Глава 2
ТИПЫ ДАННЫХ И ОПЕРАТОРЫ

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

Базовые типы данных и литералы

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

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

Тип	Размер (бит)	По умолчанию	Значения (диапазон или максимум)
`boolean`	8	`false`	`true, false`
`byte`	8	0	`-128..127`
`char`	16	`'\u0000'`	`0..65535`
`short`	16	0	`-32768..32767`
`int`	32	0	`-2147483648..2147483647`
`long`	64	0	`9223720368547758O7L`
`float`	32	0.0	`3.40282347E+38`
`double`	64	0.0	`1.797693134486231570E+308`

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

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

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

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

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

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

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

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

```
/* пример # 1 : типы данных, литералы и операции над ними :TypeByte.java */
package chapt02;

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

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

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

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

Документирование кода

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

@author – задает сведения об авторе;

@version – задает номер версии класса;

@exception – задает имя класса исключения;

@param – описывает параметры, передаваемые методу;

@return – описывает тип, возвращаемый методом;

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

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

@throws – описывает исключение, генерируемое методом;

@see – что следует посмотреть дополнительно.

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

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

```
package chapt02;

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

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

Сгенерированный для этого класса HTML-документ будет иметь вид:

chapt02

Class User

java.lang.Object
  └ **chapt02.User**

---

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

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

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

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

Рис. 2.1. Фрагмент документации для класса User

Операторы

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

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

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

Арифметические операторы

+	Сложение	/	Деление (или деление нацело для целочисленных значений)
+=	Сложение (с присваиванием)	/=	Деление (с присваиванием)
–	Бинарное вычитание и унарное изменение знака	%	Остаток от деления (деление по модулю)
–=	Вычитание (с присваиванием)	%=	Остаток от деления (с присваиванием)
*	Умножение	++	Инкремент (увеличение значения на единицу)
*=	Умножение (с присваиванием)	––	Декремент (уменьшение значения на единицу)

Битовые операторы над целочисленными типами

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

Операторы отношения

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

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

Логические операторы

||	*Или*	&&	*И*
!	Унарное отрицание		

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

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

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

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

Результатом выполнения данного кода будет:

```
5%1=0  5%2=1
14|9 = 15
14&9 = 8
14^9 = 7
~9 = -10
14>>1 = 7
14<<1 = 28
14>>>2 = 3
```

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

booleanзначение ? первое : второе

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

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

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

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

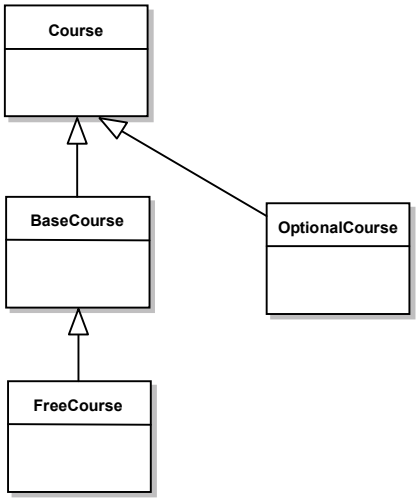

Рис. 2.2. Иерархия наследования

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

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

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

Классы-оболочки

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

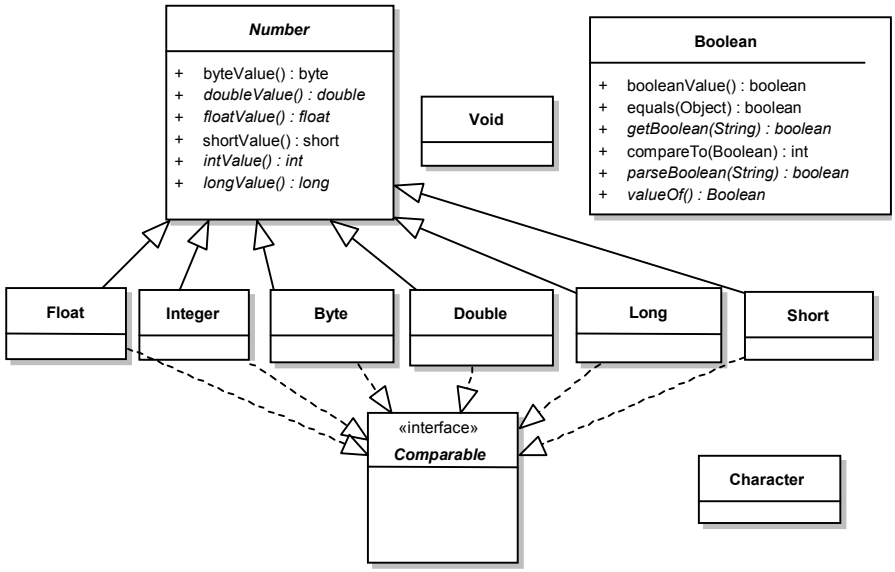

Рис. 2.3. Классы-оболочки

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

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

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

```
/* пример # 3 : преобразование типов данных: CastTypes.java */
package chapt02;

public class CastTypes {
    public static void main(String[] args) {
        Float f1 = new Float(10.71); // double в Float
        String s1 = Float.toString(0f); // float в String
        String s2 = String.valueOf(f1); // Float в String
        Byte b = Byte.valueOf("120"); // String в Byte
        double d = b.doubleValue(); // Byte в double
```

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

Результатом выполнения данного кода будет:

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

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

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

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

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

```
Integer iob = 71;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

вызывает ошибку компиляции.

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

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

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

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

Операторы управления

Оператор выбора **if** имеет следующий синтаксис:

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

Если выражение **boolexp** принимает значение **true**, то выполняется группа операторов **1**, иначе – группа операторов **2**. При отсутствии оператора **else** операторы, расположенные после окончания оператора **if** (строка 2), выполняются

вне зависимости от значения булевского выражения оператора **if**. Допустимо также использование конструкции-лесенки **if {} else if {}**.

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

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

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

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

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

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

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

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

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

Некоторые рекомендации при проектировании циклов:

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

- для индексов следует применять осмысленные имена.
- циклы не должны быть слишком длинными. Такой цикл претендует на выделение в отдельный метод.
- вложенность циклов не должна превышать трех.

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

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

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

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

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

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

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

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

Массивы

Массив представляет собой объект, где имя массива является объектной ссылкой. Элементами массива могут быть значения базового типа или объекты. Индексирование элементов начинается с нуля. Все массивы в языке Java являются динамическими, поэтому для создания массива требуется выделение памяти с помощью оператора **new** или прямой инициализации. Значения элементов неинициализированного массива, для которого выделена память, устанавливаются в значения по умолчанию для массива базового типа или **null** для массива объектных ссылок. Для объявления ссылки на массив можно записать пустые квадратные скобки после имени типа, например: **int a[]**. Аналогичный результат получится при записи **int[] a**.

```
/* пример # 5 : массивы и ссылки: ArrRef.java */
package chapt02;

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

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

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

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

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

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

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

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

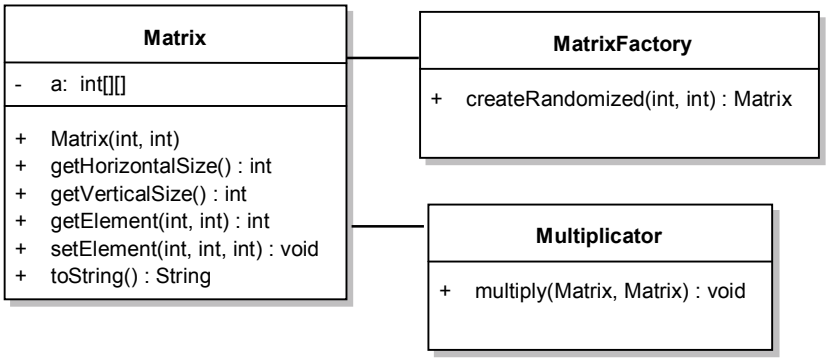

Рис. 2.4. Диаграмма классов для создания и умножения матриц

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

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

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

```
/* пример # 7 : класс-создатель матрицы : MatrixFactory.java */
package chapt02;

public class MatrixFactory {

      public static Matrix createRandomized(int n, int m) {
            Matrix matrix = new Matrix(n, m);
            for (int i = 0; i < n; i++) {
                  for (int j = 0; j < m; j++) {
      matrix.setElement(i, j, (int)(Math.random()*m*m));
                  }
            }
            return matrix;
      }
}
/* пример # 8 : произведение двух матриц : Multiplicator.java */
package chapt02;

public class Multiplicator {

      public static Matrix multiply(Matrix p, Matrix q)
                  throws MultipleException {
            int v = p.getVerticalSize();
            int h = q.getHorizontalSize();
            int temp = p.getHorizontalSize();
            // проверка возможности умножения
            if (temp != q.getVerticalSize())
                  throw new MultipleException();
            // создание матрицы результата
            Matrix result = new Matrix(v, h);
            // умножение
            for (int i = 0; i < v; i++)
                  for (int j = 0; j < h; j++) {
                        int value = 0;
                        for (int k = 0; k < temp; k++) {
      value += p.getElement(i, k) * q.getElement(k, j);
                        }
                        result.setElement(i, j, value);
                  }
            return result;
      }
}
/* пример # 9 : исключительная ситуация матрицы : MultipleException.java */
package chapt02;
public class MultipleException extends Exception {}

/* пример # 10 : класс, запускающий приложение : Runner.java */
package chapt02;
```

```
public class Runner {
      public static void main(String[] args) {
          int n = 2, m = 3, l = 4;
          Matrix p = MatrixFactory.createRandomized(n, m);
          Matrix q = MatrixFactory.createRandomized(m, l);
            System.out.println("Matrix first is: " + p);
            System.out.println("Matrix second is: " + q);

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

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

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

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

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

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

Класс Math

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

Все методы класса вызываются без создания экземпляра класса (создать экземпляр класса **Math** невозможно). В классе определено большое количество методов для математических вычислений, а также ряд других полезных методов,

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

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

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

Один из вариантов выполнения кода представлен ниже:

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

Управление приложением

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

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

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

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

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

Properties getProperties() – получение всех свойств;

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

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

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

void loadLibrary(String libname) – загрузка библиотеки;

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

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

```
/*пример # 12 : информация о состоянии оперативной памяти:
RuntimeDemo.java*/
package chapt02;

public class RuntimeDemo {
     public static void main(String[] args) {
          Runtime rt = Runtime.getRuntime();
          System.out.println("Полный объем памяти: "
                    + rt.totalMemory());
          System.out.println("Свободная память: "
                    + rt.freeMemory());
          double d[] = new double[10000];
          System.out.println("Свободная память после" +
             " объявления массива: " + rt.freeMemory());
             //инициализация процесса
          ProcessBuilder pb =
          new ProcessBuilder("mspaint","c:\\temp\\cow.gif");

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

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

```
Полный объем памяти: 2031616
Свободная память: 1903632
Свободная память после объявления массива: 1823336
Свободная память после запуска mspaint.exe: 1819680
Список команд: [mspaint, c:\temp\cow.gif]
```

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

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

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

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

В результате будет выведено:

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

Задания к главе 2

Вариант А

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

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

Вариант В

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

1. Упорядочить строки (столбцы) матрицы в порядке возрастания значений элементов **k**-го столбца (строки).
2. Выполнить циклический сдвиг заданной матрицы на **k** позиций вправо (влево, вверх, вниз).
3. Найти и вывести наибольшее число возрастающих (убывающих) элементов матрицы, идущих подряд.
4. Найти сумму элементов матрицы, расположенных между первым и вторым положительными элементами каждой строки.
5. Транспонировать квадратную матрицу.
6. Вычислить норму матрицы.
7. Повернуть матрицу на 90 (180, 270) градусов против часовой стрелки.
8. Вычислить определитель матрицы.
9. Построить матрицу, вычитая из элементов каждой строки матрицы ее среднее арифметическое.
10. Найти максимальный элемент(ы) в матрице и удалить из матрицы все строки и столбцы, его содержащие.
11. Уплотнить матрицу, удаляя из нее строки и столбцы, заполненные нулями.
12. В матрице найти минимальный элемент и переместить его на место заданного элемента путем перестановки строк и столбцов.
13. Преобразовать строки матрицы таким образом, чтобы элементы, равные нулю, располагались после всех остальных.
14. Округлить все элементы матрицы до целого числа.
15. Найти количество всех седловых точек матрицы. (Матрица **A** имеет седловую точку $\mathbf{A}_{i,j}$, если $\mathbf{A}_{i,j}$ 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 наибольший среди локальных максимумов. (Элемент матрицы называется локальным максимумом, если он строго больше всех своих соседей.)
19. Перестроить заданную матрицу, переставляя в ней столбцы так, чтобы значения их характеристик убывали. (Характеристикой столбца прямоугольной матрицы называется сумма модулей его элементов).
20. Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине – в позиции (2,2), следующий по величине – в позиции (3,3) и т. д., заполнив таким образом всю главную диагональ.

Тестовые задания к главе 2

Вопрос 2.1.

Какие из следующих строк скомпилируются без ошибки?

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

Вопрос 2.2.

Какие варианты записи оператора условного перехода корректны?

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

Вопрос 2.3.

Какие из следующих идентификаторов являются корректными?

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

Вопрос 2.4.

Какие из приведенных объявлений массивов корректны?

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

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