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 — именнованная 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.

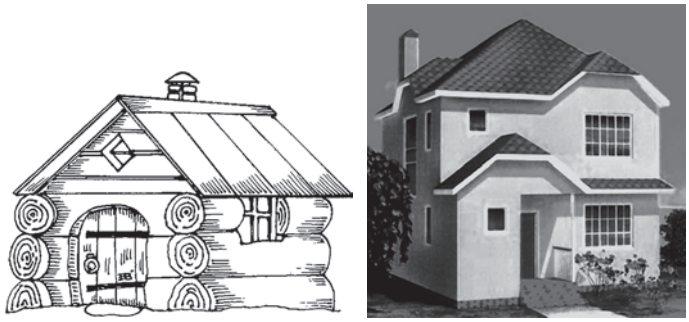

Рис. 2.1. *Описание класса (абстракция) и реальный объект*

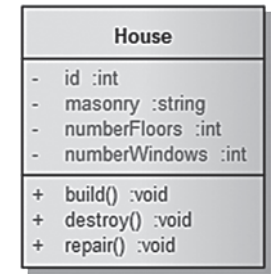

Рис. 2.2. *Графическое изображение класса*

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 объектно-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 обьекта, доступ к полям и методам обьекта # 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_.

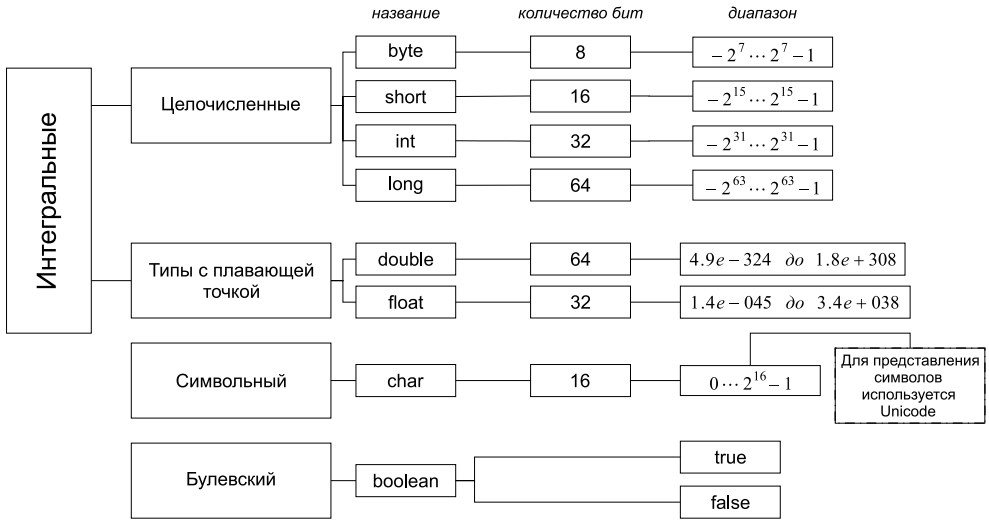

Рис. 2.1. *Базовые типы данных и их свойства*

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:

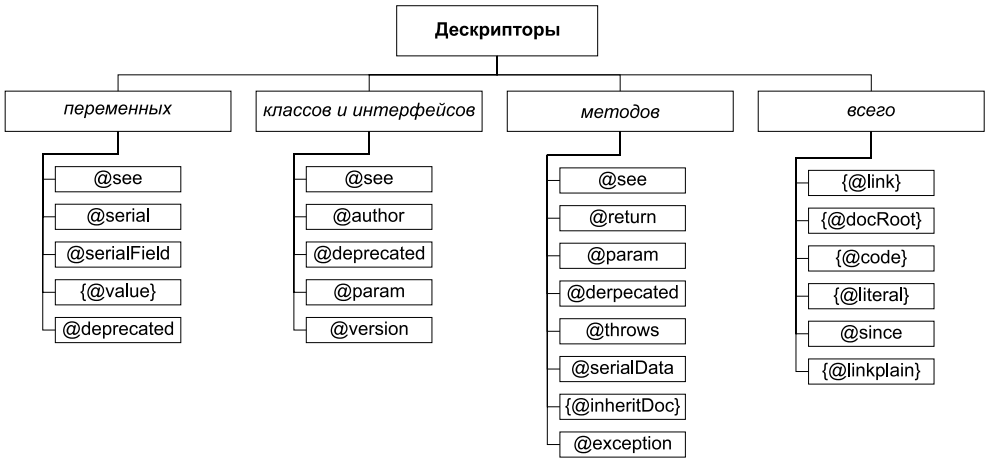

Рис. 2.2. *Дескрипторы документирования кода*

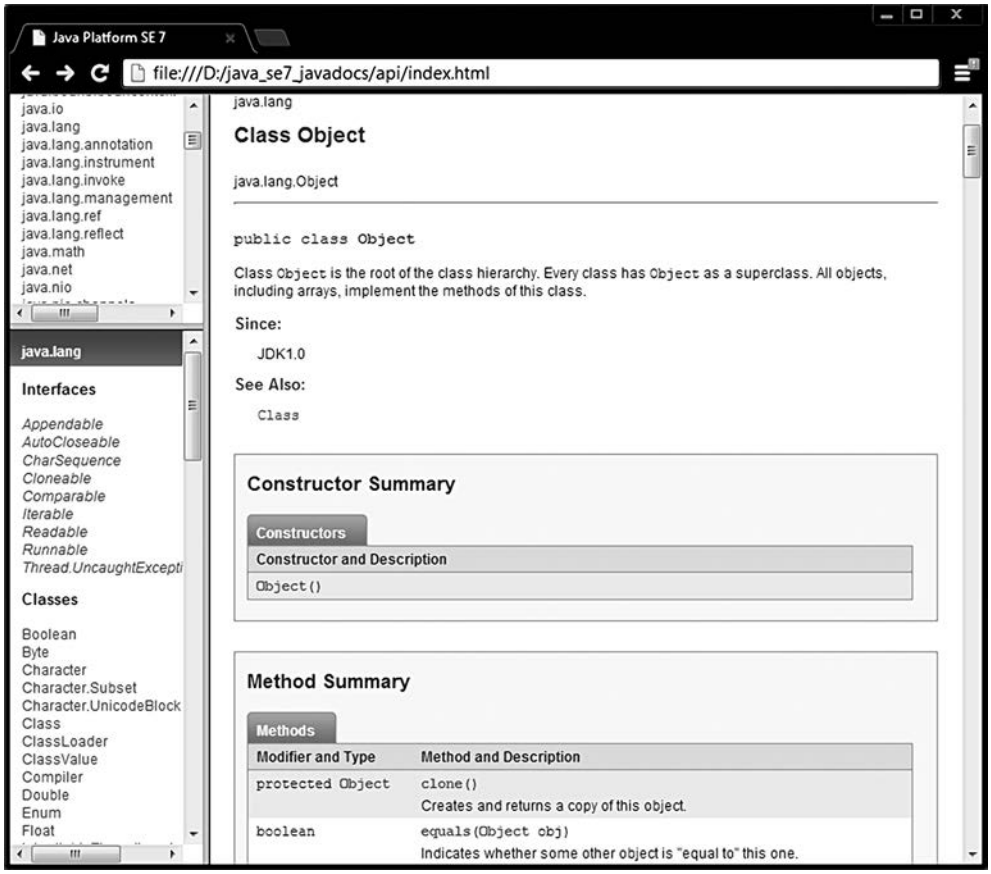

Рис. 2.3. *Сгенерированная документация для класса Object*

@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.