более понятными. Множественные связи заменяются связью многих через единственный экземпляр.

Интерфейс **Mediator** определяет методы для обращения клиентов. Реализация **ConcreteMediator** обеспечивает функциональность посредника и может иметь ссылки на клиентов для вызова их методов на основе переданных медиатору данных. Клиентский класс, в свою очередь, также может ссылаться на медиатор для информирования других клиентов об изменении своих данных.

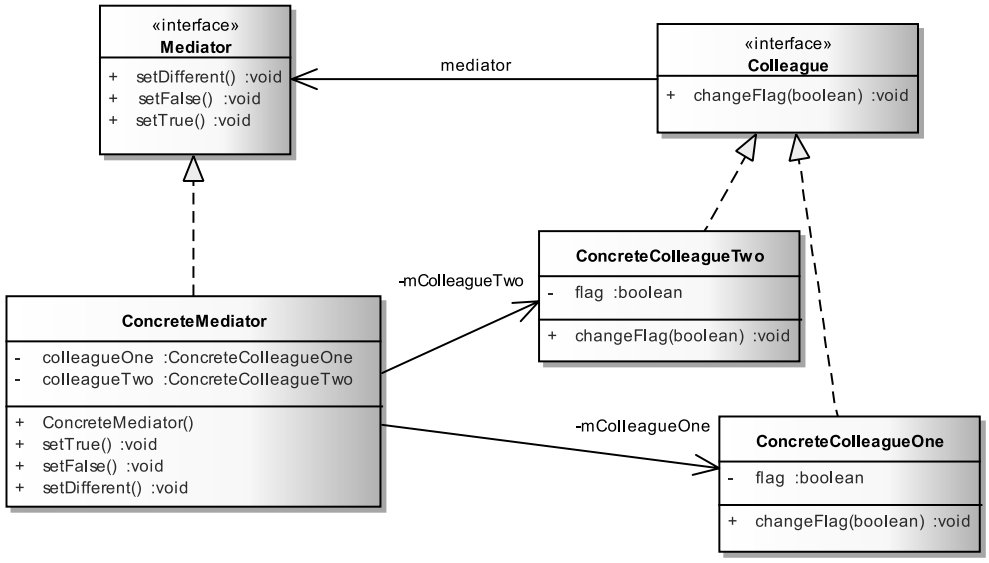

Рис. 22.8. *Базовая реализация шаблона Mediator*

/ # 26 # клиенты посредника # Colleague.java # ConcreteColleagueOne.java # ConcreteColleagueTwo.java */*

```
package by.bsu.mediator.base;
public interface Colleague {
          // может ссылаться на Mediator
          void changetFlag(boolean flag);
}
package by.bsu.mediator.base;
public class ConcreteColleagueOne implements Colleague {
    private boolean flag;
```

```
    public void changetFlag(boolean flag) {
        this.flag = flag;
    }
}
package by.bsu.mediator.base;
public class ConcreteColleagueTwo implements Colleague {
    private boolean flag;
    public void changetFlag(boolean flag) {
        this.flag = flag;
    }
}
```

/ # 27 # медиатор и его реализация # Mediator.java # ConcreteMediator.java */*

```
package by.bsu.mediator.base;
public interface Mediator {
          void setTrue();
          void setFalse();
          void setDifferent();
}
package by.bsu.mediator.base;
public class ConcreteMediator implements Mediator {
    private ConcreteColleagueOne colleagueOne;
    private ConcreteColleagueTwo colleagueTwo;
    public ConcreteMediator() {
        colleagueOne = new ConcreteColleagueOne();
        colleagueTwo = new ConcreteColleagueTwo();
    }
    public void setTrue() {
        colleagueOne.changetFlag(true);
        colleagueTwo.changetFlag(true);
        System.out.println("Both set to true");
    }
    public void setFalse() {
        colleagueOne.changetFlag(false);
        colleagueTwo.changetFlag(false);
        System.out.println("Both set to false");
    }
    public void setDifferent() {
        colleagueOne.changetFlag(true);
        colleagueTwo.changetFlag(false);
        System.out.println("First - true. Second - false");
    }
}
```

/ # 28 # использование медиатора # RunnerDemoMediator.java */*

```
package by.bsu.mediator.base;
public class RunnerDemoMediator {
    public static void main(String[] args) {
        ConcreteMediator cm = new ConcreteMediator();
```

```
        cm.setTrue();
        cm.setFalse();
        cm.setDifferent();
    }
}
```

Результатом попытки звонка в телефонной системе может быть установленное соединение, сообщение о занятости абонента, установка конференц-связи, сообщение о блокировке и многое другое. Для поддержки подобного функционала и будет использован шаблон **Mediator**.

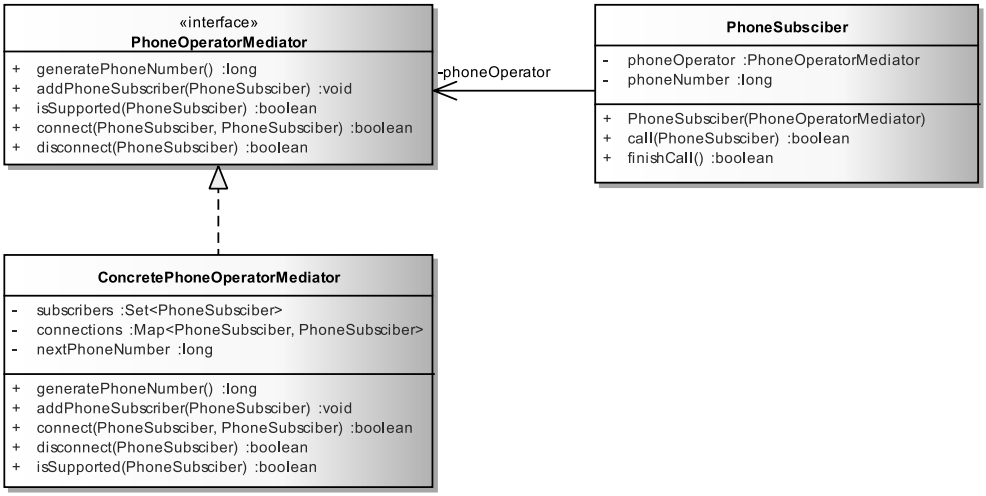

Рис. 22.9. *Пример реализации шаблона Mediator*

/ # 29 # функционал медиатора и его реализация # PhoneOperatorMediator.java # ConcretePhoneOperatorMediator.java */*

```
package by.bsu.mediator;
public interface PhoneOperatorMediator {
    long generatePhoneNumber();
    void addPhoneSubscriber(PhoneSubsciber ps);
    boolean isSupported(PhoneSubsciber ps);
    boolean connect(PhoneSubsciber ps1, PhoneSubsciber ps2);
    boolean disconnect(PhoneSubsciber ps);
}
package by.bsu.mediator;
public class ConcretePhoneOperatorMediator implements PhoneOperatorMediator {
    private Set<PhoneSubsciber> subscribers;
    private Map<PhoneSubsciber, PhoneSubsciber> connections;
    private long nextPhoneNumber;
```

```
    public ConcretePhoneOperatorMediator() {
          subscribers = new HashSet<PhoneSubsciber>();
          connections = new HashMap<PhoneSubsciber, PhoneSubsciber>();
          nextPhoneNumber = 200_00_01;
    }
    @Override
    public long generatePhoneNumber() {
          return nextPhoneNumber++;
    }
    @Override
    public void addPhoneSubscriber(PhoneSubsciber ps) {
          subscribers.add(ps);
    }
    @Override
    public boolean connect(PhoneSubsciber ps1, PhoneSubsciber ps2) {
          if (!isSupported(ps1) || !isSupported(ps2)) {
              return false;
          }
          if (connections.containsKey(ps1) || connections.containsKey(ps2)) {
              System.out.println("Line is busy...");
              return false;
          }
          connections.put(ps1, ps2);
          connections.put(ps2, ps1);
          System.out.println(ps1 + " connected to " + ps2);
          return true;
    }
    @Override
    public boolean disconnect(PhoneSubsciber ps1) {
          if (connections.containsKey(ps1)) {
              PhoneSubsciber ps2 = connections.get(ps1);
              connections.remove(ps1);
              connections.remove(ps2);
              System.out.println(ps1 + " disconnected fom " + ps2);
              return true;
          } else {
              System.out.println(ps1 + " not connected to any subscriber");
              return false;
          }
    }
    @Override
    public boolean isSupported(PhoneSubsciber ps) {
          boolean isSupported = subscribers.contains(ps);
          if (!isSupported) {
              System.out.println("Not supported " + ps);
          }
          return isSupported;
    }
}
```

/ # 30 # клиент посредника # PhoneSubsciber.java */*

```
package by.bsu.mediator;
public class PhoneSubsciber {
    private PhoneOperatorMediator phoneOperator;
    private long phoneNumber;
    public PhoneSubsciber(PhoneOperatorMediator phoneOperator) {
          this.phoneOperator = phoneOperator;
          this.phoneNumber = phoneOperator.generatePhoneNumber();
    }
    public boolean call(PhoneSubsciber otherPhoneSubsciber) {
          return phoneOperator.connect(this, otherPhoneSubsciber);
    }
    public boolean finishCall() {
          return phoneOperator.disconnect(this);
    }
    public long getPhoneNumber() {
          return phoneNumber;
    }
    public void setPhoneNumber(long phoneNumber) {
          this.phoneNumber = phoneNumber;
    }
    @Override
    public String toString() {
          return "subscriber #" + phoneNumber;
    }
    @Override
    public boolean equals(Object obj) { // simplest
          if (obj != null && obj instanceof PhoneSubsciber) {
              return this.getPhoneNumber() == ((PhoneSubsciber) obj).getPhoneNumber();
          } else {
              return false;
          }
    }
}
```

/ # 31 # демонстрация использования посредника # RunnerPhoneMediator.java */*

```
package by.bsu.mediator;
public class RunnerPhoneMediator {
    public static void main(String[] args) {
          ConcretePhoneOperatorMediator phoneOperator = new ConcretePhoneOperatorMediator();
          PhoneSubsciber ps1 = new PhoneSubsciber(phoneOperator);
          PhoneSubsciber ps2 = new PhoneSubsciber(phoneOperator);
          PhoneSubsciber ps3 = new PhoneSubsciber(phoneOperator);
          PhoneSubsciber ps4 = new PhoneSubsciber(phoneOperator);
          phoneOperator.addPhoneSubscriber(ps1);
          phoneOperator.addPhoneSubscriber(ps2);
          phoneOperator.addPhoneSubscriber(ps3);
          // попытка двух абонентов позвонить на один номер
          ps1.call(ps2);
```

```
            ps3.call(ps2);
            // завершение разговора и повтор попытки
            ps2.finishCall();
            ps3.call(ps2);
            // попытка звонка незарегистрированным абонентом
            ps4.call(ps1);
        }
}
```

В результате будет выведено:

subscriber #2000001 connected to subscriber #2000002
Line is busy...
subscriber #2000002 disconnected fom subscriber #2000001
subscriber #2000003 connected to subscriber #2000002
Not supported subscriber #2000004

Пример с телефонной станцией выглядел бы более естественно при использовании потоков. Шалон **Mediator** хорошо применяется для задач с конкурирующими операциями.

Шаблон может применяться для односторонних задач, когда клиенты имеют возможность только отправлять сообщения или только их принимать. Функционал самого клиента может быть упрощен за счет передачи части полномочий классу медиатору.

Шаблон Memento

Позволяет зафиксировать и извлечь все свойства объекта (значения его полей) с возможностью последующего восстановления объекта до данного состояния.

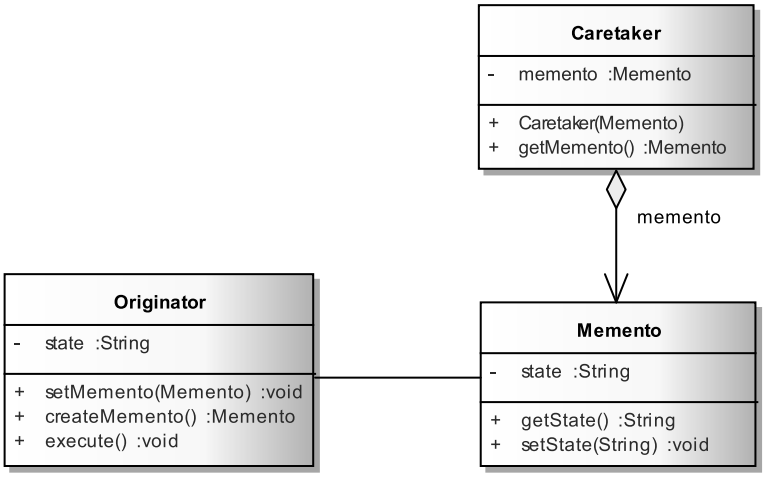

Рис. 22.10. *Базовая реализация шаблона Memento*

Инкапсуляция объекта, для которого выполняется «моментальный снимок», не нарушается.

Класс-создатель или **Originator** создает экземпляр **Memento** и оповещает его о своем состоянии, и только он может сохранить и получить информацию из объекта **Memento**. Ни один другой класс такой возможности не имеет. Статический внутренний класс **Memento**, сохраняющий информацию об объекте **Originator**. Класс **Caretaker** не знает об информации, сохраняемой в объекте **Memento**, но знает, почему и когда **Originator** может себя восстановить.

/ # 32 # хранитель состояния # Memento.java */*

```
package by.bsu.memento.base;
public class Memento {
    private String state;
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}
```

/ # 33 # класс, сохранением состояния которого занимается Memento # Originator.java */*

```
package by.bsu.memento.base;
public class Originator {
    private String state = "Initial state";
    public void setMemento(Memento memento) {
        state = memento.getState();
    }
    public Memento createMemento() {
        Memento memento = new Memento();
        memento.setState(state);
        return memento;
    }
    public void execute() {
        state = "New state";
    }
    @Override
    public String toString() {
        return state;
    }
}
```

/ # 34 # восстанавливает сохраненное состояние # Caretaker.java */*

```
package by.bsu.memento.base;
public class Caretaker {
    private Memento memento;
```

```
    public Caretaker(Memento memento) {
        this.memento = memento;
    }
    public Memento getMemento() {
        return memento;
    }
}
```

/ # 35 # сохранение и восстановление состояния # RunnerMemento.java */*

```
package by.bsu.memento.base;
public class RunnerMemento {
    public static void main(String[] args) {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker(originator.createMemento());
        System.out.println(originator);
        originator.execute();
        System.out.println(originator);
        originator.setMemento(caretaker.getMemento());
        System.out.println(originator);
    }
}
```

В качестве примера можно привести сохранение состояния счета клиента перед каким-либо серьезным действием с ним. Например, перед транзакцией, чтобы в случае неудачно завершенной транзакции восстановить предшествующее состояние.

При обращении клиентской страницы к серверу вместе с запросом передается набор параметров, представляющих информацию, введенную клиентом в поля формы. Если какая-либо часть информации неккорректна, то приложение предлагает клиенту вернуться на форму и исправить ошибки, при этом все поля формы, передаваемой клиенту для коррекции, должны быть заполнены той же самой информацией. То есть необходимо выполнить откат на один шаг.

/ # 36 # интерфейс для идентификации обьекта Memento # Memento.java */*

```
package by.bsu.memento;
public interface Memento {
}
```

/ # 37 # интерфейс для идентификации обьекта Memento # RequestParameter.java */*

```
package by.bsu.memento;
import java.util.HashMap;
public class RequestParameter {
          private HashMap<String, String> param = new HashMap<String, String>();
          public RequestParameter(HashMap<String, String> param) {
                    this.param = param;
          }
```

```
        public Memento getMemento() {
                HashMap<String, String> state = (HashMap<String, String>) (param.clone());
                return new RequestParameterMemento(state);
        }
        public void setMemento(Memento object) {
                if (object instanceof RequestParameterMemento) {
                      RequestParameterMemento memento = (RequestParameterMemento) object;
                      param = memento.state;
                }
        }
        private class RequestParameterMemento implements Memento { // внутренний класс
                private HashMap<String, String> state;
                RequestParameterMemento(HashMap<String, String> state) {
                          this.state = state;
                }
        } // окончание внутреннего класса
        public void addParam(String key, String value) {
                param.put(key, value);
        }
        public HashMap<String, String> getParams() {
                return param;
        }
        public void removeParam(String key) {
                param.remove(key) ;
        }
        public void clearParams() {
                param = new HashMap<String, String>();
        }
}
```

/ # 38 # хранитель Memento # Caretaker.java */*

```
package by.bsu.memento;
public class Caretaker {
    private Memento memento;
    public Caretaker(Memento memento) {
        this.memento = memento;
    }
    public Memento getMemento() {
        return memento;
    }
}
```

/ # 39 # сохранение и восстановление состояния # RequestRunner.java */*

```
package by.bsu.memento;
import java.util.HashMap;
public class RequestRunner {
        public static void main(String[] args) {
                HashMap<String, String> p = new HashMap<String, String>() {
```

```
                    {
                        this.put("1", "Winner");
                    }
                };
                RequestParameter req = new RequestParameter(p);
                System.out.println("first " + req.getParams());
                   Memento memento = req.getMemento();
                   Caretaker care = new Caretaker(memento);
                req.addParam("1", "Loser");
                System.out.println("second " + req.getParams());
                   memento = care.getMemento();
                   req.setMemento(memento);
                System.out.println("undo to first " +req.getParams());
        }
}
```

В результате будет выведено:

first {1=Winner}
second {1=Loser}
undo to first {1=Winner}

Для сохранения параметров, переданных с запросом в класс **RequestParameter** в виде **HashMap**, используется внутренний закрытый класс **RequestParameterMemento**. Получить текущее и восстановить необходимое состояние объекта можно методами **getMemento()** и **setMemento()**. Внутренний класс **RequestParameterMemento** и его интерфейс **Memento** не предоставляют никаких методов по доступу к своему внутреннему содержанию. Экземпляр **Caretaker** может только сохранять экземпляр **memento**, не имея никакой возможности его прочитать.

При частом использовании для больших по занимаемой памяти объектов шаблон **Memento** может использовать значительные ресурсы приложения и снижать эффективность его эксплуатации.

Шаблон Observer

В системах данные бывают отделены от их представлений. Для сохранения согласованности информации необходим механизм немедленного уведомления всех заинтересованных объектов об изменениях состояния связанного с ними объекта. То есть требуется определить связь «один ко многим» между объектами таким образом, чтобы при изменении состояния одного объекта все связанные с ним объекты должны автоматически изменить свое представление. Объект, изменяющий состояние, ничего не знает о том, что будет делать **Observer** с переденной ему информацией. Позволяет наблюдать за несколькими объектами одному экземпляру **Observer**. В языке Java этот шаблон известен под именем **Listener**.

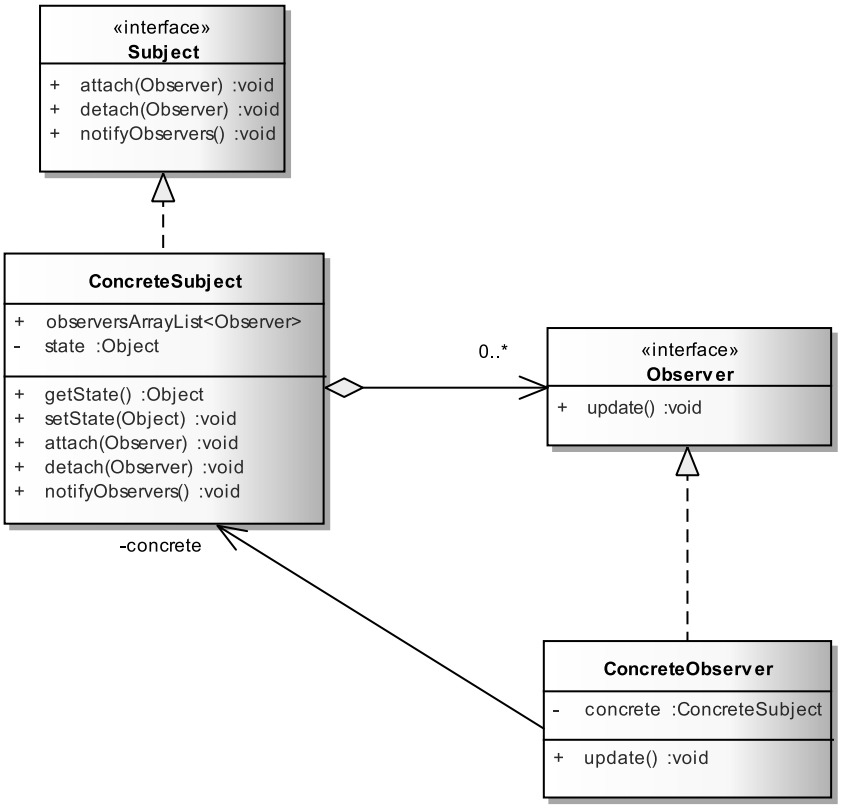

Рис. 22.11. *Базовая реализация шаблона Observer*

/ # 40 # интерфейс и изменение состояний объекта # Subject.java # ConcreteSubject.java */*

```
package by.bsu.observer.base;
public interface Subject {
    public void attach(Observer ob);
    public void detach(Observer ob);
    public void notifyObservers();
}
package by.bsu.observer.base;
import java.util.ArrayList;
public class ConcreteSubject implements Subject {
    public ArrayList<Observer> observer; // может быть единичным объектом
    private Object state;
    public Object getState () {
        return state;
    }
    public void setState(Object value) {
          // реализация
```

```
    }
    public void attach(Observer ob) {
          // реализация
    }
    public void detach(Observer ob) {
          // реализация
    }
    public void notifyObservers () {
          // запуск метода(ов) Observer
    }
}
```

/ # 41 # интерфейс и реализация слушателя# Observer.java # ConcreteObserver.java */*

```
package by.bsu.observer.base;
public interface Observer {
    public void update();
}
package by.bsu.observer.base;
public class ConcreteObserver implements Observer {
    private ConcreteSubject subject;
    public void update() {
          // реализация
    }
}
```

Шаблон **Observer** делает классы системы слабо связанными, что важно при реализации сложных моделей процессов удаления или обновления информации.

При простой модели обработки событий, генерируемые сообщения предназначены конкретному получателю, который выполняет некоторый набор действий. В более сложной модели отправка сообщения наблюдателю влечет за собой изменение, в том числе, генератора событий и сторониих объектов.

В сетевом аукционе делаются ставки на товар. Как только клиент изменил значение своей ставки, генерируется событие и обработчик проверяет ставку на возможность лидерства. Если ставка становится лидером, то сам объект-генератор ставки будет уведомлен об этом посредством изменения значения поля **leader**. То есть обработчик событий влияет на самого генератора. Если ставка не зарегистрирована на покупку данного товара, то никаких действий предприниматься не будет в ответ на изменение значения цены.

Класс **Bid** (субъект) располагает информацией о своих наблюдателях и предоставляет интерфейс для регистрации и уведомления наблюдателей. Интерфейс **Observer** (наблюдатель) определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта. Класс **AuctionObserver** (конкретный наблюдатель) хранит и/или получает объект-событие, содержащий экземпляр **Bid**, сохраняет данные и реализует интерфейс обновления, определенный в классе **AuctionObserver** для поддержки согласованности с субъектом.

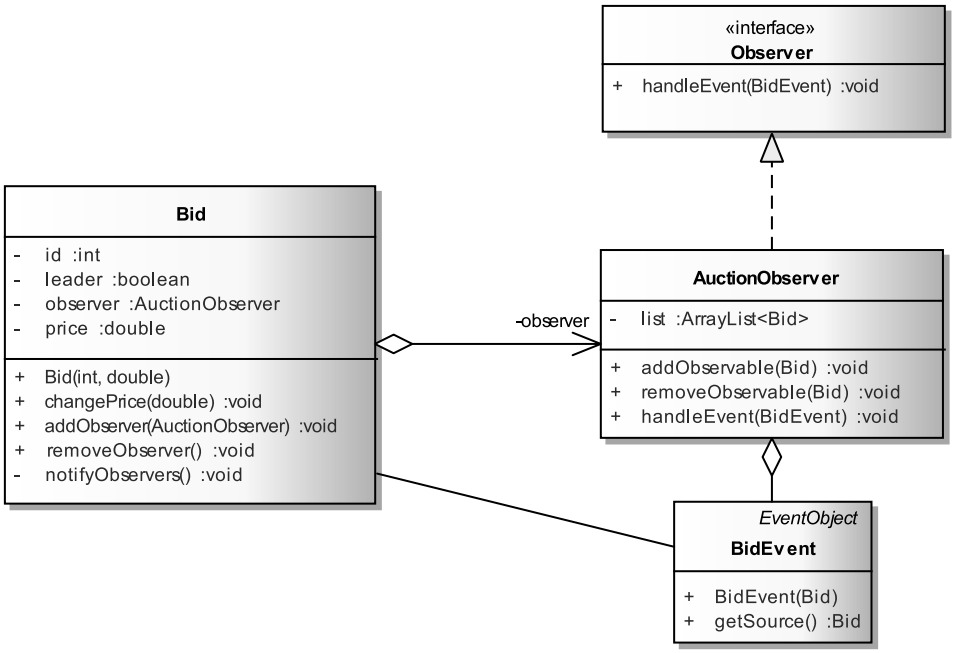

Рис. 22.12. *Пример реализации шаблона Observer*

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 один или некоторое число наблюдателей, каждый из которых подчиняется интерфейсу абстрактного класса-наблюдателя.

/ # 42 # класс-субъект с функционалом управления наблюдателем и событие, ему соответствующее # Bid.java # BidEvent.java */*

```
package by.bsu.observer;
public class Bid {
          private int id;
          private double price;
          private AuctionObserver observer;
          private boolean leader = false;
          public Bid(int id, double price) {
                    super();
                    this.id = id;
                    this.price = price;
          }
          public void changePrice(double price) {
                    this.price = price;
                    notifyObservers();
          }
```

```
        public int getId() {
                return id;
        }
        public double getPrice() {
                return price;
        }
        public void setLeader(boolean leader) {
                this.leader = leader;
        }
        public boolean isLeader() {
                return leader;
        }
        public void addObserver(AuctionObserver observer) {
                this.observer = observer;
                observer.addObservable(this);
        }
        public void removeObserver() {
                observer.removeObservable(this);
                observer = null;
        }
        private void notifyObservers() {
                if(observer != null) {
                        observer.handleEvent(new BidEvent(this));
                }
        }
        @Override
        public String toString() {
                return "Bid [id=" + id + ", price=" + price + ", leader=" + leader + "]";
        }
}
package by.bsu.observer;
import java.util.EventObject;
public class BidEvent extends EventObject {
        public BidEvent(Bid source) {
                super(source);
        }
        @Override
        public Bid getSource() {
                return (Bid)super.getSource();
        }
}
```

Классы **AuctionObserver** реализует интерфейс **Observer** и является наблюдателем. Как только субъект **Bid** изменяется, генерируется объект **BidEvent**, он передается методу **handleEvent()**, который выполнит действия в соответствии с реализованным интерфейсом и изменит, если необходимо, состояние объекта **LeaderInfo** и объектов, зарегистрированных у наблюдателя.

/ # 43 # интерфейс наблюдателя и его реализация # Observer.java # AuctionObserver.java */*

```
package by.bsu.observer;
public interface Observer {
          void handleEvent(BidEvent event);
}
package by.bsu.observer;
import java.util.ArrayList;
import java.util.Iterator;
public class AuctionObserver implements Observer {
          private ArrayList<Bid> list = new ArrayList<Bid>();
          public void addObservable(Bid bid) {
                    list.add(bid);
          }
          public void removeObservable(Bid bid) {
                    // удаление объекта из списка наблюдателя
          }
          public void handleEvent(BidEvent event) {
                    double newPrice = event.getSource().getPrice();
                    double price = 0;
                    Iterator<Bid> iterator = list.iterator();
                    boolean lead = true;
                    while (iterator.hasNext()) {
                              Bid bid = iterator.next();
                              price = bid.getPrice();
                              if (newPrice > price) {
                                        bid.setLeader(false);
                              } else if (newPrice < price) {
                                        lead = false;
                              }
                    }
                    if (lead) {
                              event.getSource().setLeader(true);
                              LeaderInfo.currentPrice = newPrice;
                              System.out.println("Leading Price " + newPrice);
                    }
          }
}
```

/ # 44 # класс, чье состояние будет меняться наблюдателем # LeaderInfo.java */*

```
package by.bsu.observer;
public class LeaderInfo {
          public static double currentPrice;
}
```

/ # 45 # использование шаблона Observer # DemoAuction.java */*

```
package by.bsu.observer;
import java.util.ArrayList;
```

```
public class DemoAuction {
          public static void main(String[] args) {
                    Bid bid1 = new Bid(1, 34);
                    Bid bid2 = new Bid(2, 35);
                    Bid bid3 = new Bid(3, 14);
                    Bid bid4 = new Bid(4, 20);
                    Bid bid5 = new Bid(5, 39);
                    AuctionObserver observer = new AuctionObserver();
                    bid1.addObserver(observer);
                    bid2.addObserver(observer);
                    bid3.addObserver(observer);
                    bid4.addObserver(observer);
                    bid5.addObserver(observer);
                    ArrayList<Bid> list = new ArrayList<Bid>();
                    list.add(bid1);
                    list.add(bid2);
                    list.add(bid3);
                    list.add(bid4);
                    list.add(bid5);
                    System.out.println("First:");
                    bid3.changePrice(45); // предложение больше максимального
                    for (Bid bid : list) {
                              System.out.println(bid);
                    }
                    System.out.println("Second:");
                    bid2.changePrice(40); // предложение меньше максимального
                    for (Bid bid : list) {
                              System.out.println(bid);
                    }
                    System.out.println("Third:");
                    bid4.changePrice(50); // предложение больше максимального
                    for (Bid bid : list) {
                              System.out.println(bid);
                    }
          }
}
```

В результате будет выведено:

First:
Leading Price 45.0
Bid [id=1, price=34.0, leader=false]
Bid [id=2, price=35.0, leader=false]
Bid [id=3, price=45.0, leader=true]
Bid [id=4, price=20.0, leader=false]
Bid [id=5, price=39.0, leader=false]
Second:
Bid [id=1, price=34.0, leader=false]
Bid [id=2, price=40.0, leader=false]

```
Bid [id=3, price=45.0, leader=true]
Bid [id=4, price=20.0, leader=false]
Bid [id=5, price=39.0, leader=false]
Third:
Leading Price 50.0
Bid [id=1, price=34.0, leader=false]
Bid [id=2, price=40.0, leader=false]
Bid [id=3, price=45.0, leader=false]
Bid [id=4, price=50.0, leader=true]
Bid [id=5, price=39.0, leader=false]
```

Если две и более ставок окажутся максимальными, то все они будут помечены как лидеры. Данная задача станет более демонстративной, если ее реализовать с применением конкурирующих операций.

Шаблон State

Позволяет отслеживать состояния объекта класса на основе агрегированных им объектов, классы которых в свою очередь могут быть организованы в некоторую иерархию классов-состояний или по другому принципу.

Шаблон **State** очень часто используется без всякого упоминания разработчика о нем. Отслеживание состояния объекта некоторого класса и его изменений в процессе работы приложения могут иметь значение для корректного функционирования. Поведение такого объекта часто и определяется его внутренним состоянием. Объект *Счет Клиента* может быть в состояниях «доступен»-«блокирован», объект *Заказ* может быть в состояниях «в обработке»-«выполнен». Отслеживание и изменение таких биполярных состояний легко реализуется введением в класс поля типа

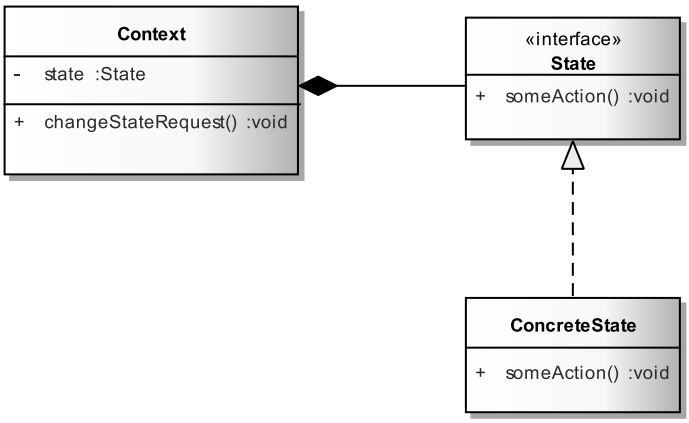

Рис. 22.13. *Базовая реализация шаблона State*

boolean и соответствующих методов по его изменению и извлечению. Если же состояний несколько и процесс перехода из одного состояния в другое подчиняется набору правил, то для организации переходов между состояниями используется иерархия состояний и специальный класс-контекст, управляющий изменениями состояний. В общем случае объект может находиться в одном состоянии из их конечного набора. Нарушение этого правила при проектировании организации состояний делает восприятие класса практически невозможным.

/ # 46 # базовая реализация State # State.java # Context.java # ConcreteState.java */*

```
package by.bsu.state.base;
public interface State {
          void someAction();
}
package by.bsu.state.base;
public class ConcreteState implements State {
          @Override
          public void someAction() { // some code here
          }
}
package by.bsu.state.base;
public class Context {
          private State state;
          public Context() { // some code here
          }
          public void changeStateRequest() {
                    state.someAction();
          }
}
```

Реализация интерфейса **State** отвечает за одно конкретное состояние. Метод **someAction()** изменяет состояние. Класс **Context** представляет описание сущности, изменение состояния которой интересует наблюдателя. Этот класс должен, как правило, обладать методами по изменению поля **state**. Но за упрвление состояниями может отвечать и другой класс.

Пусть для приема-передачи данных используется экземпляр **TCPConnection**. Соединение может находиться в трех соостояниях: открытое, закрытое и установленное, то есть надежное и работоспособное. При задании состояния оно должно открываться, а перед переходом в другое состояние обязательно закрываться. В период нахождения в состоянии оно может быть синхронизировано.

/ # 47 # класс с изменяемым состоянием # TCPConnection.java */*

```
package by.bsu.state;
public class TCPConnection {
          private TCPState state;
          // другие поля
          public TCPConnection() {
```

```
                state = new TCPClosedState();
        }
        public void open() {
                state.open(this);
                this.changeState(new TCPOpenState());
        }
        public void close() {
                state.close(this);
                this.changeState(new TCPClosedState());
        }
        public void synchronize() {
                state.synchronize(this);
                this.changeState(new TCPEstablishedState());
        }
        private void changeState(TCPState state) {
                // проверка на возможность изменения состояния
                this.state = state;
        }
}
```

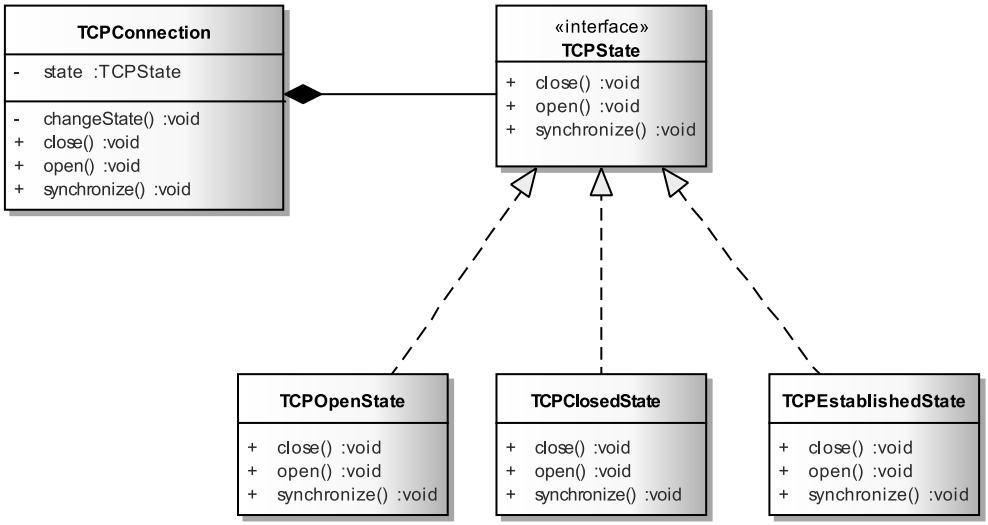

Рис. 22.14. *Состояния класса TCPConnection*

/ # 48 # интерфейс и классы состояний # TCPState.java # TCPClosedState.java # TCPOpenState.java # TCPEstablishedState.java */*

```
package by.bsu.state;
public interface TCPState {
        void open(TCPConnection context);
        void close(TCPConnection context);
        void synchronize (TCPConnection context);
}
```

```
package by.bsu.state;
public class TCPClosedState implements TCPState {
          @Override
          public void open(TCPConnection context) {
                    System.out.println("State Closed: Opening");
          }
          @Override
          public void close(TCPConnection context) {
                    System.out.println("State Closed: Closing");
          }
          @Override
          public void synchronize(TCPConnection context) {
                    System.out.println("State Closed: Synchronizing");
          }
}
package by.bsu.state;
public class TCPOpenState implements TCPState {
          @Override
          public void open(TCPConnection context) {
                    System.out.println("State Open: Opening");
          }
          @Override
          public void close(TCPConnection context) {
                    System.out.println("State Open: Closing");
          }
          @Override
          public void synchronize(TCPConnection context) {
                    System.out.println("State Open: Synchronizing");
          }
}
package by.bsu.state;
public class TCPEstablishedState implements TCPState{
          @Override
          public void open(TCPConnection context) {
                    System.out.println("State Established: Opening");
          }
          @Override
          public void close(TCPConnection context) {
                    System.out.println("State Established: Closing");
          }
          @Override
          public void synchronize(TCPConnection context) {
                    System.out.println("State Established: Synchronizing");
          }
}
```

Объявление класов-состояний можно перенести внутрь класса **Context**. Действие будет вполне правомерным, так как соятояния являются логической и неотделимой частью класса-контекста. В системе управления образованием

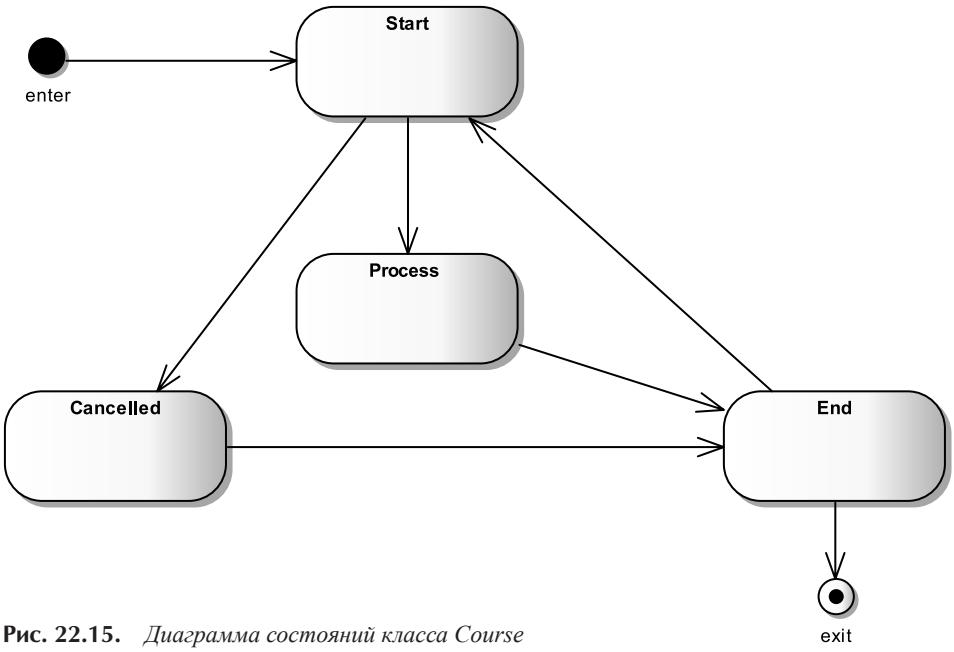

Рис. 22.15. *Диаграмма состояний класса Course*

состояние объекта класса *Учебный курс* в процессе обучения могут изменяться по правилам, приведенным на диаграмме состояний. Из некоторых состояний, в отличие от примера с соединениями, перейти в другое состояние невозможно. Например, нельзя отменить уже начавшийся курс или стартовать его еще раз, если процесс обучения уже начат.

/ # 49 # интерфейс и абстрактный класс состояний # IState.java # AbstractState.java */*

```
package by.bsu.state.univ;
public interface IState {
          void learning();
          void toCancel();
}
package by.bsu.state.univ;
public abstract class AbstractState implements IState {
          protected IState nextState;
}
```

Объявление внешних интерфейса и абстрактного класса для внутреннего класса необходимо для организации более удобного процесса управления состоянием из внешнего класса, если такое понадобится.

/ # 50 # некоторый служебный Bean-класс # Teacher.java */*

```
package by.bsu.state.univ;
public class Teacher { // поля, конструкторы методы
}
```

/ # 51 # класс с внутренним классом состояния # Course.java */*

```
package by.bsu.state.univ;
public class Course { // класс Context
    private long id;
    private String name;
    private Teacher teacher;
    private IState currentState;
    public Course(long id, String name, Teacher teacher) {
        this.id = id;
        this.name = name;
        this.teacher = teacher;
        currentState = new StartState();
    }
    public void setTeacher(Teacher teacher) {
        /* проверка имени и id курса на соответствие специализации преподавателя */
        this.teacher = teacher;
    }
    // методы set и get
    public IState getCurrentState() {
        System.out.println(currentState.getClass().getSimpleName());
        return currentState;
    }
    public void learning() {
        currentState.learning();
    }
    public void cancel() {
        currentState.toCancel();
    }
// классы состояний
    public class StartState implements IState {
        private IState nextState;
        public void learning() {
            if (Course.this.teacher != null) {
                currentState = new ProcessState();
                System.out.println("обучение начато");
            } else {
                this.toCancel();
                System.out.println("обучение не начато: нет преподавателя");
            }
        }
        public void toCancel() {
            currentState = new CancelledState();
            System.out.println("курс обучения отменен");
        }
    }
    public class ProcessState implements IState {
        private IState nextState = new EndState();
        public void learning() {
            currentState = nextState;
```

```
            System.out.println("обучение успешно завершено");
          // формирование отчета
          //сохранение результатов
        }
        public void toCancel() {
            throw new UnsupportedOperationException("Невозможно отменить уже начатый курс");
        }
    }
    public class EndState implements IState {
        private IState nextState = new StartState();
        public void learning() {
            currentState = nextState;
            // назначение нового преподавателя
            Course.this.setTeacher(new Teacher());
            System.out.println("курс готов к началу обучения");
        }
        public void toCancel() {
            throw new UnsupportedOperationException(
                                "Курс уже закончен. Его проведение отменять нет смысла");
        }
    }
    public class CancelledState implements IState {
        private IState nextState = new EndState();
        public void learning() {
            currentState = new StartState();
            // назначение нового преподавателя
            Course.this.setTeacher(new Teacher());
            System.out.println("курс готов к продолжению обучения");
        }
        public void toCancel() {
            throw new UnsupportedOperationException("Курс уже отменен");
        }
    }
}
```

Предыдущий способ определения классов состояний можно заменить на вариант без использования закрытых внутренних классов. Решение для класса **Course** будет выглядеть более эффектно, но вследствие статичности перечислений его внутренняя реализация утратит доступ к полям класса-владельца. Такой подход реализуем, когда взаимозависимость между состоянием и объектом достаточно проста и не требует манипуляций значениями полей класса со стороны состояния.

/ # 52 # класс с внутренним перечислением состояний # Course.java */*

```
package by.bsu.state.univenum;
public class Course { // класс Context
    private long id;
    private String name;
    private Teacher teacher;
```

```
private static State currentState;
enum State {
    // start nested enum State
    START {
        private State nextState;
        public void learning() {
            currentState = PROCESS;
        }
        public void toCancel() {
            currentState = CANCELLED;
            System.out.println("курс обучения отменен");
        }
    },
    PROCESS {
        private State nextState;
        public void learning() {
            currentState = nextState;
            System.out.println("обучение успешно завершено");
                        // формирование отчета
                        // сохранение результатов
        }
        public void toCancel() {
            throw new UnsupportedOperationException(
                                                    "Невозможно отменить уже начатый курс");
        }
    },
    CANCELLED {
        private State nextState;
        public void learning() {
            currentState = START;
            System.out.println("курс готов к продолжению обучения");
        }
        public void toCancel() {
            throw new UnsupportedOperationException("Курс уже отменен");
        }
    },
    END {
        private State nextState;
        public void learning() {
            currentState = nextState;
            System.out.println("курс готов к началу обучения");
        }
        public void toCancel() {
            throw new UnsupportedOperationException(
                                "Курс уже закончен. Его проведение отменять нет смысла");
        }
    };
    abstract void learning();
    abstract void toCancel();
}
// end nested enum State
```

```
public Course(long id, String name, Teacher teacher) {
        this.id = id;
        this.name = name;
        this.teacher = teacher;
        State.currentState = State.START;
    }
    public void setTeacher(Teacher teacher) {
        /* проверка имени и id курса на соответствие специализации преподавателя */
        this.teacher = teacher;
    }
    public static State getCurrentState() {
        System.out.println(State.currentState);
        return State.currentState;
    }
    public void learn() {
        State.currentState.learning();
    }
    public void cancel() {
        State.currentState.toCancel();
    }
   // методы set и get
}
```

В конце раздела приведена реализация шаблона **State** для проверки процесса оплаты заказа и выставления штрафных санкций в случае задержки оплаты. Данную реализацию следует в качестве упражнения попробовать переделать, заменив поле **IPayState** класса order на поле типа **boolean** и оценить, как это отразится на организации класса и способах вычисления сумм к оплате.

/ # 53 # реализация состояний # IPayState.java # PurchasedState.java # UnPurchasedState.java */*

```
package by.bsu.pay.state;
public interface IPayState {
          public void check(Order order);
          public void purchase(Order order);
}
package by.bsu.pay.state;
public class PurchasedState implements IPayState  {
          public void check(Order order) {
                        // проверка
                System.out.println("Оплаченный заказ проверен");
          }
          public void purchase(Order order) {
                        // оплата
                System.out.println("Заказ оплачен");
          }
}
package by.bsu.pay.state;
public class UnPurchasedState implements IPayState {
```

```
    public void check(Order order) {
        double percent = 0;
        if (order.getDays() <= 0) {
            percent = order.getCost() + order.PERCENT;
            order.setCost(order.getCost() + percent);
                    // не оплачен вовремя - начислены проценты
        }
        System.out.println("Заказ не оплачен: штраф ->" + percent);
    }
    public void purchase(Order order) {
        order.setCurrentOrderState(new PurchasedState());
    }
}
```

// # 54 # класс-контекст # Order.java

```
package by.bsu.pay.state;
package by.bsu.pay.state;
public class Order {
          private IPayState currentOrderState;
          private int days;
          private double cost;
          public final double PERCENT = 0.01;
          public Order(double cost,int days) {
                    currentOrderState = new UnPurchasedState();
                    this.days = days; /* в течение этого срока заказ должен быть оплачен */
                    this.cost = cost; // стоимость заказа
          }
          public void purchased(){
                    currentOrderState.purchase(this);
          }
          public void setCurrentOrderState(IPayState currentOrderState) {
                    this.currentOrderState = currentOrderState;
          }
          public void checked() {
                    currentOrderState.check(this);
          }
          public double getCost() {
                    return cost;
          }
          public void setCost(double cost) {
                    this.cost = cost;
          }
          public int getDays() {
                    return days;
          }
          public void setDays(int days) {
                    this.days = days;
          }
}
```

// # 55 # запуск демонстрации # RunnerStateProcess.java

```
package by.bsu.pay.state;
public class RunnerStateProcess {
          public static void main(String[] args) {
                    Order order = new Order(1000,14);
                    order.checked();
                    order.setDays(-1);
                    order.checked();
                    order.purchased();
                    order.checked();
          }
}
```

Шаблон Strategy

Необходимо определить семейство родственных алгоритмов, инкапсулировать каждый из них и сделать их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются. Алгоритмы решают близкие задачи и обладают идентичным интерфейсом. Отличия начинаются только в реализации. К примерам таких задач относятся алгоритмы сортировки, поиска, архивирования, кодирования, генерации случайных чисел и прочих. Известен также под названием Policy.

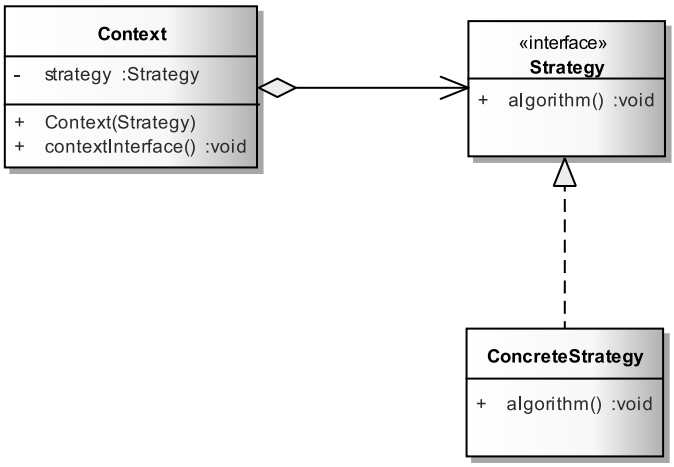

Рис. 22.16. *Базовая реализация шаблона Strategy*

// # 56 # интерфейс стратегии и его реализация # Strategy.java # ConcreteStrategy.java

```
package by.bsu.strategy.base;
public interface Strategy {
          public void algorithm();
```

```
}
package by.bsu.strategy.base;
public class ConcreteStrategy implements Strategy {
          public void algorithm() {
                    System.out.println("Using concrete algorithm.");
          }
}
```

Все алгоритмы полностью реализуются в подклассах **Strategy** так, что все алгоритмы внешне отличаются только по названиям классов, их инкапсулирующих. Для смены алгоритма достаточно изменить объект в поле класса **Context**.

// # 57 # контекст выбора и исполнения алгоритма # Context.java

```
package by.bsu.strategy.base;
public class Context {
          private Strategy strategy;
          public Context(Strategy strategy) {
                    this.strategy = strategy;
          }
          public void contextInterface() {
                    strategy.algorithm();
          }
}
```

// # 58 # запуск # BaseRunner.java

```
package by.bsu.strategy.base;
public class BaseRunner {
          public static void main(String[] args) {
                    Strategy strategy = new ConcreteStrategy();
                    Context context = new Context(strategy);
                    context.contextInterface();
          }
}
```

Интерфейс **IConversion,** определяющий **Strategy**, объявляет общий для всех поддерживаемых алгоритмов интерфейс, которым пользуется класс **Convert** (Context) для вызова конкретного алгоритма преобразования изображения, определенного в классах **ConversionGif, ConversionJpg** или **ConversionPng**.

/ # 59 # интерфейс стратегии и его реализации # IConversion.java # ConversionGif.java # ConversionJpg.java # ConversionPng.java */*

```
package by.bsu.strategy;
import java.net.URL;
public interface IConversion {
          void convert(URL urlFileImg);
}
```

```
package by.bsu.strategy;
import java.net.URL;
public class ConversionGif implements Conversion {
          @Override
          public void convert(URL urlFileImg) {
                    System.out.println("GIF Conversion");
          }
}
package by.bsu.strategy;
import java.net.URL;
public class ConversionJpg implements Conversion {
          @Override
          public void convert(URL urlFileImg) {
                    System.out.println("JPG Conversion");
          }
}
package by.bsu.strategy
import java.net.URL;
public class ConversionPng implements Conversion {
          @Override
          public void convert(URL urlFileImg) {
                    System.out.println("PNG Conversion");
          }
}
```

/ # 60 # контекст исполнения стратегии # Convert.java */*

```
package by.bsu.strategy;
import java.net.URL;
public class Convert {
          private Conversion conversion;
          public Convert(Conversion conversion) {
                    this.conversion = conversion;
          }
          public void convert(URL fileImg) {
                    conversion.convert(fileImg);
          }
}
```

Класс **Convert** конфигурируется объектом класса, реализовавшего интерфейс **Conversion**, объявленного в виде поля, и может задавать интерфейс, позволяющий объекту типа **Conversion** получить доступ к информации, в данном случае для преобразования файлов-изображений, загруженных с заданного адреса, в необходимый формат.

/ # 61 # демонстрация выбора алгоритма конвертации # DemoStrategy.java */*

```
package by.bsu.strategy;
import java.net.MalformedURLException;
import java.net.URL;
```

```
public class DemoStrategy {
          public static void main(String[] args) throws MalformedURLException {
                    URL fileUrl = new URL("image_file_url");
                  Convert convertToJpg = new Convert(new ConversionJpg());
                  convertToJpg.convert(fileUrl);
                    Convert convertToGif = new Convert(new ConversionGif());
                    convertToGif.convert(fileUrl);
                  Convert convertToPng = new Convert(new ConversionPng());
                  convertToPng.convert(fileUrl);
          }
}
```

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. Класс **Context** инкапсулирует алгоритмы в виде динамически изменяемого набора пар «ключ-алгоритм», но при обязательном наличии ненулевого алгоритма по умолчанию.

/ # 62 # хранение и выбор алгоритмов без условных операторов # Context.java */*

```
package by.bsu.strategy.plus;
import java.util.HashMap;
import java.util.Map;
public class Context {
          public final static int DEFAULT_ALGORITHM = 0;
          private Map<Integer, Strategy> algorithms = new HashMap<Integer, Strategy>();
          public Context(Strategy strategy) {
                    // проверка на null
                    algorithms.put(DEFAULT_ALGORITHM, strategy);
          }
          public Context() {
                    this(new DefaultStrategy());
          }
          public void registerAlgorithm(int key, Strategy strategy) {
                    if (key != 0) {
                              algorithms.put(key, strategy);
                    }
          }
          public void registerDefaultAlgorithm(Strategy strategy) {
                    // проверка на null
                    algorithms.put(DEFAULT_ALGORITHM, strategy);
          }
          public void contextStrategy(int key) {
                    algorithms.get(key).doAlgorithm();
          }
}
```

/ # 63 # интерфейс стратегии и его реализации # Strategy.java # DefaultStrategy.java # ConcreteStrategy.java */*

```
package by.bsu.strategy.plus;
public interface Strategy {
          void doAlgorithm();
}
package by.bsu.strategy.base.plus;
public class DefaultStrategy implements Strategy {
          @Override
          public void doAlgorithm() {
          }
}
package by.bsu.strategy.plus;
public class ConcreteStrategy implements Strategy {
          @Override
          public void doAlgorithm() {
          }
}
```

Шаблон Template Method

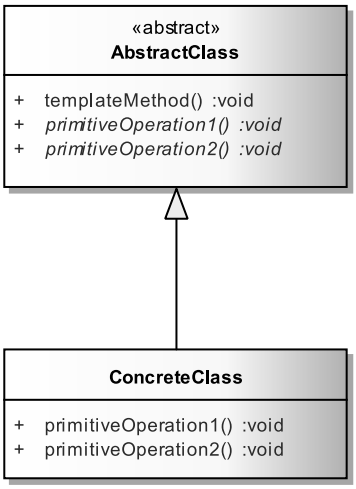

Рис. 22.17. *Базовая реализация шаблона Template Method*

Определяет основу алгоритма действий, оставляя элементы реализации подклассам. Алгоритм и последовательность действий задаются абстрактными методами, навязывая контракт подклассам для определения и расширения специфичного поведения в полиморфных методах. Подкласс в этой ситауции может заменять части метода, не переписывая их заново.

Если отвлечься от академического определения, то **Template Method** — не что иное, как механизм переопределения методов суперкласса или реализации абстрактных методов интерфейса или абстрактного класса. Если абстрагироваться от организации правила вызова реализуемых методов, в некотором методе, определяющем правила и последовательность их вызовов, то изучение этого шаблона есть повторение принципов полиморфизма.

/ # 64 # сигнатуры методов и их реализация в подклассе # AbstractClass.java # ConcreteClass.java */*

```
package by.bsu.templatemethod;
public abstract class AbstractClass {
          public abstract void primitiveOperation1();
```

```
        public abstract void primitiveOperation2();
        public void templateMethod() {
                primitiveOperation1();
                primitiveOperation2();
        }
}
package by.bsu.templatemethod;
public class ConcreteClass extends AbstractClass {
        public void primitiveOperation1() {
                // some code here
        }
        public void primitiveOperation2() {
                // some code here
        }
}
```

Часть метода базового компонента может быть неизменной, а другая часть — настраиваемой в подклассах. Тогда суперкласс будет содержать реализацию неизменяемой части и абстракцию изменяемой.

Пусть в классе **AbstractFramework** в методе **templateMethodLogin()** задана общая последовательность проверки введенных пользователем логина и пароля, определения его прав и их добавления к профилю пользователя.

Класс **BaseFramework** получает возможность задать конкретную реализацию метода проверки пароля и добавление прав в профиль пользователя.

/ # 65 # сигнатуры методов и их реализация в подклассе # AbstractFramework.java # ListPermission.java # BaseFramework.java */*

```
package by.bsu.templatemethod;
public abstract class AbstractFramework {
        public AbstractFramework() {
        }
        // методы, задающие контракт для подклассов
        protected abstract boolean check(User user);
        protected abstract ListPermission getAvaliablePermissions();
        // определение общего алгоритма
        public void templateMethodLogin(User user) { // template method
                int count = 0;
                // вызов переопределяемых в подклассах методов
                while(!check(user)) {
                        if(++count == 3) {
                                System.out.println("access denied for " + user);
                        return;
                        }
                }
        // получение списка прав (list), доступных данному пользователю
        ListPermission list = getAvaliablePermissions();
        }
}
```

```
package by.bsu.templatemethod;
public class ListPermission {
          // организация списка прав пользователя в зависимости от его роли
}
package by.bsu.templatemethod;
// классов с конкретным поведением может быть несколько
public class BaseFramework extends AbstractFramework {
          // конкретное поведение
          protected boolean check(User user) {
                    System.out.println("check User");
                    return true;
          }
          protected ListPermission getAvaliablePermissions() {
                    // получение списка прав пользователя
                    System.out.println("list of user permissions");
          }
}
```

Шаблон **Template Method** применим в случаях, если:

— существует общее для всех подклассов поведение, но оно может быть разделено на фрагменты и помещено в суперкласс;

— каркас алгоритма однократно задан жестко, а конкретное изменяемое поведение возложено на подклассы.

Шаблон Visitor

Необходимо иметь возможность добавлять новую функциональность классам некоторой иерархии, не изменяя интерфейс базового класса. Варианты возможного поведения централизуются и остаются неизменными в то время, как операции для разных классов должны выполняться по-разному.

Шаблон **Visitor** позволяет сосредоточить все такие операции в одном классе. Классов типа **ConcreteElement** может быть много, но для каждого

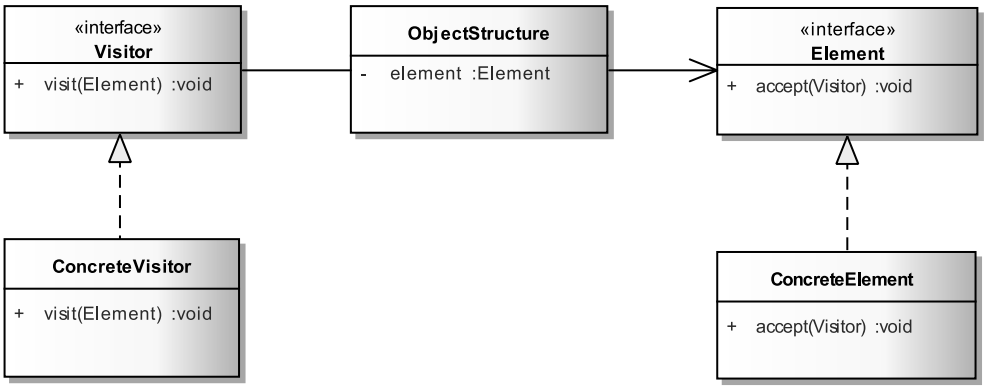

Рис. 22.18. *Базовая реализация шаблона Visitor*

из них в классе **ConcreteVisior** будет реализован метод **visit()**, определяющий именно его алгоритм.

/ # 66 # определение и реализация метода для каждого подкласса класса Element # Visitor.java # ConcreteVisitor.java */*

```
package by.bsu.visitor.base;
public interface Visitor {
          void visit (Element element);
}
package by.bsu.visitor.base;
public class ConcreteVisitor implements Visitor {
          public void visit (Element element) { // реализация
          }
}
```

Шаблон **Visitor** легко расширяем для новых операций посредством простой реализации интерфейса новым классом с соответствующей реализацией метода.

// # 67 # структура, инкапсулирующая класс Element # ObjectStructure.java

```
package by.bsu.visitor.base;
public class ObjectStructure {
          private Element element;
          // some methods
}
```

Класс **ObjectStructure** может агрегировать один или группу экземпляров типа **Element** с определенным набором активации шаблона.

/ # 68 # представление, над которым выполняет действия Visitor # Element.java */*

```
package by.bsu.visitor.base;
public interface Element {
          void accept (Visitor visitor);
}
```

/ # 69 # реализация метода accept() в конкретной сущности системы # ConcreteElement.java */*

```
package by.bsu.visitor.base;
public class ConcreteElement implements Element {
          public void accept (Visitor visitor) {
                    visitor.visit();
          }
}
```

Для представления системы проката используются два класса — **ApplianceRenting** и **DVDRenting**. Причем оба реализуют интерфейс **Visitable**

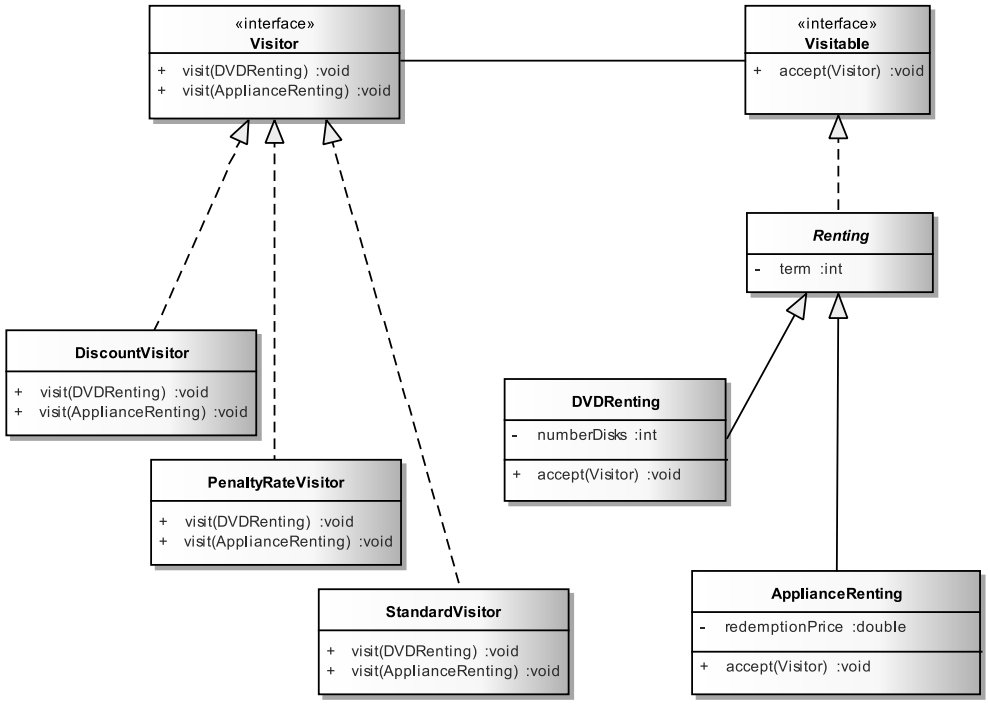

Рис. 22.19. *Пример реализации шаблона Visitor*

через абстрактный класс **Renting**, инкапсулирующий длительность проката. Метод **accept()** призван рассчитать стоимость обслуживания в зависимости от способов и условий проката. Способы и условия могут быть стандартными, льготными и штрафными. Возможно появление новых условий. Классы, реализующие интерфейс **Visitor,** представляют конкретные алгоритмы расчета для каждого сочетания предлагаемой услуги и способа ее приобретения.

/ # 70 # Visitor и его реализации # Visitor.java # StandardVisitor.java # DiscountVisitor.java # PenaltyRateVisitor.java */*

```
package by.bsu.visitor.action;
import by.bsu.visitor.entity.ApplianceRenting;
import by.bsu.visitor.entity.DVDRenting;
public interface Visitor {
          void visit(DVDRenting service);
          void visit(ApplianceRenting service);
}
package by.bsu.visitor.action;
import by.bsu.visitor.entity.ApplianceRenting;
import by.bsu.visitor.entity.DVDRenting;
public class StandardVisitor implements Visitor {
```

```
        @Override
        public void visit(DVDRenting service) {
                System.out.println("Standard DVD renting service");
        }
        @Override
        public void visit(ApplianceRenting service) {
                System.out.println("Standard Appliance renting service");
        }
}
package by.bsu.visitor.action;
import by.bsu.visitor.entity.ApplianceRenting;
import by.bsu.visitor.entity.DVDRenting;
public class DiscountVisitor implements Visitor {
        @Override
        public void visit(DVDRenting service) {
                System.out.println("Discount DVD renting service");
        }
        @Override
        public void visit(ApplianceRenting service) {
                System.out.println("Discount Appliance renting service");
        }
}
package by.bsu.visitor.action;
import by.bsu.visitor.entity.ApplianceRenting;
import by.bsu.visitor.entity.DVDRenting;
public class PenaltyRateVisitor implements Visitor{
        @Override
        public void visit(DVDRenting service) {
                System.out.println("Penalty DVD renting service");
        }
        @Override
        public void visit(ApplianceRenting service) {
                System.out.println("Penalty Appliance renting service");
        }
}
```

/ # 71 # реализация иерархии сущностей с настраиваемым поведением # Visitable.java # Renting.java # DVDRenting.java # ApplianceRenting.java */*

```
package by.bsu.visitor.action;
public interface Visitable {
        void accept(Visitor v);
}
package by.bsu.visitor.entity;
import by.bsu.visitor.action.Visitable;
public abstract class Renting implements Visitable {
        private int term;
        public int getTerm() {
                return term;
        }
```

```
        public void setTerm(int term) {
                this.term = term;
        }
}
package by.bsu.visitor.entity;
import by.bsu.visitor.action.Visitor;
public class DVDRenting extends Renting {
        private int numberDisks;
        public int getNumberDisks() {
                return numberDisks;
        }
        public void setNumberDisks(int numberDisks) {
                this.numberDisks = numberDisks;
        }
        @Override
        public void accept(Visitor v) {
                // some code here
                v.visit(this);
        }
}
package by.bsu.visitor.entity;
import by.bsu.visitor.action.Visitor;
public class ApplianceRenting extends Renting {
        private double redemptionPrice;
        public double getRedemptionPrice() {
                return redemptionPrice;
        }
        public void setRedemptionPrice(double redemptionPrice) {
                this.redemptionPrice = redemptionPrice;
        }
        @Override
        public void accept(Visitor v) {
                // some code here
                v.visit(this);
        }
}
```

Если реализация методов **accept()** во всех классах иерархии будет одинаковой, то нельзя прибегать к услугам наследования методов и убирать реализацию из подклассов. При вызове метода **accept()** суперкласса экземпляр, передаваемый в метод **visit()**, также будет экземпляром суперкласса, тогда как необходимо передавать экземпляр класса, вызывающего метод.

Применение шаблона позволяет избавить иерархию **Element** от насыщения большим количеством логического функционала или сложной конфигурации.

При определении новых подклассов-наследников **Visitor** желательно добавлять функциональность ко всем классам иерархии. Однако существует проблема: переопределить придется метод **visit()** для каждого типа в иерархии. Чтобы

этого избежать, можно по умолчанию определить класс **AbstractVisitor** и все тела его методов **visit()** оставлять пустыми.

Шаблон хорошо работает, если иерархия классов типа **Element** остается неизменной. Если добавляются новые подклассы, то обычно это ведет к изменению в классах типа **Visitor**, причем довольно значительных.

Шаблон Interpreter

Существует много подходов к решению сложной задачи, состоящей из некоторого количества подзадач. Подзадачи в такой ситуации каким-то образом связаны между собой. Для описания межзадачных связей создается некоторое подобие простого языка, с помощью которого можно переопределить исходную задачу целиком.

Деление на части преследует цель упрощения решения как подзадачи, так и, в конечном итоге, возможность на основе решения всех подзадач составить решение основной проблемы.

Шаблон **Interpreter** задает описания подзадач с помощью некоторого простого языка после выполнения декомпозиции сложной задачи на ряд небольших. Полученные в итоге выражения обрабатываются и решаются другой частью шаблона с построением некоторого синтаксического дерева.

Интерфейс **AbstractExpression** задает основные правила взаимодействия с клиентом. Его реализации типа **TerminalExpression** представляют листья синтаксической обработки выражений. Класс **NoTerminalExpression** отличается от терминальных классов тем, что содержит ссылку на следующий объект типа **TerminalExpression** и вызывает, если необходимо, методы **interpret()** для других подклассов. В класс **Context** помещается информация, передаваемая при необходимости интерпретатору для выполнения действий. Роль объекта типа **Client** заключается в инициализации и предоставлении объектов типа **AbstractExpression**, для составления конкретного выражения.

/ # 72 # базовая реализация Interpreter # AbstractExpression.java # TerminalExpression.java */*

```
package by.bsu.interpreter.base;
public interface AbstractExpression {
          public void interpret ();
}
package by.bsu.interpreter.base;
public class TerminalExpression implements AbstractExpression {
          public void interpret () { // реализация
          }
}
```

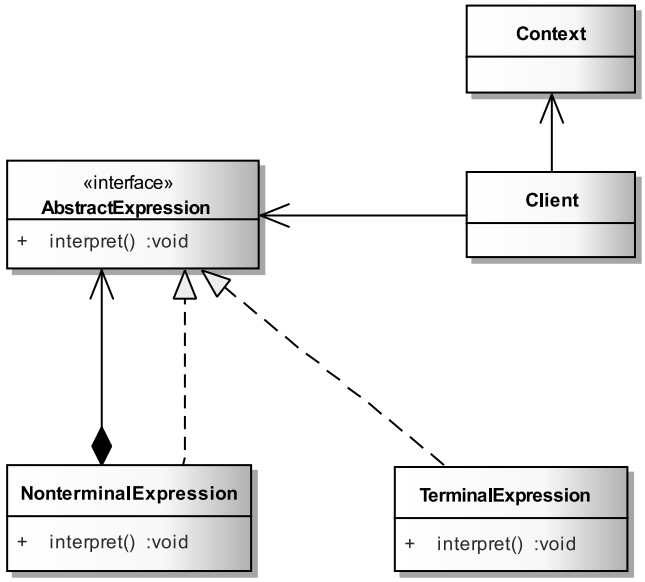

Рис. 22.20. *Базовая реализация шаблона Interpreter*

/ # 73 # базовая реализация Interpreter # NonterminalExpression.java */*

```
package by.bsu.interpreter.base;
public class NonterminalExpression implements AbstractExpression {
          private AbstractExpression expression;
          public void interpret () { // some code here
          }
}
```

/ # 74 # базовая реализация Interpreter # Context.java */*

```
package by.bsu.interpreter.base;
public class Context { // some code here
}
```

/ # 75 # базовая реализация Interpreter # Client.java */*

```
package by.bsu.interpreter.base;
public class Client {
          private AbstractExpression expression;
          private Context context;
          // some code here
}
```

Интерпретатор был создан в первую очередь для обработки регулярных выражений, изменения действия логических, арифметических и алгебраических операций.

Пусть дано некоторое простое математическое выражение, записанное в виде «8 2 7 4 + * -», что эквивалентно более общепринятому выражению

«(4+7)*2-8». Задача представляет собой синтаксический анализ и интерпретацию исходного выражения для вычисления результата.

В класс **Context** будут помещаться исходные числовые значения выражения, а также результаты промежуточных вычислений и конечный результат.

/ # 76 # реализация класса Context.java */*

```
package by.bsu.interpreter;
import java.util.ArrayDeque;
public class Context {
          private ArrayDeque<Integer> contextValues = new ArrayDeque<>();
          public Integer popValue() {
                    return contextValues.pop();
          }
          public void pushValue(Integer value) {
                    this.contextValues.push(value);
          }
}
```

В терминальных классах, ассоциированных с математическими действиями, метод **interpret()** извлекает значения из объекта **Context**, выполняет действие и записывает результат в тот же объект. Реализация для чисел только добавляет число в контекст.

/ # 77 # реализация вычислительной логики Interpreter # AbstractMathExpression.java # NonterminalExpressionNumber.java # TerminalExpressionPlus.java # TerminalExpressionMinus.java # TerminalExpressionMultiply.java # TerminalExpressionDivide.java */*

```
package by.bsu.interpreter;
public abstract class AbstractMathExpression {
          public abstract void interpret(Context context);
}
package by.bsu.interpreter;
public class NonterminalExpressionNumber extends AbstractMathExpression {
          private int number;
          public NonterminalExpressionNumber(int number) {
                    this.number = number;
          }
          @Override
          public void interpret(Context c) {
                    c.pushValue(number);
          }
}
package by.bsu.interpreter;
public class TerminalExpressionDivide extends AbstractMathExpression {
          @Override
          public void interpret(Context c) {
                    c.pushValue((c.popValue() / c.popValue()));
```

```
        }
}
package by.bsu.interpreter;
public class TerminalExpressionMinus extends AbstractMathExpression {
        @Override
        public void interpret(Context c) {
                c.pushValue(c.popValue() - c.popValue());
        }
}
package by.bsu.interpreter;
public class TerminalExpressionMultiply extends AbstractMathExpression {
        @Override
        public void interpret(Context c) {
                c.pushValue(c.popValue() * c.popValue());
        }
}
package by.bsu.interpreter;

public class TerminalExpressionPlus extends AbstractMathExpression {
        @Override
        public void interpret(Context c) {
                c.pushValue(c.popValue() + c.popValue());
        }
}
```

В классе **Client** происходит синтаксический разбор исходной задачи (выражения) и в зависимости от результатов анализа инициализация соответствующих терминальных объектов и организация их в виде списка. Метод **calculate()** отвечает за сборку окончательного решения на основе выполнения элементарных задач, которые инициируются запуском метода **interpret(),** соответствующего извлеченному из списка терминальному объекту.

/ # 78 # реализация синтаксической логики Interpreter и запуск процесса # Client.java # InterpreterRunner.java */*

```
package by.bsu.interpreter;
import java.util.ArrayList;
import java.util.Scanner;
public class Client {
        private ArrayList<AbstractMathExpression> listExpression;
        public Client(String expression) {
                listExpression = new ArrayList<>();
                parse(expression);
        }
        private void parse(String expression) { // синтаксический анализ
                for (String lexeme : expression.split("\\p{Blank}+")) {
                        if (lexeme.isEmpty()) {
                                continue;
                        }
```

```
                    char temp = lexeme.charAt(0);
                    switch (temp) {
                    case '+':
                              listExpression.add(new TerminalExpressionPlus());
                              break;
                    case '-':
                              listExpression.add(new TerminalExpressionMinus());
                              break;
                    case '*':
                              listExpression.add(new TerminalExpressionMultiply());
                              break;
                    case '/':
                              listExpression.add(new TerminalExpressionDivide());
                              break;
                    default:
                              Scanner scan = new Scanner(lexeme);
                              if (scan.hasNextInt()) {
                                   listExpression.add(
                                     new NonterminalExpressionNumber(scan.nextInt()));
                              }
                    }
          }
     }
     public Number calculate() {
          Context context = new Context();
          // выполнение простых задач и сборка результата
          for (AbstractMathExpression terminal : listExpression) {
                    terminal.interpret(context);
          }
          return context.popValue();
     }
}
package by.bsu.interpreter;
public class InterpreterRunner {
     public static void main(String[] args) {
          String expression = "8 2 7 4 + * -"; // expression in polska form
          Client interpreter = new Client(expression);
          System.out.println("[ " + expression + " ] = " + interpreter.calculate());
     }
}
```

В результате будет выведено:

[8 2 7 4 + * -] = 14

что и соответствует более общепринятому выражению (4+7)*2-8 = 14.

Интерпретатор легко заменяется при изменении правил работы с выражениями.

Задания к главе 22

В любой задаче кроме предложенного шаблона можно использовать при необходимости и другие шаблоны.

1. Паттерн Interpreter. Создать иерархию выражений и их логического анализа операциями «И», «ИЛИ», «EQUALS», «ОТРИЦАНИЕ», «CONTAINS», «COMPARISON», «COMPOUND».
2. Паттерн Interpreter. Реализовать арифметические и логические операции для комплексных чисел.
3. Паттерн Interpreter. Изменить действие логических операторов на действие битовых операторов.
4. Паттерн Strategy. Разработать модель игровой системы. Предусмотреть наличие фентезийных персонажей: орки, тролли, пегасы, эльфы, вампиры, гарпии и др. Учесть, что некоторые персонажи ходят, другие — летают, третьи — и ходят и летают. Летать также может группа персонажей с помощью магии.
5. Паттерн Observer. Разработать систему Почтовое отделение. Из издательства в почтовое отделение поступают издаваемые газеты и журналы. Почтовое отделение отправляет полученные печатные издания соответствующим подписчикам.
6. Паттерн Strategy. Разработать модель выбора способов сортировки и поиска максимального/минимального значения массива числовых объектов.
7. Паттерн State. Заказ на получение гранта для обучения может находиться в нескольких состояниях: создан, рассматривается, отложен, отклонен, подтвержден, отозван и т. д. Определить логику изменения состояний и разработать модель системы.
8. Паттерн State. Учебное задание, выполняемое студентом, может находиться в состояниях: выдано, выполнено, сдано на проверку, проверено, пересдано на проверку, не выполнено. Определить логику изменения состояний и разработать модель системы.
9. Паттерн Chain of responsibility. Прохождение платежа через банковскую систему сопровождается целым рядом действий: фиксирующих, контролирующих, снимающих процент банка и прочие вычеты и действия. Построить цепочки для различного вида платежей (обычных, льготных, государственных, внутрибанковских) в соответствии с предметной областью и разработать модель системы.
10. Паттерн Memento. Реализовать алгоритм игры судоку. Реализовать возможность «взять назад ход».
11. Паттерн Memento. Реализовать алгоритм игры «крестики-нолики». Реализовать возможность «взять назад ход».
12. Паттерн Memento. Существует набор статей в википедии. Реализовать процесс раздачи статей по требованию для изменения, сохраняя исходный вариант для возможного восстановления статьи в исходном виде.

Глава 23

СТРУКТУРНЫЕ ШАБЛОНЫ

Структурные шаблоны GoF отвечают за композицию объектов и классов, и не только за объединение частей приложения, но и за их разделение.

К структурным шаблонам относятся:

Adapter (**Адаптер**) — применяется при необходимости использовать классы вместе с несвязанными интерфейсами. Поведение адаптируемого класса при этом изменяется на необходимое (interface to an object);

Bridge (**Мост**) — разделяет представление класса и его реализацию, позволяя независимо изменять то и другое (implementation of an object);

Composite (**Компоновщик**) — группирует объекты в иерархические древовидные структуры и позволяет работать с единичным объектом так же, как с группой объектов (structure and composition of an object);

Decorator (**Декоратор**) — представляет способ изменения поведения объекта без создания подклассов. Позволяет использовать поведение одного объекта в другом (responsibilities of an object without subclassing);

Facade (**Фасад**) — создает класс с общим интерфейсом высокого уровня к некоторому числу интерфейсов в подсистеме (interface to a subsystem);

Flyweight (Легковес) — разделяет свойства класса для оптимальной поддержки большого числа мелких объектов (storage costs of objects);

Proxy (**Заместитель**) — подменяет сложный объект более простым и осуществляет контроль доступа к нему (how an object is accessed... its location).

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 может быть выделена иерархия абстракций и независимая иерархия реализаций. Точное соответствие между абстракциями и реализациями в общем случае невозможно. Обычно абстракция определяет операции более высокого уровня, чем реализация.

Шаблон позволяет снизить общее число классов за счет того, что несколько абстракций могут использовать одну реализацию, в простейшем случае определяя ее как ссылку на интерфейс в иерархии абстракций.

Реализация во время выполнения при необходимости может быть изменена.

Каркас базовой реализации шаблона представлен в виде:

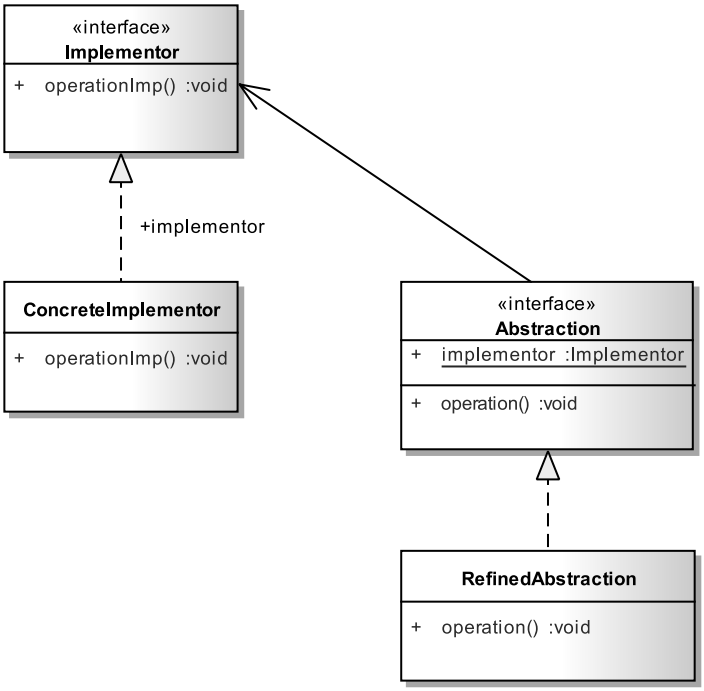

Рис. 23.1. *Базовая реализация шаблона Bridge*

/ # 1 # Implementors # Implementor.java # ConcreteImplementor.java */*

```
package by.bsu.bridge.base;
public interface Implementor {
    void operationImp();
}
package by.bsu.bridge.base;
public class ConcreteImplementor implements Implementor {
    public void operationImp() { // more code
    }
}
```

/ # 2 # Abstractions # Abstraction.java # RefinedAbstraction.java */*

```
package by.bsu.bridge.base;
public interface Abstraction {
    public static Implementor implementor;
    public void operation();
}
package by.bsu.bridge.base;
public class RefinedAbstraction implements Abstraction {
    public void operation() { // more code
    }
}
```

Элементы шаблона:

1. **Abstraction** — собственно абстракция. Определяет базовый интерфейс абстракции и агрегирует объект типа **Implementor**;
2. **RefinedAbstraction** — уточненный элемент абстракции. Подкласс основной абстракции, реализующий интерфейс ею определенный;
3. **Implementor** — исполнитель. Определяет интерфейс для классов реализации. Обычно интерфейс **Implementor** содержит только элементарные операции, а тип **Abstraction** определяет операции более высокого уровня или композиции элементарных операций;
4. **ConcreteImplementor** — конкретный исполнитель, содержащий конкретную реализацию интерфейса, определяемого типом **Implementor**.

Пусть существует некое банковское учреждение, совершающее действия по кредитным, депозитным и прочим счетам. Экземпляры счетов-**Abstraction** выполняют действия. Действия могут быть обычными, срочными и другими. Срочное обслуживание стоит клиенту дороже, так как повышаются ежемесячные платежи, теряются проценты и проч. Видов счетов в реальной банковской сфере существует достаточно много, как и действий, производимых над ними.

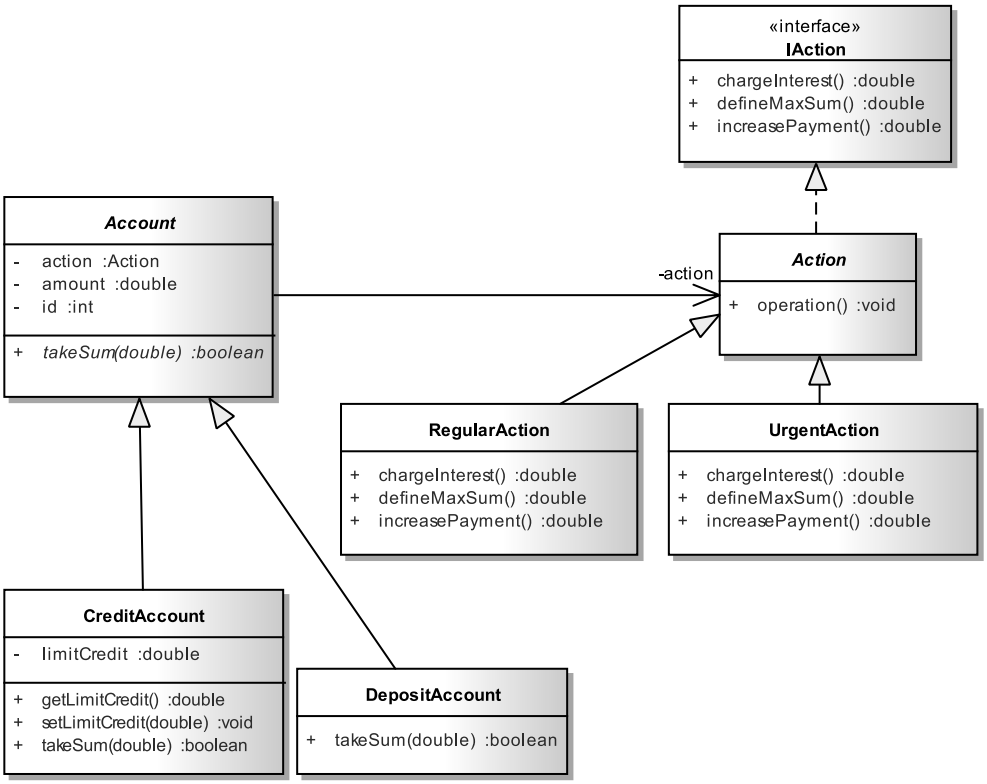

Рис. 23.2. *Реализация шаблона Bridge*

Если каждому типу счета ставить в соответствие свою реализацию действия, то общее число классов, требуемых к созданию, будет равно произведению числа типов счетов на число возможных действий. Добавление же одного нового типа счета, потребует создания классов в количестве, равном числу действий.

Классы **Action-Implementor** отделены от **Account-Abstraction**, причем так, чтобы реализации ничего не знали в идеале об абстракциях.

/ # 3 # Implementors # IAction.java # Action.java # RegularAction.java # UrgentAction.java */*

```
package by.bsu.bridge.bank;
public interface IAction {
          double chargeInterest();
          double defineMaxSum();
          double increasePayment();
}
package by.bsu.bridge.bank;
public abstract class Action implements IAction {
          // поля и методы, общие для всех реализаций
          public void operation() { // more code
          }
}
package by.bsu.bridge.bank;
public class RegularAction extends Action {
          private final static int MAX_SUM = 100; // read from base
          private final static int NORMAL_INTEREST = 3; // read from base
          @Override
          public double chargeInterest() {
                    // charge NORMAL interest on account"
                    return NORMAL_INTEREST;
          }
          @Override
          public double defineMaxSum() {
                    // max sum is unbounded"
                    return MAX_SUM;
          }
          @Override
          public double increasePayment() {
                    return 0; // stub
          }
}
package by.bsu.bridge.bank;
public class UrgentAction extends Action {
          final static int MONTHLY_PAYMENT = 10; // read from base
          private final static int MAX_SUM = 50; // read from base
          @Override
          public double chargeInterest() {
                    // charge LOW interest on accounts
```

```
                return 0; // stub
        }
        @Override
        public double defineMaxSum() {
                // check credit
                // max sum is bounded"
                return MAX_SUM;
        }
        @Override
        public double increasePayment() {
                // MAX increase in monthly payments
                return MONTHLY_PAYMENT;
        }
}
```

Класс **Action** — абстрактный, реализующий **Implementor** (**IAction**). Может содержать общие методы для всех действий, например: проверку прав пользователя, блокировку счета и т. д. Классы **RegularAction** и **UrgentAction** уточняют подклассы класса **Action**.

/ # 4 # абстракция и ее уточнения # Account.java # DepositAccount.java # CreditAccount.java */*

```
package by.bsu.bridge.bank;
public abstract class Account {
    private int id;
    private double amount;
    private Action action;
    protected Account(Action action) {
          this.action = action;
    }
    public Action getAction() {
          return action;
    }
    protected void setAction(Action action) {
          this.action = action;
    }
    public int getId() {
          return id;
    }
    public void setId(int id) {
          this.id = id;
    }
    public double getAmount() {
          return amount;
    }
    public void setAmount(double amount) {
          this.amount = amount;
    }
    public abstract boolean takeSum(double sum);
```

```
}
package by.bsu.bridge.bank;
public class DepositAccount extends Account {
          public DepositAccount(Action action) {
                    super(action);
          }
          @Override
          public boolean takeSum(double sum) {
                    System.out.println("Performing by deposit account:");
                    double interest = getAction().chargeInterest();
                    double maxSum = getAction().defineMaxSum();
                    // check amount
                    System.out.print("accountID: "+ getId() +" : interest is " + interest);
                    System.out.print(" : recording of changes in the state accounts");
                    System.out.println(": withdrawal : " + sum);
                    return true;
          }
}
package by.bsu.bridge.bank;
public class CreditAccount extends Account {
          private double limitCredit;
          public double getLimitCredit() {
                    return limitCredit;
          }
          public void setLimitCredit(double limitCredit) {
                    this.limitCredit = limitCredit;
          }
          public CreditAccount(Action action) {
                    super(action);
          }
          public boolean takeSum(double sum) {
                    System.out.println("Performing by credit account:");
                    double maxSum = getAction().defineMaxSum();
                    double payment = getAction().increasePayment();
                    System.out.print("accountID: "+ getId() +" increase monthly payments: "
                                                                        + payment);
                    System.out.print(": recording of changes in the state accounts");
                    System.out.println(" : withdrawal :" + sum);
                    return true;
          }
}
```

Класс **Account** — абстракция, классы **DepositAccount** и **CreditAccount** — уточненные абстракции.

/ # 5 # использование шаблона Bridge # BridgeClient.java */*

```
package by.bsu.bridge.bank;
public class BridgeClient {
          public static void main(String[] args) {
```

```
                Action action = new RegularAction();
                DepositAccount depositAccount = new DepositAccount(action);
                depositAccount.setId(777);
                depositAccount.setAmount(1500);
                depositAccount.takeSum(200);
                action = new UrgentAction();
                depositAccount.setAction(action); // replacement action
                depositAccount.takeSum(100);

                new CreditAccount(action).takeSum(50);
        }
}
```

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 могут иметь любое число подклассов.

Ниже приведен более простой пример использования шаблона **Bridge** при создании собственного логгера. Реализация может быть применена и для разработки с собственным псевдо-логгером.

/ # 6 # Implementors # LoggerImplementor.java # MultiThreadedLogger.java # SingleThreadedLogger.java */*

```
package by.bsu.bridge.logger;
public interface LoggerImplementor {
        void logToConsole();
        void logToFile();
        void logToSocket();
}
package by.bsu.bridge.logger;
public class MultiThreadedLogger implements LoggerImplementor {
        @Override
        public void logToConsole() {
                System.out.println("Multithreaded console log");
        }
        @Override
        public void logToFile() {
                System.out.println("Multithreaded file log");
        }
        @Override
        public void logToSocket() {
                System.out.println("Multithreaded socket log");
        }
}
package by.bsu.bridge.logger;
public class SingleThreadedLogger implements LoggerImplementor {
        @Override
        public void logToConsole() {
```

```
                System.out.println("Singlethreaded console log");
        }
        @Override
        public void logToFile() {
                System.out.println("Singlethreaded file log");
        }
        @Override
        public void logToSocket() {
                System.out.println("Singlethreaded socket log");
        }
}
```

/ # 7 # абстракция и ее уточнения # Logger.java # ConsoleLogger.java # FileLogger.java */*

```
package by.bsu.bridge.logger;
public abstract class Logger {
        protected LoggerImplementor logger;
        public Logger() {
        }
        public Logger(LoggerImplementor logger) {
                this.logger = logger;
        }
        public void setLogger(LoggerImplementor logger) {
                this.logger = logger;
        }
        public abstract void log();
}
package by.bsu.bridge.logger;
public class ConsoleLogger extends Logger {
        public ConsoleLogger() {
        }
        public ConsoleLogger(LoggerImplementor logger) {
                super(logger);
        }
        public void log(){
                logger.logToConsole();
        }
}
package by.bsu.bridge.logger;
public class FileLogger extends Logger {
        public FileLogger() {
        }
        public FileLogger(LoggerImplementor logger) {
                super(logger);
        }
        public void log(){
                logger.logToFile();
        }
}
```

/ # 8 # использование шаблона Bridge # BridgeClientMain.java */*

```
package by.bsu.bridge.logger;
public class BridgeClientMain {
          public static void main(String[] args) {
                    LoggerImplementor loggerImpl = new SingleThreadedLogger();
                    Logger logger = new ConsoleLogger(loggerImpl);
                    logger.log();
                    loggerImpl = new MultiThreadedLogger();
                    logger.setLogger(loggerImpl);
                    logger.log();
                    new FileLogger(loggerImpl).log();
          }
}
```

Шаблон Decorator

Расширяет функциональные возможности объекта, изменяя его поведение. Расширяющий класс реализует тот же самый интерфейс, что и исходный класс, делегируя исходному классу выполнение базовых операций. Может добавлять собственные операции. Представляет собой альтернативу множественному наследованию, то есть может добавлять функциональность классу, от которого

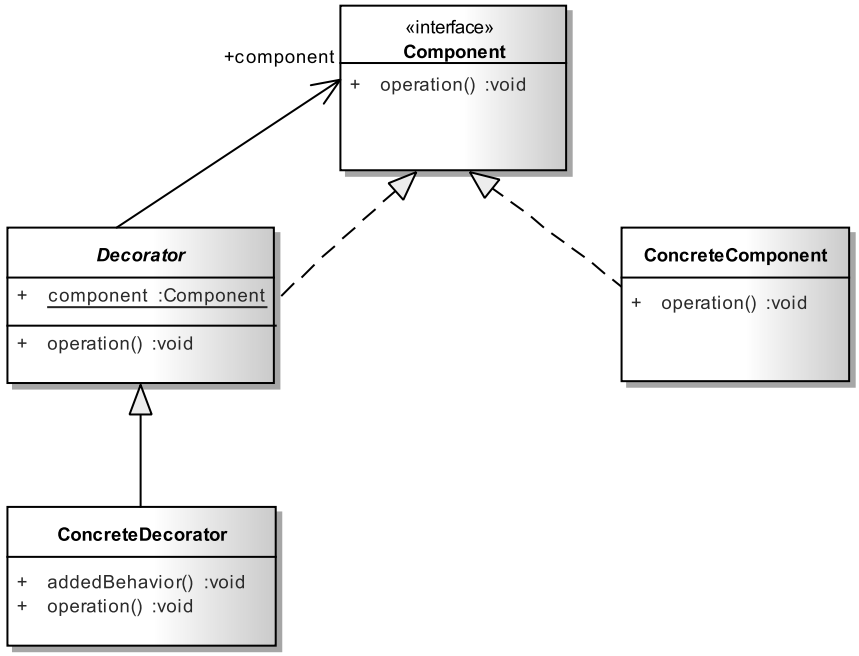

Рис. 23.3. *Базовая реализация шаблона Decorator*

нельзя наследоваться. Добавляемая функциональность может быть легко исключена при переработке кода. Шаблон **Decorator** позволяет динамически изменять поведение экземпляров в процессе выполнения приложения.

Каркас базовой реализации шаблона представлен в виде:

```
/* # 9 # декорируемые типы # Component.java # ConcreteComponent.java */

package by.bsu.decorator.base;
public interface Component {
    void operation();
}
package by.bsu.decorator.base;
public class ConcreteComponent implements Component {
    public void operation() { // more code
    }
}
```

```
/* # 10 # абстракция декоратора и конкретный декорируемый тип # Decorator.java #
ConcreteDecorator.java */

package by.bsu.decorator.base;
public abstract class Decorator implements Component {
    public static Component component;
    public void operation(){ // more code
    }
}
package by.bsu.decorator.base;
public class ConcreteDecorator extends Decorator {
    public void addedBehavior() { // more code
    }
    public void operation() { // использует реализацию типа Component и метод addBehavior()
    }
}
```

Элементы шаблона:

1) **Component** — определяет базовый интерфейс декорируемого типа;
2) **ConcreteComponent** — декорируемый тип с реализацией базовых операций. Таких классов может быть несколько;
3) **Decorator** — агрегирует декорируемый тип **Component** и наследует его реализацию. Определяет интерфейс для подклассов декораторов;
4) **ConcreteDecorator** — конкретный декоратор, содержащий конкретную реализацию интерфейса, определяемого типом **Decorator**, может объявлять дополнительное поведение. Может использовать как агрегированный тип, так и переопределять унаследованный интерфейс.

Позволяет уменьшить число подклассов по сравнению с аналогичным решением без использования декоратора. Однако при большом количестве классов-декораторов решение резко утяжеляется, становится малопонятным и утрачивает все преимущества от коротких применений шаблона.

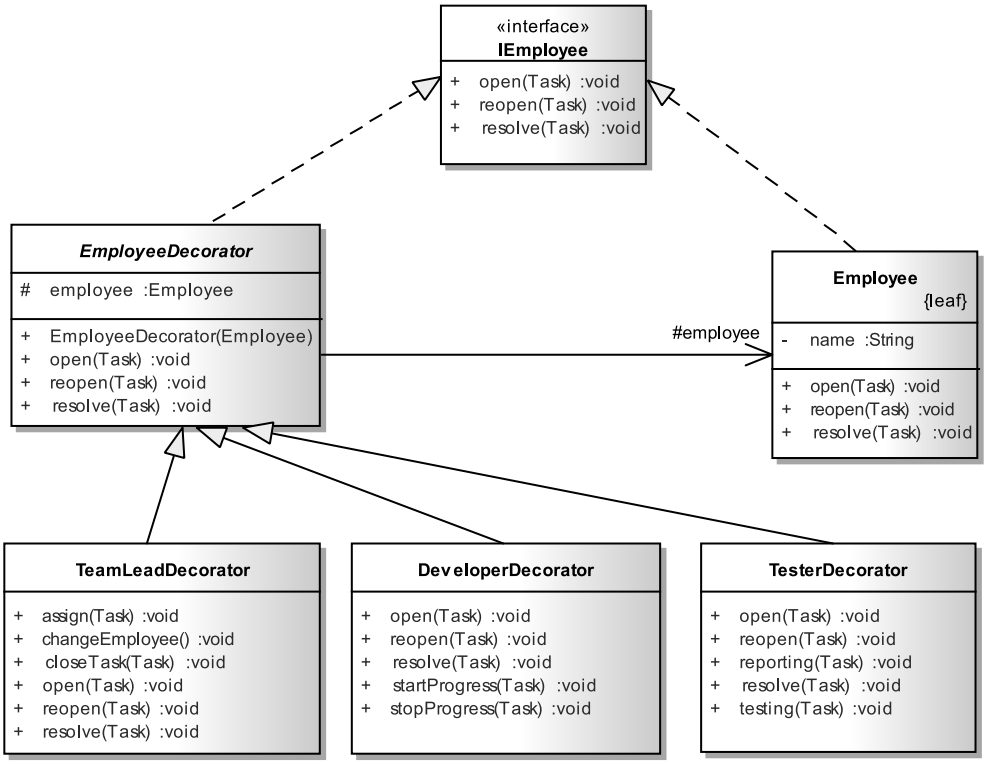

Рис. 23.4. *Реализация шаблона Decorator*

В качестве примера можно рассмотреть систему управления заданиями в IT-проекте. Система предназначена для визуализации процесса выполнения отдельных частей проекта участниками с различными профессиональными навыками. Проект выполняется сотрудниками, которые могут выполнять общие для всех действия по открытию задания для выполнения, выставлению пометки о выполнении, «переоткрытию» задания в случае, например, нахождения ошибки тестировщиком. Каждое из этих действий может иметь дополнительные особенности, зависящие от роли сотрудника в проекте. Использование шаблона **Decorator** позволяет учесть эти особенности без построения дополнительной иерархии сотрудников.

/ # 11 # определение интерфейса для общих действий # IEmployee.java */*

```
package by.bsu.decorator;
public interface IEmployee { // может быть представлен абстрактным классом
          void openTask();
          void reopenTask();
          void resolveTask();
}
```

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

/ # 12 # класс-декоратор для интерфейса IEmployee # EmployeeDecorator.java */*

```
package by.bsu.decorator;
public abstract class EmployeeDecorator implements IEmployee {
          protected Employee employee;
          public EmployeeDecorator() {
                    super();
          }
          public EmployeeDecorator(Employee employee) {
                    this.employee = employee;
          }
          @Override
          public void resolveTask() {
                    employee.resolveTask();
          }
          @Override
          public void openTask() {
                    employee.openTask();
          }
          @Override
          public void reopenTask() {
                    employee.reopenTask();
          }
}
```

Класс **Employee** определяет класс, функциональность которого будет расширена за счет применения декоратора. Сам класс в общем случае может даже запрещать наследование, то есть быть объявленным как **final**.

/ # 13 # декорируемый класс, наследования которого нежелательны # Employee.java */*

```
package by.bsu.decorator;
public class Employee implements IEmployee {
          private String name;
          public Employee() {
          }
          public Employee(String name) {
                    this.name = name;
          }
          public String getName() {
                    return name;
          }
          @Override
          public void openTask() {
                    System.out.println(this.getName() + " open task");
          }
```

```
        @Override
        public void reopenTask() {
                System.out.println(this.getName() + " reopen task");
        }
        @Override
        public void resolveTask() {
                System.out.println(this.getName() + " resolve task");
        }
}
```

Класс **DeveloperDecorator** объявляет дополнительные функциональности **startProgress()** и **endProgress()**, необходимые для разработчика, дополняя (декорируя) функциональности **openTask()**, **reopenTask()**, **resolveTask()** класса **Employee**.

/ # 14 # класс-декоратор, уточняющий базовую функциональность сотрудника # DeveloperDecorator.java */*

```
package by.bsu.decorator;
public class DeveloperDecorator extends EmployeeDecorator {
        // поля, методы
        public DeveloperDecorator(Employee employee) {
                super(employee);
        }
        @Override
        public void openTask() {
                super.openTask();
                startProgress();
        }
        @Override
        public void reopenTask() {
                super.reopenTask();
                startProgress();
        }
        @Override
        public void resolveTask() {
                super.resolveTask();
                stopProgress();
        }
        public void startProgress() {
                System.out.println(employee.getName() + " starting task");
        }
        public void stopProgress() {
                System.out.println(employee.getName() + " stopping task");
        }
}
```

Классы **TesterDecorator** и **TeamLeadDecorator** каждый в свою очередь добавляют функциональность, свойственную его деятельности, но никак не меняющую функциональность основного класса **Employee**.

/ # 15 # классы-декоратор сотрудника тестировщика и team-лидера # TesterDecorator.java # TeamLeadDecorator.java */*

```
package by.bsu.decorator;
public class TesterDecorator extends EmployeeDecorator{
          // поля, методы
          public TesterDecorator(Employee employee) {
                    super(employee);
          }
          @Override
          public void openTask() {
                    super.openTask();
                    testing();
          }
          @Override
          public void reopenTask() {
                    super.reopenTask();
                    testing();
          }
          @Override
          public void resolveTask() {
                    reporting();
                    super.resolveTask();
          }
          public void testing() {
                    System.out.println(employee.getName() + " testing task");
          }
          public void reporting() {
                    System.out.println(employee.getName() + " create report");
          }
}
package by.bsu.decorator;
public class TeamLeadDecorator extends EmployeeDecorator {
          // поля, методы
          public TeamLeadDecorator(Employee employee) {
                    super(employee);
          }
          @Override
          public void openTask() {
                    super.openTask();
                    assignTask();
          }
          @Override
          public void reopenTask() {
                    super.reopenTask();
                    changeEmployee();
          }
          @Override
          public void resolveTask() {
                    super.resolveTask();
```

```
                closeTask();
        }
        public void assignTask() {
                System.out.println(employee.getName() + " is assigning task");
        }
        public void changeEmployee() {
                System.out.println(employee.getName() + " is changing employee");
        }
        public void closeTask() {
                System.out.println(employee.getName() + " is closing task");
        }
}
```

Создав экземпляр класса **Employee,** можно делегировать ему выполнение задач, связанных с разработчиком, тестировщиком или team-лидером, без создания специализированных подклассов.

/ # 16 # использование шаблона Decorator # RunnerDecorator.java */*

```
package by.bsu.decorator;
public class RunnerDecorator {
        public static void main(String[ ] args) {
                IEmployee employee = new TesterDecorator(new Employee("Ivanov"));
                employee.reopenTask();
                employee = new TeamLeadDecorator(new Employee("Petrov"));
                employee.openTask();
        }
}
```

Шаблон Façade

Шаблон позволяет упростить доступ к сложной системе, объединив несколько действий различных классов под одним интерфейсом и делегировав ему обязанности отправлять сообщения этим классам. Клиент системы знает только об интерфейсе фасада и ничего не знает о структуре классов и последовательности вызовов, приводящих его к желаемому результату. В роли интерфейса в общем случае будет выступать совокупность классов, а не один класс со множеством обязанностей. Примером **Façade** может служить библиотека **mysql-connector-[***версия***].jar** для организации соединения и выполнения запросов к базе данных MySQL.

При разработке приложения часто приходится использовать классы/пакеты/библиотеки, созданные другими разработчиками. Причем не суть важно, являются ли эти программисты членами вашей команды или это библиотека от фирмы-производителя. При использовании таких библиотек для получения результата обычно необходимо выполнить вызов нескольких методов по определенному правилу. Причем обращение к этому функционалу может производиться довольно часто из разных частей кода программиста. В такой ситуации имеет смысл выделить этот интерфейс

в отдельный код, организованный в классы и пакеты отдельного класса и пользоваться им при необходимости, не усложняя собственный код.

Как частный случай решения в качестве **Façade** может выступать единственный класс с единственным методом. Такое примитивное решение уже конкурирует с аналогичными шаблонами. Существует опасность получить сильно связанный класс с громоздким «волшебным» методом.

/ # 17 # базовая реализация шаблона Facade # IFacade.java # Facade.java */*

```
package by.bsu.facade.base;
public interface IFacade {
          void generate();
          void find();
}
package by.bsu.facade.base;
public class Facade implements IFacade {
          private SecuritySystem securitySystem;
          private SubSystem subSystem;
          public Facade() {
                    // варианты инициализации могут быть разными
                    this.subSystem = new SubSystem();
                    this.securitySystem = new SecuritySystem();
          }
          @Override
          public void generate() {
                    // проверка пользователя и его прав
                    securitySystem.checkUser();
                    securitySystem.checkRights();
                    // действие create
                    subSystem.createNode();
          }
          @Override
          public void find() {
                    // проверка пользователя
                    securitySystem.checkUser();
                    // действие parse
                    subSystem.parse();
          }
}
```

Скрывать классы, используемые фасадом, не обязательно. Клиент может даже предоставлять данные для их инициализации.

/ # 18 # реализация шаблона Facade # SecuritySystem.java # SubSystem.java # ClientRunner.java */*

```
package by.bsu.facade.base;
public class SecuritySystem {
          boolean checkUser() {
```

```
                    return true;
          }
          boolean checkRights() {
                    return true;
          }
}
package by.bsu.facade.base;
public class SubSystem {
          public void parse() {
                    System.out.println("Parsing");
          }
          public void createNode() {
                    System.out.println("Creating program node");
          }
}
package by.bsu.facade.base;
public class ClientRunner {
          public static void main(String[] args) {
                    Facade facade = new Facade();
                    facade.generate();
          }
}
```

Снижается число связей клиентского класса с системой. Каждый метод фасада решает свою конкретную задачу, упрощая клиенту обращение к системе. При реализации фасада не следует допускать прямое обращение класса-клиента к классу системы, минуя фасад.

Например, в главе «XML & Java» в примере даны примеры кода для валидации XML-документа. Одну из вариаций можно записать в виде:

/ # 19 # проверка корректности документа XML без определения Façade # ValidatorSAXwithXSD.java */*

```
package by.bsu.valid;
import java.io.*;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
public class ValidatorSAXwithXSD {
          public static void main(String[ ] args) {
                    String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
                    String filename = "data/students.xml";
                    String schemaname = "data/students.xsd";
                    SchemaFactory factory = SchemaFactory.newInstance(language);
                    File schemaLocation = new File(schemaname);
```

```
                    try {
                              Schema schema = factory.newSchema(schemaLocation);
                              Validator validator = schema.newValidator();
                              Source source = new StreamSource(filename);
                              StudentErrorHandler sh = new StudentErrorHandler("log.txt");
                              validator.setErrorHandler(sh);
                              validator.validate(source);
                              System.out.println(filename + " validating is ended correctly");
                    } catch (SAXException e) {
                              // запись log
                    } catch (IOException e) {
                              // запись log
                    }
          }
}
```

Данный код выполняет некоторую последовательность обязательных действий, определенных разработчиком библиотеки. Чтобы решением можно было воспользоваться повторно, следует сделать фасад для него и фактически скрыть детали реализации за фасадом метода, а именно:

/ # 20 # вынесение реализации за Facade # FacadeValidator.java */*

```
package by.bsu.valid;
import java.io.File;
import java.io.IOException;
import javax.xml.XMLConstants;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class FacadeValidator {
          public boolean validateSAXwithXSD(String xmlFileName,
                              String schemaFileName, DefaultHandler handler) {
                    boolean result = false;
                    String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
                    SchemaFactory factory = SchemaFactory.newInstance(language);
                    File schemaLocation = new File(schemaFileName);
                    try {
                              Schema schema = factory.newSchema(schemaLocation);
                              Validator validator = schema.newValidator();
                              Source source = new StreamSource(xmlFileName);
                              validator.setErrorHandler(handler);
                              validator.validate(source);
                              result = true;
                    } catch (SAXException e) {
```

```
                    // запись log
                } catch (IOException e) {
                    // запись log
                }
                return result;
            }
}
```

Имена файлов и экземпляр класса-обработчика ошибок передаются в метод в качестве параметров. Метод же содержит стандартную последовательность действий, вмешиваться в которую нет разумных оснований.

Шаблон Composite

Предоставляет возможность строить сложные объекты с использованием рекурсии. Позволяет рассматривать объект как комбинацию более простых в целом составляющих древовидную структуру. Составной элемент представляет собой набор из частей с аналогичной природой. Часть целого в таком случае представляется как набор более мелких частей, и так до тех пор, пока не будет выделена некая элементарная часть. Элементарная часть уже неделима.

Примером может служить соотношение между текстом и его частями, текст может состоять из абзацев и листингов, последние, в свою очередь, — из предложений, предложения — из слов и знаков препинания, неделимый элемент — символ. В другом случае **Composite** может описывать карту местности, состоящей из областей, районов, городов и т. д.

Все классы из древовидной структуры реализуют общий интерфейс, тогда при создании любого объекта из любой части «ветки» действия будут выполняться идентичные и способ обращения ко всем элементам будет абсолютно одинаковым.

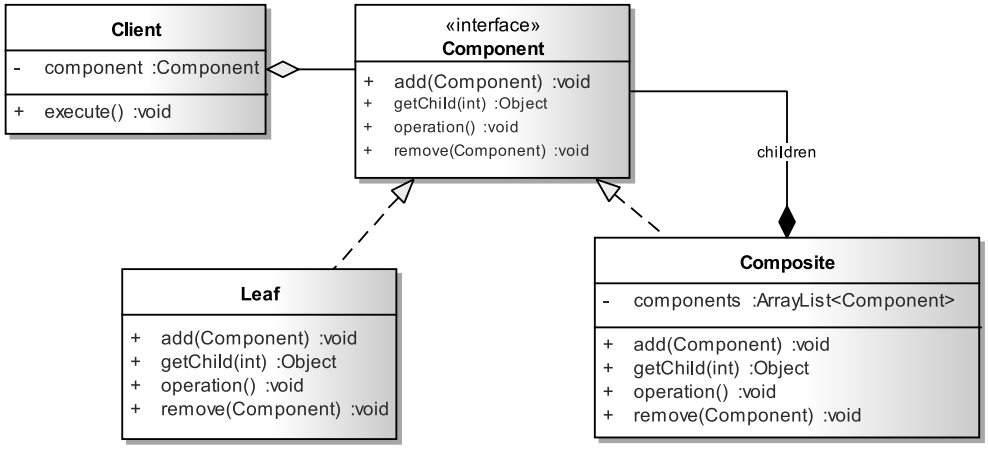

Рис. 23.5. *Базовая реализация шаблона Composite*

Интерфейс **Component** задает интерфейс для всех составных объектов. У интерфейса **Component** обычно существует одна или несколько реализаций типа **Leaf**, не имеющие потомков и описывающие неделимые элементы структуры. Тип **Composite** хранит составные и неделимые компоненты и определяет их поведение. Способ организации и поведения **Composite** зависит от задач, решаемых этим самым составным объектом.

/ # 21 # общий интерфейс древовидной структуры # Component.java */*

```
package by.bsu.composite.base;
public interface Component {
    void operation();
    void add(Component c);
    void remove(Component c);
    Object getChild (int index);
}
```

/ # 22 # неделимый элемент древовидной структуры # Leaf.java */*

```
package by.bsu.composite.base;
public class Leaf implements Component {
    public void operation() {
       System.out.println("Leaf -> Performing operation");
    }
    public void add(Component c) {
       System.out.println("Leaf -> add. Doing nothing");
       // генерация исключения и return false (если метод не void)
    }
    public void remove(Component c) {
       System.out.println("Leaf -> remove. Doing nothing");
       // генерация исключения и return false (если метод не void)
    }
    public Object getChild(int index) {
        throw new UnsupportedOperationException();
    }
}
```

/ # 22 # составной обьект # Composite.java */*

```
package by.bsu.composite.base;
import java.util.ArrayList;
public class Composite implements Component {
    private ArrayList<Component> components = new ArrayList<Component>();
    public void operation() {
        System.out.println("Composite -> Call children operations");
        int size = components.size();
        for (int i = 0; i < size; i++) {
            components.get(i).operation();
        }
```

```
    }
    public void add(Component component) {
        System.out.println("Composite -> Adding component");
        components.add(component);
    }
    public void remove(Component component) {
        System.out.println("Composite -> Removing component");
        components.remove(component);
    }
    public Object getChild(int index) {
        System.out.println("Composite -> Getting component");
        return components.get(index);
    }
}
```

/ # 23 # клиентский класс (необязателен), которому необходим составной в качестве поля # Client.java */*

```
package by.bsu.composite.base;
public class Client {
    private Component component;
    public Client(Component component) {
        this.component = component;
    }
    public void execute() {
        component.operation();
    }
}
```

/ # 24 # запуск процесса организации Composite # CompositeRunner.java */*

```
package by.bsu.composite.base;
public class CompositeRunner {
    public static void main(String[] args) {
        Component composite = new Composite();
        Component leaf = new Leaf();
        leaf.add(composite); // nothing happens;
        composite.add(leaf);
        Client client = new Client(composite);
        client.execute();
    }
}
```

Методы, управляющие добавлением/удалением/изменением любой составной части, не могут поддерживаться неделимыми частями в принципе.

Пусть необходимо разработать элементы организации набора каналов для загрузки больших информационных файлов, где пулы каналов могут состоять как из отдельных соединений разных видов, так и из пулов, которые, в свою очередь, могут состоять из отдельных каналов и других пулов. В этом случае явно необходима реализация рекурсивного включения одного пула в другой.

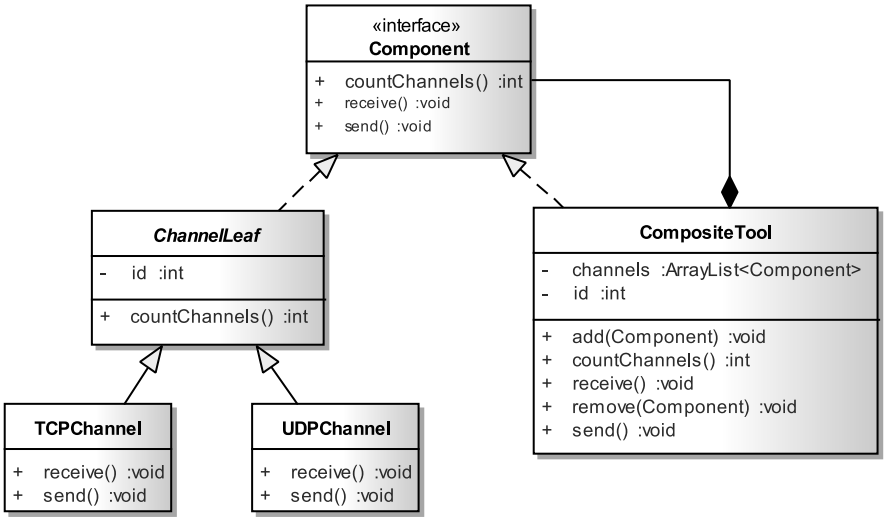

Рис. 23.6. *Реализация шаблона Composite*

В интерфейсе **Component** отсутствуют объявления методв **add()** и **remove()**, что исключает необходимость для классов типа **Leaf** предоствлять их реализацию.

Каналы могут собираться в **CompositTool**. Сам **CompositTool** может быть частью другого экземпляра **CompositTool**. Каналы и экземпляры **CompositTool** могут удаляться и добавляться в набор в любой момент по требованию пользователя.

/ # 25 # общий интерфейс для каналов и их наборов # Component.java */*

```
package by.bsu.composite.channel;
public interface Component {
    void send();
    void receive();
    int countChannels();
}
```

/ # 26 # реализация интерфейса для неделимых каналов # ChannelLeaf.java # TCPChannel.java # UDPChannel.java */*

```
package by.bsu.composite.channel;
package by.bsu.composite.channel;
public abstract class ChannelLeaf implements Component {
    private int id;
    public ChannelLeaf(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public int countChannels() {
```

```
        return 1;
    }
}
package by.bsu.composite.channel;
public class TCPChannel extends ChannelLeaf {
    public TCPChannel(int id) {
         super(id);
    }
   @Override
    public void send() {
         System.out.println("tcp send " + getId());
   }
   @Override
   public void receive() {
        System.out.println("tcp receive");
   }
}
package by.bsu.composite.channel;
public class UDPChannel extends ChannelLeaf {
    public UDPChannel(int id) {
        super(id);
    }
   @Override
   public void send() {
       System.out.println("udp send " + getId());
    }
   @Override
   public void receive() {
       System.out.println("udp receive");
    }
}
```

/ # 27 # реализация для составного элемента (Composite) # CompositeTool.java */*

```
package by.bsu.composite.channel;
import java.util.ArrayList;
public class CompositeTool implements Component {
    private int id;
    private ArrayList<Component> channels;
    public CompositeTool(int toolId) {
        this.id = toolId;
        channels = new ArrayList<Component>();
    }
    public void add(Component channel) {
        channels.add(channel);
    }
    public void remove(Component channel) {
        channels.remove(channel);
    }
    public int countChannels() {
```

```
        int count = 0;
        for(Component channel : channels) {
          count+=channel.countChannels();
        }
        return count;
    }
    @Override
    public void send() {
        System.out.println("\tComposite Tool #" + id + ", size tool: " + channels.size()
                                                  + ", number channels: " + countChannels() );
        for(Component channel : channels) {
          channel.send();
        }
    }
  @Override
   public void receive() {
       // some code here
   }
}
```

/ # 28 # организация набора каналов и демонстрация процесса # RunComposite.java */*

```
package by.bsu.composite.channel;
public class RunComposite {
    public static void main(String[] args) {
        TCPChannel channel1 = new TCPChannel(1);
        TCPChannel channel2 = new TCPChannel(2);
        UDPChannel channel3 = new UDPChannel(3);
        UDPChannel channel4 = new UDPChannel(4);
        UDPChannel channel9 = new UDPChannel(9);

        CompositeTool mainTool   = new CompositeTool(777);
        CompositeTool childTool1 = new CompositeTool(10);
        CompositeTool childTool2 = new CompositeTool(11);

        childTool1.add(channel1);
        childTool1.add(channel2);
        childTool1.add(channel3);
        childTool2.add(channel4);

        mainTool.add(childTool1); // add channels tool (1,2,3)
        mainTool.add(childTool2); // add channels tool (4)
        mainTool.add(channel9); // add single channel 9
        System.out.println("main tool send:");
        mainTool.send();
        childTool1.remove(channel2);
        mainTool.remove(childTool2);
        System.out.println("main tool send after remove:");
        mainTool.send();
    }
}
```

В результате будет выведено:

```
main tool send:
        Composite Tool #777, size tool: 3, number channels: 5
        Composite Tool #10, size tool: 3, number channels: 3
tcp send 1
tcp send 2
udp send 3
        Composite Tool #11, size tool: 1, number channels: 1
udp send 4
udp send 9
main tool send after remove:
        Composite Tool #777, size tool: 2, number channels: 3
        Composite Tool #10, size tool: 2, number channels: 2
tcp send 1
udp send 3
udp send 9
```

Шаблон Adapter

Пусть существуют две системы, выполняющие сходные действия, но несовместимые по интерфейсу, которые, тем не менее, необходимо заставить работать в одном приложении.

Позволяет определить интерфейс-адаптер, доступный его пользователям-клиентам, в то же время способ реализации отделяет от клиентов и не известен им. Однако функциональность, предоставляемая клиентам, соответствует заданному контракту. Обеспечивается взаимодействие несовместимых интерфейсов, предоставляемых классами типа **Adaptee**.

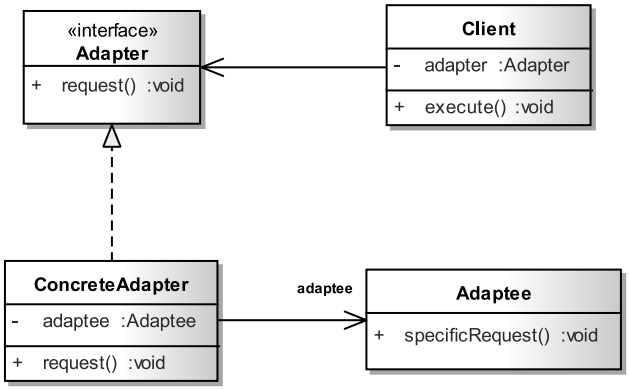

Рис. 23.7. *Базовая реализация шаблона Adapter*

/ # 29 # организация адаптера # Adapter.java # ConcreteAdapter.java */*

```
package by.bsu.adapter.base;
public interface Adapter {
    void request();
}
package by.bsu.adapter.base;
public class ConcreteAdapter implements Adapter{
    private Adaptee adaptee;
    public ConcreteAdapter (Adaptee adaptee) {
        this.adaptee = adaptee;
    }
    public void request() {
        System.out.println("Return type - void.");
        adaptee.specificRequest();
    }
}
```

/ # 30 # адаптируемый класс # Adaptee.java */*

```
package by.bsu.adapter.base;
public class Adaptee {
    public boolean specificRequest() {
        System.out.println("Return type - boolean");
        return true; // stub
    }
}
```

Класс **Client** может взаимодействовать только с экземплярами, реализующими интерфейс **Adapter**. Экземпляр класса **Client**, вызывая метод **execute()**, не будет знать, метод какого класса он вызывает — основного или адаптируемого. Предназначение шаблона **Adapter** — без серьезной переработки системы включить необходимый функционал в общем случае несовместимый с существующим.

/ # 31 # адаптируемый класс # Adaptee.java */*

```
package by.bsu.adapter.base;
public class Client {
    private Adapter target;
    public Client(Adapter target) {
        this.target = target;
    }
    public void execute(Adapter target) {
        target.request();
    }
}
```

/ # 32 # запуск процесса адаптации # RunnerAdapter.java */*

```
package by.bsu.adapter.base;
public class RunnerAdapter {
    public static void main(String[] args) {
        Adapter target = new ConcreteAdapter(new Adaptee());
        Client client = new Client(target);
        client.execute();
    }
}
```

Реализация, представленная выше, использует один класс типа **Adaptee**. В общем случае эти классы могут быть организованы в иерархию и при инициализации объекта типа **Adapter** будет передаваться объект из иерархии, и скрытое действие будет выполняться в соответствии с конкретным типа объекта. Но это уже будет не совсем шаблон **Adapter**.

Класс **Adapter** должен обладать механизмом согласования параметров и способом трансляции вызова методов основного приложения и адаптируемого класса.

Пусть в систему JSON-парсинга необходимо подключить разбор XML-документа. Для этого выделяется интерфейс **Parser** с методом **parse()**. Существующий класс **JsonParser** теперь должен имплементировать новый интерфейс. Для адаптации интерфейса класса **XmlParser** создается класс **XmlParserAdapter**, объявляющий адаптируемый класс в качестве поля, а также в качестве полей объявляет необходимые для согласования интерфейса параметры. Метод **parse()** адаптера теперь может корректно вызвать метод парсинга адаптируемого класса.

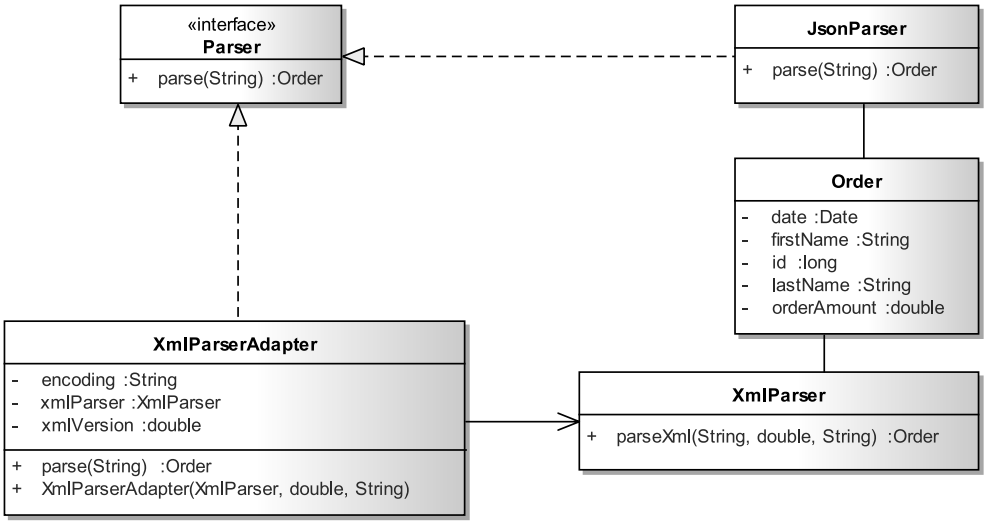

Рис. 23.8. *Парсинг документов с шаблоном Adapter*

/ # 33 # адаптируемый функционал # XmlParser.java */*

```
package by.bsu.adapter;
public class XmlParser {
    public Order parseXml(String str, double xmlVersion, String encoding) {
        Order order = new Order();
        System.out.println("Parse xml file.");
        return order;
    }
}
```

/ # 34 # организация адаптера # Parser.java # XmlParserAdapter.java */*

```
package by.bsu.adapter;
public interface Parser {
    Order parse(String order);
}
package by.bsu.adapter;
public class XmlParserAdapter implements Parser {
    private XmlParser xmlParser;
    private double xmlVersion; // согласование параметров для адаптации
    private String encoding; // согласование параметров для адаптации
    public XmlParserAdapter(XmlParser xmlParser, double xmlVersion, String encoding) {
        this.xmlParser = xmlParser;
        this.xmlVersion = xmlVersion;
        this.encoding = encoding;
    }
    @Override
    public Order parse(String order) {
        return xmlParser.parseXml(order, xmlVersion, encoding);
    }
}
```

/ # 35 # существующий функционал # JsonParser.java */*

```
package by.bsu.adapter;
public class JsonParser implements Parser {
    public Order parse(String jsonOrder) {
        Order order = new Order();
        System.out.println("Parse JSON.");
        return order;
    }
}
```

/ # 36 # запуск процесса адаптации # RunDemoAdapter.java */*

```
package by.bsu.adapter;
public class RunDemoAdapter {
  public static void main(String args[]) {
    String jsonOrder = "\"id\": \"1456\", \"firstName\": \"John\", \"lastName\": \"Smith\" ...";
    Parser parser = new JsonParser();
```

```
    Order order = parser.parse(jsonOrder);
    System.out.println(order.getOrderAmount());
    String xmlOrder =
       "<order id=\"1456\"><person firstName=\"John\" lastName=\"Smith\">EPAM</person></order>";
    parser = new XmlParserAdapter(new XmlParser(), 1.0, "UTF-8");
    order = parser.parse(xmlOrder);
    System.out.println(order.getOrderAmount());
  }
}
```

/ # 37 # создаваемая сущность # Order.java */*

```
package by.bsu.adapter;
public class Order {
    // some fields
    // some methods
}
```

Шаблон Flyweight

Экземпляр класса может использоваться как набор независимых экземпляров. С помощью разделяемого содержимого экземпляра класса возможна экономия ресурсов при инициализации большого числа схожих объектов. **Flyweight**, точнее, разделяемое содержание, которое в тоже время представляет собой единое целое, можно использовать в различных представлениях, не утрачивая исходного смысла класса-приспособленца. Приспособленцам недоступно представление, в котором они существуют.

В основе лежит различие между внутренней частью класса и его внешним состоянием. Внутренняя часть отделена от окружения, в котором работает объект и недоступна для изменения с его стороны. Отделение внешней части делает все экземпляры идентичными. Внешняя часть зависит от окружения и может быть им изменена, вследствие чего не может быть отделена. Объекты-клиенты имеют возможность передавать внешнее состояние приспособленцу. Применение **Flyweight** позволяет снизить число объектов в системе, а следовательно, экономить ресурсы, увеличивать скорость работы.

/ # 38 # интерфейс приспособленца и его реализация # Flyweight.java # ConcreteFlyweight.java */*

```
package by.bsu.flyweight.base;
public interface Flyweight {
    void operation();
}
package by.bsu.flyweight.base;
public class ConcreteFlyweight implements Flyweight {
    public void operation () { // реализация
    }
}
```

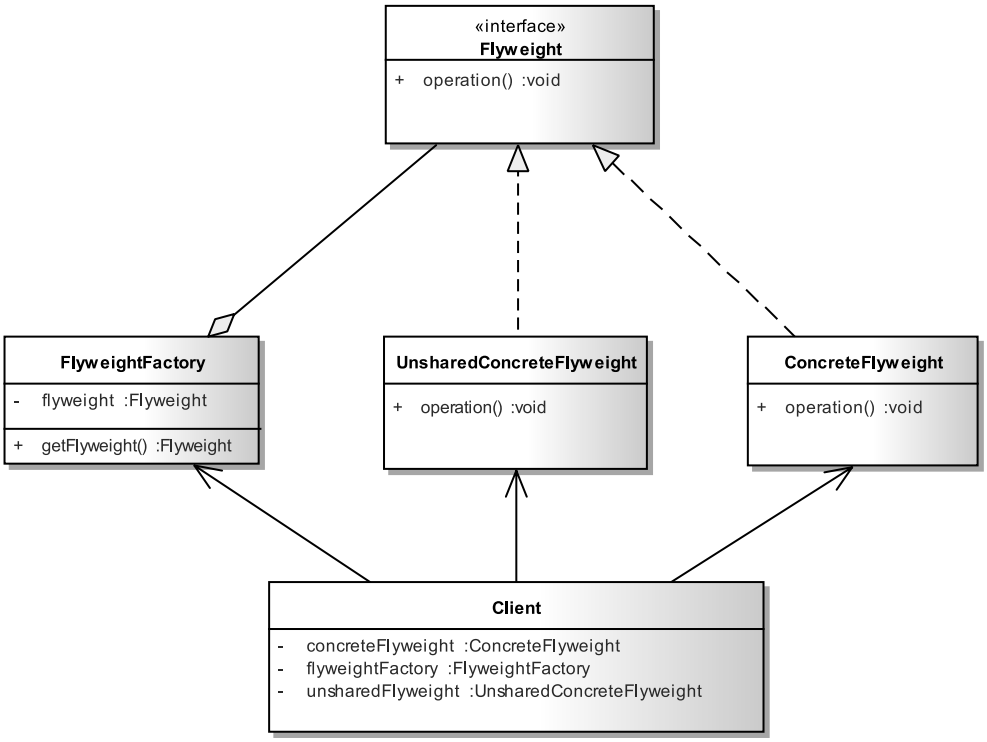

Рис. 23.9. *Базовая реализация шаблона Flyweight*

/ # 39 # неразделяемая сущность, существующая вне рамок шаблона # UnsharedConcreteFlyweight.java */*

```
package by.bsu.flyweight.base;
public class UnsharedConcreteFlyweight implements Flyweight {
    public void operation () { // more code
    }
}
```

/ # 40 # фабрика легковесов и их использование # FlyweightFactory.java # Client.java */*

```
package by.bsu.flyweight.base;
public class FlyweightFactory {
    private Flyweight flyweight;
    public Flyweight getFlyweight () {
          return new ConcreteFlyweight();
    }
}
package by.bsu.flyweight.base;
public class Client {
    private FlyweightFactory flyweightFactory;
```

```
    private ConcreteFlyweight concreteFlyweight;
    private UnsharedConcreteFlyweight unsharedFlyweight;
          // some code here
}
```

При моделировании динамической имитационной модели термитника и его обитателей может понадобиться создание нескольких миллионов однотипных объектов.

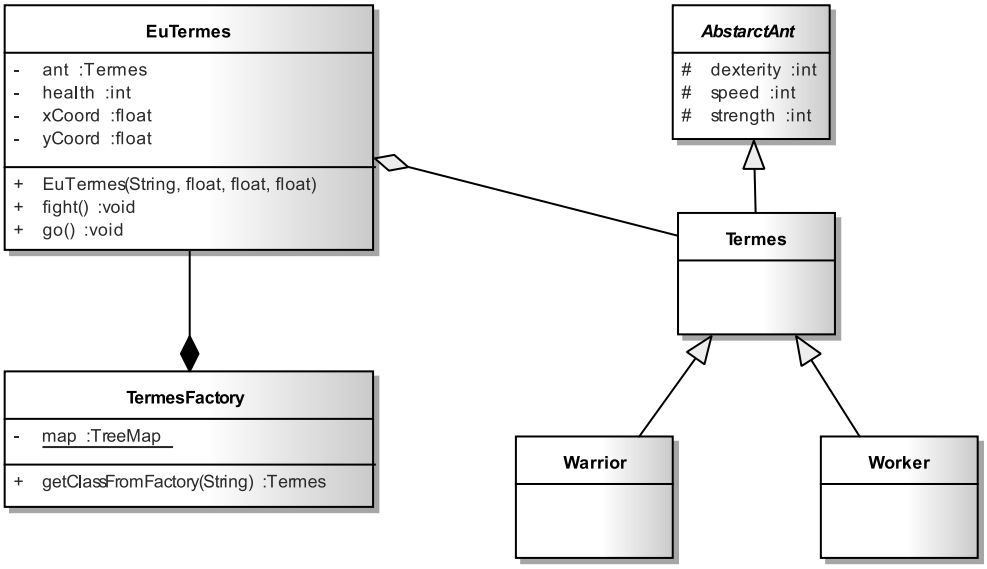

Рис. 23.10. *Термитник на основе шаблона Flyweight*

Пусть определяются два вида термитов: **Worker** и **Warrior**.

/ # 41 # иерархия термитов # AbstarctAnt.java # Termes.java # Worker.java # Warrior.java # TermesType.java */*

```
package by.bsu.flyweight;
public abstract class AbstarctAnt {
    protected int strength;
    protected int dexterity;
    protected int speed;
    public int getStrength() {
        return strength;
    }
    public void setStrength(int strength) {
        this.strength = strength;
    }
    public int getDexterity() {
        return dexterity;
    }
```

```
    public void setDexterity(int dexterity) {
        this.dexterity = dexterity;
    }
    public int getSpeed() {
        return speed;
    }
    public void setSpeed(int speed) {
        this.speed = speed;
    }
}
package by.bsu.flyweight;
import java.io.Serializable;
public class Termes extends AbstarctAnt implements Serializable {
    // more code here
}
package by.bsu.flyweight;
import java.io.Serializable;
public class Warrior extends Termes implements Serializable {
    public Warrior() {
        strength = 10;
        dexterity = 4;
        speed = 6;
    }
    // more code here
}
package by.bsu.flyweight;
import java.io.Serializable;
public class Worker extends Termes implements Serializable {
    public Worker() {
        strength = 6;
        dexterity = 9;
        speed = 10;
    }
    // more code here
}
package by.bsu.flyweight;
public enum TermesType {
    WORKER, WARRIOR
}
```

Основной задачей является грамотное разделение состояний на внутренне и внешнее. Сами термиты представляются практически идентичными, но каждый из них располагается в конкретной точке пространства.

/ # 42 # разделение состояний на основе агрегирования # EuTermes.java */*

```
package by.bsu.flyweight;
import java.io.Serializable;
// flyweight (приспособленец)
public class EuTermes implements Serializable {
```

```
    private Termes ant; // внутренняя часть
    // внешнее состояние: начало описания
    private int health;
    private float xCoord;
    private float yCoord;
    // внешнее состояние: окончание описания
    public EuTermes(String type, float xCoord, float yCoord, float zCoord) {
        this.ant = TermesFactory.getClassFromFactory(type);
        health = 180;
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
    public int getHealth() {
        return health;
    }
    public void setHealth(int health) {
        this.health = health;
    }
    public float getXCoord() {
        return xCoord;
    }
    public void setXCoord(float xCoord) {
        this.xCoord = xCoord;
    }
    public float getYcoord() {
        return yCoord;
    }
    public void setYcoord(float ycoord) {
        this.yCoord = ycoord;
    }
    public void go() {
          // more code here
    }
    public void fight() {
          // more code here
    }
}
```

Чем больше атрибутов можно сделать внешними, тем меньше потребуется экземпляров-приспособленцев, но чем больше атрибутов сделать внутренними, тем меньше времени будет затрачено экземплярами для доступа к своим собственным полям.

/ # 43 # фабрика # TermesFactory.java */*

```
package by.bsu.flyweight;
import java.util.Map;
import java.util.TreeMap;
public class TermesFactory {
    private static TreeMap<String, Termes> map = new TreeMap<>();
```

```
    public static Termes getClassFromFactory(String name) {
        if (map.containsKey(name)) {
            return map.get(name);
        } else {
            TermesType type = TermesType.valueOf(name.toUpperCase());
            switch (type) {
                case WORKER: {
                    Worker worker = new Worker();
                    map.put(name, worker);
                    return worker;
                }
                case WARRIOR: {
                    Warrior warrior = new Warrior();
                    map.put(name, warrior);
                    return warrior;
                }
                default:
                throw new EnumConstantNotPresentException(TermesType.class, type.name());
            }
        }
    }
}
```

/ # 44 # клиентское приложение # FlyweightRunner.java */*

```
package by.bsu.flyweight;
import java.util.ArrayList;
public class FlyweightRunner {
    private final static int SIZE = 4_000_000;
    public static void main(String[] args) {
        ArrayList<EuTermes> legion = new ArrayList<>();
        try {
            for (int i = 0; i < SIZE; i++) {
                legion.add(new EuTermes("Worker", 12.3f, 10.1f, 5.0f));
            }
        } catch (OutOfMemoryError e) {
            System.err.println("OutOfMemoryError for Termes Worker");
        } finally {
            System.out.println("Instance: " + i);
        }
    }
}
```

В результате будет получено:

OutOfMemoryError for Termes Worker
Instance: 2261945

т.е. память закончится только после создания **2261945** объектов.

Если же создавать объекты в «лоб», как показано в следующем коде, то память закончится значительно раньше.

/ # 45 # реализация без разделения состояний # TermesBad.java # BadRunner.java */*

```
package by.bsu.flyweight;
public class TermesBad {
    private int strength;
    private int dexterity;
    private int speed;
    private int health;
    private float xCoord;
    private float yCoord;
    {
        strength = 10;
        dexterity = 4;
        speed = 6;
        health = 180;
        xCoord = 10.2f;
        yCoord = 12.8f;
    }
    // constructors, setters, getters & more
}
package by.bsu.flyweight;
import java.util.ArrayList;
public class BadRunner {
    private static int size = 4_000_000;
    public static void main(String[] args) {
        ArrayList<TermesBad> legion = new ArrayList<>();
        try {
            for (int i = 0; i < size; i++) {
                legion.add(new TermesBad());
            }
        } catch (OutOfMemoryError e) {
            System.err.println("OutOfMemoryError for TermesBad");
        } finally {
            System.out.println("Instance: " + i);
        }
    }
}
```

Получено будет:

OutOfMemoryError for TermesBad
Instance: 1507963

Шаблон Proxy

Прокси-объект представляет другой объект. С точки зрения клиента интерфейс и функциональность класса остаются неизменными. Чтобы это представление было естественным, прокси-объект обязан реализовывать тот же интерфейс, что

и реальный класс. Кроме того, прокси-объект должен содержать реальный класс в качестве поля, чтобы при необходимости обращаться к функционалу реального класса. В общем случае реализует технологию обертывания (wrapping) класса с целью повышения безопасности или оптимизации взаимодействия.

/ # 46 # интерфейс и класс # BaseInterface.java # BaseClass.java */*

```
package by.bsu.proxy.base;
public interface BaseInterface {
          public void perform();
}
package by.bsu.proxy.base;
public class BaseClass implements BaseInterface {
          public void perform() {
                    System.out.println("...performing...");
          }
}
```

/ # 47 # класс proxy # Proxy.java */*

```
package by.bsu.proxy.base;
public class Proxy implements BaseInterface {
          private BaseClass base = null;
          public void perform() {
                    base = new BaseClass();
                    base.perform();
          }
}
```

/ # 48 # запуск # DemoProxy.java */*

```
package by.bsu.proxy.base;
public class DemoProxy {
          public static void main(String args[]) {
                    BaseInterface base = new Proxy();
                    base.perform();
          }
}
```

Существует некоторый пул соединений с БД. Соединения по мере необходимости извлекаются из пула и после использования возвращаются. При такой реализации пул оказывается незащищенным от попадания в него «диких» соединений, созданных вне пула, в то время как соединение, взятое из пула, не сможет возвратиться в него, так как пул может оказаться заполнен «дикими» соединениями. В итоге приложения из-за создания несанкционированных соединений и нарушений работы пула будет работать медленнее и с ошибками, возможно, критическими.

/ # 49 # небезопасный пул соединений # ConnectionPool.java */*

```
package by.bsu.proxy;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class ConnectionPool {
    private BlockingQueue<Connection> connectionQueue;
    public ConnectionPool(final int POOL_SIZE) throws SQLException {
        connectionQueue = new ArrayBlockingQueue<Connection> (POOL_SIZE);
        for (int i = 0; i < POOL_SIZE; i++) {
            Connection connection = // create connection
            connectionQueue.offer(connection);
        }
    }
    public Connection getConnection() throws InterruptedException {
        Connection connection = null;
        connection = connectionQueue.take();
        return connection;
    }
    public void closeConnection(Connection connection) {
        connectionQueue.offer(connection);
    }
}
```

Для решения проблемы следует создать прокси-соединение и переписать реализацию класса-пула.

/ # 50 # прокси-класс для Connection # ProxyConnection.java */*

```
package by.bsu.proxy;
import java.sql.Connection;
// imports
public class ProxyConnection implements Connection {
          private Connection connection;
          ProxyConnection(Connection connection) { // только в пакете
                    this.connection = connection;
          }
          @Override
          public Statement createStatement() throws SQLException {
                    return connection.createStatement();
          }
          @Override
          public void close() throws SQLException {
                    connection.close();
          }
          @Override
          public void commit() throws SQLException {
                    connection.commit();
          }
```

```
        @Override
        public boolean isClosed() throws SQLException {
                return connection.isClosed();
        }
        @Override
        public PreparedStatement prepareStatement(String sql) throws SQLException {
                return connection.prepareStatement(sql);
        }
        @Override
        public void rollback() throws SQLException {
                connection.rollback();
        }
        @Override
        public void setAutoCommit(boolean flag) throws SQLException {
                connection.setAutoCommit(flag);
        }
        // more override methods
}
```

/ # 51 # безопасный пул соединений # ConnectionPool.java */*

```
package by.bsu.proxy;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.concurrent.ArrayBlockingQueue;
public class ConnectionPool<T> {
        private ArrayBlockingQueue<T> connectionQueue;
        public ConnectionPool(final int POOL_SIZE) throws SQLException {
                connectionQueue = new ArrayBlockingQueue<T>(POOL_SIZE);
                for (int i = 0; i < POOL_SIZE; i++) {
                        T connection = // create connection
                        connectionQueue.offer(connection);
                }
        }
        public T getConnection() throws InterruptedException {
                T connection = null;
                connection = connectionQueue.take();
                return connection;
        }
        public void closeConnection(T connection) {
                connectionQueue.offer(connection); // возможны утечки соединений
        }
        // more methods
}
```

Если пул соединений создается с параметром **ProxyConnection**:

```
ConnectionPool<ProxyConnection> pool = new ConnectionPool<>(20);
```

то соединение, извлеченное из пула, позволяет выполнять все необходимые для объекта такого рода действия. При возвращении соединения в пул методом

closeConnection(T c) можно передать только прокси-объект. Попытка возвращения обычного экземпляра **Connection** приведет к ошибке компиляции. То есть «дикие» соединения попасть в пул не могут, и безопасность пула обеспечена.

Задания к главе 23

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 прививке деревьев в зависимости от возраста и плодоношения. Учесть, что некоторые варианты скрещивания видов невозможны.

Вариант В

В любой задаче обязательно создание иерархии классов сущностей создаваемой системы.

1. Паттерн Composite. Разработать структуру данных для хранения информации о государствах, их административных частях и городах.
2. Паттерн Composite. Разработать структуру хранения текста, который может состоять из абзацев, предложений, листингов, заголовков, слов, знаков препинания и проч.
3. Паттерн Composite. Разработать структуру организации армии в игре фэнтези. Армия может состоять из отрядов эльфов, орков, минотавров, кентавров, циклопов, драконов, гидр, рыцарей. Армия может содержать как отряды, так и одиночных воинов, отряд может состоять из других отрядов и одиночных воинов.

4. Паттерн Composite. Разработать структуру организации дискового пространства, где каталоги содержат другие каталоги и файлы
5. Паттерн Flyweight. Разработать систему учета процессов размножения колонии бактерий.
6. Существует модель системы Рецепт. Модель позволяет в неизменяемом виде хранить назначения врача и срок действия рецепта. Написать код приложения, позволяющий продлевать срок действия уже существующего рецепта. Выберите подходящий паттерн для реализации этого задания.
7. Существует модель системы Магазин цветов. Напишите код, позволяющий добавлять к букету новый атрибут (например, ленту с фирменным названием), используя при этом существующую модель порождения букетов. Выберите подходящий паттерн для реализации этого задания.

УКАЗАНИЯ И ОТВЕТЫ

Глава 1

Вопрос 1.1.

Код класса **Quest21** ошибок не содержит; статический метод **main()** класса **Quest22** пытается получить доступ к нестатическому полю, описанному в этом классе, что является ошибкой; код класса **Quest23** включает логический блок инициализации, внутри которого вызов методов допустим, код данного класса скомпилируется без ошибок, однако при запуске возникнет исключительная ситуация типа **java.lang.NoClassSuchMethodError**.

Ответ: 5.

Вопрос 1.2.

Класс — это тип данных; если он разработан программистом-пользователем системы, то говорят, что класс — это пользовательский тип данных. На объект класса действительно может не ссылаться объектная переменная, если при создании объекта этого явно не указать, отработать такой объект может только один раз в точке своего создания. Объект класса может использоваться всюду, где используется объект ЕГО (а не любого) подкласса; так, имея отношение наследования *Человек –> Доктор* получаем, что доктора (объект производного класса) можно отправить за покупками в магазин так же, как и человека (объект базового класса). Если конструкцией класса не определено, то объектов класса можно создать сколько угодно, пока есть свободная память.

Ответ:1, 4.

Вопрос 1.3.

Наследование — приобретение свойств другого объекта или класса. Полиморфизм позволяет каждому типу объекта определять свое собственное поведение. Связыванием называется сопоставление вызова функции с телом функции. Позднее связывание производится во время выполнения программы в зависимости от фактического типа объекта. Инкапсуляция — это механизм, позволяющий объединить в классе данные и методы, работающие с ними, и скрыть детали реализации.

Ответ: 4.

Вопрос 1.4.

Ошибок компиляции данный фрагмент кода не вызовет. Ссылка **s** инициализируется перед первым использованием. Сравнение в операторе **if** производится по ссылкам на строку, определенную в пуле литералов.

Ответ: 3.

Вопрос 1.5.

Данное приложение не запустится, так как статический метод **main** имеет атрибут доступа **private**, который не позволяет вызвать данный метод на выполнение вне класса **Quest5**. После успешной компиляции и запуска с помощью командной строки **Quest6 Java 7 ""** появится сообщение **Error: Main method not found in class Quest5, please define the main method as: public static void main(String[] args)**.

Ответ: 6.

Вопрос 1.6.

При запуске программы вызовется метод **main** с атрибутом доступа **public**, принимающий в качестве параметра массив строк. В теле этого метода вызывается перегруженный метод **main**, принимающий в качестве параметра только один строковый аргумент.

Ответ: 3.

Вопрос 1.7.

Класс **Book** не имеет собственной реализации метода **equals**, данный метод (возвращающий **true** только при тождественности сравниваемых ссылок) наследуется от класса **Object**.

Ответ: 3.

Ответы:

1.1. — 5 1.4. — 3 1.6. — 3
1.2. — 1, 4 1.5. — 6 1.7. — 3
1.3. — 4

Глава 2

Вопрос 2.1.

Константа **356f** — константа типа **float**. Неявное преобразование константы типа **float** возможно к типу **double, float, Object (Float)**.

Ответ: 2, 3, 9.

Вопрос 2.2.

Теги **@author** и **@version** не используются для описания конструкторов и методов, их применяют для описания пакетов и классов.

Ответ: 2, 4.

Вопрос 2.3.

При делении на вещественный нуль приводит к возбуждению исключительной ситуации. Результатом, в зависимости от делимого, будет **+/- Infinity**.

Ответ: 2.

Вопрос 2.4.

Объект, на который ссылается объектная переменная **medical**, является объектом класса **HeadDoctor**, производным от класса **Doctor**. Объект типа **HeadDoctor** является экземпляром типа **Doctor**.

Ответ: 2.

Вопрос 2.5.

Искомому фрагменту кода и фрагментам под номерами 1 и 4 соответствует блок-схема:

А фрагменты под номерами 2 и 3 отвечают следующей блок-схеме:

Ответ: 1, 4.

Вопрос 2.6.

В циклах под номерами 3 и 5 допущены синтаксические ошибки. Циклы под номерами 1, 3 и 4, вычисляют сумму первых ста натуральных чисел.

Ответ: 3, 5.

Вопрос 2.7.

Классы оболочки расположены в пакете **java.lang**. Объекты классов оболочек хранят неизменяемые значения.

Ответ: 2, 4, 5.

Вопрос 2.8.

Компиляция строки 3 приведет к ошибке **Type mismatch: cannot convert from Character[] to char[]**. Строка 4 содержит синтаксическую ошибку — отсутствуют квадратные скобки, указывающие, что создается массив.

Ответ: 3, 4.

Ответы:

2.1. — 2, 3, 9	2.4. — 2	2.7. — 2, 4, 5
2.2. — 2, 4	2.5. — 1, 4	2.8. — 3, 4
2.3. — 2	2.6. — 3, 5	

Глава 3

Вопрос 3.1.

Корректными являются описания классов 1) и 5). При описании класса **Quest1** в файле **Quest1.java** допустимо использовать модификаторы **public**, **abstract** или **final**.

Ответ: 2, 3, 4.

Вопрос 3.2.

Константное поле экземпляра класса может быть проинициализировано только один раз либо при объявлении, либо в логическом блоке инициализации, либо в конструкторе класса.

Ответ: 1, 3, 5.

Вопрос 3.3.

При компиляции данной программы будет ошибка в строке 1, так как локальная переменная используется без инициализации.

Ответ: 3.

Вопрос 3.4.

Статические **private**-поля класса доступны во всех методах этого класса, как и статические **public**-поля. К **protected**-полям класса есть доступ из подклассов, находящихся в другом пакете. Поля класса, объявленные без модификатора доступа (friendly), видны в классах этого же пакета.

Ответ: 2, 3.

Вопрос 3.5.

При создании объекта класса **Quest5** в строке 1 вызовется 3 конструктора, один класса **Object**, и два класса **Quest5**.

Ответ: 3.

Вопрос 3.6.

Для разрешения перегрузки сначала ищется метод, тип формального параметра которого совпадает с типом фактического параметра, и только в случае неудачи ищется другой метод, к типу формального параметра которого можно преобразовать передаваемый объект.

Ответ: 5.

Вопрос 3.7.

Параметризованным типам разрешено вызывать только методы класса **Object**.

Ответ: 3.

Вопрос 3.8.

Второй вариант создания экземпляра параметризованного класса содержит неверное число аргументов, четвертый — объявлен с неправильным синтаксисом, пятый и шестой содержат ошибку преобразования типа.

Ответ: 1, 3.

Вопрос 3.9.

Объекты перечисления создаются в единственном экземпляре, поэтому сравнение оператором «==» элементов перечисления приводит к сравнению их объектных ссылок. Нумерация позиций элементов перечисления начинается с нуля.

Ответ: 2.

Вопрос 3.10.

В теле перечисления можно объявлять методы, поля и конструкторы. Конструктор перечисления определяется без модификатора доступа или с модификатором **private**. Перечисления неявно наследуются от класса **java.lang.Enum**.

Ответ: 2,3,4.

Ответы:

3.1. — 2, 3, 4
3.2. — 1, 3, 5
3.3. — 3
3.4. — 2, 3
3.5. — 3
3.6. — 5
3.7. — 3
3.8. — 1, 3
3.9. — 2
3.10. — 2, 3, 4

Глава 4

Вопрос 4.1.

Класс **Quest41** объявлен в пакете с атрибутом friendly, значит, доступен для наследования классам этого пакета. Для наследования применяется ключевое слово **extends**, а не **implements**.

Ответ: 1, 6.

Вопрос 4.2.

Класс не может наследоваться от самого себя. В конструкторе класса нельзя совместно использовать вызовы **super()** и **this()**, поскольку такой вызов должен быть всегда первым оператором конструктора. Компилятор не помешает переопределить статические методы в подклассах, однако при их вызове будет использоваться механизм раннего связывания. Аннотацию **@Override** к статическим методам применять нельзя. Статические методы можно перегружать в подклассах, доступность таких методов зависит от типа ссылки и атрибута доступа. При динамическом связывании версия вызываемого метода определяется на этапе выполнения.

Ответ: 3, 4, 5.

Вопрос 4.3.

Компиляция этого кода завершится без ошибок, закрытый **final**-метод класса **Quest43** не переопределяется открытым методом класса **Quest431**.

Ответ: 3.

Вопрос 4.4.

Если программист не определил ни одного конструктора, компилятор добавляет в класс конструктор по умолчанию с атрибутом доступа **public**.

Ответ: 1.

Вопрос 4.5.

При вызове **getClass().getSimpleName()** в виде строки вернется имя класса, от которого был создан объект.

Ответ: 2.

Вопрос 4.6.

Для массивов Java переопределен метод **clone()**, который производит поэлементное копирование.

Ответ: 3.

Ответы:

4.1. — 1,6	4.3. — 3	4.5. — 2
4.2. — 3,4,5	4.4. — 1	4.6. — 3

Глава 5

Вопрос 5.1.

Новая версия метода **getNumber()** будет вызываться для элементов перечисления ***TWO*** *и* ***THREE***.

Ответ: 3.

Вопрос 5.2.

Внутренние классы могут быть производными, а ссылка базового типа может ссылаться на объект производного.

Ответ: 3.

Вопрос 5.3.

Для получения доступа из внутреннего класса к экземпляру его внешнего класса необходимо в ссылке указать имя класса и ключевое слово **this**, поставив между ними точку (например, **Outer.this**).

Ответ: 5.

Вопрос 5.4.

Во втором фрагменте кода создается объект от анонимного класса, на объект указывает ссылка типа **Object**, для которого метод **hello()** не определен.

Ответ: 2.

Вопрос 5.5.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса. Внутренние классы не могут содержать **static**-полей, кроме **final static**. Внутренние классы могут реализовывать интерфейсы. Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса; видимость класса регулируется видимостью того блока, в котором он объявлен; однако класс сохраняет доступ ко всем полям и методам внешнего класса, а также константам, объявленным в текущем блоке кода. Внутренние классы могут быть как производными, так и базовыми классами.

Ответ: 1, 3, 4, 6, 8.

Ответы:

5.1. — 3 5.3. — 5 5.5. — 1,3,4,6,8

5.2. — 4 5.4. — 2

Глава 6

Вопрос 6.1.

Метод **returner()** класса **Quest1** реализует соответствующий абстрактный метод класса **Quest100** и одновременно метод **returner()** интерфейса **Quest10**. В последнем случае возвращаемый тип заменяется его подклассом (метод подставки).

Ответ: 4.

Вопрос 6.2.

Поля интерфейса имеют модификаторы **public final static** и должны быть инициализированы при объявлении.

Ответ: 3,4.

Вопрос 6.3.

На объект **obj** класса **Cl2** ссылается ссылка базового типа (интерфейса) **Inter1**, метод **f()** для класса **Cl2** переопределен, следовательно, при выполнении кода механизм позднего связывания вызовет метод **f()** класса **Cl2**.

Ответ: 2.

Вопрос 6.4.

Для объявления аннотации используется объявление вида **@interface**

Ответ: 1.

Ответы:

6.1. — 4 6.3. — 2

6.2. — 3,4 6.4. — 1

Глава 7

Вопрос 7.1.

Синтаксис языка Java позволяет создавать строковые объекты с использованием упрощенного синтаксиса, но в этом случая строка автоматически размещается в пуле литералов. Метод **intern()** класса **String** размещает строку в пуле литералов и возвращает ссылку на нее, ссылку нужно сохранить для дальнейшего использования.

Ответ: 1.

Вопрос 7.2.

Для класса **StringBuffer** не переопределены методы **equals()** и **hashCode()**.

Ответ: 2.

Вопрос 7.3.

Начальное состояние объекта типа **Matcher** неопределенно, поэтому до вызова метода **group()** на объекте необходимо вызвать любой из методов обработчиков (например, **find()** или **lookingAt())**.

Ответ: 4.

Вопрос 7.4.

В результате компиляции и запуска должно выводиться название страны для локали ("ru","RU") так, как оно пишется в локали US.

Ответ: 4.

Вопрос 7.5.

Классы **NumberFormat** и **DateFormat** расположены в пакете **java.text**.

Ответ: 1.

Ответы:

7.1 — 1 7.3 — 4 7.5 — 1
7.2 — 2 7.4 — 4

Глава 8

Вопрос 8.1.

Проверяемые исключения в Java являются наследниками класса **Exception**, непроверяемые — наследниками класса **RuntimeException**, который, в свою очередь, наследуется от **java.lang.Exception**. Проверяемые исключения обязательно в коде обрабатывать либо в секции **try-catch**, либо с помощью ключевого слова **throws**. Непроверяемые исключения можно обрабатывать в коде так же, как и проверяемые.

Ответ: 1,3,4.

Вопрос 8.2.

При выполнении оператора **FileReader fr2 = new FileReader("test2.txt");** возникнет исключение **FileNotFoundException** во вложенном блоке **try**, которое является производным от **IOException**.

Ответ: 6.

Вопрос 8.3.

Подклассы исключений в блоках **catch** должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения.

Ответ: 1,2,5.

Вопрос 8.4.

Переопределяемый метод в подклассе не может содержать в инструкции **throws** исключений, необрабатываемых в соответствующем методе суперкласса. Это относится только к checked-исключениям.

Ответ: 2,4,5,6.

Вопрос 8.5.

Если в конструкторе выбрасывается исключение, то объект не создается. **ArithmeticException** гасится соответствующим блоком **catch**, не обработается только **NullPointerException**, полученный в строке 5.

Ответ: 5.

Ответы:

8.1. — 1,3,4 8.3. — 1,2,5 8.5. — 5
8.2. — 6 8.4. — 2,4,5,6

Глава 9

Вопрос 9.1.

На вершине иерархии байтовых потоков ввода-вывода два **абстрактных** класса: **InputStream** и **OutputStream**. На вершине иерархии символьных потоков ввода-вывода два **абстрактных** класса: **Reader** и **Writer**. **InputStreamReader** — поток ввода, который переводит байты в символы. **OutputStreamWriter** — поток ввода, который переводит символы в байты.

Ответ: 3,4.

Вопрос 9.2.

При закрытии потока, на который указывает ссылка **sc1**, закроется и поток **System.in**. Дальнейшее считывание с помощью него становится невозможным.

Ответ: 4.

Вопрос 9.3.

При *десериализации* производного *класса, наследуемого от несериализуемого класса*, вызывается *конструктор без параметров* родительского несериализуемого класса.

Ответ: 1.

Вопрос 9.4.

Классы **InputStreamReader** и **StringWriter** относятся к символьным потокам ввода-вывода.

Ответ: 1, 3.

Вопрос 9.5.

Для классов **ByteArrayInputStream**, **InputStreamReader**, **BufferedReader** не предусмотрены конструкторы, принимающие параметр типа **File**.

Ответ: 2, 3, 4.

Ответы:

9.1. — 3,4 9.3. — 1 9.5. — 2,3,4
9.2. — 4 9.4. — 1,3

Глава 10

Вопрос 10.1.

Классы **HashSet** и **LinkedHashSet** не предоставляют индексный доступ к хранимым элементам.

Ответ: 2.

Вопрос 10.2.

Интерфейс **Set** уникальность хранимых объектов определяет реализацией метода **equals()**.

Ответ: 2.

Вопрос 10.3.

Использование **Iterator** более безопасно, чем **Enumeration**, поскольку **Iterator** не позволяет менять объект коллекции иным способом, кроме как с использованием собственного метода **remove()**, а при попытке изменения объекта коллекции выбрасывает исключение **ConcurrentModificaitonException**.

Ответ: 2.

Вопрос 10.4.

Вызовом метода **EnumSet<T> complementOf(EnumSet<T> s) —** создается множество, содержащее все элементы, которые отсутствуют в указанном множестве **s**.

Ответ: 3.

Вопрос 10.5.

По умолчанию **TreeMap** сортирует элементы по ключу по возрастанию, метод **headMap()** с одним параметром возвращает элементы с начала набора до указанного элемента, не включая его.

Ответ: 1.

Ответы:

10.1. — 2 10.3. — 2 10.5. — 1
10.2. — 2 10.4. — 3

Глава 11

Вопрос 11.1.

Потоковым объектом является объект класса **Thread**. Чтобы создать потоковый объект с помощью объекта, класс которого реализует интерфейс **Runnable**, необходимо придать этому объекту функциональность класса **Thread** — передать объект **Runnable** в конструктор класса **Thread**.

Ответ: 4.

Вопрос 11.2.

Методы **wait()** и **notifyAll()** определены в классе **java.lang.Object**. Остальные методы определены в классе **java.lang.Thread**.

Ответ: 1, 2, 5, 6.

Вопрос 11.3.

Метод **isAlive()** возвращает **true**, когда поток находится в работоспособном состоянии или в состоянии, из которого он может вернуться в работоспособное.

Ответ: 2, 3, 4, 5.

Вопрос 11.4.

С помощью объекта, представляющего собой пул потоков фиксированного размера, выполняются все шесть запускаемых из метода **main()** потоков, по два одновременно.

Ответ: 4.

Вопрос 11.5.

Последовательность действий для метода **turnOn()** следующая: если лампочка горит — ждем, пока ее не выключат, включаем лампочку, оповещаем возможных остальных ждущих.

Ответ: 5.

Вопрос 11.6.

Класс **ReentrantLock** находится в пакете **java.util.concurrent.locks**, класс **AtomicInteger** в пакете **java.util.concurrent.atomic**. Остальные классы и интерфейсы находятся в пакете **java.util.concurrent**.

Ответ: 1, 3, 4, 5.

Ответы:

11.1. — 4	11.3. — 2,3,4,5	11.5. — 5
11.2. — 1,2,5,6	11.4. — 4	11.6. — 1,3,4,5

Глава 12

Вопрос 12.1.

Классы и интерфейсы JDBC находятся в пакетах **java.sql** и **javax.sql.**

Ответ: 2.

Вопрос 12.2.

Последовательность действий для подключения к БД с помощью JDBC следующая: загрузка драйвера, установление соединения, создание объекта для выполнения запросов, выполнение запроса, обработка результатов, закрытие открытых соединений.

Ответ: 2.

Вопрос 12.3.

В API JDBC существует три интерфейса, реализации которых могут предоставлять statement-объекты: **Statement**, **PreparedStatement**, **CallableStatement.**

Ответ: 1, 3, 6.

Вопрос 12.4.

Метод **rollback()**, выполненный на объекте типа **Connection**, откатит состояние БД до выполнения последней операции commit; в таблицу добавится информация (2, Petrov) и (5, Blinov).

Ответ: 1, 4.

Вопрос 12.5.

Хранимую процедуру можно выполнить на объекте **CallableStatement**, после установки входных и выходных параметров используются методы **execute(), executeQuery()** или **executeUpdate().**

Ответ: 1, 2, 3.

Ответы:

12.1. — 2 12.3. — 1,3,6 12.5. — 1,2,3

12.2. — 2 12.4. — 1,4

Глава 13

Вопрос 13.1.

TCP — протокол образует постоянное соединение и гарантирует доставку пакета. UDP не образует постоянного соединения и не гарантирует доставку пакета, однако делает это быстро.

Ответ: 2, 5.

Вопрос 13.2.

IP адрес ресурса можно узнать с помощью методов **getHostAddress()**, **getAddress**(). **getLocalHost()** — возвратит номер хоста текущей машины.

Ответ: 1, 3.

Вопрос 13.3.

Метод **accept()** не возвращает управление, пока не подключится клиент.

Ответ: 4.

Вопрос 13.4.

Если соединение с сервером не установлено, возникает исключение типа **ConnectException**.

Ответ: 4.

Вопрос 13.5.

Метод **isConnected()** говорит о том, что сокет был соединен с удаленной стороной. Информацию о том, был ли данный сокет закрыт, этот метод не предоставляет.

Ответ: 1.

Ответы:

13.1. — 2,5 13.3. — 4 13.5. — 1

13.2. — 1,3 13.4. — 4

Глава 14

Вопрос 14.1.

Параметры xml-документа: **version** — номер версии XML, **encoding** — кодировка, **standalone** — определяет автономность XML.

Ответ: 2, 4, 6.

Вопрос 14.2.

Неправильные документы не следуют синтаксическим правилам XML. Правильные документы следуют синтаксическим правилам XML и правилам, определенным в их DTD или в XSD. Документы форматированные правильно не имеют DTD или XSD.

Ответ: 2.

Вопрос 14.3.

JAXP располагается в пакете **javax.xml.parsers**, в состав которого входят четыре класса: **DocumentBuilder**, **DocumentBuilderFactory**, **SAXParser**, **SAXParserFactory**.

Ответ: 3, 4, 6, 7.

Вопрос 14.4.

Возможно обратное создание на основе XML-схемы классов на языке Java с помощью команды **xjc.exe**.

Ответ: 4.

Вопрос 14.5.

К событиям, которые обрабатывает синтаксический анализатор SAX, относятся события начала документа, начала элемента, пары значений атрибута, текстовые данные, конец элемента и конец документа.

Ответ: 6.

Ответы:

14.1. — 2,4,6 14.3. — 3,4,6,7 14.5. — 6
14.2. — 2 14.4. — 4

Глава 15

Вопрос 15.1.

Центральной абстракцией API сервлета является интерфейс **Servlet**. Интерфейс **ServletRequest** инкапсулирует связь клиента с сервером. Интерфейс **ServletResponse** инкапсулирует обратную связь сервлета с клиентом. Интерфейс **ServletContext** используется для взаимодействия с контейнером сервлетов. Интерфейс **ServletConfig** представляет собой конфигурацию сервлета, используется, в основном, на этапе инициализации.

Ответ: 4.

Вопрос 15.2.

Послать http-запрос из адресной строки браузера методом **POST** нельзя**.**

Ответ: 5.

Вопрос 15.3.

Веб-контейнер создает объект сервлета, вызывает метод **init()**. Метод **service()** вызывается в сервлете для управления каждым запросом. Перед уничтожением объекта контейнер вызовет метод **destroy()**.

Ответ: 2.

Вопрос 15.4.

Запрос от параметров отделяется символом «?», параметры между собой — символом «&».

Ответ: 3.

Ответы:

15.1. — 4 15.3. — 2
15.2. — 5 15.4. — 3

Глава 16

Вопрос 16.1.

Жизненный цикл JSP включает следующие шаги (фазы): трансляция страницы JSP — создание исходного кода сервлета; компиляция страницы JSP — компилятор переведет полученный код в байт-код; загрузка класса; создание экземпляра; вызов метода **jspInit()**; вызов метода **_jspService()** для каждого запроса; вызов метода **jspDestroy()**.

Ответ: 3.

Вопрос 16.2.

Форма **<jsp:directive.include file="URL" />** JSP представляет вариант записи директивы **<%@ include %>**. Эта директива рассматривает ресурсы как статические объекты, в отличие от одноименного действия. При использовании действия **jsp:include** нет возможности использовать во вставляемом файле код JSP.

Ответ: 2.

Вопрос 16.3.

К неявным JSP-объектам относят: **request**, **response**, **out**, **pageContext**, **session**, **application**, **config**, **page**, **exception**.

Ответ: 1, 2, 3, 4, 5.

Вопрос 16.4.

В jsp определены три основные директивы: **page**, **include**, **taglib**.

Ответ: 1, 4, 5.

Ответы:

16.1. — 3 16.3. — 1,2,3,4,5
16.2. — 2 16.4. — 1,4,5

Глава 17

Вопрос 17.1.

Сессия может использоваться разными сервлетами для доступа к одному клиенту. Сессия пользователя может быть завершена вручную или, в зависимости от того, где запущен сервлет, автоматически. Метод **invalidate()** выдает **InvalidStateException**, если вызывается для сессии, которая была завершена. Отрицательное или нулевое значение времени жизни для сессии приводит к тому, что сессия никогда не завершается.

Ответ: 1, 3.

Вопрос 17.2.

Сессия клиента идентифицируется с помощью передачи идентификатора сессии (session id) между клиентом, осуществляющим запрос, и сервером.

Ответ: 1.

Вопрос 17.3.

Файлы Cookie хранятся на стороне клиента и отправляются клиентом серверу автоматически при осуществлении запроса.

Ответ: 5.

Вопрос 17.4.

Слушатели уровня сессии, определяющие жизненный цикл сессии — это: session creation, session invalidation, session timeout. Остальные относятся к событиям, изменяющим атрибуты сессии.

Ответ: 1,5,6.

Вопрос 17.5.

Фильтры могут изменять заголовок и содержимое запроса и ответа. Фильтры не создают запрос или ответ. Фильтр можно использовать для фильтрации более одного сервлета.

Ответ: 1, 4.

Ответы:

17.1. — 1,3	17.3. — 5	17.5. — 1,4
17.2. — 1	17.4. — 1,5,6	

Глава 18

Вопрос 18.1.

Директива **page** определяет атрибуты, относящиеся ко всей jsp-странице. Директива **include** позволяет вставлять текст или код в процессе трансляции страницы JSP в сервлет. Директива **taglib** объявляет, что данная страница JSP использует библиотеку тегов.

Ответ: 3.

Вопрос 18.2.

Выражения EL записываются внутри маркера **${ }.**

Ответ: 4.

Вопрос 18.3.

Библиотека Formatting tags(fmt) содержит теги **bundle** и **setBundle**, позволяющие обрабатывать ресурсные источники.

Ответ: 2.

Вопрос 18.4.

Значение слов-операторов: **eq** — проверка на равенство, **ne** — проверка на неравенство, **lt** — строго менее чем, **gt** — строго более чем, **le** — меньше либо равно чему-то, **ge** — больше или равно чему-то.

Ответ: 4.

Вопрос 18.5.

Оператор [ ] используется для доступа к элементам массива, списков типа **List** и отображений типа **Map**.

Ответ: 2.

Ответы:

18.1. — 3 18.3. — 2 18.5. — 2
18.2. — 4 18.4. — 4

Глава 19

Вопрос 19.1.

Если в определении тега отсутствует тело, метод **doStartTag()** должен возвратить константу **SKIP_BODY**, дающую указание системе игнорировать любое содержимое между начальными и конечными элементами создаваемого тега.

Ответ: 1.

Вопрос 19.2.

Метод **doInitBody()** вызывается один раз перед первой обработкой тела тега, **doEndTag()** — вызывается один раз, когда отработаны все остальные методы, **doAfterBody()** — вызывается после каждой обработки тела тега. **doStartTag()** вызывается, если обнаруживается начальный элемент тега.

Ответ: 3.

Вопрос 19.3.

Директива **taglib** содержит два атрибута: **uri** (копирует значение **<taglib-uri>** в **web.xml** или прямо указывает на **tld**-файл) и **prefix**.

Ответ: 2.

Вопрос 19.4.

Поддерживаются следующие значения для **body-content**: **empty** — пустое тело; **jsp** — тело состоит из всего того, что может находиться в JSP-файле; **tagdependent** — тело интерпретируется классом, реализующим данный тег.

Ответ: 1, 3, 5.

Вопрос 19.5.

При создании jsp-документа, регистрация пользовательской библиотеки тегов происходит в корневом элементе **jsp:root**.

Ответ: 1.

Ответы:

19.1. — 1 19.3. — 2 19.5. — 1
19.2. — 3 19.4. — 1,3,5

Приложение 1

JUnit

Оболочки модульного тестирования — это программные средства для разработки тестов, включающие: построение, выполнение тестов и создание отчетов. Первая оболочка модульного тестирования SUnit была создана Кентом Беком в 1999 году для языка Smalltalk. Позднее Эрик Гамма создал JUnit. Сейчас существует множество оболочек для модульного тестирования, которые известны как программные средства семейства XUnit. JUnit — реализация XUnit, наиболее широко используемая и расширенная версия оболочек модульного тестирования. JUnit создан на языке Java и используется для тестирования Java кода.

Модульное тестирование или unit testing — процесс проверки на корректность функционирования отдельных частей исходного кода программы путем запуска тестов в искусственной среде. Под частью кода в Java следует понимать исполняемый компонент. С помощью модульного тестирования обычно тестируют низкоуровневые элементы кода — такие, как методы. JUnit позволяет вне исследуемого класса создавать тесты, при выполнении которых произойдет корректное завершение программы. Кроме основного положительного сценария может выполняться проверка работоспособности системы в альтернативных сценариях, например, при генерации методом исключения как реакция на ошибочные исходные данные.

Оценивая каждую часть кода изолированно и подтверждая корректность ее работы, точно установить проблему значительно проще, чем если бы элемент был частью системы.

Технология позволяет и предлагает сделать более тесной связь между разработкой кода и его тестированием, а также предоставляет возможность проверить корректность работы класса, не прибегая к пробному выводу при отладке кода.

JUnit 4 в отличие от JUnit 3 полностью построен на аннотациях.

Для использования технологии необходимо загрузить библиотеку JUnit с сервера **junit.org** и включить архив **junit.jar** в список библиотек приложения.

При включении модульного тестирования в проект:

— тесты разрабатываются для нетривиальных методов системы;
— ошибки выявляются в процессе проектирования метода или класса;
— в первую очередь разрабатываются тесты на основной положительный сценарий;
— разработчику приходится больше уделять внимания альтернативным сценариям поведения, так как они являются источником ошибок, выявляемых на поздних стадиях разработки;

— разработчику приходится создавать более сфокусированные на своих обязанностях методы и классы, так как сложный код тестировать значительно труднее;
— снижается число новых ошибок при добавлении новой функциональности;
— устаревшие тесты можно игнорировать;
— тест отражает элементы технического задания, то есть некорректное завершение теста сообщает о нарушении технических требований заказчика;
— каждому техническому требованию соответствует тест;
— получение работоспособного кода с наименьшими затратами.

Аннотация @Test

Аннотация помечает метод как тестовый, что позволяет использовать возможности класса **org.junit.Assert** и запускать его в режиме тестирования. Тестовый метод должен всегда объявляться как **public void**. Аннотация может использовать параметры: **expected** — определяет ожидаемый класс исключения; **timeout** — определяет время, превышение которого делает тест ошибочным, применение которых будет рассмотрено ниже.

Пусть необходимо создать тест на метод, производящий простые вычисления студенческой стипендии.

/ # 1 # расчет стипендии в зависимости от повышающего коэффициента # IScholarshipCalculator.java # ScholarshipCalculatorImpl.java */*

```
package by.bsu.calculation;
public interface IscholarshipCalculator {
          public double scholarshipCalculate(double stepUpCoefficient);
}
package by.bsu.calculation;
public class ScholarshipCalculatorImpl implements IscholarshipCalculator{
          public static final double BASIC_SCHOLARSHIP = 100;
          public double scholarshipCalculate(double stepUpCoefficient) {
                    return BASIC_SCHOLARSHIP * stepUpCoefficient;
          }
}
```

Метод, предназначенный для функционирования в качестве теста, достаточно промаркировать аннотацией **@Test**. Простейший тест на метод будет выглядеть следующим образом.

/ # 2 # тест с аннотацией @Test # ScholarshipCalculatorTest.java */*

```
package test.by.bsu.calculation;
import org.junit.Assert;
import org.junit.Test;
import by.bsu.calculation.IscholarshipCalculator;
import by.bsu.calculation.ScholarshipCalculatorImpl;
```

```
public class ScholarshipCalculatorTest {
        @Test
        public void scholarshipCalculateTest() {
                IscholarshipCalculator scholarshipCalculator =
                                                        new ScholarshipCalculatorImpl();
                double basicScholarship = ScholarshipCalculatorImpl.BASIC_SCHOLARSHIP;
                double stepUpCoefficient = 1.1;
                double expected = basicScholarship * stepUpCoefficient;
                double actual = scholarshipCalculator
                                .scholarshipCalculate(stepUpCoefficient);
                // проверка на совпадение с погрешностью 0,01
                Assert.assertEquals(expected, actual, 0.01);
                // Assert.assertEquals( "Тест не прошел, т.к.", expected, actual, 0.01);
        // устаревшие варианты :
        // Assert.assertEquals(expected, actual); // на точное совпадение – deprecated
        // Assert.assertEquals( "Тест не прошел, т.к.", expected, actual); // deprecated
        }
}
```

Метод **assertEquals()** проверяет на равенство значений **expected** и **actual** с возможной погрешностью **delta**. При выполнении заданных условий сообщает об успешном завершении, в противном случае — об аварийном завершении теста. При аварийном завершении генерируется ошибка **java.lang.AssertionError**. Все методы класса **Assert** в качестве возвращаемого значения имеют тип **void**. Среди них можно выделить:

assertTrue(boolean condition)/assertFalse(boolean condition) — проверяет на истину/ложь значение **condition**;

assertSame(Object expected, Object actual) — проверяет, ссылаются ли ссылки на один и тот же объект;

assertNotSame(Object unexpected, Object actual) — проверяет, ссылаются ли ссылки на различные объекты;

assertNull(Object object)/assertNotNull(Object object) — проверяет, имеет или не имеет ссылка значение **null**;

assertThat(T actual, Matcher<T> matcher) — проверяет выполнение условия;

fail() — вызывает ошибку, используется для проверки, достигнута ли определенная часть кода или для заглушки, сообщающей, что тестовый метод пока не реализован.

Все перечисленные методы имеют перегруженную версию с параметром типа **String** в первой позиции, в который можно передавать сообщение, выводимое при аварийном завершении теста.

В дополнение к классу **Assert** с его методами жесткой проверки прохождения теста разработан класс **org.junit.Assume**. Методы этого класса в случае невыполнения предполагаемого условия при работе теста сообщают только о том, что предположение не исполнилось, не генерируя при этом никаких ошибок. Методы класса предполагают, что:

assumeNoException(Throwable t) — тестируемый метод завершится, не вызвав исключения;

assumeNotNull(Object…objects) — передаваемый аргумент(ы) не является ссылкой на **null**;

assumeThat(T actual, Matcher<T> matcher) — условие выполнится;

assumeTrue(boolean b) — значение передаваемого аргумента истинно.

Фикстуры

Фикстура (Fixture) — состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Может быть представлено набором каких-либо объектов, состоянием базы данных, наличием определенных файлов, соединений и проч.

В версии JUnit 4 аннотации позволяют исполнять одну и ту же фикстуру для каждого теста или всего один раз для всего класса, или не исполнять ее совсем. Предусмотрено четыре аннотации фикстур — две для фикстур уровня класса и две для фикстур уровня метода.

— **@BeforeClass** — запускается только один раз при запуске теста.
— **@Before** — запускается перед каждым тестовым методом.
— **@After** — запускается после каждого метода.
— **@AfterClass** — запускается после того, как отработали все тестовые методы.

Использование фикстур позволяет выделить этапы и точно определять моменты, например, создания/удаления объекта, инициализации необходимых ресурсов, очистки памяти и проч.

Пусть метод **stepUpCoefficientCalculate(int averageMark)** возвращает повышающий коэффициент стипендии в зависимости от среднего балла студента.

/ # 3 # тестируемый класс # ScholarshipCalculatorImpl.java */*

```
package by.bsu.calculation;
public class ScholarshipCalculatorImpl implements IScholarshipCalculator {
          public static final double BASIC_SCHOLARSHIP = 100;
          public double scholarshipCalculate(double stepUpCoefficient) {
                    return BASIC_SCHOLARSHIP * stepUpCoefficient;
          }
          public double stepUpCoefficientCalculate(int averageMark) {
                    double stepUpCoefficient;
                    switch (averageMark) {
                       case 3:
                              stepUpCoefficient = 1;
                              break;
                       case 4:
                              stepUpCoefficient = 1.3;
                              break;
```

```
                    case 5:
                        stepUpCoefficient = 1.5;
                        break;
                    default:
                        stepUpCoefficient = 0;
                }
                return stepUpCoefficient;
        }
}
```

Фикстурой **@Before** задан момент создания объекта класса перед каждым тестом.

/ # 4 # тест с фикстурами @Before и @After # ScholarshipCalculatorTest2.java */*

```
package test.by.bsu.calculation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTest2 {
        private ScholarshipCalculatorImpl scholarshipCalculator;
        @Before
        public void initScholarshipCalculator() {
                scholarshipCalculator = new ScholarshipCalculatorImpl();
        }
        @After
        public void clearScholarshipCalculator() {
                scholarshipCalculator = null;
        }
        @Test
        public void stepUpCoefficientForFiveTest() {
                double expected = 1.5;
                double actual = scholarshipCalculator.stepUpCoefficientCalculate(5);
                assertEquals("Coefficient for mark 5 is wrong:", expected, actual, 0.01);
        }
        @Test
        public void stepUpCoefficientForThreeTest() {
                double expected = 1;
                double actual = scholarshipCalculator.stepUpCoefficientCalculate(3);
                assertEquals("Coefficient for mark 3 is wrong:", expected, actual, 0.01);
        }
}
```

Метод **initScholarshipCalculator()** выполнит инициализацию поля **scholarshipCalculator** перед запуском каждого тестового метода. Метод **clearScholarshipCalculator()** очистит это поле после завершения работы каждого из тестовых методов.

Тест можно переписать с использованием фикстуры **@BeforeClass**, чье использование позволит создавать только один экземпляр класса, на котором и будут выполнены оба теста.

/ # 5 # тест с фикстурой @BeforeClass # ScholarshipCalculatorTest3.java */*

```
package test.by.bsu.calculation;
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.Test;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTest3 {
          private static ScholarshipCalculatorImpl scholarshipCalculator;
          @BeforeClass
          public static void initScholarshipCalculator() { // static обязателен
                    scholarshipCalculator = new ScholarshipCalculatorImpl();
          }
          @Test
          public void stepUpCoefficientForFiveTest() {
                    double expected = 1.5;
                    double actual = scholarshipCalculator.stepUpCoefficientCalculate(5);
                    assertEquals("Coefficient for mark 5 is wrong:", expected, actual, 0.01);
          }
          @Test
          public void stepUpCoefficientForThreeTest() {
                    double expected = 1;
                    double actual=scholarshipCalculator.stepUpCoefficientCalculate(3);
                    assertEquals("Coefficient for mark 3 is wrong:", expected, actual , 0.01);
          }
}
```

Метод **initScholarshipCalculator()** отработает один раз перед вызовом первого тестового метода. Метод, отмеченный аннотацией **@BeforeClass** или **@AfterClass**, должен быть статическим.

JUnit 4 позволяет указать для тестового сценария более одной фикстуры. Новые фикстуры на основе аннотаций не препятствуют созданию нескольких фикстурных методов **@BeforeClass**. Однако их порядок исполнения задать нельзя.

Тестирование исключительных ситуаций

При тестировании альтернативных сценариев работы метода часто требуется точно определить тип генерируемого методом исключения на основе переданных некорректных параметров. Если тест выдает исключение, то инфраструктура тестирования сообщает о корректном результате его исполнения.

Аннотацию **@Test** при необходимости тестирования генерации конкретного исключения следует использовать с параметром **expected**. Параметр предназначен

для задания типа исключения, которое данный тест должен генерировать в процессе своего выполнения.

/ # 6 # метод с инструкцией throws # */*

```
public double stepUpCoefficientCalculate(int averageMark) throws NotSuchMarkException {
          double stepUpCoefficient;
          switch (averageMark) {
            case 2:    stepUpCoefficient = 0;
                     break;
            case 3:
                     stepUpCoefficient = 1;
                     break;
            case 4:
                     stepUpCoefficient = 1.3;
                     break;
            case 5:
                     stepUpCoefficient = 1.5;
                     break;
            default:
                     throw new NotSuchMarkException("There is no mark: " + averageMark);
          }
          return stepUpCoefficient;
}
```

Метод **stepUpCoefficientCalculate(int averageMark)** генерирует исключение **NotSuchMarkException**, если число, переданное в метод, не входит в интервал возможных оценок.

/ # 7 # исключение # NotSuchMarkException.java */*

```
package by.bsu.calculation;
public class NotSuchMarkException extends Exception {
          public NotSuchMarkException() {
          }
          public NotSuchMarkException(String message) {
                     super(message);
          }
}
```

Тогда метод, тестирующий генерацию исключения, будет записан в виде:

/ # 8 # метод тестирования генерации исключения # */*

```
@Test( expected = NotSuchMarkException.class )
public void stepUpCoefficientForElevenTest() throws NotSuchMarkException {
          double expected = 1;
          double actual = scholarshipCalculator.stepUpCoefficientCalculate(11);
          Assert.assertEquals("For mark 11 wasn't exception:", expected, actual, 0.01);
}
```

Тест завершится успешно лишь в том случае, если возникнет исключительная ситуация. Исключение **NotSuchMarkException** необходимо указать в секции **throws** тестового метода по правилам работы с проверяемыми исключениями.

Может возникнуть необходимость проверить не только возникновение исключительной ситуации, но и текст сообщения в экземпляре исключения. В этом случае лучше прибегнуть к обычному подходу без параметра **expected**.

/ # 9 # метод тестирования сообщения в исключении после его генерации # */*

```
@Test
public void stepUpCoefficientForElevenTest() {
          int averageMark = 11;
          try {
                    scholarshipCalculator.stepUpCoefficientCalculate(averageMark);
                    fail("Test for mark " + averageMark
                                        + " should have thrown a NotSuchMarkException");
          } catch (NotSuchMarkException e) {
                    Assert.assertEquals("There is no mark: " + averageMark, e.getMessage());
          }
}
```

В блоке **try**, если исключение не возникло, вызывается метод **fail()**, сигнализирующий о провале теста. В блоке **catch**, если исключительная ситуация произошла, проверяется на эквивалентность текстов сообщения об ошибке.

Ограничение по времени

В качестве параметра тестового сценария аннотации **@Test** может быть использовано значение лимита времени **timeout**. Параметр **timeout** определяет максимальный временной промежуток в миллисекундах, отводимый на исполнение теста. Если выделенное время истекло, а тест продолжает выполняться, то тест завершается неудачей.

/ # 10 # тест с ограничением по времени # ScholarshipCalculatorTestTime.java */*

```
package test.by.bsu.calculation;
import org.junit.Assert;
import org.junit.Test;
import by.bsu.calculation.IScholarshipCalculator;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTestTime {
          private static ScholarshipCalculatorImpl scholarshipCalculator;
          @BeforeClass
          public static void initScholarshipCalculator() { // static обязателен
                    scholarshipCalculator = new ScholarshipCalculatorImpl();
          }
          @Test( timeout = 10 ) // заменить на 50
          public void scholarshipCalculateTest() {
```

```
                for (int i = 1; i < 100_000; i++) {
                        double stepUpCoefficient = 1 / i;
                        double expected = 100 * stepUpCoefficient;
                        double actual =
                        scholarshipCalculator.scholarshipCalculate(stepUpCoefficient);
                        Assert.assertEquals(expected, actual, 0.01);
                }
        }
}
```

Через 10 миллисекунд после запуска тест провалится, так как на выполнение метода уходит несколько больше времени. Если увеличить время до 50 миллисекунд, то тест пройдет успешно.

Игнорирование тестов

При контроле корректности функционарования бизнес-логики приложений п до появления JUnit 4 игнорирование неудачных, незавершенных или устаревших тестов представляло определенную проблему. Аннотация **@Ignore** заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Аннотация предусматривает наличие комментария о причине игнорирования теста, полезного при следующем к нему обращении.

/ # 11 # игнорирование теста # */*

```
@Ignore("this test is not ready yet")
@Test
public void scholarshipCalculateTest() {
          double expected = 10 * stepUpCoefficient;
          double actual = scholarshipCalculator.scholarshipCalculate(stepUpCoefficient);
          Assert.assertEquals(expected, actual, 0.1);
}
```

Правила

Аннотация **@Rule** позволяет более гибко работать с классами утилитами, определяющими правила работы с тестами. Некоторые классы пакета **org.junit.rules** повторяют функциональность параметров аннотации **@Test**, а именно, класс **Timeout** определяет таймаут для теста, класс **ExpectedException** — ожидаемое исключение. С другой стороны, класс **TemporaryFolder** дает возможность протестировать возможность управления временными файлами и директориями.

Пусть в класс **ScholarshipCalculatorImpl** добавлена возможность печати отчета в файл в методе **writeResult(File f)**. Тогда с использованием правил можно создать тест для метода записи в файл и два теста, проверяющих генерацию исключений.

```
/* # 12 # тест с правилами # ScholarshipCalculatorTestRule.java */
package test.by.bsu.calculation;
import java.io.File;
import java.io.IOException;
import org.junit.BeforeClass;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import org.junit.rules.TemporaryFolder;
import org.junit.rules.Timeout;
import by.bsu.calculation.ScholarshipCalculatorImpl;
public class ScholarshipCalculatorTestRule {
          private static ScholarshipCalculatorImpl scholarshipCalculator;
          @Rule
          public final TemporaryFolder folder = new TemporaryFolder();
          @Rule
          public final Timeout timeout = new Timeout(100);
          @Rule
          public final ExpectedException thrown = ExpectedException.none();
          @BeforeClass
          public static void initScholarshipCalculator() {
                    scholarshipCalculator = new ScholarshipCalculatorImpl();
          }
          @Test
          public void writeResultTest() throws IOException {
                    File file = folder.newFile("result.txt");
                    scholarshipCalculator.writeResult(file);
          }
          @Test
          public void stepUpCoefficientCalculateTest() throws NotSuchMarkException {
                    thrown.expect(NotSuchMarkException.class);
                    scholarshipCalculator.stepUpCoefficientCalculate(11);
          }
          @Test
          public void writeResultTestTwo() throws IOException {
                    thrown.expect(NullPointerException.class);
                    scholarshipCalculator.writeResult(null);
          }
}
```

В методе **expect()** класса **ExpectedException** следует указать класс ожидаемого исключения.

Временные файлы по окончании теста уничтожаются.

Метод **writeResult()** класса **ScholarshipCalculatorImpl** представлен в виде:

```
public void writeResult(File f) throws IOException {
          FileWriter fw = new FileWriter(f);
          fw.append(this.toString());
          fw.flush();
          fw.close();
}
```

Наборы тестов и параметризованные тесты

При контроле корректности функционарования бизнес-логики приложений приходится создавать тесты, число которых достаточно велико и непостоянно. Аннотация **@RunWith** задает способ запуска теста, в случае **@RunWith(Suite.class)** — запуск набора тестов, перечисляемых в аннотации **@Suite.SuiteClasses**.

/ # 13 # запуск набора тестов # ScholarshipSuite.java */*

```
package test.by.bsu.calculation;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@Suite.SuiteClasses( { ScholarshipCalculatorTest.class, ScholarshipCalculatorTestTime.class } )
@RunWith(Suite.class)
public class ScholarshipSuite {
}
```

Наличие класса после описания способа группы тестов в общем случае — необходимая формальность.

JUnit 4 позволяет создавать тест, который может работать с различными наборами значений параметров, что позволяет разработать единый тестовый сценарий и запускать его несколько раз — по числу наборов параметров. Запуск параметризованного теста с набором данных также требует аннотации **@RunWith(Parameterized.class)**, где в качестве значения передается указание на способ запуска. У класса должны быть поля по числу параметров в каждом наборе. Аннотация **@Parameters** маркирует статический метод, который возвращает данные для теста, представленные типом **Collection**. Кроме того, необходим конструктор, связывающий данные для теста и поля класса.

/ # 14 # запуск параметризованного набора для теста # ScholarshipCalculatorTest4.java */*

```
package test.by.bsu.calculation;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
@RunWith(Parameterized.class)
public class ScholarshipCalculatorTest4 {
          // объявление параметров в виде полей
          private int averageMark;
          private double stepUpCoefficient;
          // public-конструктор с параметрами для инициализации полей
          public ScholarshipCalculatorTest(int averageMark, double stepUpCoefficient) {
                    this.averageMark = averageMark;
                    this.stepUpCoefficient = stepUpCoefficient;
          }
```

```
        // определение набора параметров в виде коллекции
        @Parameters public static Collection<Object[]> stepUpCoefficientTableValues() {
                return Arrays.asList(new Object[][] {
                                        { 3, 1.0 },
                                        { 4, 1.3 },
                                        { 5, 1.5 }
                                });
        }
        @Test
        public void stepUpCoefficientTest() throws NotSuchMarkException {
                ScholarshipCalculatorImpl scholarshipCalculator =
                                                        new ScholarshipCalculatorImpl();
                double expected = this.stepUpCoefficient;
                double actual =
                        scholarshipCalculator.stepUpCoefficientCalculate(this.averageMark);
                assertEquals(expected, actual, 0.01);
        }
}
```

Тест будет запущен по числу наборов данных, в данном случае — четыре раза.

События

При выполнении больших или сложных тестов бывает недостаточно информации от JUnit о процессе их выполнения. Возникает необходимость документирования результатов работы или выполнения определенных программных действий. Для реализации таких возможностей применяется модель прослушивания событий.

Чтобы реализовать прослушивание событий, следует создать подкласс класса **org.junit.runner.notification.RunListener** и зарегистрировать обработчик событий в блоке прослушивания с помощью класса-регистратора наследника класса **org.junit.runners.BlockJUnit4ClassRunner**. Класс, содержащий тесты и заинтересованный в обработке событий, необходимо пометить аннотацией **@RunWith** с указанием класса регистратора событий.

Класс **RunListener** позволяет переопределять следующие методы:

testStarted(Description description) — вызывается перед запуском каждого теста. В параметр **description** передается имя метода и класса, запустившего тест;

testFinished(Description description) — вызывается после успешного завершения каждого теста;

testIgnored(Description description) — вызывается, если тест не был запущен, как правило, из-за аннотации **@Ignore**;

testFailure(Failure failure) — вызывается при неудачном завершении теста. Параметр **failure** содержит информацию о неудачном тесте и об исключении, сгенерированном в итоге;

testAssumptionFailure(Failure failure) — вызывается, если не выполнено условие в методе класса **Assume**;

testRunFinished(Result result) — вызывается после завершения всех тестов. Параметр **result** содержит информацию о числе успешных/провальных тестов, о времени выполнения и проч.;

testRunStarted(Description description) — вызывается перед запуском всех тестов.

Примерная реализация класса обработчика событий:

/ # 15 # реализация обработки событий # ScholarshipRunListener.java */*

```
package test.by.bsu.calculation;
import org.junit.runner.Description;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
import org.junit.runner.notification.RunListener;
public class ScholarshipRunListener extends RunListener {
          @Override
          public void testStarted(Description description) throws Exception {
                    System.out.println("тест стартовал: " + description.getMethodName());
          }
          @Override
          public void testFinished(Description description) throws Exception {
                    System.out.println("тест завершен: " + description.getMethodName()
                                        + "\n----");
          }
          @Override
          public void testFailure(Failure failure) throws Exception {
                    System.out.println("тест провален с исключением: "
                                        + failure.getException());
          }
          @Override
          public void testIgnored(Description description) throws Exception {
                    System.out.println("тест игнорирован: " + description.getMethodName()
                                        + "\n----");
          }
          @Override
          public void testRunFinished(Result result) throws Exception {
                    System.out.println("результаты выполнения тестов:");
                    System.out.println("время выполнения: (" + result.getRunTime()
                                        + ") millis");
                    System.out.println("было запущено тестов: " + result.getRunCount());
                    System.out.println("провалено тестов: " + result.getFailureCount());
                    System.out.println("игнорировано тестов: " + result.getIgnoreCount());
                    System.out.println("все тесты завершены успешно: "
                                        + result.wasSuccessful());
          }
}
```

Регистрация обработчика событий:

/ # 16 # регистрация обработчика # ScholarshipRunner.java */*

```
package test.by.bsu.calculation;
import org.junit.runners.BlockJUnit4ClassRunner;
import org.junit.runners.model.InitializationError;
import org.junit.runner.notification.RunNotifier;
public class ScholarshipRunner extends BlockJUnit4ClassRunner {
          private ScholarshipRunListener runListener;
          public ScholarshipRunner(Class<ScholarshipRunListener> clazz)
                                                              throws InitializationError {
                    super(clazz);
                    runListener = new ScholarshipRunListener();
          }
          public void run(RunNotifier notifier) {
                    notifier.addListener(runListener);
                    super.run(notifier);
          }
}
```

Для демонстрации документирования обработки событий использован класс с тестами **ScholarshipCalculatorTestRule,** несколько измененый так, чтобы были задействованы все предопределенные методы класса **RunListener**, а именно: успешный тест, заведомо неудачный тест и пропускаемый тест.

/ # 17 # тест, запускаемый с обработкой событий # ScholarshipCalculatorTestRule.java */*

```
package test.by.bsu.calculation;
import java.io.File;
import java.io.IOException;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import org.junit.rules.TemporaryFolder;
import org.junit.rules.Timeout;
import org.junit.runner.RunWith;
import by.bsu.calculation.ScholarshipCalculatorImpl;
@RunWith(ScholarshipRunner.class)
public class ScholarshipCalculatorTestRule {
          private static ScholarshipCalculatorImpl scholarshipCalculator;
          @BeforeClass
          public static void initScholarshipCalculator() {
                    scholarshipCalculator = new ScholarshipCalculatorImpl();
          }
          @Rule
          public final TemporaryFolder folder = new TemporaryFolder();
          @Rule
          public final Timeout timeout = new Timeout(300);
```

```
        @Rule
        public final ExpectedException thrown = ExpectedException.none();
        @Test
        public void writeResultTest() throws IOException {
                File file = folder.newFile("a:/result.txt"); // тест будет провален
                scholarshipCalculator.writeResult(file);
        }
        @Test
        public void stepUpCoefficientCalculateTest() throws NotSuchMarkException {

                thrown.expect(NotSuchMarkException.class);
                scholarshipCalculator.stepUpCoefficientCalculate(11);
        }
        @Ignore("this test is not ready yet")
        @Test
        public void writeResultTestTwo() throws IOException {
                thrown.expect(NullPointerException.class);
                scholarshipCalculator.writeResult(null);
        }
}
```

В результате отработки теста обработчиком событий в консоль будет выведен отчет:

тест стартовал: stepUpCoefficientCalculateTest
тест завершен: stepUpCoefficientCalculateTest

тест стартовал: writeResultTest
тест провален с исключением: java.io.IOException: The filename, directory name, or volume label syntax is incorrect
тест завершен: writeResultTest

тест игнорирован: writeResultTestTwo

результаты выполнения тестов:
время выполнения: (57) millis
было запущено тестов: 2
провалено тестов: 1
игнорировано тестов: 1
все тесты завершены успешно: false

Приложение 2

Log4J

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, сбоев и т. д. Существуют различные API регистрации сообщений и ошибок, среди которых на данный момент можно выделить следующие.

1. **Log4J.** Apache Log4J был первым регистратором. Изначально обладает качественной архитектурой, в следствие чего быстро занял доминирующие позиции и применяется в большинстве промышленных приложений. Разработан в рамках проекта Jakarta Apache.
2. **java.util.logging.** Пакет появился в JavaSE в версии 1.4 в 2001 году после появления Log4J. Возможностей фреймворк предоставляет меньше, чем Log4J, тем не менее, у **java.util.logging** есть некоторое преимущество — он является частью JavaSE.
3. **Apache Commons Logging.** Предназначен для абстрагирования разработчика от конкретной библиотеки логгирования. Он предоставляет некоторый унифицированный интерфейс, транслируя его вызовы в использование конкретных возможностей фреймворков. Commons Logging абстрагирует следующие фреймворки: Log4J, java.util.logging, Avalon LogKit, Lumberjack.
4. **SLF4J.** Simple Logging Facade for Java абстрагирует еще больше технологий логгирования, чем Commons Logging.
5. **Logback.** Расширение Log4J с добавлением к нему новых возможностей.

Любой регистратор событий состоит из трех элементов:

— собственно регистрирующего — logger;
— направляющего вывод — appender;
— форматирующего вывод — layout.

В итоге logger регистрирует и направляет вывод события в пункт назначения, определяемый направляющим элементом, в формате, заданном форматирующим элементом.

Log4j

В современном практическом программировании представляет основной инструмент журналирования событий. Формирует журнал сообщений (отладочных, информационных, системных, security, сообщений об ошибках). Log4j

можно загрузить по адресу: **http://logging.apache.org/log4j/**. Перед использованием необходимо зарегистрировать библиотеку **log4j-[***версия***].jar** в приложении.

Logger

Основным элементом API регистрации событий и ошибок является регистратор **org.apache.log4j.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. Apache Log4J поддерживает несколько способов конфигурации. Позволяет управлять своим поведением во время исполнения.

Каждый класс приложения может иметь свой собственный logger или быть прикреплен к общему для всего приложения. Регистраторы образуют иерархию, как и пакеты Java. Каждый логгер имеет имя, описывающее иерархию, к которой он принадлежит. Разделителем является точка. Принцип полностью аналогичен формированию имени пакета в Java.

Регистратор может быть создан или получен с помощью статического метода **getLogger(String name)** или **getLogger(Class name)**, где **name** — имя пакета или класса. На вершине иерархии находится корневой регистратор. Он всегда существует и у него нет имени. Ссылку на корневой регистратор можно получить статическим методом **getRootLogger()**.

У каждого регистратора есть уровень сообщения по возрастанию (**TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, **FATAL, OFF**), который управляет выводом сообщений. Для вывода сообщений конкретного уровня используются методы **trace()**, **debug()**, **info()**, **warn()**, **error()**, **fatal()**. Чтобы вывести информацию о возникшем исключении в качестве второго параметра, в перечисленные 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** < **OFF**), 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 применяется уровень, унаследованный от его родителя. Уровень корневого регистратора **DEBUG**. Таким образом, сообщения, выводимые с уровнем ниже установленного, в лог не попадут. И в этом заключается основное преимущество — можно вставлять в программный код вывод информации на различных уровнях (об ошибках — на уровне **ERROR**, о нормальном ходе выполнения — на уровне **INFO**, отладочную — на уровне **DEBUG**), а потом гибко регулировать, что именно будет выводиться.

Некоторые общие 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

Вывод регистратора может быть направлен в различные места назначения: файл, консоль и т. д. Каждому из них соответствует класс, реализующий интерфейс **org.apache.log4j.Appender**. Кроме того, вывод в базу данных можно произвести с помощью класса **JDBCAppender**, в журнал событий ОС — **NTEventLogAppender**, на SMTP-сервер — **SMTPAppender**.

Если логгер — это та точка, откуда уходят сообщения в коде, то аппендер — это та точка, куда они приходят в конечном итоге. Например, файл или консоль. Список таких точек, поддерживаемых Log4J:

— консоль;
— файлы (несколько различных типов);
— JDBC;
— темы (topics) JMS;
— NT Event Log;
— SMTP;
— Сокет;
— Syslog;
— Telnet;
— любой **java.io.Writer** или **java.io.OutputStream**.

Существует возможность написать собственный класс аппендер и использовать его.

Основными аппендерами, использующимися наиболее широко, являются файловые аппендеры. Их есть несколько типов:

— **org.apache.log4j.FileAppender**
— **org.apache.log4j.RollingFileAppender**
— **org.apache.log4j.DailyRollingFileAppender**

Логгеры связываются с аппендерами в соотношении «многие ко многим» — у одного логгера может быть несколько аппендеров, а к одному аппендеру может быть привязано несколько логгеров. Важно понимать, что аппендеры наследуются от родительских логгеров.

Уровень логирования наследуется (или устанавливается) независимо от аппендера. Иначе, если на логгере root сконфигурирован вывод в вывод в консоль с уровнем **ERROR**, а на дочернем логгере — в файл с уровнем **INFO**, то вывод в дочерний логгер с уровнем **INFO** попадет и в файл, и в консоль.

Существует возможность отказаться от наследования аппендеров. Для этого логгеру надо выставить свойство **additivity** в **false**, по умолчанию оно выставлено

в **true**. Конкретные детали описания и использования наиболее употребимых аппендеров будут приведены ниже.

Layout

Вывод регистратора может иметь различный формат. Каждый формат представлен классом, производным от **Layout**. Все методы класса **Layout** предназначены только для создания подклассов.

Для конфигурирования формата вывода используются наследники класса **org.apache.log4j.Layout**:

— **org.apache.log4j.SimpleLayout**
— **org.apache.log4j.HTMLLayout**
— **org.apache.log4j.xml.XMLLayout**
— **org.apache.log4j.PatternLayout**

Установить **Layout** для **FileAppender** или **ConsoleAppender** можно с помощью метода **setLayout(Layout layout)** или передать его в конструкторы перечисленных классов.

org.apache.log4j.SimpleLayout — наиболее простой вариант. На выходе читается уровень вывода и сообщение.

org.apache.log4j.HTMLLayout — данный компоновщик форматирует сообщения в виде HTML-страницы.

org.apache.log4j.xml.XMLLayout — формирует сообщения в виде XML.

org.apache.log4j.PatternLayout и **org.apache.log4j.EnhancedPatternLayout —** используют шаблонную строку для форматирования выводимого сообщения.

Например:

```
<layout class="org.apache.log4j.PatternLayout" >
          <param name="ConversionPattern"
                    value="%d{dd.MM.yyyy HH:mm:ss} [%t] %-5p %c - %m%n"/>
</layout>
```

Данный форматтер принимает параметром **ConversionPattern** — шаблон вывода лога, где

%d{DATE} — выводит дату-время. В скобках можно указать собственный формат вывода. Также применимы именованые шаблоны, а именно **DATE**, **ISO8601** и **ABSOLUTE**. Последний содержит формат **HH:mm:ss,SSS**, подходящий для логов с кратким сроком хранения. По умолчанию дата будет выведена в формате **ISO8601**;

%t — выводит имя потока, выводящего сообщение, для однопоточного приложения будет выводить **main**;

%5p — выводит уровень лога (**ERROR**, **DEBUG**, **INFO** и пр.), где цифра указывает число выводимых символов, если символов меньше, то сообщение будет дополнено пробелами;

%c{6} — категория с числом выдаваемых уровней. Категорией в общем случае будет имя класса с пакетом. Обычно это строка, где уровни разделены точками. По умолчанию без {} будет выводить полный путь к корню проекта. Верхний уровень при значении **1** будет выводить только имя класса;

%M — имя метода, в котором произошел вызов записи в лог;

%L — номер строки, в которой произошел вызов записи в лог;

%m — собственно сообщение, передаваемое в лог;

%n — перевод строки.

Конфигурация

Перед использованием Log4j его необходимо сконфигурировать. Конфигурирование осуществляется способами — через файл свойств, через xml-файл, через программное конфигурирование и по умлчанию. Преимущество следут отдать способам, использующим конфигурационные файлы, как динамичным и легко изменяемым. Более удобным для понимания считается xml-конфигурирование.

Простейший конфигурационный файл **log4j.xml** может находиться в корне проекта в виде:

```
<?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="Encoding" value="UTF-8" />
                <param name="File" value="logs/log.txt" />
                <layout class="org.apache.log4j.SimpleLayout" />
        </appender>
        <logger name="by.bsu">
                <level value="debug" />
        </logger>
        <root>
                <appender-ref ref="TxtAppender" />
        </root>
</log4j:configuration>
```

В итоге создан файловый аппендер с именем **TxtAppender** для записи в файл **logs/log.txt** с поддержкой кодировки UTF-8 и упрощенным компоновщиком. Сконфигурирован корневой логгер уровня **debug**. Аппендер **FileAppender** добавляет данные в файл до бесконечности, поэтому файл может быть очень большого размера, что значительно усложняет его чтение при превышении разумных размеров. Запись в большой текстовый файл также может замедлять работу системы. В чистом виде практически не используется. Он является основой для других, предлагая ключевые способы взаимодействия с файлами. Поддерживает следующие свойства: **append** — дописывать

существующий файл или каждый раз создавать новый, **bufferedIO** — буферизовать ли вывод в файл (по умолчанию **false**), **file** — имя файла, **encoding** — кодировка вывода, **bufferSize** — размер буфера (по умолчанию 8192).

Подключается данная конфигурация вызовом метода **doConfigure()** класса **org.apache.log4j.xml.DomConfigurator**:

```
new DOMConfigurator().doConfigure("log4j.xml", LogManager.getLoggerRepository());
```

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).

/ # 1 # регистратор ошибок # DemoLog.java */*

```
package by.bsu.log4j.base;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
public class DemoLog {
          static {
               new DOMConfigurator().doConfigure("log4j.xml", LogManager.getLoggerRepository());
          }
          static Logger logger = Logger.getLogger(DemoLog.class);
          public static void main(String[ ] args) {
                    try {
                              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 than 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;
          }
}
```

Вывод регистратора **"by.bsu.log4j.base.DemoLog"**, в файл **log.txt** будет следующим:

DEBUG - Argument n is 9
INFO - Result is 362880
ERROR - negative argument:
java.lang.IllegalArgumentException: argument -3 less than zero
at by.bsu.log4j.base.DemoLog.factorial(DemoLog.java:22)
at by.bsu.log4j.base.DemoLog.main(DemoLog.java:14)

Порции выводимых данных в Log4J называются сообщениями.

Для вывода одновременно в текстовый файл и в файл в виде XML необходимо добавить следующую информацию о новом аппендере с простым XML компоновщиком:

```
<appender name="XMLAppender" class="org.apache.log4j.FileAppender">
          <param name="File" value="logs/log.xml" />
          <layout class="org.apache.log4j.XMLLayout"/>
</appender>
```

и добавить строку о появлении еще одного корневого логгера:

```
<root>
          <appender-ref ref="TxtAppender" />
          <appender-ref ref="XMLAppender" />
</root>
```

Таким образом, логгеров может быть несколько и использовать их можно не только для дублирования информации в разные точки сохранения, но и независимо друг от друга.

Файл **log.xml** будет содержать следующую информацию:

```
<log4j:event logger="by.bsu.log4j.base.DemoLog" timestamp="1355923596119" level="DEBUG"
thread="main">
<log4j:message><![CDATA[Argument n is 9]]></log4j:message>
</log4j:event>
<log4j:event logger="by.bsu.log4j.base.DemoLog" timestamp="1355923596121" level="INFO"
thread="main">
<log4j:message><![CDATA[Result is 362880]]></log4j:message>
</log4j:event>
<log4j:event logger="by.bsu.log4j.base.DemoLog" timestamp="1355923596122" level="ERROR"
thread="main">
<log4j:message><![CDATA[negative argument: ]]></log4j:message>
<log4j:throwable><![CDATA[java.lang.IllegalArgumentException: argument -3 less than zero
          at by.bsu.log4j.base.DemoLog.factorial(DemoLog.java:27)
          at by.bsu.log4j.base.DemoLog.main(DemoLog.java:19)
]]></log4j:throwable>
</log4j:event>
```

Если использовать вместо компоновки **SimpleLayout**

```
<layout class="org.apache.log4j.xml.SimpleLayout"/>
```

компоновку **PatternLayout**, приведенную выше в виде

```
<layout class="org.apache.log4j.PatternLayout" >
         <param name="ConversionPattern"
                  value="%d{dd.MM.yyyy HH:mm:ss} [%t] %-5p %c - %m%n"/>
</layout>
```

то в файл **log.txt** будет выведено

```
19.12.2012 17:21:59 [main] DEBUG by.bsu.log4j.base.DemoLog - Argument n is 9
19.12.2012 17:21:59 [main] INFO  by.bsu.log4j.base.DemoLog - Result is 362880
19.12.2012 17:21:59 [main] ERROR by.bsu.log4j.base.DemoLog - negative argument:
java.lang.IllegalArgumentException: argument -3 less than zero
        at by.bsu.log4j.base.DemoLog.factorial(DemoLog.java:27)
        at by.bsu.log4j.base.DemoLog.main(DemoLog.java:19)
```

Следует привести еще один востребованный способ конфигурирования с помощью файлов **properties**:

```
PropertyConfigurator.configure("log4j.properties");
```

Пример простейшего файла **log4j.properties** для вывода ссобщений на консоль:

```
log4j.rootLogger=debug, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.target=System.out
log4j.appender.stdout.layout=org.apache.log4j.SimpleLayout
```

У компоновщика **HTMLLayout** есть два свойства — **Title** и **LocationInfo**, задающие заголовок HTML-документа и режим вывода информации о точке, где сгенерировано сообщение: имя файла и номер строки в нем. По умолчанию **LocationInfo** имеет значение **false**.

Файл **log4j.properties** для вывода сообщений в html-файл:

```
log4j.rootLogger = DEBUG, html
log4j.appender.html=org.apache.log4j.FileAppender
log4j.appender.html.File=logs/log.html
log4j.appender.html.layout=org.apache.log4j.HTMLLayout
log4j.appender.html.layout.Title=HTML Layout Example
log4j.appender.html.layout.LocationInfo=true
```

У данного компоновщика есть существенный недостаток: формат HTML требует корректного закрытия документа. А при генерации вывода непрерывно добавляются сообщения, т. е. строки в таблицу, и автоматического закрытия документа не происходит.

Log4j поддерживает конфигурирование по умолчанию без всяких файлов, только при этом все сообщения будут выводиться на консоль:

```
BasicConfigurator.configure();
```

Те же действия можно выполнить, не прибегая к конфигурационным файлам непосредственно в коде приложения. С помощью метода **addAppender(Appender newAppender)** класса **Logger** можно добавить **Appender** 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 filename)**. По умолчанию любые сообщения, записываемые в файл, будут добавляться к уже имеющимся. Но с помощью метода **setAppend(boolean append)** это можно отменить, сбросив флаг **append**.

```
ConsoleAppender consAppender = new ConsoleAppender(new SimpleLayout());
FileAppender xmlAppender = new FileAppender(new XMLLayout(), "logs/log.xml");
          logger.addAppender(consAppender);
          logger.addAppender(xmlAppender);
FileAppender fileAppender = new FileAppender(new SimpleLayout(), "logs.log.txt");
          logger.addAppender(fileAppender);
          logger.setLevel(Level.DEBUG);
```

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.

Программная конфигурация логгеров используется редко из-за своей громоздкости и отсутствии гибкост при изменении настроек.

При конфигурировании Log4j для веб-проекта следует поместить код процесса конфигурации в метод **init()** сервлета:

```
String prefix = getServletContext().getRealPath("/");
String filename = getInitParameter("init_log4j");
if (filename != null) {
          PropertyConfigurator.configure(prefix + filename);
}
```

где путь вместе с именем файла вида **WEB-INF/classes/log4j.properties** объявляется в параметрах инициализации сервлета.

Аппендеры RollingFileAppender и DailyRollingFileAppender

Аппендер **RollingFileAppender** позволяет создавать новый файл по достижении определенного размера. «Создавать» — означает изменить имя текущего файла путем добавления ему расширения «.0» и открыть следующий. По достижении им максимального размера — первому вместо расширения «.0» выставляется «.1», текущему — «.0», открывается следующий. А именно:

logs.txt.2
logs.txt.1
logs.txt.0
logs.txt

Максимальный размер файла и максимальный индекс, устанавливаемый сохраняемым предыдущим файлам, задаются свойствами **maximumFileSize** и **maxBackupIndex** соответственно. Если индекс должен быть превышен — файл не переименовывается, а удаляется. Таким образом, всегда будет в наличии больше определенного количества файлов, каждый из которых не больше определенного объема.

Пример применения **RollingFileAppender** в **log4j.properties**:

```
log4j.rootLogger=info, fileout
log4j.appender.fileout =org.apache.log4j.RollingFileAppender
log4j.appender.fileout.file=logs/log.txt
log4j.appender.fileout.file.maxBackupIndex=10
log4j.appender.fileout.maximumFileSize=15KB
log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern=%p %d %t %c - %m%n
```

При этом каждый раз при превышении файлом размера, указанного в свойстве **maximumFileSize**, будет создаваться новый файл.

Аппендер **DailyRollingFileAppender** в отличие от **RollingFileAppender**, создающего новый файл по достижении определенного размера, **DailyRollingFileAppender** создает файл с определенной частотой, которая зависит от шаблона, указанного в конфигурации:

'.'yyyy-MM — раз в месяц,
'.'yyyy-ww — раз в неделю,
'.'yyyy-MM-dd — раз в день,
'.'yyyy-MM-dd-a — раз в полдня,
'.'yyyy-MM-dd-HH — раз в час,
'.'yyyy-MM-dd-HH-mm — раз в минуту.

В кавычках в начале шаблона указан символ, который будет использоваться как разделитель между значением даты-времени и именем файла. При создании к имени файла в конце приписываются текущие дата и время, отформатированные согласно указанному шаблону (с помощью класса **java.text.SimpleDateFormat**).

log_time.txt
log_time.txt.2013-01-30-14-29
log_time.txt.2013-01-30-14-30

```
log4j.rootCategory=INFO, fileout
log4j.appender.fileout = org.apache.log4j.DailyRollingFileAppender
log4j.appender.fileout.File=logs/log_time.txt
log4j.appender.fileout.Append = true
log4j.appender.fileout.DatePattern ='.'yyyy-MM-dd-HH-mm
log4j.appender.fileout.layout = org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss} %c{3} [%p] %m%n
```

Этот аппендер может быть весьма удобен в случае, если организована автоматическая архивация лога. Наличие в имени файла метки времени делает его по определению уникальным.

Фильтры

Необходимость записывать не все сообщения, а только удовлетворяющее некоторым условиям, реализуется с использованием фильтров. Стандартные фильтры бывают двух видов: **LevelMatchFilter** — пишет сообщения заданного уровня и **LevelRangeFilter** — пишет сообщения в диапазоне уровней.

Например:

```
LevelMatchFilter filter = new LevelMatchFilter();
filter.setLevelToMatch("INFO");
```

в конфигурационном файле в теле тега **<appender>** вставляется тег **<filter>** в виде:

```
<filter class="org.apache.log4j.varia.LevelRangeFilter">
          <param name="LevelMax" value="WARN"/>
          <param name="LevelMin" value="INFO"/>
</filter>
```

Для создания собственного фильтра требуется создать подкласс класса **org.apache.log4j.spi.Filter**. Каждому фильтру соответсвует **Appender**. Отношение между ними может быть установлено как многие ко многим. При вызове логгирующего метода генерируется объект события **LoggingEvent**, которое и передается в метод **int decide()** фильтра. Метод, в свою очередь, должен вернуть результат своей работы в виде одного из значений: **Filter.ACCEPT** — разрешение записать сообщение в лог, **Filter.NEUTRAL** — передать дальше по цепочке фильтров или записать, если фильтр последний, **Filter.DENY** — запрет записи сообщения и обрыв цепочки.

Пусть существует журнал событий, который фиксирует сообщения, связанные с коллекцией монет. Но логгировать события следует только для монет, чьи идентификационные номера больше некоторого значения, попадают в определенный диапазон или удовлетворяют другим условиям.

Чтобы решить поставленную задачу, следует в реализации метода **decide()** указать условия разрешения/игнорирования фиксации события.

/ # 2 # собственный фильтр # CoinFilter.java */*

```
package by.bsu.log.filter;
import org.apache.log4j.spi.Filter;
import org.apache.log4j.spi.LoggingEvent;
import by.bsu.log.entity.Coin;
public class CoinFilter extends Filter {
public int decide(LoggingEvent event) {
                    int result = Filter.NEUTRAL;
                    Object object = event.getMessage();
                    if (object instanceof Coin) {
                              Coin coin = (Coin) object;
                              int id = coin.getId();
```

```
                    // игнорировать, если id меньше 1000, записывать – если больше
                              result = id < 1_000 ? Filter.DENY : Filter.ACCEPT;
                    }
                    return result;
          }
}
```

Из экземпляра события извлекается объект-сообщение, на основе содержимого которго и осуществляется решение.

/ # 3 # бизнес-класс # Coin.java */*

```
package by.bsu.log.entity;
public class Coin {
          private int id;
          private int value;
          private String currencyName;
          public Coin() {
          }
          public Coin(int id, int value, String currencyName) {
                    this.id = id;
                    this.value = value;
                    this.currencyName = currencyName;
          }
          public String getCurrencyName() {
                    return this.currencyName;
          }
          public int getId() {
                    return this.id;
          }
          public int getValue() {
                    return this.value;
          }
}
```

Как экземпляр класса **Coin** попадает в событие? Методы класса **Logger**, отвечающие за запись, принимают сообщение в виде экземпляра класса **Object**. В общем случае туда передаются объекты типа **String**, информация из которых выводится в место назначения в виде логов. Но при этом генерируется событие **LoggingEvent**, в которое и помещается экземпляр-сообщение.

```
Coin coin = // init object
logger.info(coin);
```

Программисту только и остается с помощью фильтра определить реакцию на контент сообщения, какого бы типа он не был.

Конфигурационный файл **log4j.xml** будет выглядеть:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
```

```
        <renderer renderedClass="by.bsu.log.entity.Coin"
                  renderingClass="by.bsu.log.renderer.CoinRenderer"/>
        <appender name="ConsAppender" class="org.apache.log4j.ConsoleAppender">
                  <layout class="org.apache.log4j.PatternLayout">
                            <param name="ConversionPattern" value="%t %-5p %c{5} - %m%n" />
                  </layout>
                  <filter class="by.bsu.log.filter.CoinFilter" />
        </appender>
        <logger name="by.bsu">
                  <level value="debug" />
                  <appender-ref ref="ConsAppender" />
        </logger>
</log4j:configuration>
```

Способ вывода сообщения и его внешний вывод при использовании фильтра также должен быть реализован разработчиком. После возвращения методом **decide()** значения **Filter.ACCEPT** управление передается переопределенному методу **doRender()** интерфейса **ObjectRenderer**, который определяет формат и содержание выводимого сообщения.

/ # 4 # построение сообщения для регистратора # CoinRenderer.java */*

```
package by.bsu.log.renderer;
import org.apache.log4j.or.ObjectRenderer;
import by.bsu.log.entity.Coin;
public class CoinRenderer implements ObjectRenderer {
          public String doRender(Object obj) {
                    StringBuilder builder = new StringBuilder(32);
                    if (obj instanceof Coin) {
                              Coin coin = (Coin) obj;
                              String currency = coin.getCurrencyName();
                              int id = coin.getId();
                              int value = coin.getValue();
                              builder.append( id + ": " + value + "(" + currency + ")");
                    }
                    return builder.toString();
          }
}
```

/ # 5 # регистратор ошибок # FilterDemoLog.java */*

```
package by.bsu.log.base;
import java.util.ArrayList;
import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import org.apache.log4j.xml.DOMConfigurator;
import by.bsu.log.entity.Coin;
public class FilterDemoLog {
          static {
              new DOMConfigurator().doConfigure("log4j.xml", LogManager.getLoggerRepository());
          }
```

```
        private static Logger logger = Logger.getLogger(FilterDemoLog.class);
        public static void main(String args[]) {
                ArrayList<Coin> list = new ArrayList<Coin>() {
                        {
                                this.add(new Coin(956, 1, "$"));
                                this.add(new Coin(3462, 10, "руб"));
                                this.add(new Coin(758, 2, "тенге"));
                                this.add(new Coin(2101, 5, "zl"));
                        }
                };
                for (Coin coin: list) {
                        logger.info(coin);
                }
        }
}
```

В консоль будут выведены сообщения только о двух монетах из четырех.

main INFO by.bsu.log.base.FilterDemoLog - 3462: 10(руб)
main INFO by.bsu.log.base.FilterDemoLog - 2101: 5(zl)

Из фильтров можно построить цепочку фильтров.

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 (Unified Modeling 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 все;
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 чертеж полочки для книг — это уже UML. Таким образом, графические наброски проекта всегда рядом с человеком. Для промышленного программирования, выполняемого большими и зачастую распределенными 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. Также UML популярен и вне пределов IT 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 скептически относиться к этому. С трудом можно себе представить, хотя многие менеджеры мечтают об этом, что будет создан инструмент, в котором, нарисовав все и нажав «волшебную кнопку», на выходе получишь готовое приложение.

Задача состоит в том, чтобы получить оценку того, в чем диаграммы могут быть полезны и в чем — нет.

UML — семейство графических нотаций, базирующихся на единой метамодели, помогающее 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 в том, 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 до такой степени подробно, что он станет 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 определяют 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 являются 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 средство обмена информацией между разработчиками.

Если не придерживаться согласованного понимания, то другие разработчики просто не поймут, что вы хотели выразить своей диаграммой.

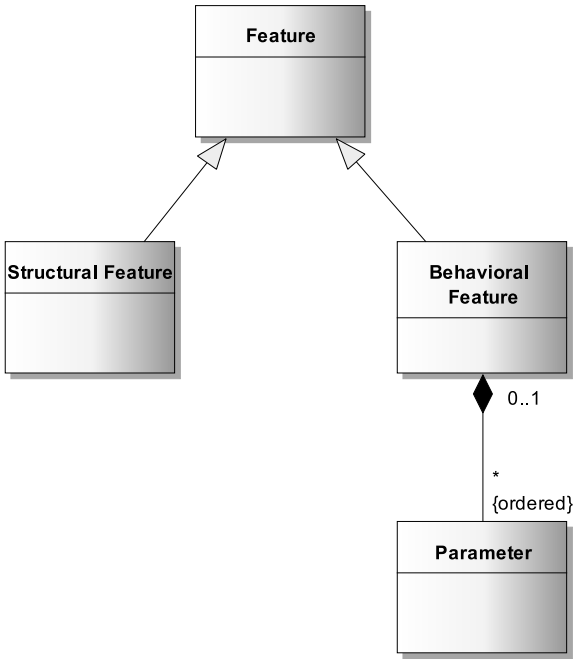

Рис. 1. *Нотации и метамодель*

— Activity — процедурное и параллельное поведение. Введено в UML 1;
— Class — классы, свойства и взаимоотношения. Введено в UML 1;
— Communication — взаимодействие между объектами; 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 распределение во времени. Новая в UML 2;
— Use case — способы взаимодействия пользователей с системой. Введено в UML 1.

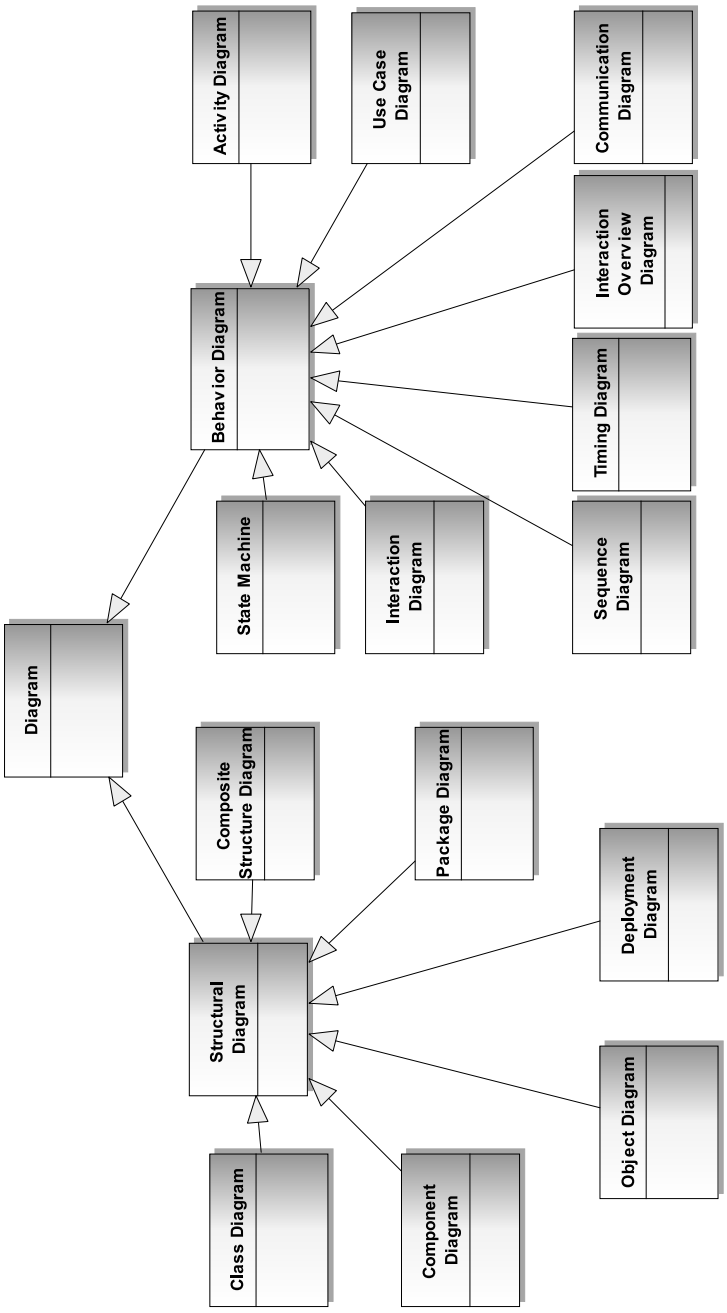

Рис. 2. *Диаграммы UML*

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 все находится в реальном мире и все не так просто. Каким языком считать UML? Предписывающим, как языки программирования, или описательным? Сложившееся понимание в IT индустрии на данный момент — рассматривать UML как описательный язык. И в этом случае все, 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

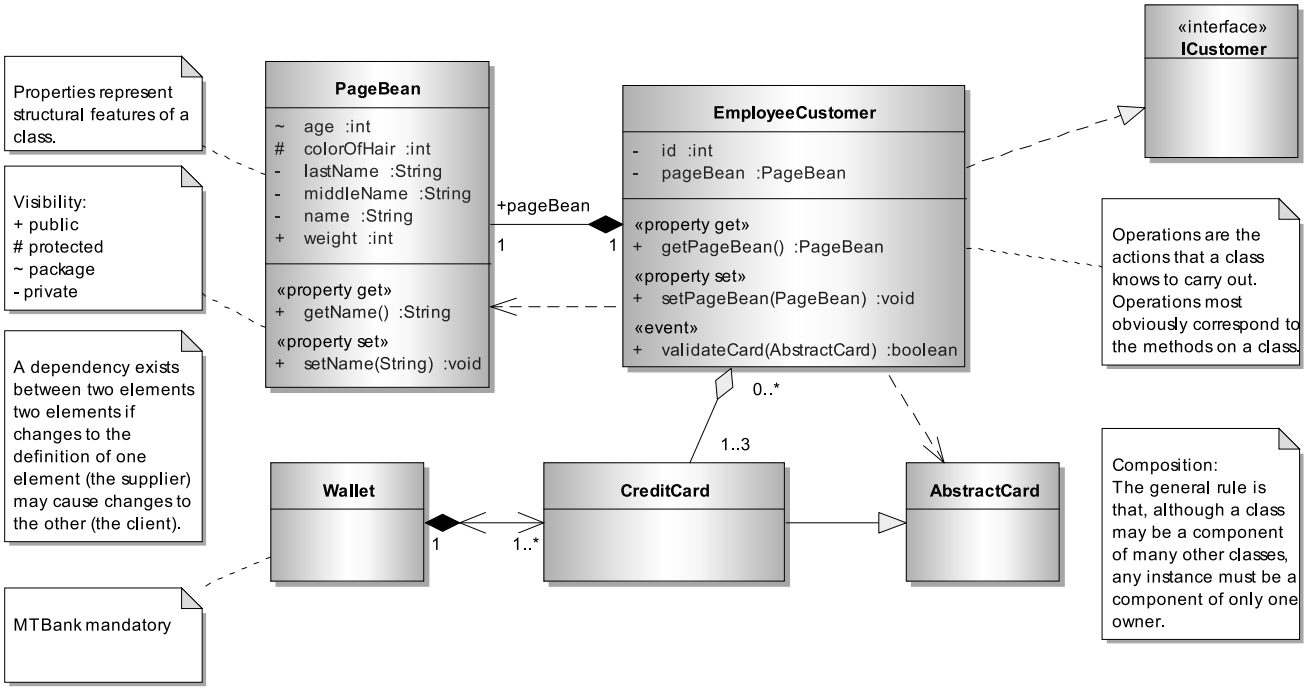

Рис. 3. *«Свойства» классов*

(методов) класса. Следует различать свойства как поднабор операторов следующих контракту Java Beans — **get***ИмяСвойства*(), **set***ИмяСвойства*().

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 отличающимися 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.

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.

Подкласс создается в случаях, если:

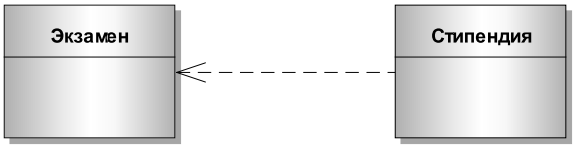

Рис. 4. *Отношение зависимости*

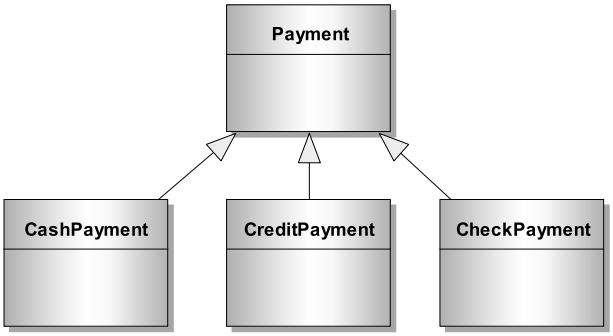

Рис. 5. *Отношение обобщения*

— 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.

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.

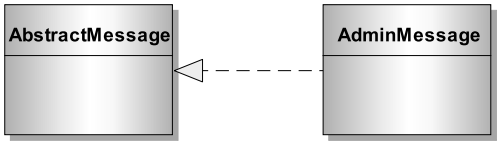

Рис. 6. *Отношение реализации*

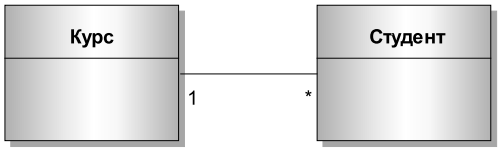

Рис. 7. *Отношение ассоциации*

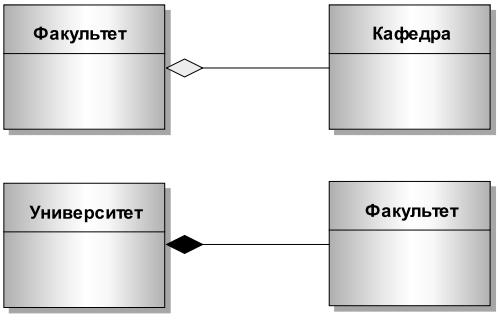

Рис. 8. *Отношения коллективной агрегации и композиции*

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.

Множественность

В общем случае множественность определяет нижнюю и верхнюю границу количества объектов, которые могут соответсвововать свойствам.

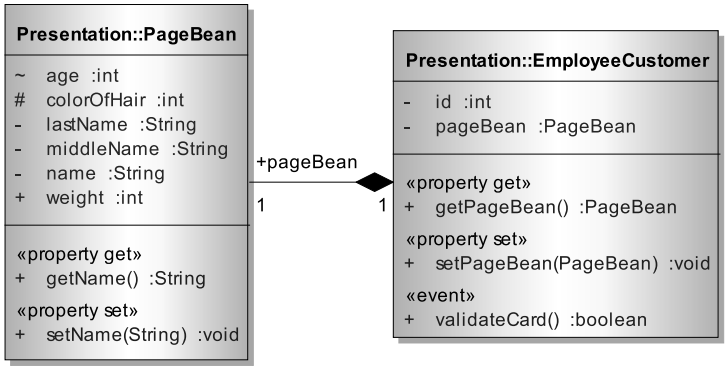

Рис. 9. *Множественность*

```
/* # 1 # множественность отношений между классами # EmployeeCustomer.java */

public class EmployeeCustomer {
          private int id;
          private PageBean pageBean;

          public int getId() {
                    return id;
```

```
        }
        public void setID(int id) {
                this.id = id;
        }
        public PageBean getPageBean() {
                return pageBean;
        }
        // more code here
}
```

Двунаправленная ассоциация

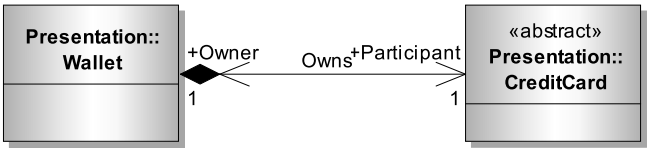

Рис. 10. *Двунаправленная ассоциация*

/ # 2 # взаимная видимость классов # Wallet.java # CreditCard.java */*

```
public class Wallet {
        private List<CreditCard> creditCards;
        public void addCreditCard(CreditCard newCreditCard) {
                creditCards.add(newCreditCard);
        }
}
public class CreditCard {
        private Wallet wallet;

        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 их модель. 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. Все 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 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 все 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. Кодом 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 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 стандартизирован, 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:

В состав дистрибутива СУБД входят и API-библиотеки, позволяющие исполнять SQL-запросы из написанного пользователем программного кода.

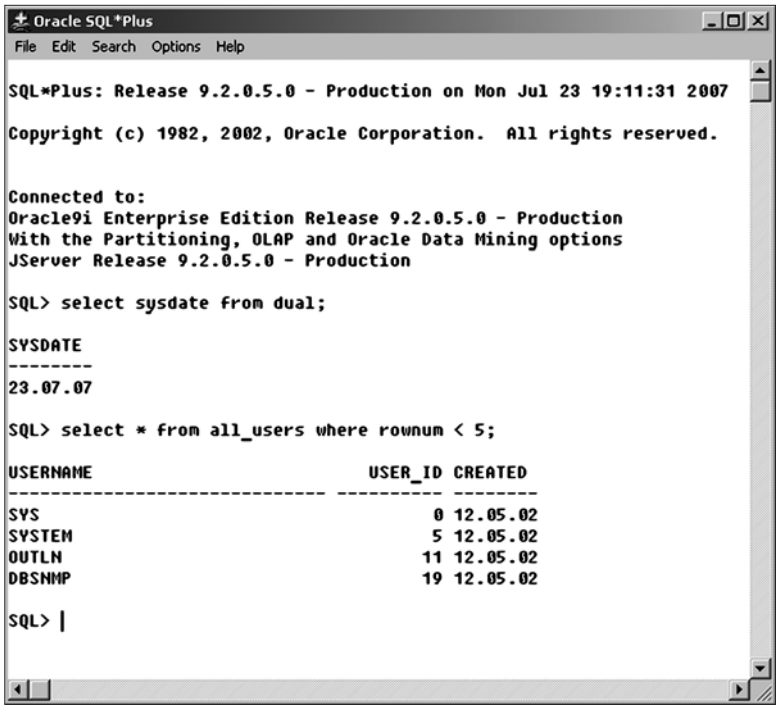

Рис. 1. *Интерактивная утилита*

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 талеров 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
	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>]
         [GROUP 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
```

Символ «*» 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	Инженер	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 = Departments.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: Си**д**оров

Выражение [**NOT**] **IN** проверяет значения на [не]вхождение в определенное множество:

```
SELECT * FROM Employees WHERE Position IN ('Начальник', 'Доярка')
```

Выражение [**NOT**] **BETWEEN** проверяет значения на [не]попадание в некоторый диапазон:

```
SELECT * FROM Employees WHERE Salary BETWEEN 500 AND 1000
```

Этот запрос вернет список работников, зарплата которых больше либо равна 500 талерам и меньше либо равна 1000 талерам.

Несколько условий поиска могут комбинироваться посредством логических операторов **AND**, **OR** или **NOT**:

```
SELECT *
          FROM Employees
          WHERE Position IN ('Начальник', 'Доярка')
                    AND Salary BETWEEN 500 AND 1000
SELECT *
          FROM Employees
          WHERE (Position = 'Начальник' OR Position = 'Доярка')
                    AND Salary BETWEEN 500 AND 1000
SELECT *
          FROM Employees
          WHERE NOT (Position = 'Начальник' OR Position = 'Доярка')
```

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 приведенном примере в результат попадут только отделы, максимальная зарплата в которых превышает 1000 талеров:

```
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>)] [<ограничение целостности уровня колонки>]
[, <имя колонки> <тип колонки>[(<размер колонки>)] [<ограничение 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 связывания>]
          [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

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 язык, основанный на использовании командной строки. 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 разветвленную систему целей, состоящую из множества мелких задач. Каждая задача является объектом, который наследует соответствующий интерфейс класса **Task**. Все это дает 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, OS/2 Warp, Novell Netware и MacOS X.

Чтобы обрабатывать и использовать Ant, необходимо иметь JAXP-compliant XML-parser (он есть в любой версии JDK), установленным и включенным 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:\jdk1.7
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.

Тег 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.

Тег Target

Цель может зависеть от других целей. Например, имеются две цели: для компиляции и для изъятия файлов из 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 ее 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 нее.

Цель также имеет возможность быть исполненной только в случае, если определенный 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	Разделенный запятыми список имен 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, запятую и пробел.

Пример **build-**файла:

```
<?xml version='1.0' encoding='UTF-8'?>
<project name="NewProject" 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 все из ${build} в NewProject-${DSTAMP}.jar файл -->
    <jar jarfile="${dist}/lib/NewProject-${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 все 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\NewProject-20070815.jar
BUILD SUCCESSFUL
Total time: 2 seconds
```

Тег 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 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=2014
```

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" filtering="true">
        <filterset refid="copy.filterset" />
    </copy>
</target>
```

Содержимое исходного текстового файла:

```
# file.txt
Version is @version@
```

В результате будет получено:

```
# 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 только с определенным ранее путем.

В качестве сокращения <**classpath**> поддерживает **path** и **location** атрибуты так:

```
<classpath>
  <pathelement path="${classpath}"/>
</classpath>
```

Может быть сокращено до:

```
<classpath path="${classpath}"/>
```

В дополнение **DirSet**, **FileSet** и **FileList** 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 логин
-noinput	Запрещает интерактивный ввод информации
-buildfile <file>	Использовать данный **build**-файл (по умолчанию берется **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**

Коротко о задачах 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
Pre-process Tasks
Property Tasks
Remote Tasks
SCM Tasks
Testing Tasks

Краткое описание основных:

Archive Tasks

Имя задачи	Описание
Jar	Упаковывает в Jar набор файлов
Unzip	Распаковывает **zip** архивы
Zip	Создает **zip** архивы

Compile Tasks

Имя задачи	Описание
Javac	Компилирует определенные исходные файлы внутри запущенной Ant'ом VM или с помощью новой VM, если **fork** атрибут определен
JspC	Запускает JSP-компилятор. Используется для предварительной компиляции JSP-страниц для более быстрого запуска их с сервера, или при отсутствии JDK на нем, 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** определен, команда исполняется, только если Ant запущен под определенную систему
Java	Исполняет Java класс внутри запущенной (Ant) VM или с помощью другой, если **fork** атрибут определен

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 определенном каталоге, или набор файлов, определенных одним или несколькими **FileSet'**ами
Mkdir	Создает директорию. Несуществующие 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 путем 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 определенное 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 своем 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**}, являющиеся Java кодами и не содержащие текста **Test** в своем имени.

PatternSet

Шаблоны могут быть сгруппированы в наборы и позже использованы путем 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, определенные 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/>
```

эквивалентны.

```
<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

Создается 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.newdomain;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;
```

```
public class NewVeryOwnTask 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="newtask" classname="com.newdomain.NewVeryOwnTask"/>

  <target name="main">
    <newtask message="Hello World! My NewVeryOwnTask 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="newtask"
        classname="com.newdomain.NewVeryOwnTask"
        classpath="build"/>
  </target>

  <target name="main" depends="declare">
   <newtask message="Hello World! My NewVeryOwnTask works!"/>
  </target>
</project>
```

Приложение 6

JPA

Объектно-реляционное преобразование (ORM) — простое сохранение Java объектов в реляционную базу данных — в настоящее время используется в большинстве проектов. Среди технологий, поддерживающих ORM: Entity Beans 2.x, TopLink, iBatis, EclipseLink, Hibernate, JDO, и даже JDBC с DAO и собственными аннотациями. На основе принципов организации перечисленных выше технологий экспертная группа Java EE разработала Java Persistence API.

Java Persistence API предоставляет POJO persistence модель объектно-реляционного преобразования. Создана экспертной группой EJB 3.0, однако ее применение не ограничивается компонентами EJB. Она может быть использована веб-приложениями и клиентами напрямую, и даже вне Java EE платформы в Java SE приложениях.

ORM технологии предоставляют следующие возможности:

— декларативный способ реализации ORM или ORM метаданных позволяет соотнести объекты с одной или несколькими таблицами из базы данных. Для хранения метаданных используются аннотации или XML;
— API для манипулирования сущностями. API позволяет сохранять, восстанавливать, обновлять или удалять объекты, освобождая программиста от написания JDBC и SQL кода;
— язык запросов для восстановления объектов из базы данных. Это один из наиболее важных элементов, так как некорректные SQL утверждения могут замедлить базу данных. Этот подход также освобождает приложение от внутренних SQL, разбросанных по всему приложению.

Java Persistence API, под названием Criteria, предоставляет стандартный механизм для ORM, EntityManager API для создания, обновления и удаления объектов и язык запросов JPA-QL или JPQL для извлечения сущностей, запросы которого могут быть проверены на корректность на этапе компиляции и могут быть динамически сформированы на этапе выполнения приложения

В EJB3 JPA аннотации используются для определения объектов и отношений. JPA также поддерживает вариант использования XML описания, однако аннотации значительно упрощают разработку.

Аннотации можно разбить на две категории: логические аннотации (позволяют описать объектную модель, связи между классами) и физические аннотации (описывающие физическую схему, таблицы, столбцы, индексы и т. д.).

Объявление сущности

Сущность (Entity) — простой объект Plain Ordinary Java Object (POJO), который требуется сохранить в реляционной базе данных. Как и любой POJO, сущность может быть представлена как абстрактным, так и конкретным классом, и даже наследником другого POJO класса. Чтобы объявить класс POJO сущностью, его нужно отметить аннотацией **@Entity**:

/ # 1 # обьявление класса сущности, для связи с таблицей в БД # Student.java */*

```
package by.bsu.jpa.entity;
import javax.persistence.*; // пакет содержит EJB3 аннотации
@Entity
public class Student implements Serializable {
          @Id
          private Long id;
          public Long getId() {
                    return id;
          }
          public void setId(Long id) {
                    this.id = id;
          }
}
```

Каждая сущность имеет первичный ключ. Задать сохраняемому полю свойство первичного ключа можно с помощью аннотации **@Id**. Сущность управляет его состоянием, используя либо поля, либо *get* и *set* методы. Это зависит от того, где используется аннотация.

Классу **Student** соответствует канонический статический класс метамодели:

```
import javax.persistence.metamodel.SingularAttribute;
@javax.persistence.metamodel.StaticMetamodel(domain.Student.class)
public class Student_ {
          public static volatile SingularAttribute<Student,Long> id;
}
```

Все сохраняемые атрибуты класса **domain.Student** представлены и в классе метамодели в виде открытых статических переменных-членов типа **SingularAttribute<Student, ?>**. Поэтому можно ссылаться на атрибуты **domain.Student** не через механизм рефлексии, а непосредственно на этапе компиляции, используя соответствующие статические члены, например **Student_.id**.

Определение таблицы

Аннотация **@Table** указывается на уровне класса. Она позволяет указать таблицу, каталог и схему, которые соответствуют сущности. Если эта аннотация не указана, в качестве имени таблицы воспринимается имя класса.

```
@Entity
@Table(name="students")
public class Student implements Serializable { /* more code */ }
```

Можно также указать уникальные столбцы с помощью аннотации **@UniqueConstraint**:

```
@Table(name="students", uniqueConstraints = {@UniqueConstraINT(columnNames={"lastName"})})
```

Но если при создании базы данных этот constraint не применялся, то и при выполнении запросов из JPA он будет проигнорирован. Обратить внимание на то, что массив **columnNames** ссылается на логические имена столбцов.

Persistence Contexts

Persistence context — множество управляемых сущностей, в котором каждому уникальному идентификатору соответствует единственная сущность. В persistence context сущности и их жизненный цикл управляются экземпляром класса **javax.persistence.EntityManager**.

В среде Java EE транзакция JTA обычно включает параллельные вызовы нескольких компонентов. Они часто должны иметь доступ к одному и тому же persistence context в пределах одной транзакции. Для облегчения такого использования объекта **EntityManager** при его создании persistence context автоматически наследуется из текущей транзакции. Это позволяет избежать передачи одного объекта **EntityManager** между компонентами. В этом случае экземпляр **EntityManager** управляется контейнером Java EE так же, как и его жизненный цикл.

В редких случаях приложению может понадобиться доступ к автономному persistence context. В этом случае используется метод **createEntityManager()** интерфейса **javax.persistence.EntityManagerFactory**. Тогда экземпляр класса **EntityManager** управляется приложением.

Создание экземпляра EntityManager

Для получения объекта **EntityManager**, управляемого JEE контейнером, используется аннотация **@PersistenceContext** с возможным атрибутом **name**, указывающим имя persistence unit.

Для получения объекта, управляемого приложением, нужно вызвать статический метод **EntityManagerFactory.*createEntityManager()***. Создание объекта **EntityManagerFactory** требует достаточно много ресурсов, к тому же его методы потокобезопасны, поэтому этот объект создается один раз за все время работы приложения. Получить объект можно либо с помощью аннотации

@PersistenceUnit в технологии Java EE, либо с помощью статического метода ***createEntityManagerFactory()*** класса **javax.persistence.Persistence** в Java SE.

Управление транзакциями

В зависимости от типа транзакции объекта **EntityManager** транзакции контролируются либо JTA, либо локальным EntityTransaction API.

В первом случае объекты **EntityManager** используют текущую JTA транзакцию, которая начинается и фиксируется вне объекта **EntityManager**.

Во втором случае объект **EntityManager** использует транзакцию, связанную с источником данных через persistence provider. Такие объекты **EntityManager** могут использовать серверные или локальные источники. В этом случае транзакция управляется методами интерфейса **EntityTransaction**. Получить объект, реализующий этот интерфейс, можно с помощью статического метода **EntityManager.*getTransaction()***.

Компоновка сущностей

Компоновка сущностей осуществляется с помощью persistence unit. Persistence unit — это логическая группировка, которая включает:

— **EntityManagerFactory** и ее **EntityManager** вместе с их конфигурационной информацией;
— множество управляемых классов;
— метаданные преобразований, которые указывают соответствия с базой данных.

Persistence unit описывается в файле конфигурации **persistence.xml**, который располагается в папке **META-INF** проекта. Структура этого файла следующая: в корневом элементе **<persistence>** находится один или несколько элементов **<persistence-unit>**. Этот элемент имеет атрибуты **name** и **transaction-type**, где последний может принимать значения **JTA** или **RESOURCE_LOCAL**. Конфигурация persistence unit определяется элементами:

— **description** — описание persistence-unit;
— **provider** — имя класса, реализующего интерфейс **PersistenceProvider** из пакета **javax.persistence.spi**;
— **jta-data-source, non-jta-data-source** — указывают глобальное JNDI имя соответствующего источника данных;
— **class** — аннотированный класс сущности;
— **properties** — специфические свойства, зависящие от реализации: параметры соединения с базой данных, указание уровня логирования и т. п.

В итоге для веб-приложения с технологией EclipseLink конфигурационный файл при использовании **javax.sql.DataSource** с СУБД MySQL выглядит приблизительно так:

 # 2 # конфигурационный файл компоновки сущностей для веб-приложения # persistence.xml

```
<persistence>
          <persistence-unit name="UniversityPU" transaction-type="JTA">
                    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
                    <jta-data-source>java:/comp/env/jdbc/UniversityDS</jta-data-source>
          </persistence-unit>
</persistence>
```

Класс-провайдер **oracle.toplink.essentials.PersistenceProvider** для технологии TopLink.

Файл **context.xml** конфигурации сервера Tomcat должен содержать определение пула соединений:

```
<Context antiJARLocking="true">
            <Resource name="jdbc/UniversityDS"
            auth="Container"
            type="javax.sql.DataSource"
            maxActive="32"
            maxIdle="8"
            maxWait="10000"
            username="root"
            password="pass"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/university?autoReconnect=true"
            />
</Context>
```

Файл **glassfish-resources.xml** конфигурации сервера GlassFish должен содержать определение пула соединений:

```
<resources>
    <jdbc-connection-pool
allow-non-component-callers="false"     associate-with-thread="false"
connection-creation-retry-attempts="0"  connection-creation-retry-interval-in-seconds="10"
connection-leak-reclaim="false"         connection-leak-timeout-in-seconds="0"
connection-validation-method="auto-commit"
datasource-classname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
fail-all-connections="false"            idle-timeout-in-seconds="300"
is-connection-validation-required="false"
is-isolation-level-guaranteed="true"    lazy-connection-association="false"
azy-connection-enlistment="false"       match-connections="false"
max-connection-usage-count="0"          max-pool-size="32"
max-wait-time-in-millis="60000"         name="mysql_university_rootPool"
non-transactional-connections="false"   pool-resize-quantity="2"
res-type="javax.sql.DataSource"         statement-timeout-in-seconds="-1"
steady-pool-size="8"                    validate-atmost-once-period-in-seconds="0"
wrap-jdbc-objects="false">
        <property name="serverName" value="localhost"/>
```

```
        <property name="portNumber" value="3306"/>
        <property name="databaseName" value="university"/>
        <property name="User" value="root"/>
        <property name="Password" value="pass"/>
        <property name="URL"
value="jdbc:mysql://localhost:3306/university?zeroDateTimeBehavior=convertToNull"/>
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource
enabled="true"
jndi-name="jdbc/UniversityDS"
object-type="user"
pool-name="mysql_university_rootPool"/>
</resources>
```

Для Java SE приложений в примерах этой главы пул соединений применен не будет.

Простое приложение

В качестве примера рассматривается простейшее приложение, сохраняющее в базу данных информацию о студенте. Эта информация будет храниться в базе данных MySQL. Из внешних библиотек понадобятся драйвер для базы данных, а также реализация JPA. В качестве последней можно взять реализацию EclipseLink. Для подключения необходимо щелкнуть правой кнопкой мыши по проекту, выбрать *Properties -> Project Facets-> Further configuration available*. Затем в выпадающем списке Platform выбрать EclipseLink2.3.*x* и в пункте *Type* выбрать *User Library* и скачать соответствующую библиотеку.

Фреймворк EclipseLink включает основные библиотеки **eclipselink.jar** и **javax.persistence_2.0.4.v[***версия***].jar**, а также ряд вспомогательных библиотек.

Далее создается класс сущности **Student** со следующими полями: **id** — идентификатор (номер зачетки), **firstName** — имя, **lastName** — фамилия.

Указывается, что класс является сущностью:

```
@Entity
public class Student implements Serializable
```

Определяется идентификатор и стратегия его генерирования:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

Указываются поля и соответствующие столбцы в таблице:

```
@Column(name = "FIRST_NAME")
@Size(max = 15)
private String firstName;
```

```
@Column(name = "LAST_NAME")
@Size(max = 25)
private String lastName;
```

Создается новая база данных *university* с единственной таблицей *student*:

```
CREATE DATABASE 'university' DEFAULT CHARACTER SET utf8;
USE 'university ';
CREATE TABLE 'student' (
 'id' INT(11) NOT NULL AUTO_INCREMENT,
 'first_name' VARCHAR(15) NOT NULL,
  'last_name' VARCHAR(25) NOT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Далее необходимо сконфигурировать файл **persistence.xml** для технологии EclipseLink и СУБД MySQL: указываются провайдер, параметры доступа к базе данных и отображаемый на базу данных класс:

 # 3 # рабочий конфигурационный файл компоновки сущностей при использовании технологии EclipseLink # persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="UniversityJPA" transaction-type="RESOURCE_LOCAL">
   <provider>
         org.eclipse.persistence.jpa.PersistenceProvider
   </provider>
   <class>by.bsu.jpa.entity.Student</class>
   <properties>
      <property name="eclipselink.jdbc.url" value= "jdbc:mysql://localhost:3306/university"/>
      <property name="eclipselink.jdbc.user" value="root"/>
      <property name="eclipselink.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="eclipselink.jdbc.password" value="pass"/>
      <property name="eclipselink.logging.level" value="INFO"/>
      <property name="eclipselink.target-database" value="Auto"/>
      <property name="eclipselink.ddl-generation" value="create-tables" />
      <property name="eclipselink.ddl-generation.output-mode" value="database" />
   </properties>
   </persistence-unit>
</persistence>
```

Свойство **eclipselink.ddl-generation** со значением **create-tables** проверяет наличие таблицы в базе данных и при ее отсутствии создает ее в соответствии с правилами, определяемыми сущностью. Если же таблица существует, то никаких действий не производится. Если же присвоить свойству значение **drop-and-create-tables**, то таблицы, соответствующие сущностям, будут пересоздаваться всякий раз при запуске приложения.

Для технологии TopLink и СУБД Derby стандартный конфигурационный файл представлен ниже:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
        <persistence-unit name="UniversityJPA" transaction-type="RESOURCE_LOCAL">
        <provider>oracle.toplink.essentials.PersistenceProvider</provider>
          <class>by.bsu.jpa.entity.Student</class>
          <properties>
            <property name="toplink.jdbc.url"    value="jdbc:derby:university;create=true" />
            <property name="toplink.jdbc.driver"
                              value="org.apache.derby.jdbc.EmbeddedDriver" />
            <property name="toplink.jdbc.user" value="admin" />
            <property name="toplink.jdbc.password" value="pass" />
            <property name="toplink.ddl-generation" value="create-tables" />
            <property name="toplink.application-location" value="./db-schema"/>
            <property name="toplink.logging.level" value="FINE" />
            <property name="toplink.target-database" value="Derby" />
          </properties>
        </persistence-unit>
</persistence>
```

Теперь можно приступить к работе с сущностью в Java SE.

1. Создается фабрика, ассоциирующаяся с persistence-unit посредством имени "*UniversityJPA*"

   ```
   EntityManagerFactory factory = Persistence.createEntityManagerFactory("UniversityJPA").
   ```

2. Создается объект **EntityManager** для управления сущностями

   ```
   EntityManager entityManager = factory.createEntityManager().
   ```

3. Старт транзакции

   ```
   EntityTransaction transaction = entityManager.getTransaction();
   transaction.begin().
   ```

4. Создание экземпляра сущности

   ```
   Student student = new Student();
   student.setFirstName("Yulia");
   student.setLastName("Ivanukovich").
   ```

5. Сохранение объекта в persistence context

   ```
   entityManager.persist(student).
   ```

6. Сохранение изменений в базе данных

   ```
   entityManager.flush().
   ```

7. Изменения подтверждаются и закрываются объекты **EntityManager** и **EntityManagerFactory**

   ```
   transaction.commit();
   entityManager.close();
   factory.close().
   ```

В результате выполнения программы в базе данных появляется новая запись:

id	first_name	last_name
1	Yulia	Ivanukovich

В Java EE для создания **EntityManager**, управляемого контейнером используется следующий код.

/ # 4 # взаимодействие с persistence unit в JEE # StudentServlet.java */*

```
public class StudentServlet extends HttpServlet {
          @PersistenceUnit(unitName = "UniversityJPA")
          private EntityManagerFactory factory;
          protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
                    EntityManager em = factory.createEntityManager();
                    try {
                              List studentList = em.createQuery("SELECT s FROM Student s")
                                                  .getResultList();
                              request.setAttribute("students", studentList);
                              getServletContext().getRequestDispatcher("/jsp/list_students.jsp")
                                                  .forward(request, response);
                    } finally {
                              em.close();
                    }
          }
}
```

Результат выполнения запроса на выбор всех студентов из таблицы помещается в запрос и передается для отображения странице JSP.

Построение соответствий простых свойств

Каждое нестатическое и не временное свойство сущности считается сохраняемым, если не указана аннотация **@Transient**. Наличие у поля модификаторов **static**, **final** или **transient** аналогично указанию для этого поля **@Transient** аннотации. Объявление без аннотаций соответствует объявлению с аннотацией **@Basic**. Эта аннотация позволяет указать стратегию выборки для свойства:

```
// временное свойство
public transient int counter;

// сохраняемое свойство
private String firstName;

// временное свойство
@Transient
String getLengthInMeter() { /* code */ }
```

```
// временное свойство
private static String firstName;

// временное свойство
private final String firstName;

// сохраняемое свойство
String getCode() { /* code */ }

// сохраняемое свойство
@Basic
int getLength() { /*code */ }

// сохраняемое свойство: будет извлекаться из базы
// при первом доступе к полю (отложенная выборка)
@Basic( fetch = FetchType.LAZY )
String getDetailedComment() { /* code */ }

// сохраняемое свойство
@Temporal( TemporalType.TIME )
java.util.Date getDepartureTime() { /* code */ }

// перечисление, сохраняемое в базе как строка
@Enumerated( value = STRING )
TypeNote getNote() { /* code */ }
```

Аннотация **@Lob** указывает на то, что свойство должно быть сохранено в **Blob** или **Clob**, в зависимости от его типа: **java.sql.Clob**, **Character[]**, **char[]** и **java.lang.String** будут сохранены в **Clob**, **java.sql.Blob**, **Byte[]**, **byte[]** и сериализованные объекты будут сохранены в **Blob**. Для полей типа **java.util.Calendar** и **java.util.Date** необходимо указывать тип столбца для хранения времени в БД аннотацией **@Temporal** с значениями **TIME**, **DATE**, **TIMESTAMP**.

Определение атрибутов столбца

Столбцы, соответствующие свойствам, могут быть определены с помощью аннотации **@Column**. Ее используют для переопределения значений по умолчанию. Аннотация может применяться как к обычному свойству, так и к **@Id** и **@Version** свойствам.

```
@Column(
        name="first_name";
        boolean unique() default false;
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String table() default "";
        int length() default 255;
        int precision() default 0;
        int scale() default 0;
)
```

Где:

name — имя столбца. По умолчанию соответствует имени свойства;

unique — накладывает ограничение уникальности на данный столбец. По умолчанию **false**;

nullable — указывает, может ли значение столбца принимать значение **null**. По умолчанию **true**;

insertable — указывает, будет ли столбец входить в **insert** утверждение. По умолчанию **true**;

updatable — указывает, будет ли столбец входить в update утверждение. По умолчанию **true**;

columnDefinition — переписывает sql DDL фрагмент для конкретного столбца;

table — определяет таблицу, в которой находится столбец. По умолчанию таблица, соответствующая классу;

length — длина столбца. По умолчанию принимает значение 255;

precision — количество цифр. По умолчанию принимает значение 0;

scale — количество цифр после запятой. По умолчанию принимает значение 0.

Атрибуты **scale** и **precision** применимы, если описываемое поле класса имеет числовой тип.

Вложенные объекты-компоненты

В классе сущности можно объявить вложенные компоненты и при необходимости переопределить их столбцы. Классы вложенных компонентов должны иметь аннотацию **@Embeddable** на уровне класса. Переопределить маппинг столбца вложенного объекта можно с помощью аннотаций **@Embedded** и **@AttributeOverride**:

/ # 5 # использование вложенных объектов с переопределением # Student.java */*

```
@Entity
public class Student implements Serializable {
        // используется маппинг Address по умолчанию
                private Address homeAddress;
                // переопределение маппинга Country
                @Embedded
                @AttributeOverrides( {
                        @AttributeOverride( name = "iso2",
                                column = @Column(name="bornIso2")),
                        @AttributeOverride( name = "name",
                        column = Column( name = "bornCountryName" ) )
                } )
                private Country bornIn;
                // some code here
        }
```

```
/* # 6 # embedded класс # Address.java */

@Embeddable
public class Address implements Serializable {
        private String city;
        private Country nationality;
}
```

```
/* # 7 # переопределяемый класс # Country.java */

@Embeddable
public class Country implements Serializable {
        private String iso2;
        @Column( name = "countryName" )
        private String name;

        public String getIso2() {
                return iso2;
        }
        public void setIso2(String iso2) {
                this.iso2 = iso2;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        // some code here
}
```

Вложенные типы наследуют тип доступа от содержащей его сущности. Тип доступа определяет стратегию отображения объектов на базу данных: поля объекта или свойства объекта, имеющие методы **get***Имя***()**, **set***Имя***()**.

Неаннотированные значения по умолчанию

Если перед свойством POJO класса не указана ни одна аннотация, применяется следующее правило:

— для свойств простого типа его mapping устанавливается в **@Basic**;
— для свойства аннотированного как **@Embeddable**, его mapping порлучит значение **@Embedded**;
— если тип свойства **Serializable**, то оно преобразуется как **@Basic** в столбец, содержащий объект в сериализованном виде;
— если тип свойства **java.sql.Clob** или **java.sql.Blob**, то оно преобразуется в **@Lob** соответствующего типа.

Построение соответствий идентификаторов

Аннотация **@Id** позволяет определить, какое свойство будет использовано в качестве идентификатора сущности. Оно может быть установлено в самом приложении, либо сгенерировано провайдером, реализующим JPA. Стратегию генерирования идентификатора можно указать, используя аннотацию **@GeneratedValue**:

— использование стратегии **AUTO** создает **@TableGenerator** с именем **SEQ_GEN** со значениями по умолчанию. Указание **generator** не даст никакого эффекта;
— использование стратегии **TABLE**, не указывая генератор, создает **@TableGenerator** с именем **SEQ_GEN_TABLE** со значениями по умолчанию. Указание генератора, но не типа **@TableGenerator**, создает и **@TableGenerator** со стандартными значениями;
— использование стратегии **IDENTITY** или **SEQUENCE** дает результаты, зависящие от базы.
  - для базы данных Oracle по умолчанию создается **@SequenceGenerator** с именем **SEQ_GEN_SEQUENCE** со значениями по умолчанию. Указание генератора, отличного от **@SequenceGenerator** дает те же результаты;
  - для базы данных PostgreSQL создается столбец **SERIAL** с именем
  - **<имя таблицы>_<имя ключа>_SEQ**;
  - для базы данных MySQL создается столбец **AUTO_INCREMENT**;
  - для других поддерживаемых баз данных создается **IDENTITY** столбец.

Пример автоматической генерации первичного ключа:

```
@Entity
public class Student implements Serializable{
          @Id
          @GeneratedValue
          private Long id;
}
```

Для описания составных первичных ключей используется вложеный компонент и аннотация **@IdClass**:

/ # 8 # использование составного первичного ключа # Student.java */*

```
@Entity
@IdClass (StudentPk.class)
public class Student {
          // часть первичного ключа
          @Id
          public String getFirstname() {
                    return firstname;
          }
          public void setFirstname(String firstname) {
```

```
                this.firstname = firstname;
        }
        // часть первичного ключа
        @Id
        public String getLastname() {
                return lastname;
        }
        public void setLastname(String lastname) {
                this.lastname = lastname;
        }
        // some code here
}
```

/ # 9 # класс составного первичного ключа # StudentPk.java */*

```
        @Embeddable
        public class StudentPk implements Serializable {
                // имя и тип поля такие же, как в классе Student
                public String getFirstname() {
                        return firstname;
                }
                public void setFirstname(String firstname) {
                        this.firstname = firstname;
                }
                // имя и тип поля такие же, как в классе Student
                public String getLastname() {
                        return lastname;
                }
                public void setLastname(String lastname) {
                        this.lastname = lastname;
                }
                // some code here
        }
}
```

Таким образом, **@IdClass** указывает на соответствующий класс, определяющий первичный ключ.

Управление версиями. «Оптимистическое блокирование»

В JPA возможны ситуации, когда одна транзакция может отменить изменения другой транзакции. В случае, если одновременно несколько транзакций применят свои изменения, то та, у которой операция **commit()** пройдет последней, затрет изменения предыдущей. Во избежание таких ошибок используется механизм блокирования. Механизм предусматривает два типа блокирования: оптимистическое и пессимистическое.

Оптимистическое блокирование включается добавлением в класс сущности поля с аннотацией **@Version**, которое может быть целочисленного типа или типа **java.util.Date**. В последнем случае оно будет хранить время последнего обновления обьекта.

/ # 10 # использование «оптимистического блокирования» # Student.java */*

```
@Entity
public class Student implements Serializable {
// some code here
@Version
@Column(name="OPTLOCK")
          public Integer getVersion() { /* more code */ }
}
```

Поле для значения версии будет храниться в столбце **OPTLOCK** и объекты **EntityManager** станут использовать его для выявления конфликтующих обновлений. Во время **commit()** транзакции у каждой обновляемой сущности проверяется ее версия в памяти и в базе данных. В случае их совпадения происходит сохранение и увеличение (или инкрементирование) версии. В противном случае транзакция генерирует исключение **OptimisticLockException** и транзакция откатывается.

Пессимистическое блокирование осуществляется с помощью трех состояний:

— **PESSIMISTIC_READ** блокирует для записи данные, читаемые текущей транзакцией, но не блокирует для чтения из других транзакций;
— **PESSIMISTIC_WRITE** в момент обновления записи одной транзакцией в базе данных, запись блокируется для чтения, записи или удаления из другой транзакции;
— **PESSIMISTIC_FORCE_INCREMENT** блокирует данные, которые текущая транзакция читает или для которых увеличивает проаннотированную версию.

Объявляется в виде:

```
EntityManager entityManager = new EntityManager();
entityManager.find(Student.class, student.getId(), LockModeType.PESSIMISTIC_WRITE);
```

Именованные запросы

Особый способ организации запросов к БД и повышения производительности приложения представляют именованные запросы. Для объявления именованных запросов применяется аннотация **@NamedQuery**. Аннотация определяет имя запроса и сам текст запроса в виде диалекта SQL под названием Java Persistence Query Language или JPQL. Запрос реализуется в коде как объект класса **Query**. Объект **Query** конструируется, используя экземпляр **EntityManager**

как фабрику. Запрос может определяться как статически, так и динамически. Статические запросы определяются с применением аннотаций.

```
@NamedQuery(name = "studentByLastName",
                   query = " SELECT s FROM Student s WHERE s.lastName = :lastName ")
```

Вызываются статические запросы достаточно просто:

```
Query query = entityManager.createNamedQuery("studentByLastName");
query.setParameter("lastName", lastName); // передача параметра поиска
Collection students = query.getResultList();
```

Динамические запросы определяются непосредственно в коде.

```
Query query = entityManager.createQuery("SELECT s FROM Student s");
Collection students = query.getResultList();
```

Если для класса необходимо определить более одного запроса, применяется аннотация **@NamedQueries**.

```
@NamedQueries({
    @NamedQuery(name = "Student.findAll",
          query = "SELECT s FROM Student s"),
    @NamedQuery(name = "Student.findById",
          query = "SELECT s FROM Student s WHERE s.id = :id"),
    @NamedQuery(name = "Student.findByFirstName",
          query = "SELECT s FROM Student s WHERE s.firstName = :firstName"),
    @NamedQuery(name = "Student.findByLastName",
          query = "SELECT s FROM Student s WHERE s.lastName = :lastName")})
```

Запросы могут также определяться или переопределяться с применением XML.

Связь сущностей

В информационных системах сущности обычно связаны друг с другом различными типами связей. JPA также определяет связи между классами, если один класс объявляется в качестве поля другого класса, т. е. является его частью.

Классы могут соотноситься как «один к одному», «один ко многим», «многие к одному» и «многие ко многим».

В JPA для определения этих связей используются аннотации:

@OneToOne
@OneToMany
@ManyToOne
@ManyToMany

Связи могут быть двунаправленными и однонаправленными. При двунаправленной связи оба класса содержат ссылки друг на друга, при однонаправленной — только один класс ссылается на другой. При двунаправленной связи необходимо указывать атрибут другого класса, владеющий связью с данными классом в виде **@ManyToMany(mappedBy="***имяПоляДругогоКласса***")**.

Обычной является связь, когда две сущности находятся в отношении многие ко многим.

В системе управления образовательным процессом студент может записаться на много факультативных курсов и, соответственно, один курс может посещать много студентов.

/ # 11 # связанная сущность @ManyToMany # Student.java */*

```
package by.bsu.jpa.entity;
// imports
@Entity
@Table(name = "STUDENT")
// маппинг именованного запроса
@NamedQuery(name = "studentByLastName",
                    query = "SELECT st FROM Student st WHERE st.lastName = :lastName")
public class Student implements Serializable {
          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          @Column(name = "ID", nullable = false, unique = true)
          private int id;

          @Column(name = "FIRST_NAME")
          private String firstName;

          @Column(name = "LAST_NAME")
          private String lastName;

          // маппинг отношения many-to-many на стороне владельца
          @ManyToMany( cascade = { CascadeType.PERSIST, CascadeType.MERGE } )
          // определение присоединяемой таблицы
          @JoinTable(
                    name = "COURSE_STUDENT",
                    joinColumns = { @JoinColumn(name = "STUDENT_ID") },
                    inverseJoinColumns= { @JoinColumn(name="COURSE_ID") }
          )
          private List<Course> courses;
          // getters, setters and toString methods
}
```

/ # 12 # связанная сущность @ManyToMany # Course.java */*

```
package by.bsu.jpa.entity;
// imports
import static javax.persistence.EnumType.STRING;
@Entity
@Table(name = "COURSE")
public class Course implements Serializable {
          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          @Column(name = "ID")
          private int id;
```

```
        @Column(name = "NAME")
        private String name;

        @Column(name = "LECTOR")
        private String lector;

        @Enumerated(value = STRING)
        private TypeCourse type = TypeCourse.OPTIONAL;
        // маппинг отношения many-to-many на обратной стороне
        @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE},
                    mappedBy = "courses"
        )
        private List<Student> students;
        // getters, setters and toString methods
}
```

/ # 13 # типы курсов # TypeCourse.java */*

```
package by.bsu.jpa.entity;
public enum TypeCourse {
        OPTIONAL, REQUIRED
}
```

Демонстрация создания связанных объектов и использования именованного запроса:

/ # 14 # dao взаимодействия сущностей # CourseStudentDAO.java */*

```
package by.bsu.jpa.dao;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Query;
import by.bsu.jpa.entity.Course;
import by.bsu.jpa.entity.Student;
public class CourseStudentDAO {
        private EntityManager entityManager;
        public CourseStudentDAO(EntityManagerFactory factory) {
                entityManager = factory.createEntityManager();
        }
        // демонстрация создания, связывания и сохранения объектов
        public void createDemoStudent(Student student, Course course) {
                EntityTransaction transaction = null;
                try {
                        transaction = entityManager.getTransaction();
                        transaction.begin();
                        ArrayList<Student> students = new ArrayList<>();
                        ArrayList<Course> courses = new ArrayList<>();
                        // связывание student с course
                        students.add(student);
```

```
                    courses.add(course);
                    course.setStudents(students);
                    student.setCourses(courses);
                    // сохранение объекта student в базу данных
                    entityManager.persist(student);
                    transaction.commit();
                } catch (final Exception e) {
                    if (transaction != null && transaction.isActive()) {
                        transaction.rollback();
                    }
                    e.printStackTrace();
                }
        }
        /** именованный запрос: выводится список студентов с заданной фамилией
         *  @param lastName – фамилия является параметром запроса */
        @SuppressWarnings("unchecked")
        public void loadStudentsByLastName(String lastName) {
                // создается объект именованного запроса
                Query query = entityManager.createNamedQuery("studentByLastName");
                // устанавливается параметр lastName
                query.setParameter("lastName", lastName);
                // выполняется запрос
                List<Student> studentList = (List<Student>) query.getResultList();
                // вывод результатов
                if(studentList.size() == 0) {
                    System.out.println("Student " + lastName +" not found");
                }
                for (Student s : studentList) {
                    System.out.println(s.getFirstName() + " " + s.getLastName() + ", N"
                            + s.getId());
                    System.out.println("Courses:");
                    for (Course sCourse : s.getCourses()) {
                        System.out.println("\t\"" + sCourse.getName() + "\" by "
                          + sCourse.getLector() + ", "
                          + sCourse.getStudents().size() + " students.");
                    }
                }
        }
        public void closeManager() {
                if (entityManager != null && entityManager.isOpen()) {
                    entityManager.close();
                }
        }
}
```

/ # 15 # запуск приложения UniversityManyToMany # StudentCourseDemo.java */*

```
package by.bsu.jpa.main;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import by.bsu.jpa.dao.CourseStudentDAO;
```

```
import by.bsu.jpa.entity.Course;
import by.bsu.jpa.entity.Student;
public class StudentCourseDemo {
          private static final String PERSISTENCE_UNIT_NAME = "UniversityJPAManyToMany";
          public static void main(String[] args) {
                    EntityManagerFactory factory =
                              Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
                    CourseStudentDAO dao = new CourseStudentDAO(factory);
                    // создание объекта Student
                    Student student = new Student();
                    student.setFirstName("Varvara");
                    student.setLastName("Timofeeva");
                    // создание объекта Course
                    Course course = new Course();
                    course.setLector("Blinov");
                    course.setName("Code Engeneering");
                    dao.createDemoStudent(student, course);
                    dao.loadStudentsByLastName("Timofeeva");
                    dao.closeManager();
                    if (factory != null && factory.isOpen()) {
                              factory.close();
                    }
          }
}
```

Перед запуском приложения в конфигурационный файл **persistence.xml** следует добавить информацию о POJO-классах:

```
<persistence-unit name="UniversityJPAManyToMany" transaction-type="RESOURCE_LOCAL">
          <class>by.bsu.jpa.entity.Student</class>
          <class>by.bsu.jpa.entity.Course</class>
          <properties>
                    <!--необходимый набор properties -->
          </properties>
</persistence-unit>
```

В результате выполнения кода во все три таблицы добавятся соответствующие данные и в консоль будет выведено:

Varvara Timofeeva, N1
Courses: "Code Engeneering" by Blinov, 1 students.

Для демонстрации связи **@OneToOne** выбирается вариант связи классов **Student** и **Address**. Класс **Student** переписан с учетом наличия поля адреса и определения взаимно однозначной связи между ним и классом **Address**.

/ # 16 # связанная сущность @OneToOne # Student.java */*

```
package by.bsu.jpa.entity;
// imports
@Entity
```

```
public class Student {
          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          @Column(name = "ID", nullable = false, unique = true)
          private int id;

          @Column(name = "FIRST_NAME")
          private String firstName;

          @Column(name = "LAST_NAME")
          private String lastName;
          @OneToOne(cascade = CascadeType.ALL)
          private Address address;
          // getters, setters and toString methods
}
```

/ # 17 # связанная сущность @OneToOne # Address.java */*

```
package by.bsu.jpa.entity;
// imports
@Entity
public class Address {
          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          @Column(name = "ID", nullable = false, unique = true)
          private int id;
          private String street;
          private String city;
          private String state;
          private String zip;
          // getters, setters and toString methods
}
```

/ # 18 # запуск демонстрации каскадного добавления для связи @OneToOne # StudentAddressDemo.java */*

```
package by.bsu.jpa.main;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import by.bsu.jpa.entity.Address;
import by.bsu.jpa.entity.Student;
public class StudentAddressDemo {
          static EntityManagerFactory emf = Persistence
                              .createEntityManagerFactory("JPAUnniversityOneToOneCascade");
          static EntityManager em = emf.createEntityManager();

          public static void main(String[] a) {
                    em.getTransaction().begin();
                    Student student = new Student();
```

```
                student.setLastName("Reut");
                student.setFirstName("Alex");
                Address address = new Address();
                address.setCity("Prague");
                address.setState("Czech");
                address.setStreet("Ryzarska");
                student.setAddress(address); // добавление адреса студенту

                em.persist(student);
                em.flush();
                em.getTransaction().commit();

                Query query = em.createQuery("SELECT s FROM Student s");
                List<Student> list = (List<Student>) query.getResultList();
                for (Student st : list) {
                          System.out.println(st.getAddress());
                }
                em.close();
                emf.close();
        }
}
```

В результате будут выведены адреса всех студентов. Если с каким-либо студентом адрес не был ассоциирован, то для такого студента будет выведено значение **null**.

Наследование сущностей

Классы относительно друг друга могут быть связаны не только отношением «состоит из», но и в отношении «является». Программной реализацией такого отношения будет наследование классов. Информация для суперкласса и его наследников может находиться как в нескольких таблицах базы данных, так и в одной таблице. Ниже будет рассмотрена более простая стратегия «Single-Table».

Суперкласс иерархии должен быть помечен аннотацией **@Inheritance** с указанием стратегии **SINGLE_TABLE**.

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

В таблице **course** представлен столбец **type**, который не имеет мэппинга ни на одно ни в одном классе. Наличие такого поля есть ключевое требование при использовании «Single-Table» стратегии. Такой столбец называется разделяющий столбец и должен быть помечен аннотацией **@DiscriminatorColumn** в виде:

```
@DiscriminatorColumn(name="type")
```

Такие столбцы используются чаще всего для хранения основной характеристики объекта и аналогичны перечислениям в Java. Значение для ячейки столбца при сохранении информации из объекта задается в аннотации **@DiscriminatorValue**.

Пример иерархии классов при использовании однотабличной стратегии.

/ # 19 # классы иерархии для single-table стратегии # Course.java # RequiredCourse.java # OptionalCourse.java */*

```
package by.bsu.jpa.inheritance.single;
// imports
@Entity
@Table(name = "course")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type")
public abstract class Course {
          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          @Column(name = "id")
          private int id;

          @Column(name = "name")
          private String name;

          @Column(name = "lector")
          private String lector;
          // getters, setters and toString methods
}
package by.bsu.jpa.inheritance.single;
// imports
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("REQUIRED")
public class RequiredCourse extends Course implements Serializable {
          private static final long serialVersionUID = 1L;
          public RequiredCourse() {
                    super();
          }
          public RequiredCourse(int mark) {
                    super();
                    this.mark = mark;
          }
          @Column(name = "mark")
          private int mark;
          // getters, setters and toString methods
}
package by.bsu.jpa.inheritance.single;
// imports
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("OPTIONAL")
public class OptionalCourse extends Course implements Serializable {
          private static final long serialVersionUID = 1L;
          public OptionalCourse() {
                    super();
          }
}
```

Класс **RequiredCourse** обладает полем **mark**, которого нет у класса **OptionalCourse**. Последний, в свою очередь, также может содержать поля, отсутствующие в другом классе. Информация для обоих классов будет храниться в одной таблице, поэтому такие столбцы не должны иметь свойства **NOT NULL**.

Для сохранения/извлечения логично создать DAO-класс.

/ # 20 # сохранение/извлечение информации # CourseDAO.java */*

```
package by.bsu.jpa.inheritance.single;
// imports
public class CourseDAO {
	public void saveCourse(EntityManager entityManager, Course course) {
		// obtain transaction
		EntityTransaction transaction = entityManager.getTransaction();
		try {
			transaction.begin();
			// save created object to database
			entityManager.persist(course);
			// commit transaction and close manager and factory
			transaction.commit();
		} catch (Exception e) {
			// if transaction is active roll back it
			if (transaction.isActive()) {
				transaction.rollback();
			}
			e.printStackTrace();
		}
	}
	public List<Course> getRequiredCourses(EntityManager entityManager) {
		// create query
		Query query = entityManager.createQuery("SELECT c FROM RequiredCourse c");
		// execute query
		return (List<Course>)query.getResultList();
	}
	public List<Course> getOptionalCourses(EntityManager entityManager) {
		// create query
		Query query = entityManager.createQuery("SELECT c FROM OptionalCourse c");
		// execute query
		return (List<Course>)query.getResultList();
	}
}
```

Следующий пример представляет процесс создания экземпляра, его сохранения и извлечения информации о всех классах из иерархии.

/ # 21 # запуск демонстрации стратегии single-table # SingleInheritanceDemo .java */*

```
package by.bsu.jpa.inheritance.single;
// imports
public class SingleInheritanceDemo {
```

```
        private static EntityManagerFactory factory =
                Persistence.createEntityManagerFactory("JPASingleInheritance");
        public static void main(String ... args) {
                // create entity manager
                EntityManager entityManager = factory.createEntityManager();
                // create DAO and entity
                CourseDAO dao = new CourseDAO();
                RequiredCourse course = new RequiredCourse();
                course.setName("Design Patterns in Java");
                course.setLector("Blinov,Ph.D");
                course.setMark(8);
                // save course
                dao.saveCourse(entityManager, course);
                // retrive required and optional courses
                List<Course> required = dao.getRequiredCourses(entityManager);
                List<Course> optional = dao.getOptionalCourses(entityManager);
                // close entity manager
                entityManager.close();
                // put student list into console
                System.out.println("Required Courses: " + required);
                System.out.println("Optional Courses:" + optional);
                factory.close();
        }
}
```

В файл конфигурации **persistence.xml** следует добавить все классы иерархии.

Другая стратегия «Joined» более практична вследствие использования для хранения информации о подклассах различных связанных между собой таблиц. Тогда как применение стратегии «Single-Table» повлечет за собой большое количество незаполненных полей в таблице базы данных, которые появляются из-за невостребованности их столбцов тем или иным классом из иерархии сущностей.

JPA предлагает еще ряд стратегий и возможностей работы с информацией из иерархии.

Код sql для создания базы данных *university* приведен ниже:

```
# database structure for database 'university'
CREATE DATABASE IF NOT EXISTS 'university' DEFAULT CHARACTER SET utf8;
USE 'university';
#
# table structure for table 'address'
#
CREATE TABLE 'address' (
 'id' INT(11) NOT NULL AUTO_INCREMENT,
 'zip' VARCHAR(10) DEFAULT NULL,
 'state' VARCHAR(50) DEFAULT NULL,
 'city' VARCHAR(50) DEFAULT NULL,
 'street' VARCHAR(50) DEFAULT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
#
# table structure for table 'student'
#
CREATE TABLE IF NOT EXISTS 'student' (
 'id' INT(11) NOT NULL AUTO_INCREMENT,
 'first_name' VARCHAR(15) NOT NULL,
  'last_name' VARCHAR(25) NOT NULL,
  'address_id' INT(11) DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'address_fk' ('address_id'),
CONSTRAINT 'student_ibfk_1' FOREIGN KEY ('address_id') REFERENCES 'address' ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
#
# table structure for table 'course'
#
CREATE TABLE IF NOT EXISTS 'course' (
  'id' INT(11) NOT NULL AUTO_INCREMENT,
  'name' VARCHAR(100) NOT NULL,
  'lector' VARCHAR(50) NOT NULL,
  'type' VARCHAR(11) NOT NULL,
  'mark' INT(3),
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
#
# table structure for table 'course_student'
#
CREATE TABLE IF NOT EXISTS 'course_student' (
  'course_id' INT(11) NOT NULL,
  'student_id' INT(11) NOT NULL,
  KEY 'student_fk' ('student_id'),
  KEY 'course_fk' ('course_id'),
CONSTRAINT 'course_student_ibfk_1' FOREIGN KEY ('course_id') REFERENCES 'course' ('id') ON DELETE
CASCADE ON UPDATE CASCADE,
CONSTRAINT 'course_student_ibfk_2' FOREIGN KEY ('student_id') REFERENCES 'student' ('id') ON DELETE
CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
#
# table structure for table 'lecturer'
#
CREATE TABLE IF NOT EXISTS 'lecturer' (
  'first_name' VARCHAR(50) NOT NULL,
  'last_name' VARCHAR(50) NOT NULL,
  PRIMARY KEY ('first_name','last_name')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Приложение 7

Hibernate

Hibernate — один из первых инструментов 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 или Oracle.
2. Загрузить и подключить Hibernate с сервера **http://hibernate.org/**
3. Загрузить и подключить JDBC-драйвер используемой базы данных (в случае использования Ant, в папку **lib** проекта), в данном случае **mysql-connector-java-[***версия***].jar** или **ojdbc[***версия***].jar**. Обычно JDBC-драйвер предоставляется на сайте разработчика сервера баз данных.

Библиотеки **hibernate3.**[*версия*]**.jar** и **hibernate-annotations-3.**[*версия*]**.jar** являются основными. Кроме них, устанавливается еще целый ряд необходимых библиотек.

Создание простейшего приложения

Далее следует настроить конфигурационный файл Hibernate на использование пула соединений. Этот файл должен располагаться в корне дерева классов и иметь имя **hibernate.cfg.xml**. Например, при использовании СУБД Oracle файл конфигурации будет выглядеть следующим образом.

```
<?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 name="oracleSessionFactory">
   <!-- Hibernate JDBC Properties -->
         <property name="hibernate.dialect">
                  org.hibernate.dialect.Oracle9Dialect</property>
```

```
        <property name="hibernate.connection.driver_class">
                oracle.jdbc.driver.OracleDriver</property>
        <property name="hibernate.connection.url">
                jdbc:oracle:thin:@localhost:1521:UNIVERSITY_HIB</property>
        <property name="hibernate.connection.username">admin</property>
        <property name="hibernate.connection.password">pass</property>
                <!-- Transaction API -->
        <property name="hibernate.transaction.factory_class">
                org.hibernate.transaction.JDBCTransactionFactory</property>
                <!-- Hibernate Optional Configuration Properties -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.current_session_context_class">thread</property>
        <property name="hibernate.default_entity_mode">pojo</property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <property name="hibernate.connection.pool_size">50</property>
   </session-factory>
</hibernate-configuration>
```

Здесь указан способ получения JDBC-соединения, включено логгирование команд SQL, настроен диалект SQL.

В случае использования сервера баз данных MySQL в конфигурационный файл следует внести следующие изменения:

```
<?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="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
          jdbc:mysql://localhost:3306/university_hib?zeroDateTimeBehavior=convertToNull
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">pass</property>
    <property name="hibernate.show_sql">true</property>
    <!-- enable Hibernate's automatic session context management -->
    <property name="hibernate.current_session_context_class">thread</property>
    <property name="hibernate.hbm2ddl.auto">create</property>
    <mapping class="by.bsu.hib.entity.Course"/>
    <mapping class="by.bsu.hib.entity.Student"/>
  </session-factory>
</hibernate-configuration>
```

Свойство *hibernate.hbm2ddl.auto* со значением *create* представляет приложению возможность создания таблиц в базе данных *university_hib* на основе правил, заданных в POJO классах. При каждом запуске приложения таблицы будут пересоздаваться заново, уничтожая все ранее созданные таблицы вместе с хранящимися там данными. Если необходимо использовать данные таблиц

после предыдущих запусков, то следует удалить указанное свойство из конфигурационного файла.

Для настройки параметров работы Hibernate вместо конфигурационного XML-файла можно использовать файл **hibernate.properties**.

POJO-класс

POJO (Plain Ordinary Java Object) — класс, которому по определенным правилам ставятся в соответствие некоторые таблица или набор столбцов в таблице БД. В примере, рассматривающем учебные курсы и студентов, их посещающих, будут созданы следующие классы.

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.

/ # 1 # POJO-класс сущности # Course.java */*

```
package by.bsu.hib.entity;
import java.io.Serializable;
import java.util.Set;
import javax.persistence.*;
@Entity
@Table(name = ("course"))
public class Course implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    @Column(name = "TITLE")
    private String title;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "course_student", joinColumns = {@JoinColumn(name = "COURSE_ID")},
                                        inverseJoinColumns = {@JoinColumn(name = "STUDENT_ID")})
    private Set<Student> students;
    public Course() {
    }
    public Course(String title) {
        this.title = title;
    }
```

```
    public Course(Long id, String title) {
        this.id = id;
        this.title = title;
    }
    public Long getId() {
        return id;
    }
    protected void setId(Long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public Set<Student> getStudents() {
        return students;
    }
    public void setStudents(Set<Student> students) {
        this.students = students;
    }
    @Override
    public String toString() {
        return "Course{" + "id=" + id + ", title=" + title + ", students=" + students + '}';
    }
}
```

/ # 2 # POJO-класс сущности # Student.java */*

```
package by.bsu.hib.entity;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name = ("student"))
@NamedQuery(name="findStudentByLastName",
                    query="select s from Student s where s.lastName = :lastName")
public class Student implements Serializable {
    @Id
    @GenericGenerator(name="auto_inc", strategy = "increment")
    @GeneratedValue(generator="auto_inc")
    @Column(name = ("ID"))
    private Long id;
    @Column(name = ("LAST_NAME"))
```

```
    private String lastName;
    @Column(name = ("FIRST_NAME"))
    private String firstName;
    public Student() { /* more code */ }
    public Student(String lastName, String firstName) {
      this.lastName = lastName;
      this.firstName = firstName;
    }
    public Student(Long id, String lastName, String firstName) {
      this.id = id;
      this.lastName = lastName;
      this.firstName = firstName;
    }
    // getters and setters
    @Override
    public String toString() {
      return "Student{" + "id=" + id + ", lastName=" + lastName + ", firstName=" + firstName + '}';
    }
}
```

После создания таблиц в базе данных задается соответствие между POJO-классами и таблицами. Объектно-реляционный mapping описывается в виде аннотаций.

Аннотация **@Entity** указывает, что данный класс является сущностью бизнес модели;

Аннотация **@Table(name = "COURSE")** указывает на имя таблицы в базе данных. Если имя таблицы совпадает с именем класса, то его можно опустить.

Для идентификации конкретной записи в базе данных требуется ключевое поле — аннотация **@Id.**

Аннотация **@Column(name = ("LAST_NAME"))** указывает соответствие полей класса и столбцов базы данных.

Аннотации **@GenericGenerator(name="auto_inc", strategy = "increment"),**

@GeneratedValue(generator="auto_inc") указывают способ генерация значений в данном столбце, возможные его значения: **increment**, **identity**, **sequence**, **hilo**, **seqhilo**, **uuid**, **guid**, **native**, **assigned**, **select**, **foreign.**

Кроме того следует обратить внимание на следующие аннотации:

@ManyToMany(cascade = CascadeType.ALL) — 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. Атрибут **cascade** используется для определения каскадных операций, может принимать следующие значения: **ALL**, **PERSIST**, **MERGE**, **REMOVE**, **REFRESH**.

@JoinTable(name = ("course_student"),

joinColumns = {@JoinColumn(name = ("course_id"))},

inverseJoinColumns = {@JoinColumn(name = ("student_id"))}) — определяет имя связующей таблицы и имена внешних ключей в ассоциированных сущностях.

Аннотация **@NamedQuery(name = "findStudentByLastName", query = "SELECT s FROM Student s WHERE s.lastName = :lastName")** определяет именованный HQL (Hibernate Query Language)-запрос. HQL-запросы — это аналог SQL-запросов для Hibernate, который предоставляет возможность гибкого взаимодействия с базами данных. При выполнении HQL-запросов, Hibernate генерирует соответствующий текущей базе данных SQL-запрос. HQL значительно удобнее стандартного SQL, однако он поддерживает только те функции, которые в том или ином виде реализованы в большинстве баз данных.

@NotNull — валидатор, который будет следить за тем, чтобы сохраняемое поле не было пустым.

@Length — валидатор, который будет следить за тем, чтобы содержимое строковой переменной не превышало определенную длину.

Валидаторы — часть технологии Hibernate, которая позволяет накладывать ограничения на допустимые значения полей. Срабатывание валидаторов происходит, когда выполняется попытка сохранения объекта. Помимо широкого набора стандартных валидаторов, Hibernate предоставляет возможность создания собственных.

Необходимо отметить, что, вообще говоря, для того, чтобы отобразить простой класс в базу, достаточно всего двух аннотаций: **@Entity** и **@Id**. Все остальные данные об отображаемых классах Hibernate собирает сам посредством механизма отражений.

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 **org.hibernate.cfg.Configuration**.

Для открытия нового сеанса **HibernateUtil** следует создать служебный класс

/ # 3 # конфигурация и инициализация нового сеанса # HibernateUtil.java */*

```
package by.bsu.hiber.util;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
public class HibernateUtil {
    private static final SessionFactory sessionFactory;
    static {
        try {
            // create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable e) {
            // make sure you log the exception, as it might be swallowed
```

```
            System.err.println("Initial SessionFactory creation failed." + e);
            throw new ExceptionInInitializerError(e);
        }
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

В классе **HibernateUtil** создается новая конфигурация. Если mapping-классы не были указаны в конфигурационном файле Hibernate, то их необходимо добавить в конфигурацию явно

```
Configuration aconf = new AnnotationConfiguration()
   .addAnnotatedClass(by.bsu.hib.entity.Course.class)
   .addAnnotatedClass(by.bsu.hib.entity.Student.class);
aconf.configure();
sessionFactory = aconf.buildSessionFactory();
```

Метод **addAnnotatedClass()** возвращает объект **AnnotationConfiguration**, поэтому он вызывается несколько раз подряд.

Интерфейс **Session** используется для сохранения в базу данных и восстановления из нее объектов классов **Course** и **Student**. Экземпляр **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. В конце обращения сессия закрывается

   ```
   session.close();
   ```

Интерфейс **org.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 **org.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** (текст запросов и команд пишется на Hibernate-диалекте; он похож на SQL, только в качестве сущностей-носителей данных выступают классы, а не таблицы). Получить его можно с помощью объекта **Session**:

```
session.createQuery("HQL-Запрос")
```

Интерфейс **Query** имеет следующие методы:

list() — выполняет запрос, результат возвращается в коллекции **List**;

```
session.createQuery("HQL-Запрос").list();
```

session.getNamedQuery("findStudentByLogin") — выполняет именованный запрос ***findStudentByLogin***.

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 =:status") .setString('status', 'closed').executeUpdate();

iterate() — возвращает **Iterator** по результатам запроса

```
Iterator it = createQuery("HQL-Запрос").iterate();
```

Долгоживущие Объекты и Коллекции (Persistent Objects and Collections) — однопоточные, долгоживущие объекты, содержащие сохраняемое состояние и бизнес-функции. Ими могут быть обычные JavaBean/POJO (Plain Old Java Objects) объекты, чья отличительная особенность — ассоциация с одной сессией (**Session**). Как только их сессия будет закрыта, эти объекты становятся отсоединенными и свободными для использования на любом уровне приложения, например, непосредственно как объекты передачи данных на уровень представления, так и получения данных из него.

Временные Объекты и Коллекции (Transient Objects and Collections) — экземпляры короткоживущих классов, которые в данный момент не ассоциированы с экземпляром **Session**. Это могут быть объекты, созданные приложением и в данный момент еще не переведенные в долгоживущее состояние.

Транзакция **org.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 **org.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 **org.hibernate.transaction.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 указанных классов представлено ниже:

/ # 4 # dao-класс взаимодействия с таблицей course # CourseDAO.java */*

```
package by.bsu.hib.dao;
import by.bsu.hib.entity.Course;
import by.bsu.hib.entity.Student;
```

```
import java.util.Set;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class CourseDAO {
    private Session session;
    public CourseDAO(Session session) {
        this.session = session;
    }
    public Set<Student> findRegistedOnCourse(String courseTitle) {
        Set<Student> registedOnCourse;
        Query query = session.createQuery("FROM Course WHERE title=:title");
        query.setParameter("title", courseTitle);
        Course course = (Course) query.uniqueResult();
        registedOnCourse = course.getStudents();
        return registedOnCourse;
    }
    public boolean addCourse(Course course) {
        boolean flag = false;
        Transaction t = null;
        try {
            t = session.beginTransaction();
            session.saveOrUpdate(course);
            t.commit();
            flag = true;
        } catch (HibernateException e) {
            e.printStackTrace();
            t.rollback();
        }
        return flag;
    }
}
```

/ # 5 # dao-класс взаимодействия с таблицей student # StudentDAO.java */*

```
package by.bsu.hib.dao;
import by.bsu.hib.entity.Student;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
public class StudentDAO {
    private Session session;
    public StudentDAO(Session session) {
        this.session = session;
    }
    public Student getStudent(String lastName) {
        Student student = null;
        try {
```

```
            Query query = session.getNamedQuery("findStudentByLastName");
            query.setParameter("lastName", lastName);
            student = (Student) query.uniqueResult();
            System.out.println(student);
        } catch (HibernateException e) {
            e.printStackTrace();
        }
        return student;
    }
    public boolean studentExists(String lastName) {
        Student student = null;
        try {
            Query query = session.getNamedQuery("findStudentByLastName");
            query.setParameter("lastName", lastName);
            student = (Student) query.uniqueResult();
        } catch (HibernateException e) {
            e.printStackTrace();
        }
        return student != null;
    }
    public void addStudent(Student student) {
        Transaction t = null;
        try {
            t = session.beginTransaction();
            session.save(student);
            t.commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            t.rollback();
        }
    }
    public void deleteStudent(Student student) {
        Transaction t = null;
        try {
            t = session.beginTransaction();
            session.delete(student);
            t.commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            t.rollback();
        }
    }
    public void updateStudent(Student student) {
        Transaction t = null;
        try {
            t = session.beginTransaction();
            session.update(student);
            t.commit();
        } catch (HibernateException e) {
```

```
            e.printStackTrace();
            t.rollback();
        }
    }
}
```

/ # 6 # простейшее применение hibernate # StudentCourseSimplestDemo.java */*

```
package by.bsu.hib.main;
import by.bsu.hib.dao.CourseDAO;
import by.bsu.hib.dao.StudentDAO;
import by.bsu.hib.entity.Course;
import by.bsu.hib.entity.Student;
import by.bsu.hib.util.HibernateUtil;
import java.util.HashSet;
import java.util.Set;
import org.hibernate.Session;
public class StudentCourseSimplestDemo {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSession();
        try {
            StudentDAO studentDAO = new StudentDAO(session);
            CourseDAO courseDAO = new CourseDAO(session);

            String courseTitle1 = "Java";
            Course course1 = new Course();
            course1.setTitle(courseTitle1);

            HashSet<Student> set1 = new HashSet<Student>() {
                {
                    this.add(new Student("Ivanov", "Vitalii"));
                    this.add(new Student("Reut", "Alexandra"));
                    this.add(new Student("Tomkevich", "Alina"));
                }
            };
            course1.setStudents(set1);
            courseDAO.addCourse(course1);

            String courseTitle2 = "Design Patterns for Java";
            Course course2 = new Course();
            course2.setTitle(courseTitle2);
            courseDAO.addCourse(course2);
            Set<Student> setRes = courseDAO.findRegistedOnCourse(courseTitle1);
            Student student1 = new Student("Zanko", "Vital");
            studentDAO.addStudent(student1);
            System.out.println(setRes);
            Student student2 = studentDAO.getStudent("Ivanov");
            System.out.println(student2);
            HashSet<Student> set2 = new HashSet<>();
            set2.add(student1);
            set2.add(student2);
```

```
            course2.setStudents(set2);
            courseDAO.addCourse(course2);
        } finally {
            if (session != null) {
                session.close();
            }
        }
    }
}
```

База данных после запуска приложения будет содержать следующую информацию:

Таблица *course*:

ID	TITLE
1	Java
2	Design Patterns for Java

Таблица *student:*

ID	FIRST_NAME	LAST_NAME
1	Alina	Tomkevich
2	Vitalii	Ivanov
3	Alexandra	Reut
4	Vital	Zanko

Таблица *course_student:*

COURSE_ID	STUDENT_ID
1	1
1	2
2	2
1	3
2	4

XML mapping POJO-классов

Описание связей между классом и таблицей может быть представлено с помощью xml-файла. Для этого в примере #1, рассматривающем учебные курсы и студентов, их посещающих, следует удалить все аннотации.

Объектно-реляционный mapping описывается в виде XML-документа *ИмяКласса*.**hbm.xml** в каталоге, содержащем *ИмяКласса***.class** файл соответствующего класса. 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.

 # 7 # mapping-файл для класса by.bsu.hib.entity.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="by.bsu.hib.entity.Course" table="course">
          <id name="id" column="id" type="java.lang.Long">
                    <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="by.bsu.hib.entity.Student"/>
          </set>
    </class>
</hibernate-mapping>
```

 # 8 # mapping-файл для класса by.bsu.hib.entity.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="by.bsu.hib.entity">
    <class name="Student" table="student">
          <id name="id" column="id" type="java.lang.Long">
                    <generator class="native"/>
          </id>
          <property name="lastname" type="java.lang.String"/>
          <property name="firstname" type="java.lang.String"/>
    </class>
</hibernate-mapping>
```

Пояснения по приведенному коду:

<class name="by.bsu.hib.entity.Course"> — необходимо указать класс, который будет использоваться при работе с указанной ниже таблицей базы данных. Причем 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 производиться 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>
```

Здесь 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-many name="*property_name***" column="***column_name***" class="***class_name***" 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. Описание

```
<set name="propertyName" inverse="true">
          <key column="foreignId"/>
          <many-to-many class="ClassName"/>
</set>
```

интерпретирует коллекцию «множество».

В конфигурационном файле **hibernate.cfg.xml** в теге **mapping** вместо атрибута **class** следует использовать атрибут **resource** в виде:

```
<mapping resource="by/bsu/hib/entity/Course.hbm.xml"/>
<mapping resource="by/bsu/hib/entity//Student.hbm.xml"/>
```

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. несоответствие приводит к ошибкам, которые достаточно сложно выявить.

Приложение 8

IDE Eclipse

Установка и запуск

Eclipse — сообщество открытого кода или open source, чьи проекты сфокусированы на создании открытой платформы для разработки, развертывания, управления приложениями с использованием различных фреймворков, инструментов и сред исполнения. Загрузить продукты Eclipse можно на официальном сайте http://eclipse.org/

Eclipse Java IDE for Web Developers — среда для разработки веб-приложений на Java. До загрузки IDE можно выбрать сборку для конкретной платформы. Ниже будет рассмотрена IDE из сборки Juno для операционной системы Windows (32-bit).

Для установки Eclipse Java IDE необходимо распаковать загруженный архив в ту директорию, в которой будет храниться IDE. Распакованная IDE занимает менее 300 Мб. Среда Eclipse используется для разработки приложений, поэтому предварительно необходимо установить Java Development Kit (JDK). Оптимальную версию можно загрузить с сайта http://oracle.com/ и следовать мастеру установки.

Если JDK установлен, то для запуска Eclipse Java IDE следует воспользоваться файлом **eclipse.exe**. При запуске файла на экране появится окно (Рис. 1.), где необходимо выбрать **workspace** — каталог, в котором будут храниться

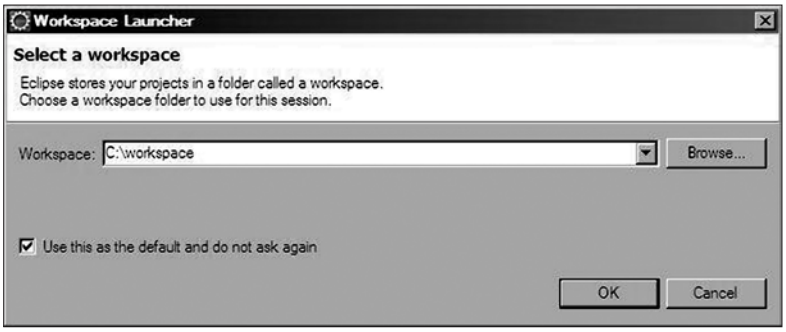

Рис. 1. *Выбор директории хранения проектов*

проекты. В случае использования одного **workspace**, необходимо установить значение по умолчанию (отметить check box ☑).

Создание и запуск проекта

При первом запуске проекта на экране появится вкладка *Welcome page* (Рис. 2), содержащая несколько ссылок. При закрытии или сворачивании вкладки *Welcome page*, среда будет выглядеть как на рис. 3.

Создать новый проект в Eclipse IDE можно несколькими способами:

— нажать **Alt+Shift+N**;

— выбрать в меню **File** пункт **New.**

После совершения одного из двух перечисленных действий необходимо выбрать тип проекта. Далее будут подробно рассмотрены два типа проектов: консольный и веб-проект.

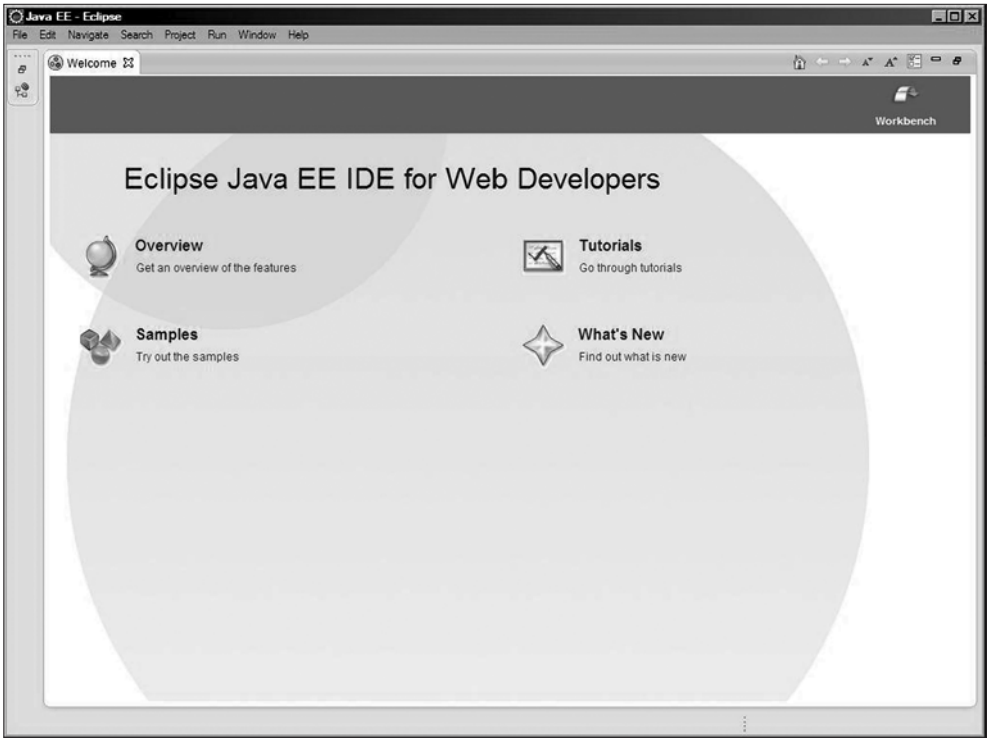

Рис. 2. *Welcome page*

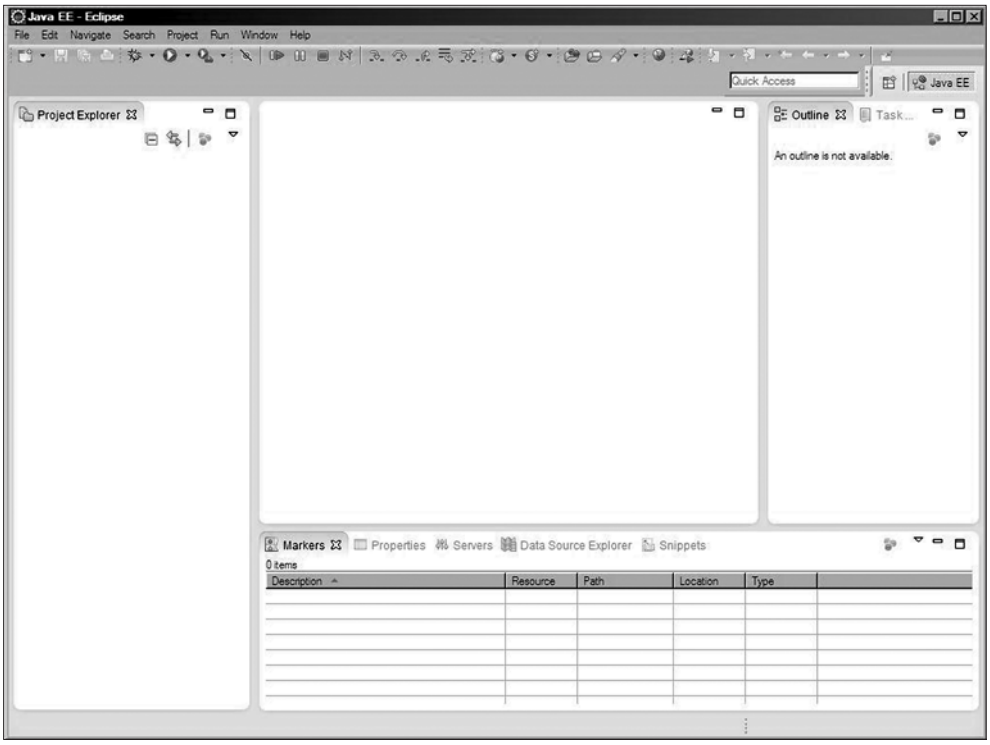

Рис. 3. *Eclipse IDE*

Создание, запуск и отладка консольного проекта

Для создания консольного проекта необходимо выбрать меню **File — New — Project**. В результате выбора откроется мастер создания проекта (Рис. 4.).

В Мастере создания проекта необходимо выбрать **Java Project** и нажать кнопку **Next**. Далее вводится имя проекта, выбирается JRE (Рис. 5.). Можно задать некоторые дополнительные параметры, нажимая кнопку **Next**, можно оставить все остальное по умолчанию и нажать кнопку **Finish** для создания проекта. В *Package explorer* слева появится только что созданный проект (Рис. 6.).

Чтобы создать пакет в новом проекте, необходимо нажать правой кнопкой мыши на название проекта в *Package Explorer* или выбрать в главном меню **File — New — Package**. Затем ввести имя пакета и нажать кнопку **Finish**. После этого в папке **src** появится пиктограмма пустого пакета с именем. С помощью **File — New** можно создать класс (**Class**), интерфейс (**Interface**), перечисление (**Enum**) или другой элемент проекта (Рис. 7.).

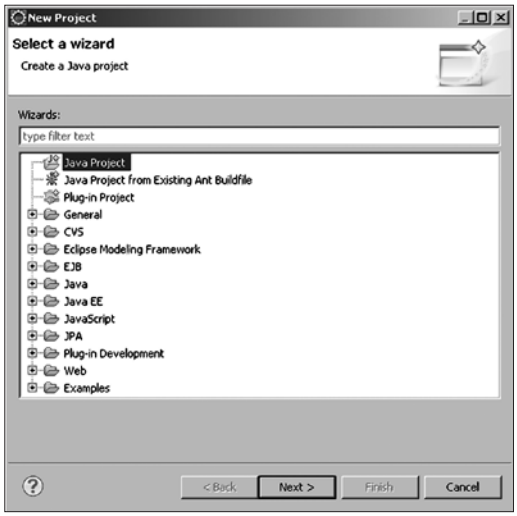

Рис. 4. *Мастер создания проекта*

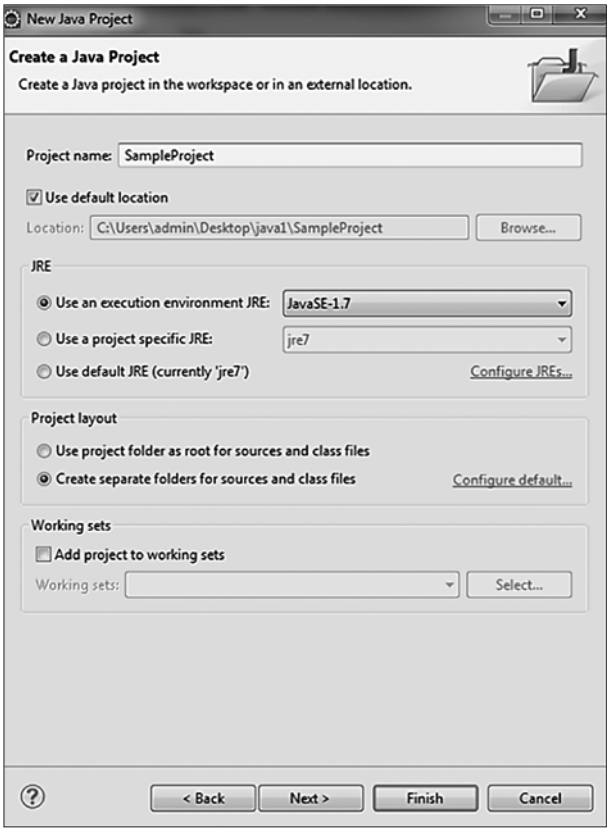

Рис. 5. *Создание Project*

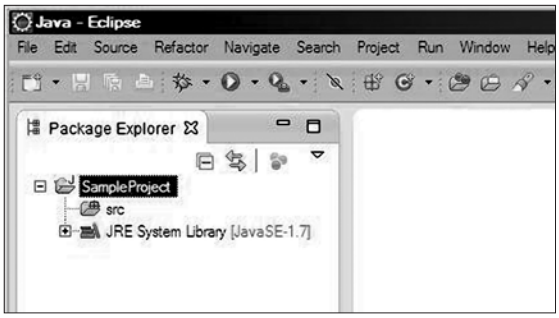

Рис. 6. *Package Explorer*

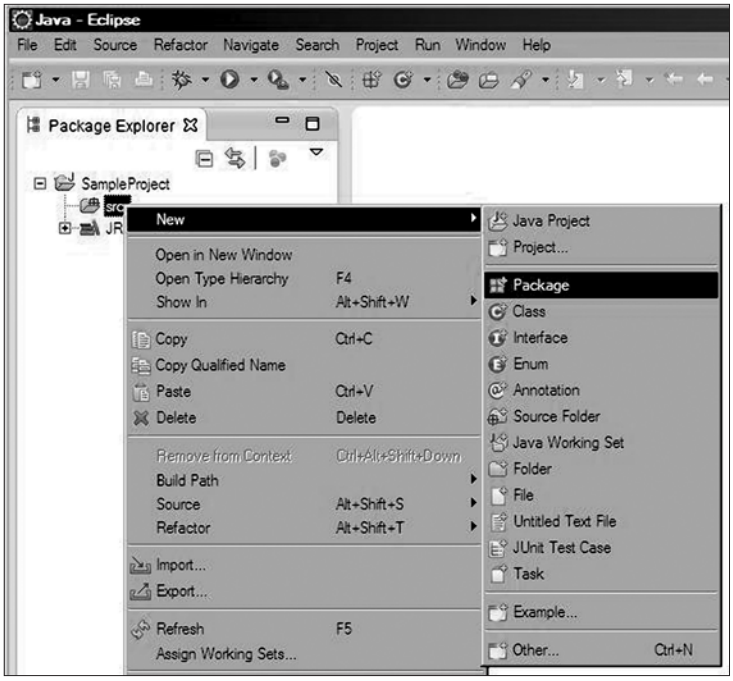

Рис. 7. *Создание нового элемента проекта*

При создании класса необходимо как минимум ввести его имя. Так же в *Мастере создания класса* можно добавить к классу интерфейсы, указать модификатор доступа, сообщить, что класс будет содержать метод **main()** и прочее (Рис. 8.).

Созданный класс будет выглядеть так (см. Рис. 9.).

Проект можно запустить нажатием сочетания клавиш **Ctrl+F11** или кнопкой **Run** на панели инструментов (зеленый круг с белым треугольником внутри).

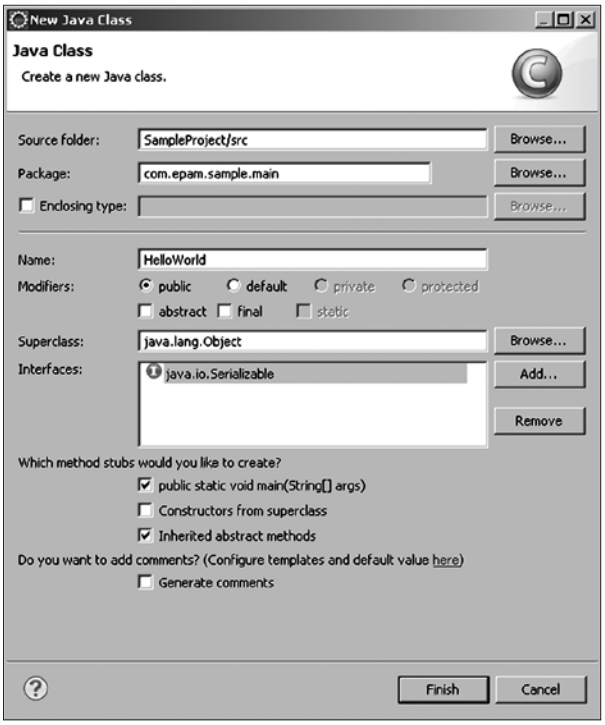

Рис. 8. *Создание нового класса*

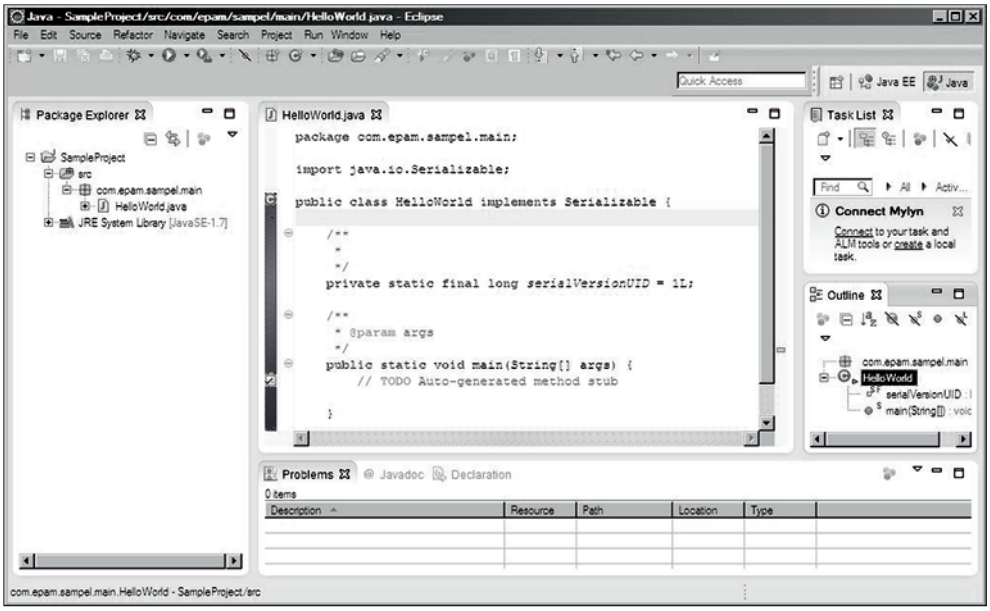

Рис. 9. *Сгенерированный класс*

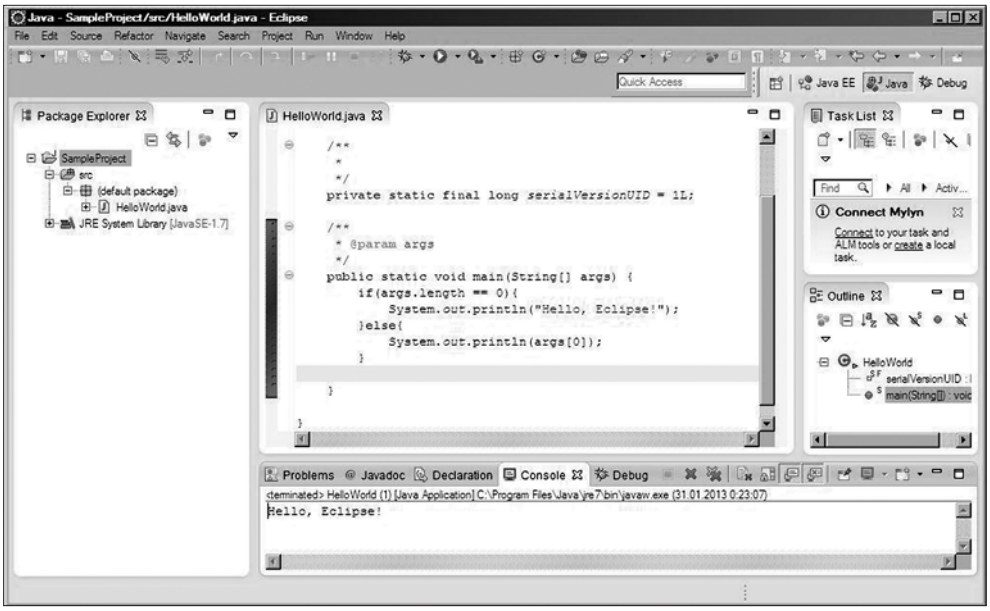

Рис. 10. *Результат запуска программы без аргументов*

В папке *src* проекта может размещаться только java-код и файлы с расширением *properties*. Все остальные ресурсы проекта всегда должны находиться в проекте, но вне папки *src*.

При разработке может потребоваться временно или постоянно закомментировать участок кода. Выполняется указанное действия после выделения фрагмента кода нажатием сочетания клавиш **Ctrl-/**. Действия по снятию комментария с выделенного фрагмента осуществляется этим же сочетанием клавиш.

Код java имеет для лучшего визуального восприятия лестничную структуру, вид которой можно изменить в настройках редактора. Но в процессе разработки она может быть нарушена программистом, тогда для придания коду форматированного вида следует нажать сочетание клавиш **Ctrl-Shift-F** или выбрать **Source — Format**.

Для передачи аргументов в метод **main()** необходимо выбрать пункт меню **Run — Run Configurations** и далее на вкладке **Arguments** в поле *Program arguments* указать параметры. Параметры разделяются пробелом (Рис. 11.).

Для отладки проект запускается кнопкой **F11** клавиатуры или кнопкой **Debug** панели инструментов (кнопка с изображением жука). Точки остановки (breakpoint) ставятся двойным щелчком мыши на сером поле слева от кода класса. Точка остановки графически выглядит как маленький голубой шарик.

В режиме отладки среда переключается в следующий вид (Рис. 12.).

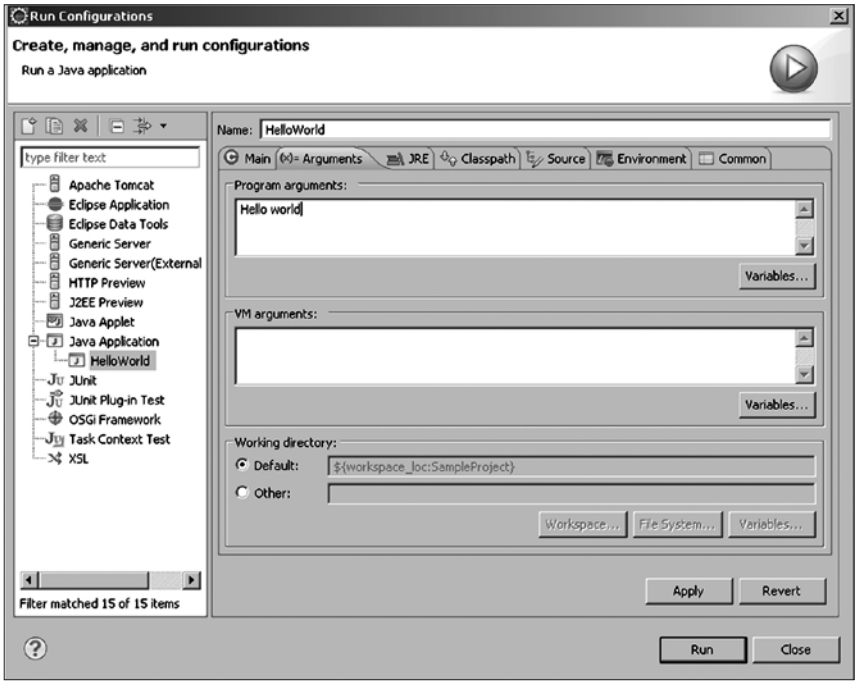

Рис. 11. *Передача параметров в main*

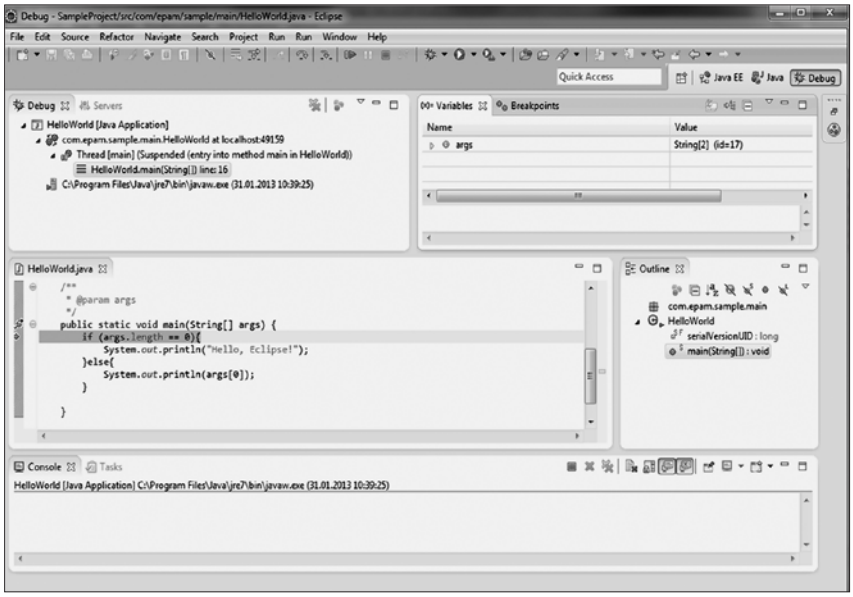

Рис. 12. *Debug*

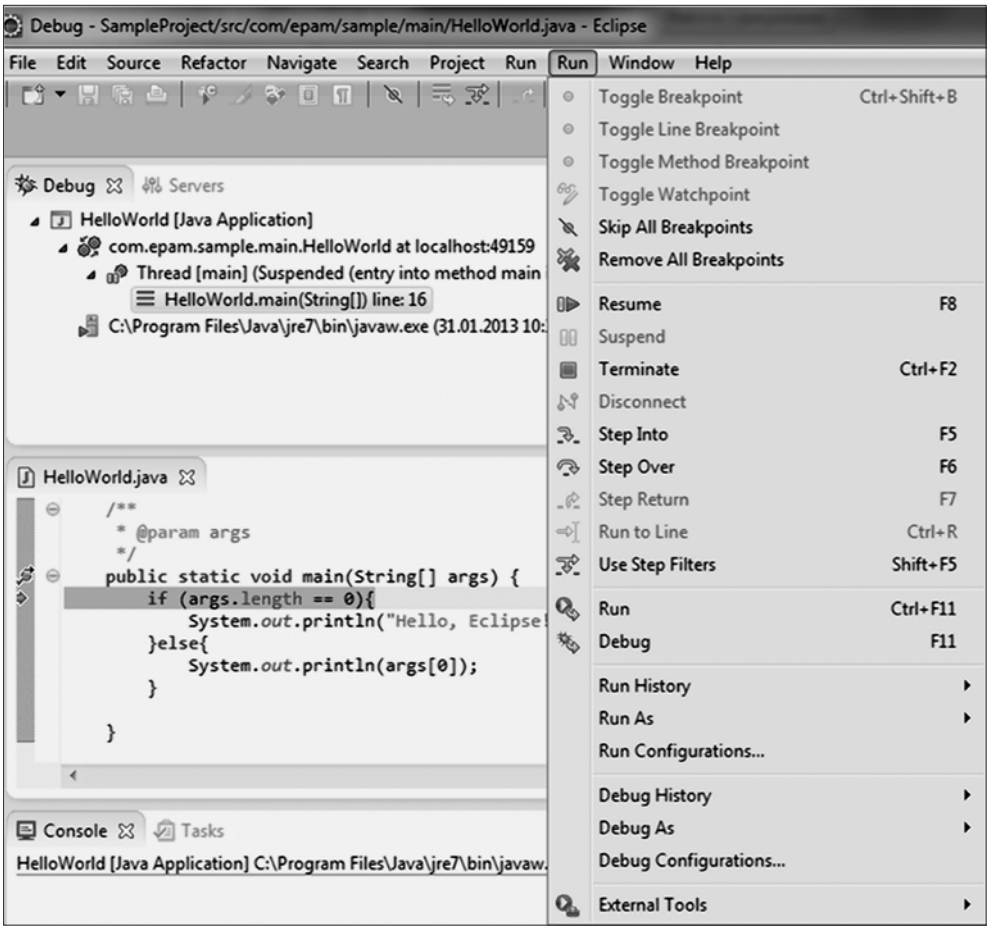

Рис. 13. *Управление отладкой*

Для управления режимом отладки можно использовать пункты меню **Run**, кнопки на вкладке **Debug** или соответствующие быстрые сочетания клавиш на клавиатуре:

F8 — продолжить,
Ctrl+F12 — прервать выполнение программы,
F5 — выполнить инструкцию,
F6 — перейти к следующей инструкции,
Ctrl+R — перейти к строке.

Генерация методов

Среда Eclipse предоставляет много дополнительных возможностей по генерации кода, что существенно сокращает рутинные действия при программировании.

Пусть создан класс **Student** с полями **int id** и **String lastName**. Определение инкапсуляции требует, чтобы поля объявлялись как **private**, а доступ к ним осуществлялся через методы **get***Имя()* и **set***Имя()*. Для получения доступа к меню генерации этих методов следует нажать правую кнопку мыши и в открывшемся меню выбрать **Source — Generate Getters and Setters**. В открытом виде будет представлено меню выбора (Рис. 14.). С его помощью можно выбрать необходимые для генерации методы, отметив их в окне *Select*. Выбрать

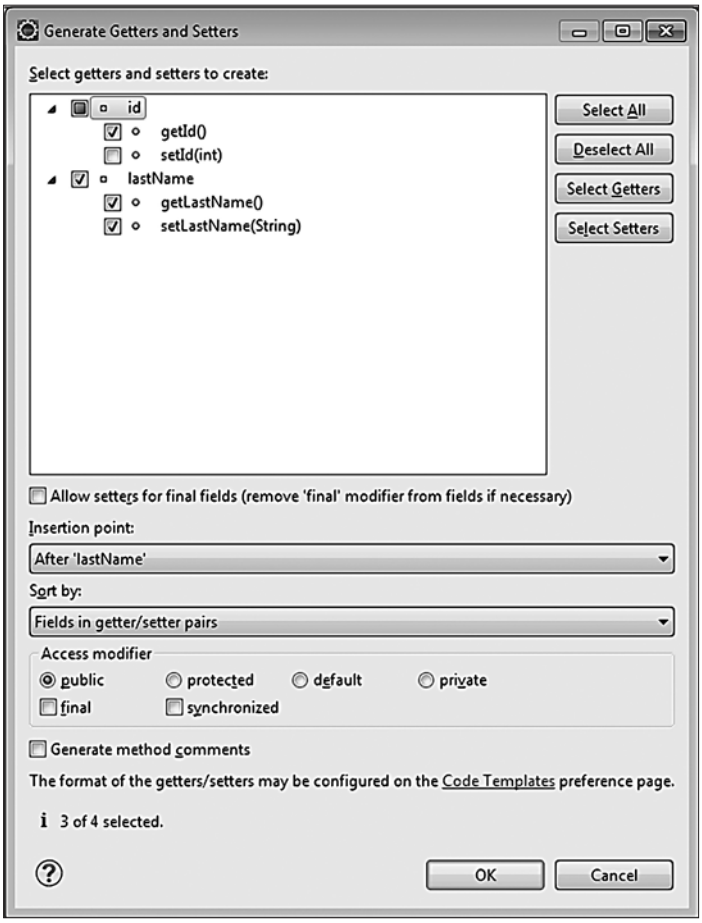

Рис. 14. *Генерация getters & setters*

место размещения в классе в выпадающем списке *Insertion point*. Задать последовательность размещения в коде в выпадающем списке *Sort by*, а также определить модификаторы доступа для создаваемых методов.

В результате выбора осуществленного в рис. 14 будет сгенерирован код в виде (см. Рис. 15).

```
*Student.java   HelloWorld.java
package com.epam.sample.main;
public class Student {
    private int id;
    private String lastName;
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public int getId() {
        return id;
    }
}
```

Рис. 15. *Созданный код getters & setters*

Обычной практикой для классов, хранящих информацию представляется автоматическая генерация методов **equals()**, **hashCode()** и **toString()**. Для доступа к меню генерации первых двух методов после щелчка правой кнопкой мыши в окне класса в появившемся меню следует выбрать **Source — Generate hachCode() and equals()**. Задать необходимые поля (по умолчанию используются все поля класса), место расположения методов в коде класса и необходимость добавления комментариев.

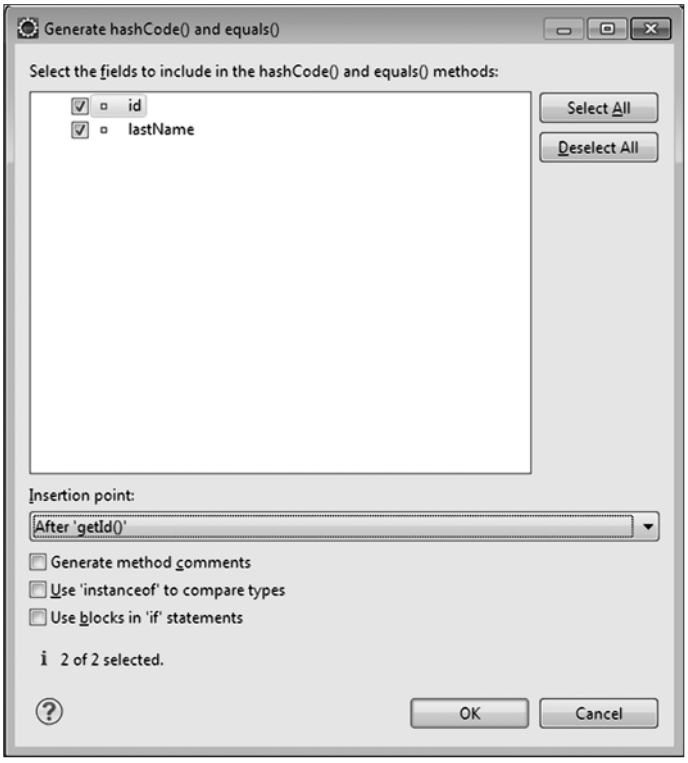

Рис. 16. *Генерация equal & hashCode*

В результате по правилам языка Java будет сгенерирован код в виде:

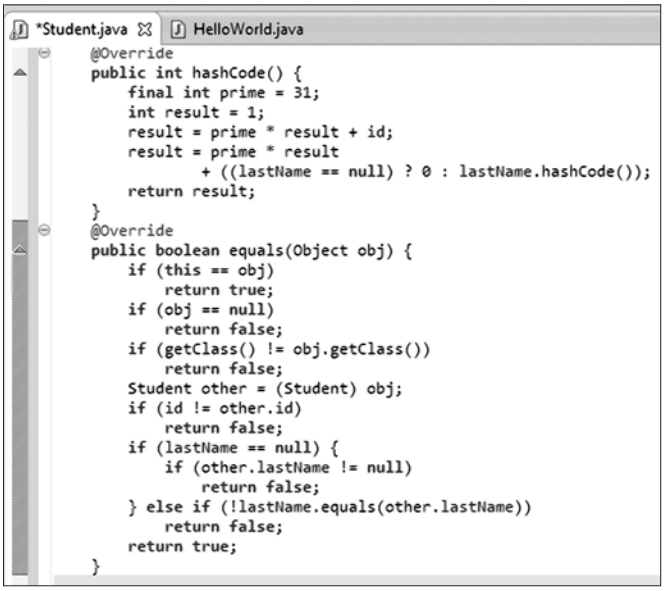

Рис. 17. *Код сгенерированных методов equal & hashCode*

Задать правила генерации метода **toString()** возможно, выбрав в том же меню **Source — Generate toString().**

В отличие от обычно неизменяемого после генерации кода методов **equals()** и **hashCode()**, код метода **toString()** часто подвергается изменениям после генерации в соответствии с потребностями разработчика.

Eclipse предоставляет удобный механизм генерации переопределяемых методов суперкласса. Пусть разработан класс **GroupStudent**, наследующий класс **ArrayList** в виде:

```
package com.epam.sample.main;
import java.util.ArrayList;
public class GroupStudent extends ArrayList<Student> { }
```

При разработке функциональности класса потребовалось переопределить методы добавления студентов в экземпляр разрабатываемого класса. Для решения этой задачи следует выбрать **Override/Implement Methods** в строке меню **Source** и в открывшемся окне развернуть список доступных для переопределения методов класса **ArrayList**. После чего выбрать необходимые методы из представленного списка.

В результате будет сгенерирован синтаксически корректный код с использованием функциональности переопределяемого метода (Рис. 20.). Разработчику остается только внести в код необходимые ему изменения.

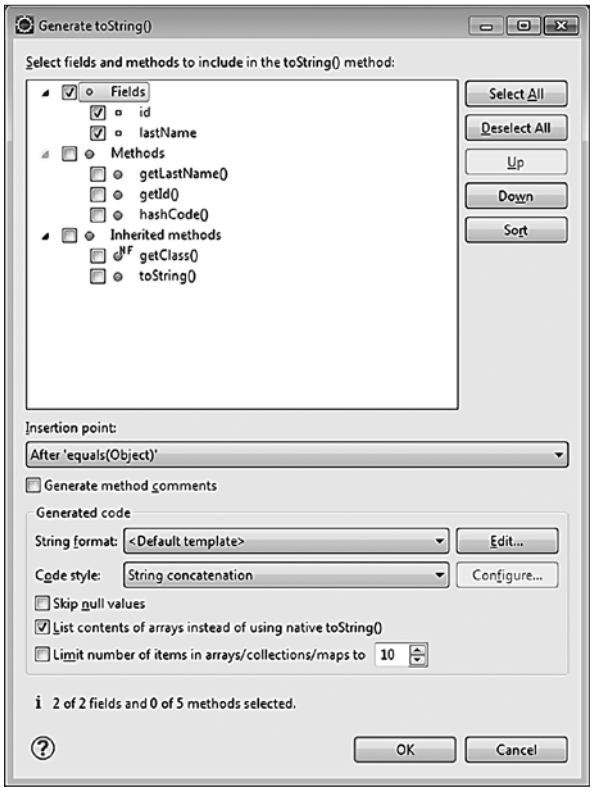

Рис. 18. *Генерация toString*

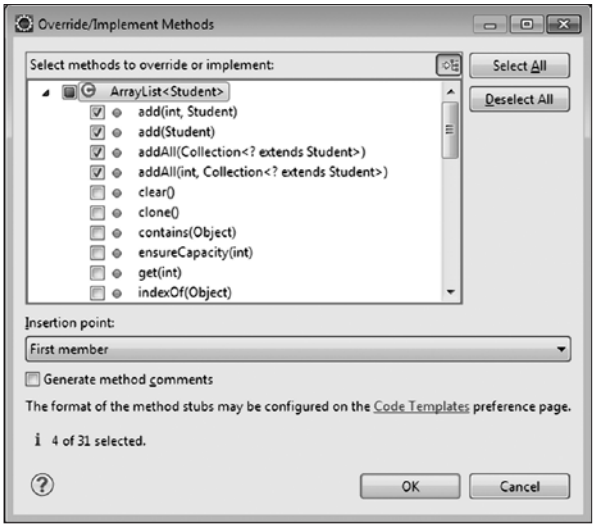

Рис. 19. *Генерация переопределяемых методов*

```
public class GroupStudent extends ArrayList<Student>{

    @Override
    public void add(int arg0, Student arg1) {
        // TODO Auto-generated method stub
        super.add(arg0, arg1);
    }

    @Override
    public boolean add(Student arg0) {
        // TODO Auto-generated method stub
        return super.add(arg0);
    }

    @Override
    public boolean addAll(Collection<? extends Student> arg0) {
        // TODO Auto-generated method stub
        return super.addAll(arg0);
    }

    @Override
    public boolean addAll(int arg0, Collection<? extends Student> arg1) {
        // TODO Auto-generated method stub
        return super.addAll(arg0, arg1);
    }
}
```

Рис. 20. *Созданный код переопределяемых методов*

Если же класс не наследует, а агрегирует свойства другого класса, как например:

```
package com.epam.sample.main;
import java.util.ArrayList;
public class GroupStudent {
          private ArrayList<Student> students;
}
```

Тогда для реализации необходимых разработчику методов с использованием имен и, возможно, функциональности агрегированного класса следует в меню **Source** выбрать **Generate Delegate Methods**. Последующие действия аналогичны действиям при генерации переопределенного кода.

Генерация конструкторов

Важную роль в коде практически любого класса выполняют конструкторы. При генерации конструкторов следует принимать во внимание особенности процесса создания экземпляра класса, для которого предназначен тот или иной конструктор. Основное назначение конструктора — инициализация полей. Для генерации конструктора на примере класса **Student** с полями **id** и **lastName** следует выбрать **Source — Generate Constructor using fields**.

В предложенном окне следует выбрать поля, которые будут использованы в качестве параметров создаваемого конструктора. Результатом будет один

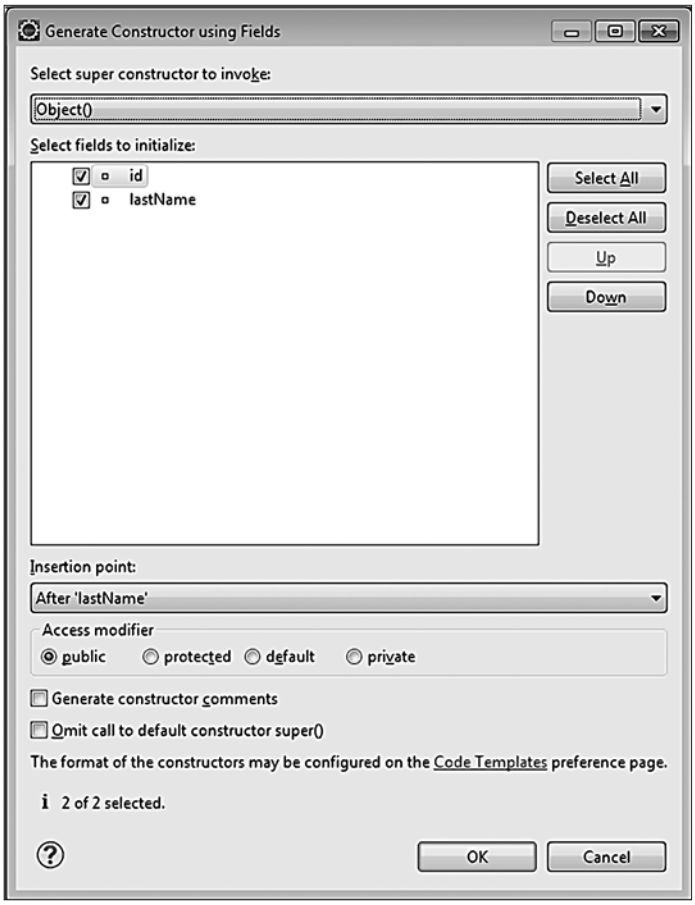

Рис. 21. *Генерация конструктора на основе полей класса*

конструктор. Если необходим конструктор с другим набором параметров, операцию создания следует повторить.

Если класс наследует другой класс, то при разработке его конструкторов может потребоваться обращение к конструкторам суперкласса. На примере класса **GroupStudent**, выбрав в списке **Source — Generate Constructors from Superclass**, будет получено окно, в котором будет представлен список конструкторов суперкласса, доступных для использования в конструкторе подкласса. Выбрав необходимый набор и нажав кнопку **OK**, будет сгенерировано столько конструкторов, сколько было отмечено.

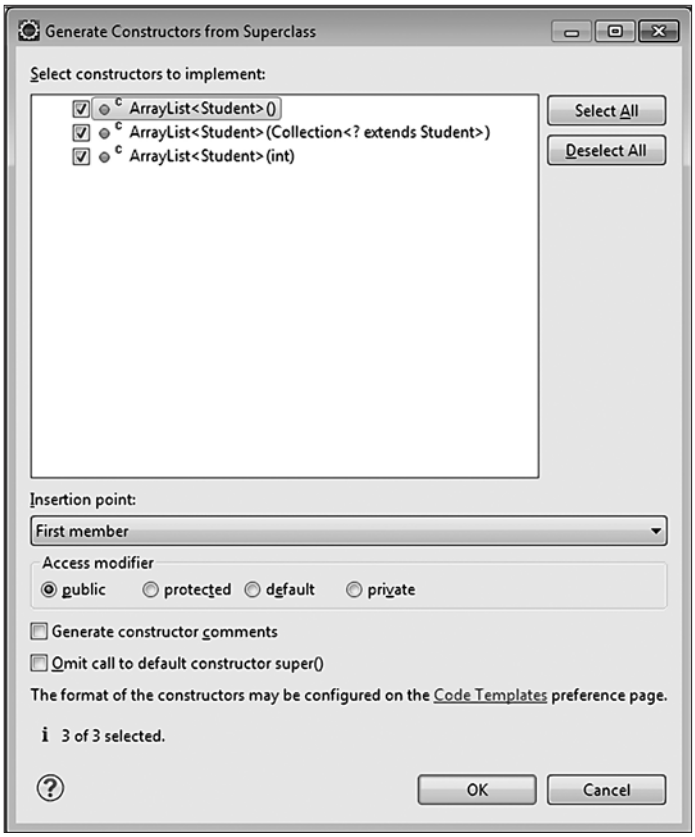

Рис. 22. *Генерация конструкторов с использованием суперкласса*

Code assist

Java редактор предоставляет возможность разработчику создавать код быстрее, предоставляя ему возможности *code assist*. Пусть в методе **main()** необходимо создать код, использующий возможности класса **System**. Достаточно набрать в редакторе

```
System.
```

Сделать небольшую паузу, и автоматически откроется окно code assist с предложением выбора статических полей и методов класса **System** (Рис. 23.). Как только выбор будет осуществлен вертикальным перемещением по списку или набором на клавиатуре начальных символов необходимого метода или поля с последующим выбором из списка, результаты выбора появятся в редакторе кода. Вызов code assist можно также осуществить нажатием клавиш **Ctrl-Space**.

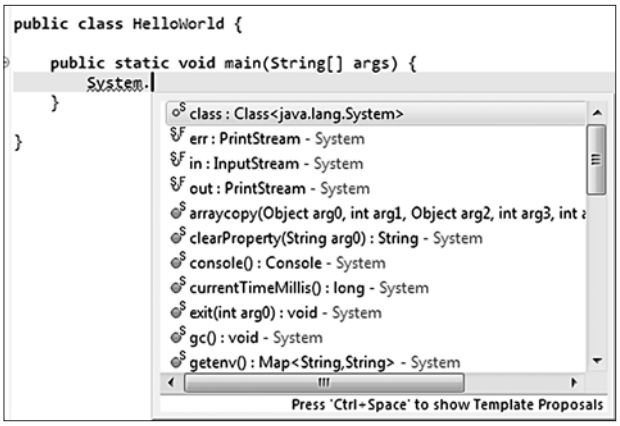

Рис. 23. *Code assist*

Шаблоны

Eclipse поддерживает короткий набор части кода с возможностью генерации части конструкции. Для этого применяется набор так называемых templates, созданных в самой среде.

Демонстрация использования шаблонов на примере циклов. В методе набирается слово **for** и нажимаются клавиши **Ctrl-Space**. Появляется меню с предложением выбрать один из четырех циклов языка или классов, начинающихся на символы, набранные в редакторе (Рис. 24.). В правой части окна располагается пример кода, который будет сгенерирован после нажатия кнопки **Enter**. После того, как код будет создан, шаблон предоставляет возможность изменения имени переменной цикла.

Список всех доступных для применения шаблонов можно получить, выбрав **Window — Preferences — Java — Editor — Templates**.

Разрешено создавать собственные шаблоны. Создание шаблона начинается с нажатия кнопки **New** (Рис. 26.).

Шаблоны, в том числе и стандартные, разрешено редактировать. Сохранить и перенести авторский набор шаблонов на другую версию Eclipse можно, применяя кнопки **Export/Import** в окне, представленном на рис. 25.

Рефакторинг

Созданный код может потребовать исправления. Чаще всего разработчику необходимо переименовать пакет, класс, метод, поле, переменную или константу. Причем требующее изменения понятие, например метод, может быть

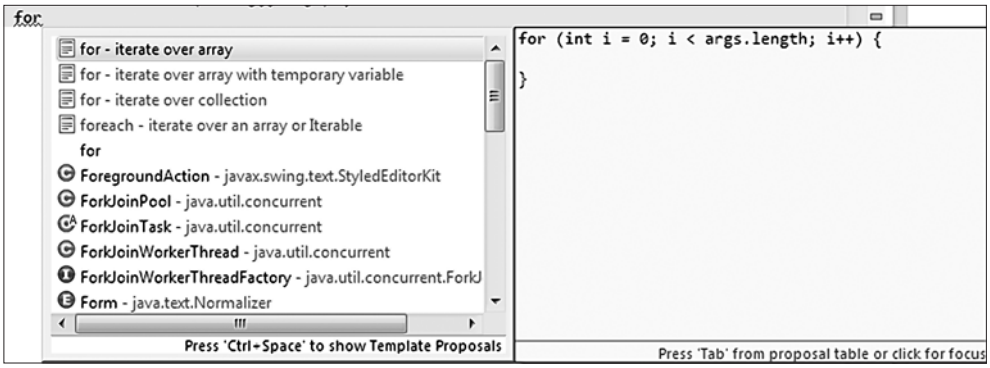

Рис. 24. *Вызов шаблона*

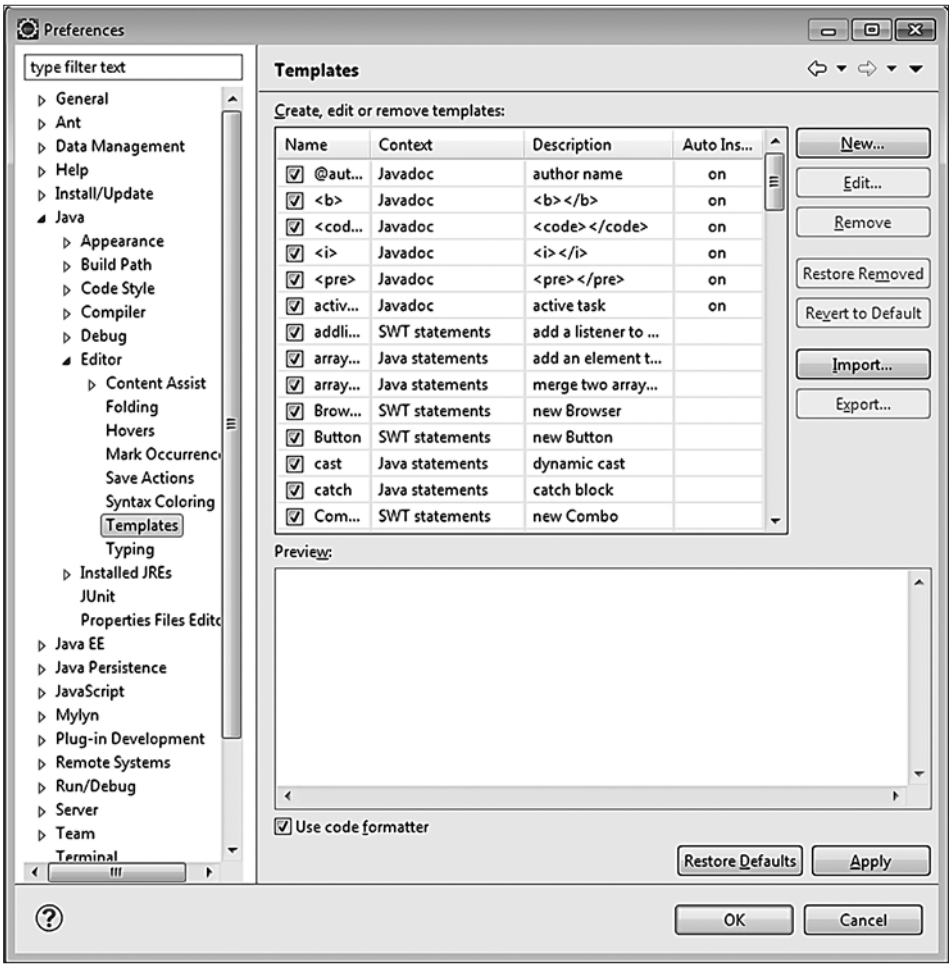

Рис. 25. *Список доступных шаблонов*

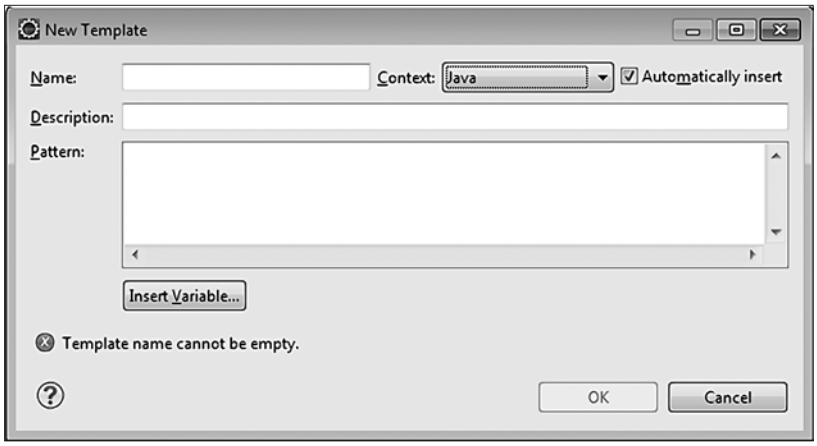

Рис. 26. *Меню создания шаблона*

многократно использовано в коде. В обычной ситуации разработчику придется перебирать все классы, в которых осуществлялся вызов такого метода и исправлять его имя, что может оказаться весьма трудоемкой задачей.

Eclipse позволяет изменить имя класса, метода, поля или переменной, распространяя изменения на все обращения к изменяемому понятию, в том числе и из других классов и пакетов приложения.

Чтобы осуществить изменение имени, следует двойным щелчком левой клавиши мыши выделить имя, требующее изменения, после чего правой кнопкой вызвать список, выбрать в нем **Source — Refactor — Rename**, ввести новое имя и нажать **Enter** для фиксации изменений в приложении.

При рефакторинге метода может понадобиться не только изменение его имени, но и корректировка всей сигнатуры. Для этого следует выделить метод и выбрать **Source — Refactor — Change Method signature** в результате чего, например, для метода **void setLastName(String lastName)** будет выведена форма редактирования. В этом окне кроме имени можно изменить тип возвращаемого методом значения, спецификатор доступа к методу. Список аргументов метода может быть увеличен добавлением новых аргументов или сокращен при необходимости. Имена аргументов также можно изменять. Предварительный вид сигнатуры будет отображаться в строке *Method signature preview*. Изменения коснутся также всех переопределенных версий метода во всех подклассах.

Рефакторинг предоставляет много других возможностей при работе с классом и его составляющими.

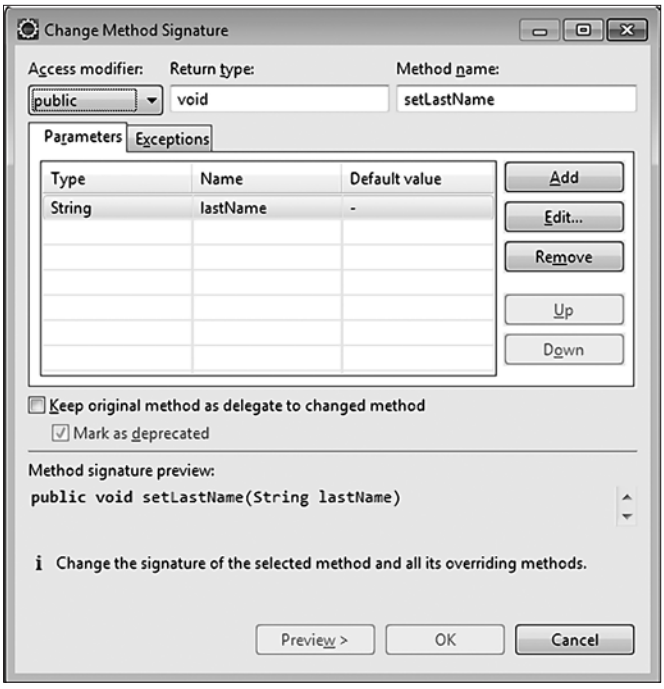

Рис. 27. *Рефакторинг сигнатуры метода*

Подключение библиотек

При разработке проектов кроме стандартных библиотек Java необходимы библиотеки сторонних разработчиков, например: Log4J, JUnit, Xerces и прочие. Чтобы воспользоваться функциональностью классов из этих библиотек, их необходимо подключить к проекту. Сначала требуется загрузить jar-архив библиотеки с определенного интернет-ресурса и разместить его на диске, желательно в директории, находящейся внутри основной директории проекта, хотя в общем случае это необязательно. Для включения библиотеки в ресурсы проекта необходимо щелкнуть правой кнопкой мыши по проекту и из выпавшего списка выбрать **Build Path — Configure Build Path**.

В появившемся окне следует выбрать закладку **Libraries** и нажать кнопку **Add External JARs** (Рис. 28.). С помощью формы файловой системы найти необходимый jar-архив, например, **log4j-[*версия*].jar** и нажать кнопку **Open**. После чего указанная библиотека появится в списке подключенных библиотек.

Библиотека Log4J будет видна при открытии папки *Referenced Libraries*.

Теперь классы библиотеки можно использовать в разрабатываемом проекте.

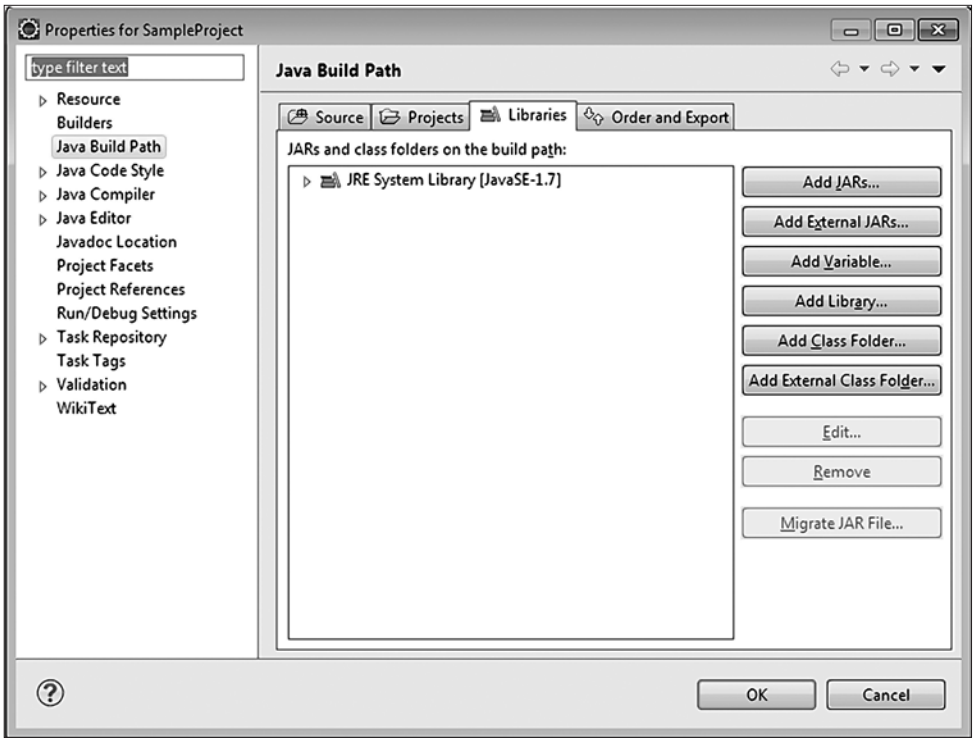

Рис. 28. *Подключение внешней библиотеки*

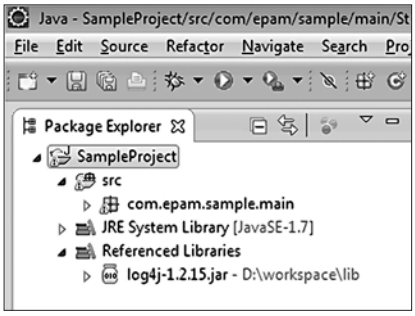

Рис. 29. *Проект с подключенной библиотекой*

Подключение plugin-ов

IDE Eclipse представляет собой настраиваемую систему, возможности которой можно наращивать и улучшать. Добавление новых возможностей производится за счет подключения plugin-ов. Plugin представляет собой некую небольшую

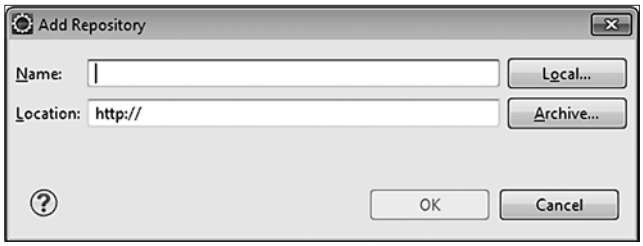

Рис. 30. *Проект с подключенной библиотекой*

программу, для подключения которой следует выбрать **Help — Install New Software** и в появившейся форме нажать кнопку **Add** (Рис. 30.). В поле *Name* указать имя плагина, в поле *Location* адрес URL его загрузки и запустить процесс. Затем плагин будет закачан на диск разработчика и подключен, после чего сразу становится доступным для использования.

Существуют еще три способа подключения плагинов, если они уже находятся на диске разработчика. Если плагин представлен в виде архива, то следует на форме нажать кнопку **Archive** и выбрать искомый архив с плагином. Если плагин представлен в виде набора папок с файлами, то следует выбрать **Local** и указать на корневую папку плагина. Два последних способа могут не сработать, так как плагины, разработанные к разным версиям Eclipse, могут быть несовместимы с представленной версией. Есть еще один кустарный способ подключения плагинов прямым копированием в папку plugins самого IDE Eclipse. Несовместимость версий в этом случае может привести к тому, что плагин может быть вообще не представлен в функционале IDE или работать некорректно.

Совместимость версии плагина с версией Eclipse следует проверять на сайте разработчика плагина или на eclipse.org.

Импорт/экспорт проектов

В процессе разработки проект может переноситься с одного компьютера на другой. Для переноса проекта следует правым кликом на проекте вызвать его меню и выбрать **Export — Java — JAR file — Next** (Рис. 31.). В открывшейся форме обязательно пометить check box с надписью *Export Java source files and resources*. Иначе архив, полученный в результате экспорта, не будет содержать файлы с исходным кодом проекта. В поле *JAR file* следует указать месторасположение и имя jar-архива и нажать кнопку **Finish**. После чего можно забрать созданный архив и переносить на другой компьютер.

Экспорт можно выполнить, и в директорию на диске для этого необходимо выбрать **Export — General –File System — Next**. В поле *To directory* открывшейся

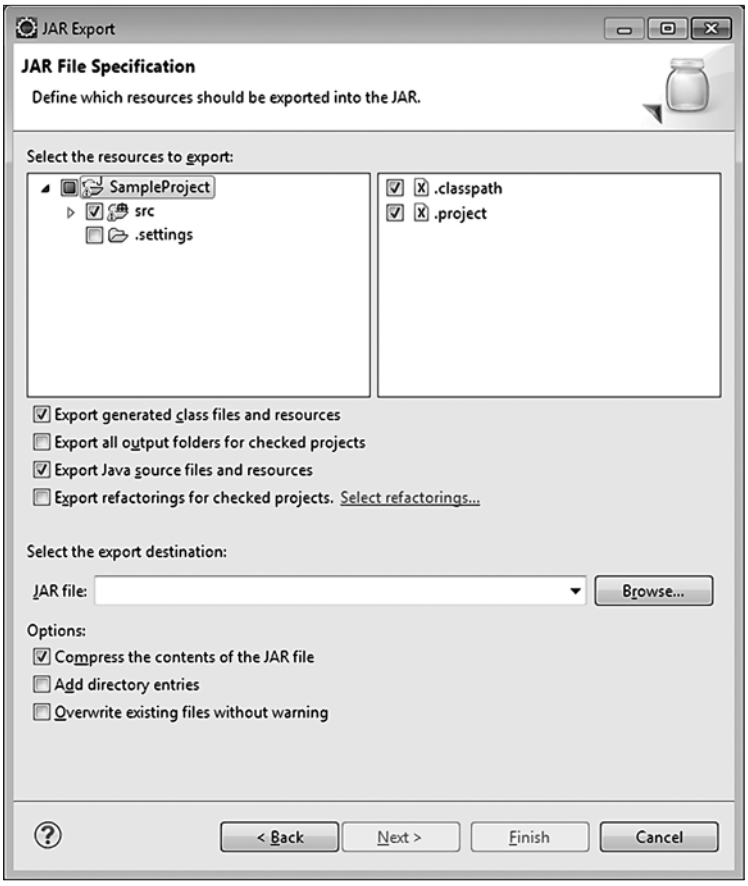

Рис. 31. *Экспорт проекта*

формы ввести путь, по которому после выполнения экспорта можно будет забрать директорию с файлами проекта.

Для импорта проекта в Eclipse следует сначала создать новый java проект, например, с именем *SampleProjectCopy*. После чего выбрать **File — Import — General — Archive file** (Рис. 32.). Далее выбрать *SampleProjectCopy/src* в поле *Into folder.* В поле *From archive file* выбрать jar-архив импортируемого проекта и нажать кнопку **Finish**.

Все исходные коды импортируемого проекта окажутся в папке *src* проекта *SampleProjectCopy*. Существуют и другие способы импорта.

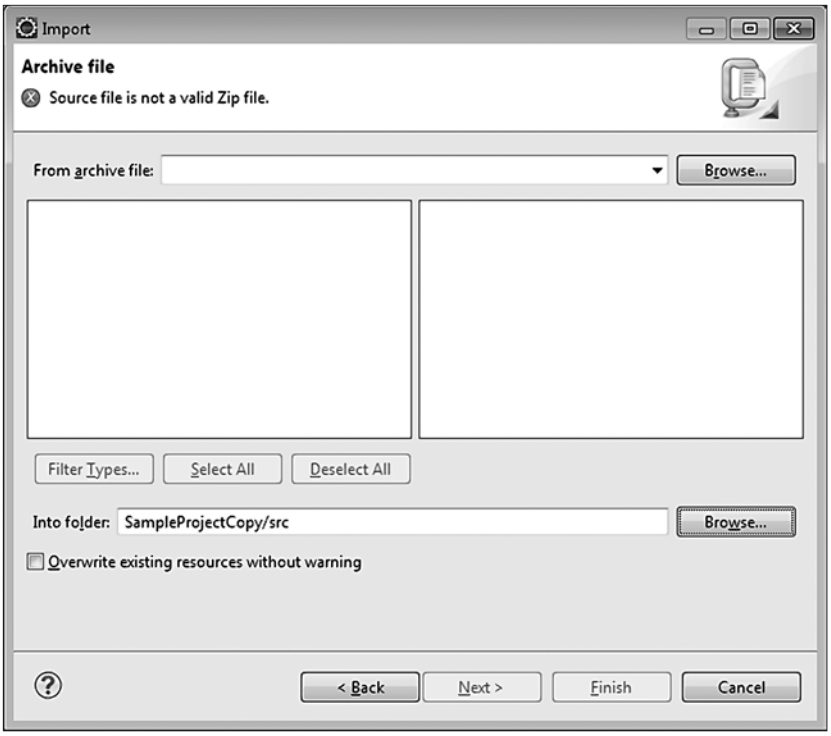

Рис. 32. *Импорт проекта*

Создание, запуск и отладка веб-проекта

Перед созданием динамического веб-проекта необходимо настроить сервер java-приложений. Для этого в меню **Window — Preferences** выбирается в дереве слева **Server — Runtime Environments**. Чтобы добавить application server, например, Apache Tomcat необходимо нажать кнопку **Add**…

После нажатия кнопки **Finish** в списке серверов появится только что добавленный сервер. Теперь необходимо выбрать меню **File — New — Dynamic web project**, задать имя проекта, выбрать сервер. Можно задать некоторые дополнительные параметры, нажимая кнопку **Next**, можно оставить все остальное по умолчанию и нажать кнопку **Finish** для создания проекта (Рис. 35.). Созданный проект появится в **Project Explorer** (Рис. 36.).

Далее следует нажать **Next**, и на следующей форме также нажать **Next**. На полученной форме конфигурирования веб-модуля выбрать *Generate web. xml deployment descriptor*. Наличие файла **web.xml** необходимо для задания важных параметров инициализации веб-приложения.

Для завершения создания веб-проекта следует нажать **Finish** и в закладке *Project Explorer* появится проект.

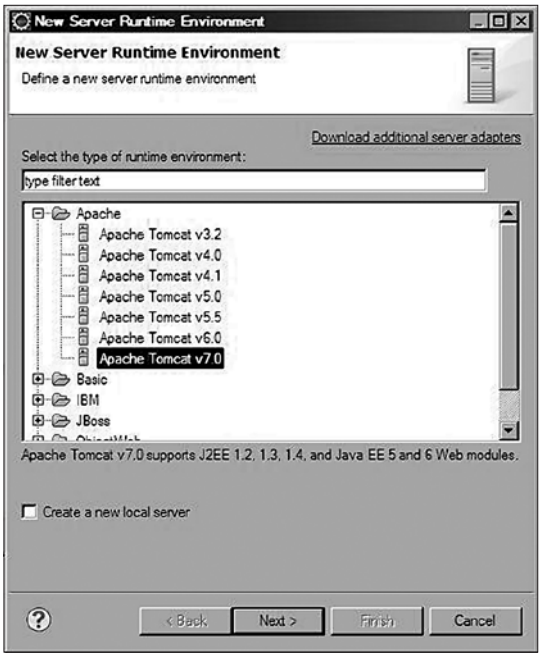

Рис. 33. *Выбор сервера для добавления*

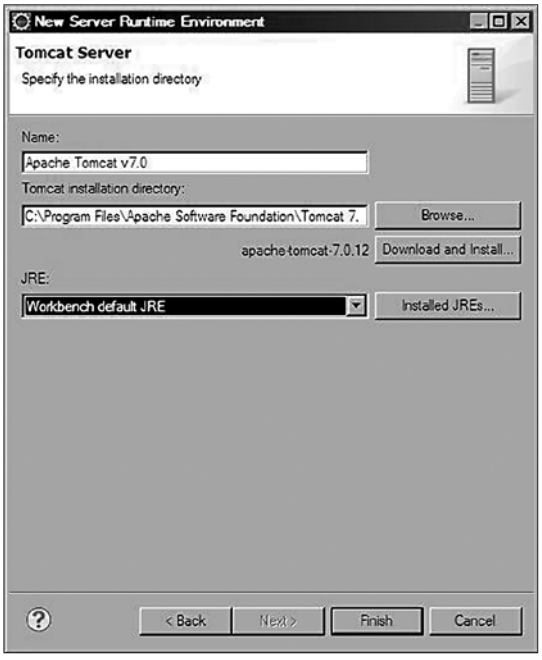

Рис. 34. *Определение директории сервера*

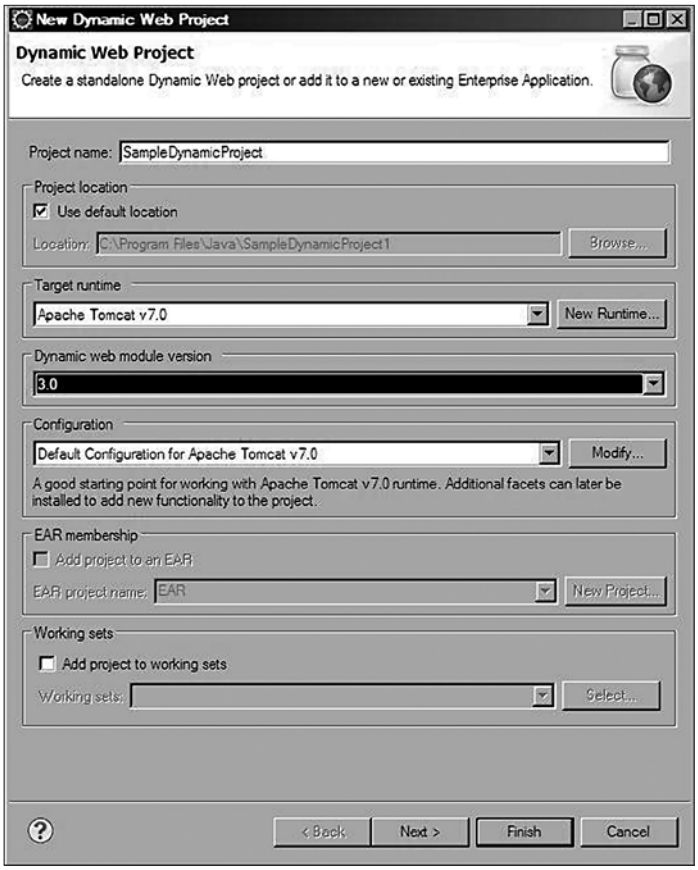

Рис. 35. *Создание нового веб-проекта*

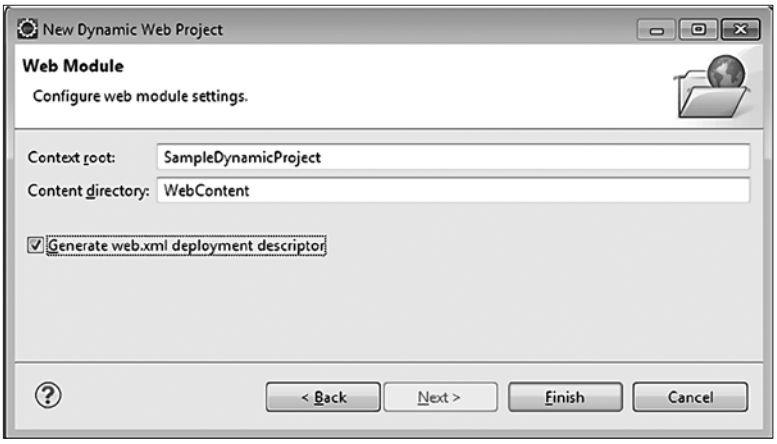

Рис. 36. *Создание нового веб-проекта*

Рис. 37. *Вид веб-проекта в Project Explorer*

Для запуска веб-проекта необходимо нажать правой кнопкой мыши на название проекта в **Project Explorer** и выбрать пункт **Run As — Run On Server**. Для отладки: **Debug As — Debug On Server**. Управление процессом отладки такое же, как в консольном проекте.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Арнольд, К., Гослинг, Дж., Холмс, Д. Язык программирования Java. — 3-е изд. — М. : Вильямс, 2001. — 624 с.
2. Эккель, Б. Философия Java. — 4-е изд. — СПб. : Питер, 2011. — 640 с.
3. Блох, Д. Java. Эффективное программирование. — М. : Лори, 2002. — 224 с.
4. Макконнелл, С. Совершенный код. — СПб. : Питер, 2005. — 896 с.
5. Хорстманн, К. С., Корнелл, Г. Библиотека профессионала. Java 2 : Том 1. Основы. — 8-е изд. — М. : Вильямс, 2013. — 816 с.
6. Хорстманн, К. С., Корнелл, Г. Библиотека профессионала. Java 2. : Том 2. Тонкости программирования. — 8-е изд. — М. : Вильямс, 2012. — 992 с.
7. Блинов, И. Н., Романчик, В. С. Java 2. Практическое руководство. — Минск : УниверсалПресс, 2005. — 400 с.
8. Блинов, И. Н., Романчик, В. С. Java. Промышленное программировние. — Минск : УниверсалПресс, 2007. — 704 с.
9. Перри, Б. У. Java сервлеты и JSP. Сборник рецептов. — М. : Кудиц-пресс, 2009. — 768 с.
10. Тейт, Б. Горький вкус Java. — СПб. : Питер, 2003. — 334 с.
11. Буч, Г., Рамбо, Дж., Джекобсон, А. Язык UML. Руководство пользователя. — М. : ДМК, 2000. — 432 с.
12. Фаулер, М. UML. Основы. — 3-е изд. — СПб. : Символ-плюс, 2006. — 192 с.
13. Ларман, К. Применение UML 2.0 и шаблонов пректирования. Введение в объектно-ориентированный анализ и проектирование. — 3-е изд. — СПб. : Вильямс, 2012. — 736 с.
14. Эдди, С. Э. XML: справочник. — СПб. ; М. ; Х. ; Минск : Питер, 2000. — 480 с.
15. Стелтинг, С., Маасен, О. Java. Применение шаблонов Java. — М. : Вильямс, 2002. — 576 с.
16. Гамма, Э., Хелм, Р., Джонсон, Р., Влиссидес, Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб. : Питер, 2007. — 366 с.
17. Кириевски, Д. Рефакторинг с использованием шаблонов. — М. : Вильямс, 2008. — 400 с.
18. Keith, M., Schincariol, M. Pro EJB3: Java Persistence API. — Apress, 2006. — 480 p.
19. Bauer, C., King, G. Java Persistence with Hibernate. — Manning, 2007. — 876 p.
20. Ceki Gülcü. The complete log4j Manual. — eBook, 2002. — 180 p.
21. Burnette, E. Eclipse IDE Pocket Guide. — O'Reilly, 2005. — 127 p.

Учебное издание

БЛИНОВ Игорь Николаевич
РОМАНЧИК Валерий Станиславович

Java
Методы программирования

Учебно-методическое пособие

Публикуется в авторской редакции

В оформлении обложки использован стилизованный фрагмент полинезийского орнамента с острова Java

Дизайн обложки *Е. В. Драченов*
Компьютерная верстка *И. П. Бондарович*
Корректоры *Е. И. Бондаренко, Л. А. Анисовец*

Подписано в печать 05.06.2013.
Формат 70×100/16. Гарнитура Times.
Бумага офсетная. Печать офсетная.
Усл. печ. л. 72,8. Уч.-изд. л. 43,70.

Отпечатано в типографии ООО «Поликрафт».
Ул. Кнорина, 50, корп. 4, к. 401а, 220103, Минск.
ЛП 02330/0494199 от 03.04.2009.
Тираж 500 экз. Заказ .