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_.

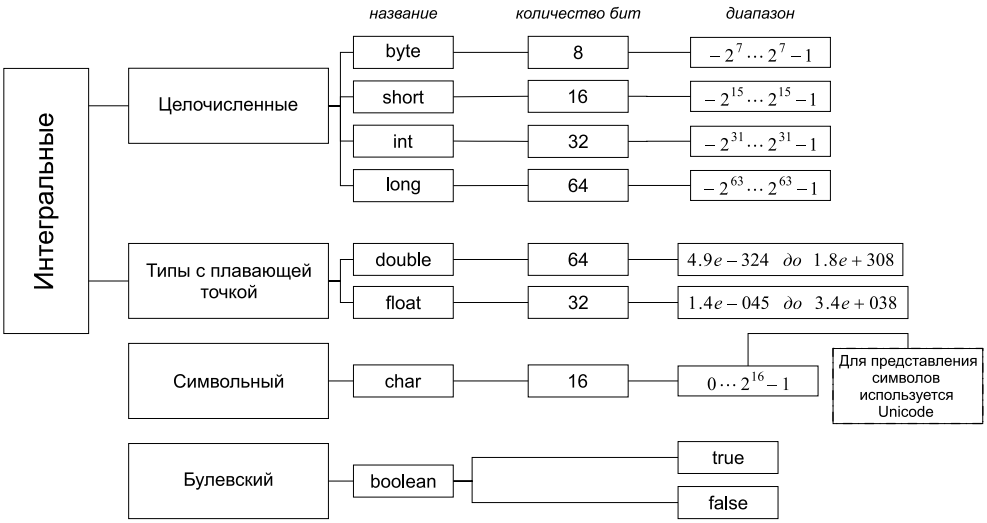

Рис. 2.1. *Базовые типы данных и их свойства*

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:

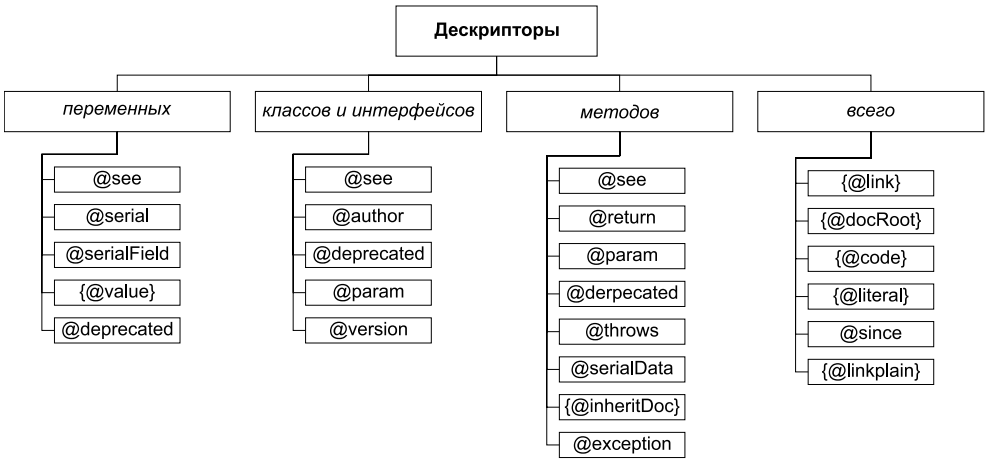

Рис. 2.2. *Дескрипторы документирования кода*

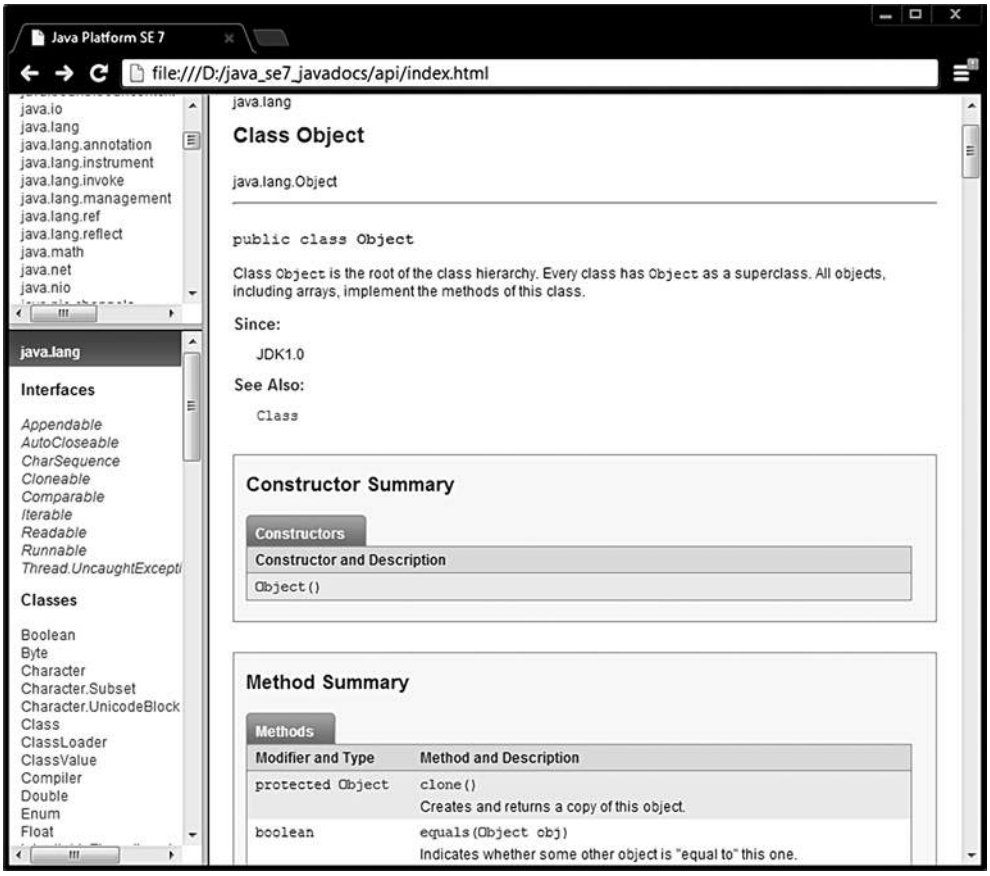

Рис. 2.3. *Сгенерированная документация для класса Object*

@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 */}
}
```

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**.

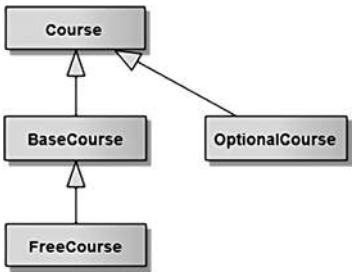

Рис. 2.5. *Иерархия наследования*

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.

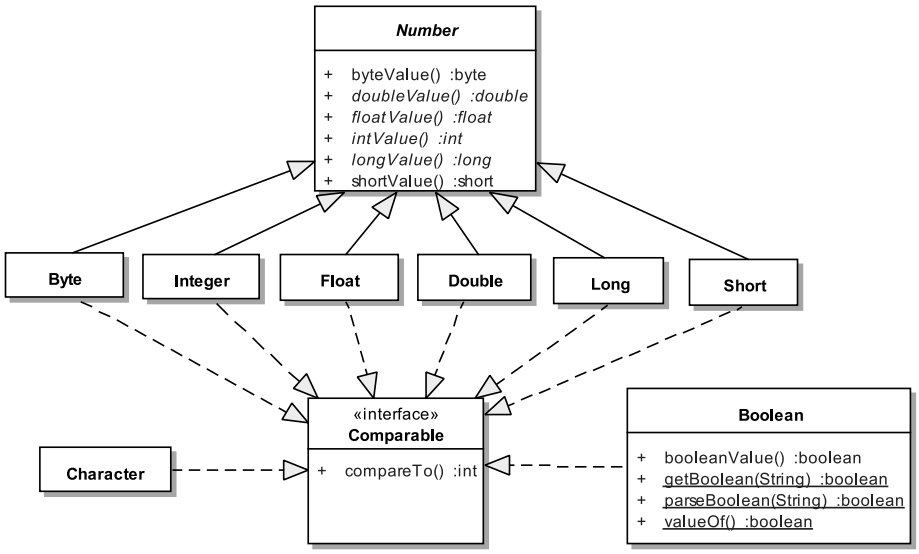

Рис. 2.6. *Иерархия классов-оболочек*

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)?