Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Ярославский государственный университет им. П. Г. Демидова

И. В. Парамонов

Язык программирования Java и Java-технологии

Учебное пособие

Ярославль 2006

УДК 004:43
ББК В185.2я73+3973.2–018.1
П 18

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2006 года

Рецензенты:
кандидат технических наук Г. П. Штерн;
кафедра прикладной математики и вычислительной
техники Ярославского государственного
технического университета

П 18 **Парамонов, И. В.** Язык программирования Java и Java-технологии: учеб. пособие / И. В. Парамонов; Яросл. гос. ун-т. — Ярославль: ЯрГУ, 2006. — 92 с.
ISBN 5–8397–0468–7

Учебное пособие содержит описание основных средств языка программирования Java и Java-технологий, а также некоторые общие сведения об объектно-ориентированном программировании и проектировании.

Предназначено для студентов IV курса факультета информатики и вычислительной техники ЯрГУ, обучающихся по специальности 010503 Математическое обеспечение и администрирование информационных систем (дисциплина «Язык программирования Java и Java-технологии», блок ДС), очной формы обучения.

Библиогр.: 11 назв.

ISBN 5–8397–0468–7

Оглавление

Введение

Учебное пособие предназначено для студентов IV курса факультета информатики и вычислительной техники ЯрГУ, обучающихся по специальности «Математическое обеспечение и администрирование информационных систем». В нём описываются средства языка программирования Java и некоторые связанные с ним технологии. Пособие содержит также некоторые общие сведения об объектно-ориентированном программировании и проектировании.

Первые две главы посвящены базовым средствам языка — типам данных, операциям и операторам. Значительная часть материала этих глав представлена в сравнении средств языка Java с аналогичными средствами языка C++.

В главах 3–5 излагаются принципы объектно-ориентированного программирования и описываются средства языка, реализующие эти принципы. Особое внимание уделено обработке ошибочных ситуаций с помощью исключений, а также некоторым вопросам проектирования классов.

Главы 6–8 описывают средства стандартной библиотеки Java. Рассмотрены вопросы обработки строковых данных, потокового ввода/вывода, а также использования контейнерных классов Java — чрезвычайно мощных и удобных средств хранения и обработки данных в приложениях.

Главы 9–11 представляют собой введение в некоторые из Java-технологий. Изложенный материал позволяет изучить идеи, лежащие в основе соответствующих технологий, а также освоить базовые средства этих технологий.

В конце пособия приведён список основной и дополнительной литературы по рассматриваемому предмету. В этом списке хотелось бы выделить книгу [1], содержащую очень подробное и глубокое изложение механизмов языка, а также книгу [2], которая может послужить справочником по основным средствам и особенно библиотекам языка. Хотелось бы также порекомендовать книгу [3], описывающую характерные ошибки проектирования Java-приложений, изучение которой может существенно повысить квалификацию программиста.

1. Типы данных, литералы, переменные

1.1. Примитивные типы данных. В Java определено 8 примитивных типов данных. К ним относятся четыре целочисленных **byte**, **short**, **int**, **long**, два вещественных **float** и **double**, символьный **char** и логический **boolean** типы данных. Их основные характеристики приведены в табл. 1.1. В отличие от C++ в Java нет типа данных «указатель».

Главная особенность примитивных типов данных в Java состоит в том, что переменные этих типов не являются объектами. Переменные всех остальных типов, включая массивы и переменные строкового типа **String**, являются ссылками на объекты (объектными ссылками). Подробнее они будут рассмотрены в п. 3.3.

1.2. Литералы. Целочисленные литералы могут задаваться в десятичном, восьмеричном и шестнадцатеричном виде. Литералы в десятичном виде записываются непосредственно, для остальных используются префиксы, аналогичные префиксам в языке C: «0» — для восьмеричных и «0x» — для шестнадцатеричных литералов. Например, целочисленный литерал 47 может быть представлен как 47, 057 или 0x2F. Все целочисленные литералы автоматически приводятся к минимально возможному типу, но не выше **int**. Литералы типа **long** должны иметь суффикс «L» или «l» (например, 47L).

Литералы с плавающей точкой могут представляться в обычном либо в научном формате. Литералы типа **float** должны иметь суффикс

Тип данных	Размер, байт	Допустимые значения
byte	1	$-128 \div 127$
short	2	$-32768 \div 32767$
int	4	$-2147483648 \div 2147483647$
long	8	$-2^{63} \div 2^{63}-1$
float	8	$\approx 1.4\cdot10^{-45} \div 3.4\cdot10^{38}$
double	16	$\approx 1.7\cdot10^{-324} \div 4.9\cdot10^{308}$
char	2	$0 \div 65535$
boolean	—	true, false

Таблица 1.1. Примитивные типы данных в Java

«F» или «f» (например, 3.5F), а литералы типа **double** — суффикс «D» или «d», который может опускаться (3.5D или просто 3.5).

Символьные литералы ограничиваются одинарными кавычками. Также возможно представление в шестнадцатеричном виде в Unicode. В последнем случае литерал записывается в виде последовательности четырёх шестнадцатеричных цифр с префиксом «\u»). Например, литерал 'ю' может быть представлен в виде '\u044E'.

Строковые литералы ограничиваются двойными кавычками ("abcd").

Определены два логических литерала **true** и **false**, *не равные* целым литералам 1 и 0 соответственно.

1.3. Переменные. Объявление переменных в Java аналогично объявлению переменных в C++. Любая переменная может быть инициализирована в момент описания любым допустимым выражением. Область действия и время жизни локальной переменной ограничивается блоком, в котором эта переменная описана.

В отличие от C++ не допускается объявление во внутреннем блоке переменной с тем же именем, что и переменная во внешнем блоке (кроме случая, когда переменная во внешнем блоке является аргументом метода).

Переменные, объявленные со спецификатором **final**, являются неизменяемыми (константами). Например:

```
final double PI = 3.141592653589793116;
```

1.4. Преобразования типов. Преобразования типов в Java делятся на неявные и явные. Неявные преобразования осуществляются компилятором автоматически. Правила автоматического приведения в Java являются значительно более жёсткими, чем в C++. В отличие от C++ неявные преобразования в Java осуществляются только в тех случаях, когда возможно гарантировать, что в результате преобразования не произойдёт потеря значимости. Например, следующая строка, являющаяся корректной с точки зрения языка C++, в Java вызовет ошибку компиляции:

```
int x = 2.5; // попытка неявного преобразования от double к int
```

Приведём более формальные правила неявного преобразования типов. Автоматическое преобразование типов при присваивании производится, если выполнены следующие условия:

- требуется приведение целочисленного типа или типа **char** к какому-либо числовому типу или вещественного к вещественному;

- тип, к которому требуется приведение, имеет больший размер, чем исходный.

Автоматическое преобразование типов в выражениях осуществляется

- для аргументов операции конкатенации строк (более подробно см. п. 6.3);
- если один из аргументов бинарной операции имеет тип **double**, второй приводится к типу **double**;
- иначе: если один из аргументов имеет тип **float**, второй приводится к типу **float**;
- иначе: если один из аргументов имеет тип **long**, второй приводится к типу **long**;
- иначе оба аргумента приводятся к типу **int**.

Здесь следует отметить последний пункт. В некоторых случаях его применение может приводить к не вполне естественным результатам. Например, следующий фрагмент является ошибочным с точки зрения языка:

```
short a = 5;
short b = a + 1; // ошибка: требуется short, а не int
```

Ошибка связана с тем, что при добавлении 1 к значению переменной **a** произошло преобразование результата к типу **int**. Так как преобразование от типа **int** к **short** может приводить к потере значимости, то оно должно поризводиться явно:

```
short b = (short)(a + 1);
```

Причиной такого «странного» поведения является тот факт, что типы данных **byte** и **short** создавались не для выполнения арифметических операций, а для обмена с внешними устройствами. Для выполнения арифметических операций следует использовать типы **int** и **long**.

Явное преобразование совместимых типов осуществляется по следующим правилам:

- сужение целочисленных типов производится путём усечения старших битов;
- усечение вещественных типов при присвоении их целым производится путём отбрасывания дробной части и применения предыдущего пункта.

Синтаксис явного преобразования типов совпадает с синтаксисом преобразования типов в C (см. пример выше).

Наконец, некоторые преобразования типов не являются допустимыми. В число таких преобразований входят: любые преобразования к типу **boolean**, преобразования классов к примитивным типам и примитивных типов к классам (с некоторыми исключениями), и т. д. Попытка выполнения таких преобразований будет приводить к ошибкам на этапе компиляции.

1.5. Массивы. Массивы в Java являются объектами, поэтому на них распространяются все правила работы с объектами (см. п. 3.3). Однако, поскольку массивы являются одними из наиболее широко применяемых структур данных, они будут рассмотрены здесь.

Для объявления одномерного массива используется синтаксис, аналогичный описанию переменных, но к имени переменной либо к имени типа данных добавляются пустые квадратные скобки. Например:

```
int a[]; // первый вариант
int[] b; // второй вариант
```

При этом сам массив *не создаётся*. Для создания массива следует использовать операцию **new**:

```
a = new int[50];
```

Возможно одновременное объявление и выделение памяти:

```
int a[] = new int[50];
```

Возможна также инициализация массива при его создании:

```
int a[] = { 20, 50, 166, 72, 0, –53 };
```

Для обращения к элементам массива используется операция []. Элементы массива нумеруются с нуля. Попытка обращения к несуществующему элементу массива приводит к ошибке времени выполнения. Массивы поддерживают получение информации о своих размерах посредством экземплярной переменной **length**.

Многомерные массивы представляют собой массивы массивов. Они могут создаваться аналогично одномерным:

```
int a[][] = new int[5][2];
```

Возможно создание непрямоугольных массивов. Способ работы с такими массивами иллюстрируется следующим примером.

```
// Пример: построение таблицы биномиальных коэффициентов
public class Binomial
```

```
{
    public static void main(String args[])
    {
        final int N = 10;
        int c[][] = new int[N][];
        // Формирование таблицы биномиальных коэффициентов
        for(int i = 0; i < N; ++i)
        {
            c[i] = new int[i + 1];
            c[i][0] = c[i][i] = 1;
            for(int j = 1; j < i; ++j)
                c[i][j] = c[i – 1][j – 1] + c[i – 1][j];
        }
        // Вывод таблицы
        for(int i = 0; i < c.length; ++i)
        {
            for(int j = 0; j < c[i].length; ++j)
                System.out.print("\t" + c[i][j]);
            System.out.println();
        }
    }
}
```

2. Операции и операторы

2.1. Общая характеристика операций. Большинство операций Java аналогичны операциям C++, хотя некоторые из них функционируют чуть иначе. Поскольку в Java нет указателей, естественным представляется отсутствие операций взятия адреса (**&**), разыменования (*) и косвенной адресации (–>). Операция **.** («точка») в Java используется для обращения к полям и методам объектов через объектные ссылки (более подробно см. п. 3.3).

В последующих пунктах описаны основные операции Java. Приоритеты операций приведены в табл. 2.1.

2.2. Арифметические операции. К арифметическим относятся следующие операции: унарные операции сохранения и изменения знака (+ и –); унарные операции инкремента и декремента (++ и ––); бинарные операции сложения, вычитания, умножения, деления и нахождения остатка от деления (+, –, *, /, **%**), а также операции с присваиванием (+=, –=, *=, /=, **%=**).

Арифметические операции применяются к числовым типам и имеют результат также числового типа. Операция деления для целочисленных типов даёт результат целочисленного типа (неполное частное), для вещественных — вещественного.

Если при выполнении операций в целочисленной арифметике происходит переполнение, то один или несколько старших битов результата могут быть усечены (при этом, поскольку старший бит отвечает за знак числа, возможно изменение знака) без генерации сообщения об ошибке. Единственной операцией в целочисленной арифметике, приводящей к ошибке, является операция деления на 0.

При выполнении операций в вещественной арифметике ошибка не возникает никогда: при переполнениях и делении на ноль результат получает одно из специальных значений NaN («не число»), Infinity («плюс бесконечность»), –Infinity («минус бесконечность»). При необходимости их можно получить, обратившись к константам **NaN**, **POSITIVE_INFINITY**, **NEGATIVE_INFINITY** соответственно в классах **Float** и **Double**. Например, результат вычисления выражения –

1.	()	[]	.	
2.	++	−−	~	!
3.	*	/	%	
4.	+	−		
5.	<<	>>	>>>	
6.	<	>	<=	>=
7.	==	!=		
8.	&			
9.	^			
10.	|			
11.	&&			
12.	||			
13.	?:			
14.	=	*operator=*		

Таблица 2.1. Приоритеты операций

3.0f/0.0f равен **Float.NEGATIVE_INFINITY**, а результат вычисления выражения 0.0/0.0 равен **Double.NaN**.

2.3. Операции сравнения. К операциям сравнения относятся операции **==**, **!=**, **<**, **>**, **<=**, **>=**. Единственным их отличием от соответствующих операций C++ является то, что их результат всегда имеет тип **boolean**.

2.4. Логические операции. Логические операции применяются к аргументам типа **boolean** и дают результат типа **boolean**. К логическим относятся: унарная операция логического «не» (**!**), бинарные операции логического «и» (**&&**, **&**), «или» (**|**, **||**) и «исключающего или» (**^**), а также аналогичные операции с присваиванием (**&=**, **|=**, **^=**).

Операции **&&** и **||** вычисляют значение своего второго аргумента только в том случае, если результат невозможно определить по значению первого. В отличие от них операции **&** и **|** всегда вычисляют значения обоих своих аргументов. Например:

```
int x = 10, y = 10;
if(x == 10 || ++y == 10)
    System.out.println(y);
```

печатает «10», так как второй аргумент операции **||** не вычисляется. Если заменить **||** на **|**, результатом работы будет «11».

2.5. Условная операция. В Java имеется аналогичная C++ условная операция **?:**. Её синтаксис:

```
условие ? выражение1 : выражение2
```

Если *условие* истинно, то результатом этой операции является *выражение1*, иначе — *выражение2*. При этом выражение, не являющееся результатом операции, не вычисляется. Существенно, что в Java условие должно иметь тип **boolean** (см. также п. 2.7).

2.6. Операция присваивания и оператор-выражение. Все операции языка Java (как и во всех других языках) имеют возвращаемое значение. Однако, кроме этого, они (как и в C, C++, но не в других языках) могут также изменять свои операнды. К таким операциям относятся операции ++, −−, = и все операции с присваиванием. По этой причине в языках C, C++ и Java нет *оператора присваивания*, но есть *операция присваивания* и *оператор-выражение*.

Операция присваивания — это бинарная операция, которая присваивает значение своего второго аргумента первому. Первый аргумент операции присваивания должен быть *lvalue*, т. е. «допускающим присваивание». Результат операции присваивания равен значению её второго аргумента. Например, следующая операция

```
x = y = 10
```

осуществляет присваивание переменной **y** значения 10, после чего результат этой операции (т. е. 10) присваивается переменной **x**.

Оператор-выражение имеет синтаксис:

```
выражение;
```

Таким образом, добавление точки с запятой к любому выражению превращает его в оператор, а возможность соединять в одном выражении несколько операций позволяет одним оператором изменить сразу несколько переменных, что невозможно при использовании оператора присваивания.

В отличие от C++, в Java есть дополнительное требование, чтобы последняя операция, входящая в оператор-выражение, изменяла свои операнды, т. е. требование отсутствия бесполезного кода (code having no effect) в программе. Например, оператор

```
x + 1;
```

является допустимым в C++ и недопустимым в Java, поскольку он не изменяет значения никаких переменных.

2.7. Операторы управления потоком. К операторам управления потоком относятся: условный оператор **if**, оператор выбора **case**, операторы циклов **while**, **do—while** и **for**, операторы досрочного выхода из цикла **break** и перехода к следующей итерации **continue**, а также оператор выхода из функции **return**. Синтаксис этих операторов аналогичен C++ с небольшими модификациями.

В частности, все операторы, выполняющие проверку условий, требуют, чтобы условие имело тип **boolean**. Это означает, что, в отличие от C++, конструкции **if**(n) и **if**(n != 0) не эквивалентны, и первая из них является синтаксически неверной, если переменная **n** имеет тип, отличный от **boolean**.

В операторе **for**, так же как и в C++, возможно объявление переменных. Например

```
for(int i = 0; i < N; ++i)
```

Объявленные таким образом переменные действительны лишь внутри цикла. Можно использовать несколько выражений в заголовке оператора **for**, разделяя их запятыми:

```
for(i = 0, j = 0; i + j < 20; ++i, ++j)
```

Управляющее выражение в операторе **case** должно быть одного из целочисленных типов.

Java не содержит оператора **goto**. Вместо него существуют расширения операторов **break** и **continue**, позволяющие осуществить выход из нескольких вложенных циклов. Они мало распространены и потому здесь не рассматриваются. Их описание можно найти, например, в книге [2].

Важно отметить, что наличие в программе недоступного кода (например, кода, следующего за оператором **return**) запрещено в Java и приводит к ошибке компиляции.

3. Классы и объекты

3.1. Основные понятия объектно-ориентированного программирования. Язык программирования Java поддерживает парадигму *объектно-ориентированного программирования* (ООП). Центральным понятием ООП является понятие класса. *Класс* — это тип данных, определяемый пользователем. Класс может содержать *поля* (переменные) и *методы* (функции). Переменные типа «класс» называются *объектами* (или *экземплярами класса*). Каждый объект имеет свой собственный набор экземпляров полей, определенных в классе. Содержимое полей в любой момент времени определяет *состояние* этого объекта. Методы существуют в одном экземпляре для каждого класса, однако вызов любого (нестатического) метода может осуществляться только с указанием объекта класса. Как правило, такой метод обрабатывает поля именно того экземпляра класса, на котором он вызывается (изменяет его состояние). Тем самым, методы класса определяют *поведение* всех объектов этого класса. Очень важно отметить, что *ни поля, ни методы класса не существуют сами по себе вне объектов этого класса.*

Понятие класса можно рассматривать как развитие понятия структуры языка C, при котором функции, обрабатывающие данные, размещённые в структурах, помещаются внутри этих структур. При этом средства объектно-ориентированных языков программирования позволяют запретить доступ к данным объектов извне и осуществлять обработку данных только посредством методов. В этом случае говорят, что *методы класса предоставляют интерфейс к его данным.*

Предыдущее изложение описывает понятия класса и объекта с технической стороны, с точки зрения средств языка программирования. Однако можно дать и другую интерпретацию. Характерной особенностью объектно-ориентированного программирования по сравнению с ранними парадигмами (структурное программирование) является то, что программист получает возможность не подбирать языковые средства для выражения механизмов обработки данных предметной области, а *моделировать* предметную область внутри программы. Объектам в программе в этом случае соответствуют (в большинстве случаев) объекты предметной области, а классам — универсальные понятия пред-

метной области, характеризующие свойства и поведение однотипных объектов. Такой подход к программированию, как правило, является более эффективным (особенно при разработке больших программных комплексов, а также при коллективной разработке), поскольку программист получает возможность описывать взаимодействие объектов на языке предметной области, а не на языке, отражающем внутренние детали работы вычислительной машины.

Отметим, что вопросы, связанные с принципами разработки объектно-ориентированных приложений, чрезвычайно обширны, однако знакомство с ними необходимо любому профессиональному программисту. Для дальнейшего изучения этих вопросов можно порекомендовать в первую очередь книгу [6], а также книги [1, 3, 5, 7, 8].

3.2. Объявление класса. Для объявления класса используется следующий синтаксис:

```
[ специф_доступа ] class имя_класса
{
  [ специф_доступа ] тип_данных имя_поля [ = начальн_знач ];
    . . .
  [ специф_доступа ] тип_данных имя_метода(список_аргументов)
    {
        тело_метода
    }
    . . .
}
```

Порядок объявления полей и методов в классе может быть произвольным. Возможные спецификаторы доступа описаны в п. 3.10. В отличие от C++ объявление класса и его реализация размещаются в одном и том же файле.

Конструктор имеет имя, совпадающее с именем класса, и не имеет возвращаемого значения. Если конструктор класса не объявлен явно, то создаётся пустой конструктор, не имеющий аргументов (конструктор по умолчанию).

Если в классе определено два метода с одним и тем же именем, то такие методы называются *перегруженными*. Выбор необходимого перегруженного метода осуществляется компилятором на основании количества и типов аргументов, передаваемых методу. Не допускается определение двух методов с одинаковыми сигнатурами (сигнатурой называется совокупность имени метода и типов его аргументов), так как в этом

случае компилятор не сможет определить, какой из методов следует вызывать.

Аргументы по умолчанию не поддерживаются. Если такая возможность требуется, то следует явно реализовать методы со всеми необходимыми наборами аргументов.

В Java не допускается наличие кода вне классов, т. е. нет нелокальных переменных, которые бы не являлись полями, и нет функций в смысле C++.

3.3. Создание объектов. В Java все объекты создаются в динамической памяти виртуальной машины. Переменных-объектов в том смысле, в каком они есть в C++, в Java не существует, а переменные, типом которых (по синтаксису) является некоторый класс, на самом деле являются не объектами, а *ссылками* на эти объекты (объектными ссылками). Например

```
MyClass myObject;
```

Здесь объявляется объектная ссылка типа **MyClass** с именем **myObject** и *не создаётся* сам объект (в C++ аналогом данного объявления было бы объявление указателя: MyClass* myObject). Собственно создание объекта осуществляется с помощью операции **new**:

```
myObject = new MyClass();
```

Здесь создаётся объект типа **MyClass** (при создании вызывается конструктор без аргументов), и ссылка на него записывается в переменную **myObject**. Возможно одновременное объявление объектной ссылки и её инициализация:

```
MyClass myObject = new MyClass();
```

3.4. Обращение к полям и методам. Когда объект создан, доступ к его полям и методам осуществляется посредством операции **.** (точка):

```
myObject.someMethod();
```

При обращении к полю или методу объекта из метода, вызванного на этом же объекте, имя объекта может опускаться либо заменяться ключевым словом **this**. Ключевое слово **this** используется также для вызова конструктора класса из другого конструктора (см. пример ниже). В последнем случае подобный вызов должен быть первым оператором конструктора.

3.5. Пример. В качестве примера рассмотрим класс **Box** («ящик»), хранящий размеры некоторого ящика (три целых числа). Класс содержит перегруженные конструкторы с различными аргументами, а также метод **volume()**, вычисляющий объём ящика. Класс **SimpleBoxDemo** демонстрирует создание и использование экземпляров класса **Box**.

В целях упрощения примера управление доступом (см. п. 3.10) не используется. На практике такой подход не должен использоваться никогда!

```
/** Класс "Ящик" */
class Box
{
    /** Размеры ящика */
    int width, height, depth;
    /** Конструктор по умолчанию */
    Box() { width = height = depth = 0; }
    /** Основной конструктор */
    Box(int width, int height, int depth)
    {
        this.width = width; this.height = height; this.depth = depth;
    }
    /** Конструктор для куба */
    Box(int size)
    {
        this(size, size, size); // width = height = depth = size;
    }
    /** Метод вычисления объема */
    int volume() { return width * height * depth; }
}
public class SimpleBoxDemo
{
    static void main(String args[])
    {
        Box b1 = new Box(5), b2 = new Box(5, 10, 20);
        b1.width = 10;
        System.out.println(b1.volume() + "\n" + b2.volume());
    }
}
```

3.6. Удаление объектов. Операция удаления объектов в Java не предусмотрена. Её заменяет механизм автоматической сборки мусора. Для обеспечения работы этого механизма JVM (виртуальная машина

Java) отслеживает количество объектных ссылок на каждый созданный объект и, как только на объект не остаётся ни одной ссылки, он становится кандидатом на удаление. Само удаление объектов осуществляется периодически и только в случае, когда объём памяти, требующей освобождения, становится существенным.

Автоматическая сборка мусора позволяет обходить значительное количество проблем, связанных с утечками памяти. К сожалению, в некоторых случаях такие проблемы всё же возникают и в Java-приложениях. Примеры можно найти в книге [3].

Деструкторы в Java не предусмотрены. Так как освобождение памяти осуществляет JVM, то необходимость в них возникает не слишком часто. Если они всё же необходимы, следует определять собственные методы и вызывать их явно. Примером здесь может служить метод **close()** поточных классов Java, осуществляющий закрытие потока ввода/вывода (см. п. 7.1).

3.7. Статические поля и методы. Статические поля и методы существуют в единственном экземпляре для каждого класса, т. е. являются разделяемыми для всех экземпляров этого класса.

Для объявления статических полей и методов используется спецификатор **static**:

```
static int static_field = 10;
```

Статические поля инициализируются в момент загрузки класса либо значением, указанным в объявлении (см. пример п. 3.5), либо в специальном **static**-блоке.

Статические методы не могут (без явного указания объекта) обращаться к нестатическим полям и нестатическим методам класса.

Вызов статических полей и методов может осуществляться без указания экземпляра класса. При таком обращении вместо имени экземпляра используется имя самого класса. Например:

```
double x = Math.sqrt(2.0);
```

Метод **main()**, который является точкой входа в Java-программу, всегда должен объявляться статическим (см. пример п. 3.5).

3.8. Передача аргументов методам и особенности использования объектных ссылок. Объектная ссылка ведёт себя как указатель в C, поэтому присвоение одной объектной ссылки другой *не приводит* к копированию объекта. Для выполнения копирования объектов в классе обычно реализуется конструктор копий и, возможно, метод **clone()**.

Передача примитивных типов методам осуществляется *по значению*. Это относится и к объектным ссылкам, поэтому сами объекты передаются *по ссылке* (аналогично копированию указателя в C).

Рассмотрим типичную ошибку при использовании объектных ссылок. Пусть имеется класс «точка плоскости» со следующей реализацией:

```
class Point
{
    public double x, y;
    public Point(double _x, double _y)
    {
        x = _x; y = _y;
    }
}
```

Пусть требуется реализовать метод, выполняющий обмен двух объектов класса **Point**. Поскольку объекты передаются по ссылке, естественной может показаться следующая реализация

```
static void swap(Point c1, Point c2)
{
    // Данная реализация неверна!
    Point s = c1;
    c1 = c2;
    c2 = s;
}
```

Однако данная реализация является неверной, поскольку она производит обмен *копий ссылок* на объекты, подлежащие обмену. Поскольку копии являются локальными переменными, то они уничтожаются при выходе из метода.

Верная реализация (реализованная в виде метода) должна обменивать *содержимое* объектов, а не ссылки на них:

```
static void swap(Point c1, Point c2)
{
    double s;
    s = c1.x; c1.x = c2.x; c2.x = s;
    s = c1.y; c1.y = c2.y; c2.y = s;
}
```

Наконец, отметим, что с точки зрения объектно-ориентированного программирования предпочтительным является решение данной за-

дачи, оформленное в виде метода *самого класса* **Point**. Пример такой реализации:

```
void swap(Point c)
{
    double s;
    s = x; x = c.x; c.x = s;
    s = y; y = c.y; c.y = s;
}
```

Здесь происходит обмен содержимого объекта, на котором вызван метод **swap()** с другим объектом, переданным этому методу в качестве аргумента. Отметим, что такая реализация не провоцирует программиста использовать описанный выше неверный подход.

3.9. Пакеты и структура модуля трансляции. Пакеты являются средством группировки типов (классов и интерфейсов) в Java-программе. Каждый пакет может содержать один или несколько классов и интерфейсов, а также других пакетов. Распределение классов по пакетам позволяет решать проблему конфликтов имён, а также более эффективно осуществлять управление доступом.

Для обращения к элементам пакета используется полный путь, состоящий из набора подпакетов, разделённых точками (например, **java.util.HashSet**). Для обращения к классам того же самого пакета можно использовать просто их имена, опуская имя пакета.

Модулем трансляции в Java является файл. Каждый модуль трансляции может начинаться утверждением **package**, определяющим пакет, которому принадлежат классы, описанные в данном модуле. Например,

```
package mypackage.mysubpackage;
```

Если **package**-утверждение отсутствует, то считается, что класс принадлежит пакету по умолчанию, который не имеет имени, не может иметь подпакетов и не является видимым из других пакетов.

Модуль трансляции может содержать одно или несколько **import**-утверждений, позволяющих опускать имена пакетов при обращении к классам. Например, если в модуле трансляции присутствует строка

```
import java.util.*;
```

то ко всем классам пакета **java.util** (но не к классам его подпакетов!) можно будет обращаться по коротким именам (**HashSet** вместо **java.util.HashSet** и т. д.). Возможно и импортирование единичного класса пакета, например:

```
import java.util.HashSet;
```

Модуль трансляции обязательно должен содержать исходный текст основного класса (или интерфейса) модуля. Имя этого класса должно совпадать с именем файла модуля трансляции. Этот класс является единственным классом модуля трансляции, который может быть объявлен открытым (**public**, см. п. 3.10). Модуль трансляции может содержать также один или несколько вспомогательных классов.

Поиск файлов, содержащих байт-код классов, осуществляется в подкаталогах, соответствующих местоположению классов в иерархии пакетов, внутри каталогов, перечисленных в переменной окружения **CLASSPATH**.

Например, для того чтобы JVM нашла байт-код класса **java.util.HashSet**, он должен находиться в файле с именем HashSet.class в подкаталоге java/util какого-либо каталога переменной окружения **CLASSPATH**. Как правило, туда включается текущий каталог, а также каталог, содержащий стандартные библиотеки Java. Файлы исходных текстов обычно также располагаются в файловой системе в соответствии с иерархией пакетов.

3.10. Управление доступом и инкапсуляция. Управление доступом используется для того, чтобы, с одной стороны, предотвратить несанкционированное изменение данных, содержащихся в объектах, в результате неправильного или злонамеренного использования, а с другой стороны, чтобы отделить интерфейс класса от его реализации (сокрытие реализации). Программист, использующий класс, видит только его интерфейс (поля и методы с открытым доступом) и может ничего не знать о внутреннем устройстве класса. Последнее позволяет изменять реализацию класса прозрачно для программиста. Например, можно заменить реализацию более эффективной, полностью перестроив внутреннее устройство класса, причём программный код, использующий класс, будет без изменений работать с новой версией класса.

Размещение полей и методов внутри классов совместно с сокрытием реализации называется *инкапсуляцией*. Инкапсуляция является характерной чертой объектно-ориентированного программирования.

Управление доступом осуществляется путём установки определённого спецификатора доступа в описании классов, полей и методов. В Java существует два спецификатора доступа для классов и четыре — для полей и методов. Все они вместе с областями видимости соответствующих элементов программы перечислены в табл. 3.1, 3.2. Уровни

Область видимости		private	по умолч.	protected	public
тот же пакет	тот же класс	+	+	+	+
	другой класс	–	+	+	+
другой пакет	субкласс	–	–	+	+
	не субкласс	–	–	–	+

Таблица 3.1. Уровни доступа полей и методов

Область видимости	по умолч.	public
тот же пакет	+	+
другой пакет	–	+

Таблица 3.2. Уровни доступа классов и интерфейсов

доступа обозначаются своими спецификаторами, кроме уровня доступа «по умолчанию», которому никакого специального спецификатора не соответствует.

Приведём некоторые рекомендации по использованию управления доступом в Java-программах.

Главное правило управления доступом: *при проектировании для всех элементов программы следует задавать максимально ограниченный уровень доступа, позволяющий этим элементам правильно функционировать*. Чем меньше классы знают друг о друге, тем больше шансов они имеют быть использованными повторно.

Для полей классов используется доступ **private**. Для чтения и изменения их значений извне используются специальные методы (setter'ы и getter'ы), имеющие спецификатор доступа **public**. Например,

```
class MyClass
{
    private int value;
    public void setValue(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

Данный подход является достаточно гибким. Например, всегда можно сделать некое поле полем только для чтения, удалив или просто не реализовав соответствующий setter, или ограничить возможные значения для поля, вставив соответствующую проверку.

Методы класса, имеющие спецификатор доступа **public**, образуют *интерфейс к данным* этого класса. Только через эти методы внешние

классы смогут обращаться к данным, инкапсулированным классом, и только они определяют возможные способы обработки этих данных.

При наличии наследования (см. п. 4.1) спецификаторы доступа элементов **private** и «по умолчанию» часто заменяются спецификатором **protected**, чтобы позволить наследникам класса иметь доступ к соответствующим элементам классов. Также возможно применение доступа по умолчанию, если вся иерархия наследников находится в том же пакете, что и наследуемый класс.

Доступ по умолчанию может использоваться для создания дружественных классов, которые могут обращаться к закрытым полям и методам друг друга. Такие дружественные классы размещаются в одном пакете. Возможности управления доступом в этом случае гибче, чем в случае дружественных классов в C++. Однако злоупотреблять этой возможностью не рекомендуется, поскольку дружественные классы оказываются сильнее связанными друг с другом, чем обычные, поэтому изменения в одном классе могут потребовать серьёзных изменений в других.

3.11. Пример. Рассмотрим в качестве примера класс «стек целых чисел» со следующей реализацией:

```
/** Класс "стек целых чисел" */
public class IntStack
{
    private final int MAXLEN = 64; // максимальный размер стека
    private int s[], size;
    /** Конструктор по умолчанию */
    public IntStack()
    {
        size = 0;
        s = new int[MAXLEN];
    };
    /** Конструктор копий */
    public IntStack(IntStack stack)
    {
        size = stack.size;
        s = new int[MAXLEN];
        for(int i = 0; i < size; ++i)
            s[i] = stack.s[i];
    }
    /** Получение размера стека */
    public int getSize() { return size; }
```

```
    /** Запись элемента в стек */
    public void push(int e) { s[size++] = e; }
    /** Получение элемента из стека */
    public int pop() { return s[--size]; }
    /** Проверка стека на пустоту */
    public boolean isEmpty() { return size == 0; }
}
```

Состояние стека определяется его текущим фактическим размером **size** и содержимым, хранящимся в массиве **s**. Для поля **size** определён метод **getSize()**, позволяющий получать текущий размер стека. Непосредственного доступа к массиву **s** нет. Обращение к нему может осуществляться только через методы **push()**, **pop()**, **isEmpty()**, которые в совокупности с методом **getSize()** и двумя конструкторами класса определяют интерфейс к данным стека. Отметим, что, для того чтобы использовать класс, необходимо иметь лишь сигнатуры этих методов и описание того, какие операции они осуществляют. Внутренние детали реализации при таком подходе скрыты и с легкостью могут быть изменены. Например, можно заменить массив **s** двусвязным списком, и это никак не отразится на классах, использующих класс **IntStack**.

3.12. Сравнение объектов. Стандартная операция Java == осуществляет *сравнение ссылок на объекты*. Однако, как правило, требуется сравнение объектов не по ссылкам, а *по их содержимому*. Для этого предназначен метод

```
boolean equals(Object obj)
```

определённый в классе **Object** (см. п. 4.1) и потому содержащийся во всех классах Java. Реализация этого метода по умолчанию функционирует так же, как и операция ==, т. е. сравнивает объектные ссылки. Программист должен переопределить этот метод так, чтобы он осуществлял сравнение требуемым образом. Например, для класса **Point** («точка плоскости»), содержащего две целочисленные координаты **x** и **y**, соответствующий метод можно реализовать следующим образом:

```
public boolean equals(Object obj)
{
    if(!(obj instanceof Point))
        return false;
    Point c = (Point)obj;
    return x == c.x && y == c.y;
}
```

Если в программе используются стандартные классы-контейнеры (см. п. 8.1), обычно требуется переопределить не только метод **equals()**, то также метод

```
int hashCode()
```

который возвращает хэш-код объекта. Хэш-код объекта — это целое число, которое вычисляется на основании состояния объекта (содержимого его полей). Равным объектам (т. е. объектам, для которых метод **equals()** возвращает значение **true**) должен соответствовать один и тот же хэш-код. Для неравных объектов хэш-коды не обязаны быть неравными, однако к этому следует стремиться. Кроме того, следует учитывать, что чем более равномерным является распределение хэш-кодов объектов, тем выше будет производительность встроенных коллекций, их использующих (множеств и ассоциативных массивов).

Для рассмотренного выше класса **Point** можно определить метод **hashCode()** следующим образом:

```
int hashCode()
{
    return x + y;
}
```

3.13. Обёртки примитивных типов. Как уже отмечалось, переменные примитивных типов в Java не являются объектами. В некоторых случаях, однако, требуется их представление как объектов (например, для хранения в контейнерах, см. п. 8.1). Для этого используются так называемые *обёртки примитивных типов* — классы, инкапсулирующие примитивные типы.

Для каждого из восьми примитивных типов определён соответствующий ему класс-обёртка. При создании экземпляров таких типов обёртываемый объект передаётся в конструктор соответствующего класса:

```
Byte(byte value)
Short(short value)
Integer(int value)
Long(long value)
Float(float value)
Double(double value)
Character(char value)
Boolean(boolean value)
```

Кроме того, все классы-обёртки, кроме **Character**, имеют конструкторы с аргументом типа **String**. Они осуществляют преобразование строки к соответствующему примитивному типу и обёртывают полученный результат.

Все классы-обёртки переопределяют методы **equals()**, **toString()** (см. п. 3.12), а также реализуют интерфейс **Comparable<T>** (см. п. 8.4).

Для выполнения обратного преобразования от объектов-обёрток к примитивным типам используются методы

```
byte byteValue()
short shortValue()
int intValue()
. . .
```

вызываемые на соответствующих объектах. Начиная с версии Java 1.5, большая часть таких преобразований производится автоматически. Все классы-обёртки являются неизменяемыми (immutable).

Классы-обёртки содержат также статические методы

```
static byte parseByte(String s)
static short parseShort(String s)
static int parseInt(String s)
. . .
```

которые могут использоваться для преобразования строки к примитивным типам. В случае, если преобразование осуществить не удаётся, выбрасывается исключение типа **NumberFormatException**.

4. Наследование и полиморфизм

4.1. Наследование. *Наследование* — это механизм построения новых классов на основе уже существующих. При этом наследники класса (*субклассы*) получают свойства и функциональность класса-родителя (*суперкласса*) и имеют возможность изменять и расширять их. Механизм наследования используется для создания более частных, специализированных классов по сравнению с суперклассом.

Для того чтобы объявить класс наследником некоторого другого класса, используется ключевое слово **extends** с последующим именем наследуемого класса, добавляемое в объявление класса:

[*специф_доступа*] **class** *имя_субкласса* **extends** *имя_суперкласса*

В Java допускается наследование только от одного класса. Субкласс наследует все поля и методы суперкласса, однако те из них, которые объявлены со спецификаторами доступа **private** и по умолчанию (последние, только если субкласс находится вне пакета, содержащего суперкласс), оказываются недоступными для субкласса.

Субкласс может переопределять поля и методы суперкласса. В этом случае доступ к родительским полям и методам закрывается и может осуществляться только из методов субкласса посредством ключевого слова **super**, синтаксис которого совпадает с синтаксисом явного обращения к элементам текущего экземпляра через **this**. Например, следующий фрагмент кода показывает вызов метода **overridenMethod** суперкласса из одноимённого метода субкласса.

```
public void overridenMethod()
{
    super.overridenMethod();
    // . . .
}
```

При переопределении полей и методов возможно изменение спецификатора доступа на более широкий (**private** на любой другой; по умолчанию на любой другой, кроме **private** и т. д.).

Конструкторы не наследуются, однако конструкторы субклассов обязательно вызывают конструкторы своих суперклассов. Последнее мо-

жет происходить явно (для этого первым оператором конструктора должен быть вызов конструктора суперкласса через **super**) или неявно (в этом случае происходит вызов конструктора по умолчанию). Вызов конструкторов родительских классов происходит сверху вниз по дереву наследования.

В следующем примере на основе класса «ящик» создаётся более специализированный класс «ящик с весом», который добавляет дополнительное поле «вес» и метод для его получения, а также использует вызовы конструкторов суперкласса.

```
/** Класс "Ящик" */
class Box
{
    /** Размеры ящика */
    protected int width, height, depth;
    /** Getter'ы */
    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public int getDepth() { return depth; }
    /** Основной конструктор */
    public Box(int w, int h, int d) { width = w; height = h; depth = d; }
    /** Конструктор для куба */
    public Box(int size) { this(size, size, size); }
    /** Метод вычисления объема */
    public int volume() { return width * height * depth; }
}
/** Класс "Ящик с весом" */
class BoxWeight extends Box
{
    /** Вес ящика */
    protected int weight;
    /** Getter для weight */
    public int getWeight() { return weight; }
    /** Основной конструктор */
    public BoxWeight(int w, int h, int d, int wg)
    {
        super(w, h, d);
        weight = wg;
    }
    /** Конструктор для куба */
    public BoxWeight(int size, int wg)
    {
```

```
            super(size);
            weight = wg;
        }
    }
```

Если в описании класса суперкласс для него не определён, то по умолчанию производится наследование от библиотечного класса **Object**. Этот класс представляет собой корень иерархии наследования в Java и содержит методы, необходимые для корректного функционирования JVM, а также методы, выполняющие такие распространённые операции, как преобразование к строке (п. 6.3), сравнение объектов на равенство (п. 3.12) и т. д.

4.2. Ограничение и форсирование наследования. Если метод объявлен со спецификатором **final**, то его переопределение запрещается. Если же со спецификатором **final** объявлен класс, то запрещается наследование от такого класса.

Отдельные методы класса могут быть объявлены абстрактными. В этом случае они объявляются со спецификатором **abstract** и не имеют тела:

```
abstract void abstractMethod();
```

Такие методы должны обязательно переопределяться в субклассах. Если класс содержит хотя бы один абстрактный метод, то он также должен быть объявлен абстрактным. В этом случае запрещается создание объектов данного класса. Чтобы иметь возможность создавать экземпляры субклассов такого класса, следует реализовать в них все абстрактные методы суперкласса. Если субкласс реализует не все абстрактные методы суперкласса, он также должен объявляться абстрактным.

4.3. Полиморфизм. *Полиморфизм* — это механизм, который позволяет обращаться к объектам, унаследованным от одного класса, как к объектам этого класса. При таком обращении к объектам субклассов они реагируют способом, определённым в соответствующем субклассе, а не в родительском классе. В этом случае методы, определённые в суперклассе, образуют *единый интерфейс доступа к данным всех субклассов*. Полиморфизм является одним из самых мощных средств объектно-ориентированного программирования, поскольку он позволяет единообразно обрабатывать наборы объектов, принадлежащих различным классам, причём выбор реализации той или иной операции осуществляется автоматически на основании типа данных объекта.

Реализация полиморфизма в Java основана на следующих правилах.

1. Ссылка на объект класса может быть приведена к ссылке на объект любого из классов, находящихся выше данного в иерархии наследования. Это преобразование является расширяющим и может производиться автоматически. Обратное преобразование также может быть произведено, но оно выполняется всегда явно. При попытке привести объектную ссылку к недопустимому типу возникает ошибка. Для проверки допустимости преобразования может быть использована операция **instanceof**:

```
if(myObject instanceof MyClass)
    ((MyClass)myObject).doSomethingSpecificForMyClass();
```

2. Если некоторые из методов суперкласса были переопределены в субклассе, то *при обращении к этим методам даже посредством ссылки на объект суперкласса будет происходить вызов именно переопределённых методов*. Это отличается от реализации полиморфизма в C++, где обращение к переопределённым методам в описанной ситуации будет происходить лишь в том случае, когда соответствующие методы были объявлены виртуальными.

Рассмотрим пример. Пусть есть базовый класс «геометрическая фигура». Он содержит метод **area()**, предназначенный для вычисления площади фигуры. Каждый из субклассов класса «фигура» переопределяет этот метод таким образом, чтобы вычислялась площадь соответствующей фигуры.

Ссылки на создаваемые экземпляры классов конкретных фигур сохраняются в массиве ссылок, имеющих тип «фигура», а затем единообразно обрабатываются: для каждой фигуры выводится её площадь. При этом никаких проверок типов фигур не требуется, для каждой фигуры автоматически вызывается нужный метод.

```
/** Абстрактный класс "Геометрическая фигура" */
abstract class Figure
{
    /** Метод вычисления площади фигуры */
    abstract public double area();
}
/** Класс "Прямоугольник" */
class Rectangle extends Figure
{
    private double a, b;
```

```
    public Rectangle(double a, double b) { this.a = a; this.b = b; }
    public double area() { return a * b; }
}
/** Класс "круг" */
class Circle extends Figure
{
    private double r;
    public Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
}
/** Класс "треугольник" */
class Triangle extends Figure
{
    private double a, b, c;
    public Triangle(double a, double b, double c)
    {
        this.a = a; this.b = b; this.c = c;
    }
    public double area()
    {
        double p = (a + b + c) / 2;
        return Math.sqrt(p * (p – a) * (p – b) * (p – c));
    }
}
public class FiguresPolymorphism
{
    public static void main(String args[])
    {
        Figure f[] = new Figure[3]; // массив объектных ссылок
        f[0] = new Rectangle(2, 3);
        f[1] = new Triangle(3, 4, 5);
        f[2] = new Circle(1);
        for(int i = 0; i < f.length; ++i)
            System.out.println(f[i].area());
    }
}
```

4.4. Интерфейсы. В Java не допускается наследование одного класса от нескольких. Однако существует ограниченный аналог множественного наследования в виде *множественной реализации одним классом нескольких интерфейсов*. Функционально интерфейсы близки к абстрактным классам с определёнными ограничениями (допускаются

только абстрактные методы и только статические поля), однако любой класс может *реализовывать* (аналог наследования) несколько интерфейсов.

Для объявления интерфейса используется следующий синтаксис:

interface *имя_интерфейса* [**extends** *список_имён_интерфейсов_предков*]

Все методы интерфейса неявно получают спецификатор **abstract public**, а все переменные интерфейса — спецификатор **final static public** и должны быть сразу же инициализированы. Использование других спецификаторов доступа в интерфейсах не допускается.

Любой класс может реализовать любое количество интерфейсов. Для объявления класса, реализующего один или несколько интерфейсов, используется следующий синтаксис:

[*специф_доступа*] **class** *имя_класса* [**extends** *имя_суперкласса*]
    implements список_имен_интерфейсов

Элементы списка имён интерфейсов разделяются запятыми. Если класс объявлен как реализующий некоторые интерфейсы, он должен либо реализовать все методы, объявленные в этих интерфейсах, либо быть абстрактным.

4.5. Интерфейсные ссылки. Подобно возможности приведения ссылки на объект класса к ссылке на объект его суперкласса, существует возможность приведения такой ссылки к ссылке на любой из интерфейсов, которые реализует класс. Такие ссылки ведут себя абсолютно аналогично ссылкам на объекты абстрактных классов. Например:

```
interface Ifc1
{
    void f();
}
interface Ifc2
{
    void g();
}
class MyClass implements Ifc1
{
    public void f() { };
    public void g() { };
}
public class InterfRef
```

```
{
    public static void main(String args[])
    {
        MyClass m = new MyClass();
        Ifc1 ir = m; // приведение к интерфейсной ссылке
        ir.f(); // вызов метода через интерфейсную ссылку
        // ir.g(); // недопустимо! интерфейс Ifc1 не содержит метода g()
        // ((Ifc2)m).g(); // недопустимо! MyClass не реализует Ifc2
        ((MyClass)ir).g(); // обратное преобразование к ссылке на MyClass
    }
}
```

4.6. Рекомендации по использованию наследования и полиморфизма. При разработке иерархии классов следует строить дерево наследования в соответствии с принципом: *субкласс является более частным, специализированным классом по отношению к суперклассу*. Несоблюдение этого принципа приводит к серьёзным ошибкам проектирования, проявляющимся в сильной связности классов, неочевидности взаимоотношений между ними и затруднении проектирования на последующих этапах.

Несмотря на очевидное сходство абстрактных классов и интерфейсов, их назначение, вообще говоря, является несколько различным. Абстрактный класс представляет собой абстрактную сущность, сущность, детали которой не до конца описаны в силу её общности. В отличие от класса интерфейс — это не сущность, а, скорее, набор потенциальных возможностей, которые могут быть реализованы различными классами, не имеющими между собой ничего общего кроме этих возможностей.

Наследование поддерживает полиморфизм «генетически» общих методов. Полиморфизм же методов, имеющий другую природу, должен быть реализован посредством интерфейсов. Примером может служить интерфейс **Comparable<T>** (см. п. 8.4). Этот интерфейс может быть реализован любым классом, допускающим упорядочение своих экземпляров. В то же время предположение о том, что все объекты, допускающие сравнение, должны быть субклассами одного класса выглядит несостоятельным.

5. Обработка исключений

5.1. Концепция исключений. Исключения представляют собой объектно-ориентированный механизм обработки исключительных ситуаций при выполнении программы. Этот механизм позволяет обрабатывать такие ситуации более простым и концептуально ясным способом по сравнению с традиционными способами обработки ошибок, такими как возврат специальных значений или использование функций **setjmp()** и **longjmp()** в C.

Исключение — объект, который *выбрасывается* программой при возникновении исключительной ситуации. Обычно он инкапсулирует описание этой ситуации. Выброшенное исключение *ловит* некоторый обработчик — программный блок, осуществляющий обработку исключительной ситуации. После выполнения обработчика управление передаётся на оператор, непосредственно следующий за обработчиком. Таким образом осуществляется переход из одной части программы в другую с выполнением некоторых специальных действий, заключённых в обработчике исключительной ситуации.

5.2. Выбрасывание и обработка исключений. Для того чтобы выбросить исключение, используется следующий синтаксис:

```
throw объект–исключение;
```

Обычно объект-исключение создаётся непосредственно в операторе **throw**, например:

```
throw new NullPointerException();
```

Оператор, выбрасывающий исключение, должен находиться внутри специального блока **try**, непосредственно за которым располагаются обработчики исключений:

```
try
{
    операторы_выбрасывающие_исключения
}
catch(имя_класса_исключения имя_переменной)
{
```

```
        тело_обработчика_исключения
    }
    . . .
[ finally
    {
        операторы
    } ]
```

Если некоторый оператор блока **try** выбрасывает исключение, то все дальнейшие операторы этого блока не выполняются и начинается поиск обработчика, в заголовке которого указан тип, совпадающий с типом выброшенного объекта-исключения, либо тип, наследником которого (не обязательно непосредственным) он является. Поиск осуществляется по следующим правилам:

- если оператор, выбросивший исключение, не находится в **try**-блоке, то происходит выход из метода и продолжение поиска обработчика в вызывающем методе (это может повторяться несколько раз, пока один из операторов, вызывающих метод, не окажется в блоке **try**);
- обработчик ищется среди **catch**-блоков, непосредственно следующих за данным **try**-блоком;
- если подходящего обработчика не находится, поиск продолжается среди **catch**-блоков, относящихся к включающему **try**-блоку и т. д. (при этом также может происходить выход из одного или нескольких методов);
- если подходящего обработчика не находится вообще, исключение ловит обработчик по умолчанию, который печатает описание исключения, трассу стека и завершает работу программы.

Если подходящий обработчик найден, то он выполнятся, причём объект-исключение передаётся обработчику в качестве аргумента. По окончании выполнения обработчика управление передаётся на оператор, непосредственно следующий за всеми блоками **catch**, находящимися после блока, обработавшего исключение.

Обработчики исключений-субклассов должны находиться до обработчиков исключений-суперклассов, так как иначе все такие исключения будут обрабатываться обработчиком исключений-суперклассов и возникнет недоступный код.

При наличии блока **finally** операторы этого блока выполняются независимо от того, было ли исключение выброшено или нет:

- если исключение произошло и было обработано, блок **finally** выполняется после обработчика исключений;
- если обработчик не был найден, то блок **finally** выполняется перед поиском обработчика во включающем блоке;
- если выход из **try**-блока был выполнен посредством оператора **return**, то блок **finally** выполняется перед выходом из метода.

5.3. Пример. Рассмотрим пример, являющийся улучшением примера из п. 3.11. Данная реализация класса «стек целых чисел» выполняет проверку правильности действий, выполняемых со стеком, и выбрасывает исключения в случае ошибок.

```
/** Класс "стек целых чисел" */
class IntStack
{
    private final int MAXLEN = 64; // максимальный размер стека
    private int s[] = new int[MAXLEN], size = 0;
    /** Получение размера стека */
    public int getSize() { return size; }
    /** Запись элемента в стек */
    public void push(int e)
    {
        if(size < MAXLEN)
            s[size++] = e;
        else
            throw new RuntimeException("Переполнение стека");
    }
    /** Получение элемента из стека */
    public int pop()
    {
        if(size != 0) return s[--size];
        throw new RuntimeException("В стеке нет элементов");
    }
    /** Проверка стека на пустоту */
    public boolean isEmpty() { return size == 0; }
}
public class IntStackExample
{
    /** Снятие n элементов с вершины стека и вывод их на экран */
    static void popAndPrint(IntStack st, int n)
    {
        for(int i = 0; i < n; ++i)
```

```
                System.out.println(st.pop());
        }
        /** Точка входа в программу */
        public static void main(String args[])
        {
            IntStack st = new IntStack();
            try
            {
                st.push(3); st.push(5); st.push(7);
                popAndPrint(st, 4);
            }
            catch(RuntimeException e)
            {
                System.out.println("Выброшено исключение: " + e);
            }
        }
}
```

Метод **main()** класса **IntStackDemo** создаёт экземпляр класса **IntStack** и последовательно помещает в него три элемента, а затем вызывает метод **popAndPrint()**, который пытается снять со стека четыре элемента. При попытке снять четвёртый элемент в методе **pop()** класса **IntStack** выбрасывается исключение. Поскольку этот метод не содержит блока **try**, то происходит выход в метод **popAndPrint()**. Этот метод также не содержит блока **try**, поэтому происходит выход и из этого метода, после чего оператор **catch()** метода **main()** обрабатывает исключение. Вывод этой программы:

```
7
5
3
Выброшено исключение: java.lang.RuntimeException: В стеке нет элементов
```

Отметим, что такое «пробрасывание» исключений через несколько методов является типичным при их обработке — исключение обрабатывается в том контексте, где это становится возможным, при этом стек вызовов «разворачивается» без участия программиста. Эта особенность во многом и обусловливает большую практическую ценность исключений.

5.4. Иерархия классов исключений и выбрасывание исключений из методов. Все объекты-исключения в Java должны принадлежать

классам, унаследованным от класса **Throwable**, имеющего два субкласса: **Exception** и **Error**. Исключения типа **Error** выбрасываются JVM в случае возникновения серьёзных ошибок. Как правило, программист не должен ни выбрасывать, ни обрабатывать такие исключения. Все остальные исключения являются экземплярами класса **Exception** и его субклассов. Кроме того, имеется особая группа исключений, представленная классом **RuntimeException** и его субклассами. Поскольку класс **Exception** является суперклассом всех пользовательских исключений, для того чтобы обработать все выброшенные исключения независимо от их типов, можно использовать конструкцию:

```
catch(Exception e)
```

Существует важная особенность, касающаяся исключений, являющихся экземплярами класса **Exception** и его субклассов, кроме класса **RuntimeException** и его субклассов. Если такое исключение выбрасывается из метода, то тип его должен быть обязательно указан в утверждении **throws** в заголовке этого метода

```
[ специиф_доступа ] тип_данных имя_метода(список_аргументов)
[ throws список_имён_классов_исключений_выбрасываемых_методом ]
```

а также всех методов, которые вызывают такие методы, но не обрабатывают соответствующие исключения. В противном случае компилятор выдаст сообщение об ошибке. Это сделано для того, чтобы привлечь внимание программиста к исключительным ситуациям, которые требуют обязательной обработки. В стандартной библиотеке к таким исключениям относятся: ошибки ввода/вывода (**IOException**), ошибки обращения к базе данных (**SQLException**) и некоторые другие. При выбрасывании исключений, являющихся экземплярами класса **RuntimeException** и его субклассов, никакой специальной декларации не требуется. В библиотеке к таким исключениям относятся: исключение при выполнении недопустимых математических операций (**ArithmeticException**), попытка обращения к несуществующему элементу массива (**IndexOutOfBoundsException**), к полю или методу несуществующего объекта (**NullPointerException**) и другие. Такие ошибки никогда не должны возникать в правильной программе, в то же время требование обязательной их обработки привело бы к загромождению исходного текста (например, к необходимости размещения каждой операции деления внутри блока **try** во избежание ошибки деления на ноль).

6. Обработка строк

6.1. Класс String. Класс **String** является основным классом, предназначенным для хранения и обработки строк символов.

Для создания экземпляров класса **String** может быть использован один из следующих конструкторов:

```
String()
String(String str)
String(StringBuffer strbuf)
String(char[] arr)
String(char[] arr, int first, int count)
```

Первый из них создаёт пустую строку, второй и третий копируют содержимое объектов классов **String** и **StringBuffer** в созданный объект. Последние два конструктора позволяют создать строку на основе символьного массива или его части. Кроме того, любая объектная ссылка типа **String** может быть проинициализирована посредством присвоения ей строкового литерала:

```
String filename = "data.txt";
```

Возможно, несколько неожиданной особенностью класса **String** является то, что экземпляры этого класса *не могут быть изменены после их создания* (immutable). Однако это не создаёт ограничений для их использования, поскольку все методы, которые должны были бы изменять строку, просто создают новую модифицированную строку, оставляя исходную без изменений. Поясним работу этого механизма на примере:

```
String s = "abcd";
s = s.toUpperCase();
```

Здесь метод **toUpperCase()** создаёт новую строку, содержащую последовательность символов "ABCD", и возвращает ссылку на эту строку, которая присваивается переменной **s**, старое значение переменной теряется. Исходная строка остаётся в неизменном виде и, поскольку на неё больше не осталось объектных ссылок, будет удалена сборщиком мусора.

Основные методы класса **String** приведены в табл. 6.1.

int length()	Получение длины строки
char charAt(**int** index)	Извлечение символа
char[] toCharArray()	Получение строки в виде символьного массива
boolean equals(String str)	Сравнение строк на равенство
boolean equalsIgnoreCase(String str)	Сравнение строк без учета регистра
int compareTo(String str)	Лексикографическое сравнение строк
int compareToIgnoreCase(String str)	Лексикографическое сравнение строк без учета регистра
boolean startsWith(String prefix)	Проверка, начинается ли строка с заданной подстроки
boolean endsWith(String suffix)	Проверка, заканчивается ли строка заданной подстрокой
int indexOf(String subStr)	Поиск первого вхождения подстроки в строке с начала строки/с заданной позиции
int indexOf(String subStr, **int** fromIndex)	
int lastIndexOf(String subStr)	Поиск последнего вхождения подстроки в строке с начала строки/с заданной позиции
int lastIndexOf(String subStr, **int** fromIndex)	
String substring(**int** beginIndex, **int** endIndex)	Получение подстроки (символ endIndex не входит в подстроку!)
String substring(**int** beginIndex)	Получение хвоста строки
String concat(String str)	Конкатенация строк
String toUpperCase()	Преобразование строки к верхнему/нижнему регистру
String toLowerCase()	
String trim()	Удаление ведущих и завершающих пробелов в строке
String replace(String target, String replacement)	Замена подстроки другой строкой
boolean matches(String regex)	Проверка строки на соответствие регулярному выражению
String replaceFirst(String regex, String replacement)	Замена первой подстроки/всех подстрок, соответствующих регулярному выражению, заданной подстрокой
String replaceAll(String regex, String replacement)	
String[] split(String regex)	Разбиение строки на подстроки (разделители задаются регулярным выражением)

Таблица 6.1. Основные методы класса **String**

Метасимвол(ы)	Значение
\	обозначение специального символа или экранирование обычного
.	любой символ, кроме ограничителей строки («\n», «\r» и пр.)
|	выбор одного из двух регулярных выражений
[ ]	один из символов, входящих в заданный набор
[^]	один из символов, не входящих в заданный набор
[–]	один из символов, входящих в диапазон
^	признак начала строки
$	признак конца строки
*	ноль и более предыдущих символов
+	один и более предыдущих символов
?	ноль или один предыдущий символ
{*k*}	ровно *k* предыдущих символов
{*k*,}	не менее *k* предыдущих символов
{*k*,*m*}	не менее *k* и не более *m* предыдущих символов
( )	группировка/сохранение строк, соответствующих регулярным выражениям
k	*k*-я сохранённая группа

Таблица 6.2. Метасимволы регулярных выражений

6.2. Регулярные выражения. Регулярные выражения — это выражения, описывающие структуру текстовой строки с использованием обычных символов и *метасимволов*, определяющих свойства строки в целом или её отдельных частей. Средства стандартной библиотеки Java позволяют выполнять проверку строк на соответствие заданному регулярному выражению, а также осуществлять замену подстрок, удовлетворяющих регулярным выражениям, другими подстроками.

Основные метасимволы, используемые в регулярных выражениях в Java, приведены в табл. 6.2. Примеры регуляных выражений приведены в табл. 6.3.

Для проверки строки на соответствие заданному регулярному выражению используется метод

```
boolean matches(String regex)
```

класса **String**. Заметим, что поскольку он проверяет *всю строку* на соответствие регулярному выражению, то использование символов «^» и «$» в этом случае излишне.

Регулярное выражение	Значение
abc|def	последовательность символов «abc» или «def»
a.c	символы «a» и «c», разделённые любым символом
https?://	последовательность символов «http://» или «https://».
^a.*z$	строка, начинающаяся с символа «a» и заканчивающаяся символом «z»
[A–Z]+	непустая последовательность прописных латинских букв
[^0–9]{5,}	последовательность не менее чем из пяти символов, не являющихся цифрами
\.\.\.	последовательность символов «…»
(.+)a\1	две одинаковые подстроки, разделённые символом «a»

Таблица 6.3. Примеры регулярных выражений

Следующие методы заменяют соответственно первое и все вхождения *подстроки*, соответствующей регулярному выражению:

```
String replaceFirst(String regex, String replacement)
String replaceAll(String regex, String replacement)
```

Например, следующий оператор заменяет многоточие, находящееся в конце строки, вопросительным знаком:

```
someString.replaceFirst("\\.\\.\\.$", "?");
```

Поскольку символ «.» является метасимволом, то для того, чтобы использовать его как обычный символ, его следует экранировать символом «\». Однако последний символ используется в Java для обозначения специальных символов (таких, как «\n»), поэтому также должен экранироваться: «\\.».

В замещающей строке можно использовать метасимвол «$» с последующим номером, обозначающий соответствующую группу в заменяемой строке. Например, следующий оператор заменяет угловые скобки в строке квадратными, при этом содержимое скобок остаётся прежним:

```
someString.replaceFirst("<(.*)>", "[$1]");
```

Следует учитывать, что если в строке содержится несколько открывающих и закрывающих угловых скобок, то заменены будут первая открывающая и последняя закрывающая, поскольку шаблону «.*» соответ-

ствует любое количество любых символов, в том числе все вложенные скобки.

Метод

```
String[] split(String regex)
```

позволяет разбить строку на подстроки, используя для описания разделителей регулярные выражения. Результат записывается в строковой массив. Например, оператор

```
String[] m = "abc:::de:f::gh".split(":+");
```

заполнит массив m строками «abc», «de», «f» и «gh».

В заключение отметим, что регулярные выражения, по-видимому, являются самым мощным инструментом обработки строковых данных, и их возможности значительно шире, чем это удалось показать на приведённых примерах. Более полную информацию о регулярных выражениях можно получить в книге [10].

6.3. Преобразование к строке и операция конкатенации. Класс **String** является в некотором смысле исключительным классом в Java, поскольку любой тип данных может быть преобразован к нему. Для примитивных типов такое преобразование даёт их естественное строковое представление, для объектов вызывается метод **toString()**, определённый в классе **Object** и, следовательно, присутствующий в любом классе Java.

Для строк определена операция конкатенации, обозначаемая знаком +. Это бинарная операция, один из аргументов которой должен иметь тип **String**. Она осуществляет автоматическое преобразование другого аргумента к типу **String** (если это необходимо) и слияние полученных строк. Заметим, что это единственный случай, когда преобразование к строке осуществляется неявно. Существует также операция конкатенации с присваиванием +=, первый аргумент которой должен иметь тип **String** (и обязательно быть *lvalue*), а второй может быть произвольным. При выполнении операции он будет преобразован к типу **String**.

Приведём пример использования преобразования к строке и операции конкатенации. Для класса «точка плоскости» можно определить операцию преобразования к строке следующим образом:

```
class Point
{
    private double x, y;
```

```
    public String toString()
    {
        return "(" + x + ", " + y + ")";
    }
    public Point(double _x, double _y)
    {
        x = _x; y = _y;
    }
}
```

Здесь автоматическое преобразование величин типа **double** к строкам осуществляется в выражении оператора **return**. Определённый выше метод **toString()** может быть использован следующим образом:

```
Point p = new Point(2.0, 5.0);
System.out.println(p); // вывод на экран: (2.0, 5.0)
```

Для того чтобы получить и вывести на экран строковое представление объекта, метод **println()** вызывает метод **toString()** класса **Point**.

Иногда требуется управлять видом строкового представления. Для этого может быть использован статический метод **format()** класса **String**:

```
static String format(String format, Object. . . args)
```

Он принимает форматную строку и переменное количество выражений и преобразует их в строку в формате, определённом форматной строкой. Правила написания форматной строки в целом соответствуют правилам написания форматной строки функции **printf()** в языке C. Например,

```
System.out.println(String.format("%04d%4.1f", 25, 3.58));
```

выведет «0025 3.6».

6.4. Класс StringBuffer. Класс **StringBuffer** предназначен для работы с модифицируемыми строками. Каждый объект этого класса содержит некоторый буфер, хранящий строку. При изменениях строки размер буфера может автоматически изменяться.

Как правило, использование объектов класса **StringBuffer** в программах не является необходимым, поскольку все те же операции можно выполнить, используя объекты класса **String**. Однако в некоторых случаях использование класса **StringBuffer** повышает эффективность программы, поскольку позволяет избежать многократного копирования модифицированных строк.

int length()	Получение длины строки
char charAt(**int** index)	Извлечение символа
char setCharAt(**int** index)	Изменение символа
StringBuffer append(String str) StringBuffer append(**int** i) StringBuffer append(Object obj) ...	Добавление строки к концу (существуют перегруженные версии для всех примитивных типов и типа **Object**)
StringBuffer insert(**int** index, String str)	Вставка строки в заданную позицию
StringBuffer delete(**int** beginIndex, **int** endIndex)	Удаление подстроки
StringBuffer replace(**int** beginIndex, **int** endIndex, String str)	Замена одной подстроки другой
StringBuffer reverse()	Обращение строки

Таблица 6.4. Методы класса **StringBuffer**

Для создания экземпляров класса **StringBuffer** используются следующие конструкторы:

```
StringBuffer()
StringBuffer(int capacity)
StringBuffer(String str)
```

Первые два из них создают объект, хранящий пустую строку с начальным буфером длины 16 в первом случае и заданной длины во втором случае. Третий конструктор инициализирует буфер заданной строкой. Размер буфера в этом случае устанавливается равным длине строки, увеличенной на 16.

Основные методы класса **StringBuffer** приведены в табл. 6.4.

В отличие от методов модификации строк класса **String**, аналогичные методы класса **StringBuffer** изменяют именно тот экземпляр объекта, на котором они вызываются. При этом они возвращают ссылку на этот объект (**this**). Последнее сделано для того, чтобы можно было объединять набор последовательных операций над строкой в одно выражение подобно тому, как это сделано в стандартной библиотеке языка C:

```
StringBuffer sb = new StringBuffer("abcd");
sb.append("ef").reverse();
System.out.println(sb); // выводит на экран: fedcba
```

7. Ввод/вывод

7.1. Потоки ввода/вывода. Ввод/вывод в Java осуществляется посредством потоковых классов. Все потоки разделяются на байтовые и символьные. Байтовые потоки предназначены для передачи бинарных данных, а символьные — текстовых. Для представления информации символьные потоки используют Unicode. Потоковая библиотека Java разработана таким образом, что ввод/вывод данных производится практически одинаково, вне зависимости от источников и приёмников передаваемых данных. Эта особенность обусловлена тем, что все потоковые классы унаследованы от одних и тех же абстрактных классов, определяющих методы, посредством которых и происходит взаимодействие пользовательских приложений с потоками. Суперклассами байтовых потоков являются классы **InputStream** и **OutputStream**, символьных — **Reader** и **Writer**. Основные методы этих классов перечислены в табл. 7.1, 7.2.

В случае ошибок ввода-вывода методы потоковых классов выбрасывают исключение класса **IOException** или его подклассов (например, **FileNotFoundException**). Классы этих исключений не являются подклассами класса **RuntimeException** и потому нуждаются в обязательной обработке.

Средства потокового ввода/вывода размещаются в пакете **java.io**.

7.2. Байтовые потоки, связанные с файлами. Байтовые потоки, связанные с файлами, используются для работы с бинарными файлами. Для этих потоков ввод/вывод данных осуществляется непосредственно без выполнения каких-либо преобразований над содержимым. Функциональность байтовых потоков, связанных с файлами, обеспечивается классами **FileInputStream** и **FileOutputStream**. Для создания объектов используются следующие конструкторы:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName, boolean append)
        throws FileNotFoundException
```

void write(**int** a) **throws** IOException	Запись одного байта/символа в поток
void write(♢ a) **throws** IOException	Запись содержимого из массива в поток
void write(♢[] a, **int** offset, **int** len) **throws** IOException	Запись содержимого части массива в поток
void flush() **throws** IOException	Сброс буферов вывода, связанных с потоком
void close() **throws** IOException	Закрытие потока

Таблица 7.1. Основные методы абстрактных классов **OutputStream** и **Writer**. Вместо символа ♢ подставляется **int** для байтовых и **char** для символьных потоков

Все конструкторы пытаются открыть файл, имя которого им передаётся в качестве аргумента. Если при создании выходного потока соответствующий файл уже существовал, он будет усечён до нулевой длины. Однако, если использовать третий из числа перечисленных выше конструкторов со значением аргумента **append** равным **true**, файл будет открыт в режиме дозаписи в конец.

7.3. Символьные потоки-обёртки для байтовых потоков. Поскольку вся символьная информация представляется в Java в Unicode, в то время как внешние данные зачастую представлены в других кодировках, то для эффективного взаимодействии Java с внешним миром требуется преобразование кодировок. Эту операцию выполняют символьные потоки-обёртки. Они могут присоединяться к любым байтовым потокам («обёртывают» его) и позволяют обращаться с полученным потоком как с символьным, выполняя все необходимые преобразования.

Рассматриваемым потокам соответствуют классы **InputStreamReader** (преобразует данные, считываемые из байтового потока в Unicode) и **OutputStreamWriter** (преобразует символьные данные, выводимые в байтовый поток из Unicode). Для создания объектов данных классов используются следующие конструкторы:

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String charSet)
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, String charSet)
```

int read() **throws** IOException	Чтение одного байта/символа из потока (в случае достижения конца файла возвращает –1)
int read(♢[] a) **throws** IOException	Заполнение массива содержимым, считанным из потока (возвращает фактическое количество считанных байтов/символов)
int read(♢[] a, **int** offset, **int** len) **throws** IOException	Заполнение части массива содержимым, считанным из потока
boolean ready() **throws** IOException	Проверка наличия данных в потоке (если метод возвращает **false**, то следующий вызов метода **read()** будет ожидать данных и завершится только при их получении). *Метод определяется только интерфейсом **Reader**!*
long skip(**long** n) **throws** IOException	Пропуск заданного количества байтов/символов в потоке
void close() **throws** IOException	Закрытие потока

Таблица 7.2. Основные методы абстрактных классов **InputStream** и **Reader**. Вместо символа ♢ подставляется **int** для байтовых и **char** для символьных потоков

Требуемая кодировка определяется аргументом **charSet**. Если она не указана, используется кодировка системной локали.

7.4. Символьные потоки, связанные с файлами. Символьные потоки, связанные с файлами, предназначены для работы с текстовыми файлами. Фактически они представляют собой комбинацию байтового потока и символьного потока-обёртки, осуществляющего преобразование. При этом всегда используется кодировка системной локали.

Данным потокам соответствуют классы **FileReader** и **FileWriter** со следующими конструкторами:

```
FileReader(String fileName) throws FileNotFoundException
FileWriter(String fileName) throws FileNotFoundException
FileWriter(String fileName, boolean append) throws FileNotFoundException
```

Используются они полностью аналогично конструкторам байтовых потоков, связанных с файлами (см. п. 7.2).

7.5. Символьный print-поток. Символьный print-поток (ему соответствует класс **PrintWriter**) является потоком-обёрткой для другого символьного потока, добавляя к нему функциональность форматированного вывода — методы **print()** и **println()**. Они определены для аргументов всех примитивных типов и типа **Object**. В последнем случае в поток выводится строковое представление объекта (см. п. 6.3).

Для создания объектов класса **PrintWriter** используются следующие конструкторы:

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
```

Отметим, что класс **PrintReader** не обеспечивает автоматического сброса буферов в файл при появлении в потоке символа перевода строки. Автоматический сброс возможен лишь при использовании метода **println()** при условии, что при создании объекта значение параметра **autoFlush** было установлено равным **true**.

В последних версиях Java класс **PrintWriter** содержит конструкторы, позволяющие присоединяться к байтовым и даже файловым байтовым потокам:

```
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName) throws FileNotFoundException
PrintWriter(String fileName, String charSet)
```

Эти конструкторы просто создают промежуточные объекты типов **InputStreamWriter** и **FileInputStream**, к которым и присоединяются.

Отметим, что наряду с классом **PrintWriter** существует класс **PrintStream** со схожей функциональностью. В новых Java-программах создание объектов этого класса не рекомендуется, однако именно такой тип имеют объекты **System.out** (стандартный поток вывода) и **System.err** (стандартный поток ошибок). Это обусловлено историческими причинами.

7.6. Буферизованные потоки. Буферизованные потоки являются обёртками для других потоков. Они обеспечивают повышение эффективности ввода/вывода за счёт буферизации. Функциональность буферизованных потоков обеспечивается классами **BufferedInputStream**, **BufferedOutputStream** (входной и выходной байтовые потоки с буферизацией), **BufferedReader** и **BufferedWriter** (аналогичные символьные

потоки). Для создания объектов этих классов используются конструкторы, принимающие в качестве аргумента ссылку на обёртываемый поток:

```
BufferedInputStream(InputStream in)
BufferedOutputStream(OutputStream out)
BufferedReader(Reader in)
BufferedWriter(Writer out)
```

Кроме того, входной символьный поток с буферизацией добавляет метод

```
String readLine() throws IOException
```

осуществляющий чтение одной строки из входного символьного потока. В случае достижения конца файла этот метод возвращает значение **null**.

7.7. Консольный ввод/вывод. Стандартные потоки вывода и ошибки представлены в Java объектами **System.out** и **System.err**, имеющими тип **PrintStream** (см. п. 7.5). Функционально они аналогичны символьным print-потокам, потому для вывода в них обычно используются методы **print()** и **println()**, например:

```
System.err.println("Это строка выводится в стандартный поток ошибки");
```

Чуть более сложная ситуация складывается со стандартным потоком ввода. В Java ему соответствует объект **System.in**, однако, в силу исторических причин, он является не символьным, а байтовым потоком и потому для ввода данных из него требуется преобразование в Unicode. Последнее можно осуществить, использовав поток-обёртку **InputStreamReader**:

```
Reader in = new InputStreamReader(System.in);
```

Поскольку обычно ввод из стандартного входного потока осуществляется построчно, а не посимвольно, то часто используется ещё одна обёртка типа **BufferedReader**:

```
Reader in = new BufferedReader(new InputStreamReader(System.in));
```

После этого можно использовать методы **read()** и **readLine()** созданного потока:

```
String s = in.readLine();
```

7.8. Пример. Следующая программа выводит в выходной файл только те строки входного файла, которые состоят лишь из чисел, разделённых пробелами. Имена входного и выходного файлов передаются в программу через аргументы командной строки. Для определения строк, которые следует выводить в выходной файл, используются регулярные выражения (см. п. 6.2).

```
import java.io.*;
class NumbersCopier
{
    /** Точка входа в программу */
    public static void main(String args[])
    {
        if(args.length != 2)
        {
            System.out.println("Неправильный вызов программы");
            return;
        }
        try
        {
            BufferedReader in = new BufferedReader(new FileReader(args[0]));
            PrintWriter out = new PrintWriter(args[1]);
            String s;
            while((s = in.readLine()) != null)
                if(s.matches("( *-?[0-9]+)+ *"))
                    out.println(s);
            out.close();
            in.close();
        }
        catch(IOException e)
        {
            System.out.println("Ошибка ввода/вывода");
        }
    }
}
```

8. Контейнеры

8.1. Обзор контейнеров. *Контейнерами* называются объекты, способные хранить другие объекты. Они различаются способом организации, временем выполнения основных операций (добавления, извлечения и удаления элементов), а также поддерживаемыми способами обхода элементов контейнера.

Контейнеры в Java можно разделить на три группы: массивы, коллекции и ассоциативные массивы. Массивы были рассмотрены в п. 1.5. Их отличительной чертой является невозможность изменения размеров после создания. Все остальные контейнеры в Java динамически изменяют свой размер при добавлении и удалении элементов.

Самую многочисленную группу контейнеров образуют *коллекции*. Все классы-коллекции в Java являются *параметризованными* типами (generics). Параметризованный тип — это класс или интерфейс, имеющий параметр — имя типа, которое должно задаваться при создании объектов. Параметризованные типы напоминают шаблоны C++ и предназначены в первую очередь для того, чтобы обеспечить контроль правильности типов на этапе компиляции. Они представляют собой очень гибкий и в то же время сложный механизм, детальное рассмотрение которого выходит за рамки данного пособия. Подробную информацию о параметризованных типах можно получить в статье [11].

Параметром классов-коллекций Java является тип объектов, которые они могут хранить (здесь и всюду ниже этот тип обозначен буквой **E**). Параметр может быть только классом или интерфейсом. Для хранения в коллекциях данных примитивных типов следует использовать классы-обёртки (см. п. 3.13).

Все классы-коллекции реализуют интерфейс **Collection<E>**. Основные методы этого интерфейса приведены в табл. 8.1. Все они соответствуют операциям, практически не зависящим от типа коллекции. В число наиболее часто используемых коллекций входят динамические массивы, двусвязные списки, множества и упорядоченные множества. Классы, реализующие функциональность этих коллекций, описаны в пп. 8.3, 8.5.

boolean add(E o)	Добавление элемента в коллекцию (возвращает **true** в случае успеха)
boolean remove(Object o)	Удаление элемента из коллекции
boolean contains(Object o)	Проверка принадлежности элемента коллекции
int size()	Получение количества элементов в коллекции
boolean isEmpty()	Проверка коллекции на пустоту
boolean equals(Object o)	Сравнение коллекций на равенство
Iterator<E> iterator()	Получение итератора коллекции
Object[] toArray()	Копирование содержимого коллекции в массив в порядке обхода коллекции итератором

Таблица 8.1. Основные методы интерфейса **Collection<E>**

Ассоциативные массивы (также употребляются термины «словарь» и «карта отображений») являются более сложными типами данных, чем коллекции. Фактически они представляют собой массивы, использующие вместо индексов произвольные объекты. Подробно они описаны в п. 8.6.

8.2. Итераторы. Итераторы предназначены для обхода коллекций в некотором порядке. Использование итератора — это самый быстрый способ перечисления всех элементов коллекции. Классы-итераторы реализуют интерфейс **Iterator<E>**, определяющий три метода: **hasNext()**, **next()** и **remove()**.

Итератор может быть получен путём вызова метода **iterator()** на соответствующем объекте-коллекции. Первоначально итератор устанавливается *перед первым* элементом коллекции. Каждый вызов метода

```
E next()
```

возвращает элемент, перед которым находился итератор до вызова, и сдвигает итератор, помещая его после возвращённого элемента. Если итератор находится после последнего элемента коллекции, метод **next()** выбрасывает исключение типа **NoSuchElementException**. Для проверки наличия элемента перед итератором используется метод

```
boolean hasNext()
```

Для удаления последнего возвращённого методом **next()** элемента может быть использован метод

```
void remove()
```

Допускается лишь однократный вызов метода **remove()** после вызова метода **next()**. В противном случае, а также в случае, когда вызов **next()** не осуществлялся вообще, выбрасывается исключение типа **IllegalStateException**.

В качестве примера рассмотрим использование итератора для вывода содержимого коллекции **collection**, содержащей элементы типа **MyClass**:

```
for(Iterator<MyClass> it = collection.iterator(); it.hasNext(); )
    System.out.println(it.next());
```

Для коллекций (а также для массивов) существует альтернативный вариант цикла **for**:

```
for([ тип данных ] переменная : коллекция)
    оператор;
```

Он осуществляет обход содержимого коллекции в порядке, определяемом итератором (для массивов в порядке от первого элемента к последнему), при этом переменной, заданной в заголовке цикла, последовательно присваиваются ссылки на элементы коллекции. Например:

```
for(MyClass e : collection)
    System.out.println(e);
```

8.3. Списки и динамические массивы. Списки и динамические массивы — это коллекции с произвольным порядком следования элементов. Порядок элементов в таких коллекциях определяется их номерами так же, как в обычных массивах. Все классы, обеспечивающие функциональность этих коллекций, реализуют интерфейс **List<E>** (субинтерфейс интерфейса **Collection<E>**), методы которого приведены в табл. 8.2.

Для рассматриваемых коллекций метод **add()**, определённый в интерфейсе **Collection<E>**, осуществляет вставку в конец коллекции, а метод **remove()** удаляет первое встретившееся вхождение заданного элемента.

Стандартная библиотека содержит следующие классы коллекций с произвольным порядком следования элементов: **LinkedList<E>**, **ArrayList<E>** и **Vector<E>**.

Класс **LinkedList<E>** представляет собой коллекцию, организованную в виде двусвязного списка. Он обеспечивает получение элемента по

void add(**int** index, E element)	Добавление элемента в указанную позицию
E get(**int** index)	Получение элемента с заданным индексом
E set(**int** index, E element)	Изменение элемента с заданным индексом (возвращает старое значение элемента)
E remove(**int** index)	Удаление элемента с заданным индексом (возвращает удалённый элемент)
int indexOf(Object o) **int** lastIndexOf(Object o)	Поиск первого/последнего вхождения элемента (в случае неудачи возвращают –1)
ListIterator<E> listIterator() ListIterator<E> listIterator(**int** index)	Получение списочного итератора, установленного на начало списка/перед заданным элементом списка

Таблица 8.2. Основные методы интерфейса **List<E>**

индексу за время порядка $O(N)$, а также вставку и удаление элементов в позицию итератора за время порядка $O(1)$.

Для создания экземпляров класса **LinkedList<E>** могут быть использованы следующие конструкторы:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Дополнительно к методам, объявленным в интерфейсе **List<E>**, класс **LinkedList<E>** предоставляет методы для вставки, получения и удаления первого и последнего элементов списка:

```
void addFirst(E o)
void addLast(E o)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

Все они требуют для своей работы время порядка $O(1)$.

Класс **ArrayList<E>** представляет собой коллекцию, организованную в виде массива переменного размера. Такие массивы имеют некоторый начальный размер (по умолчанию 10), который может увеличиваться при необходимости. Они обеспечивают получение элемента по

индексу за время порядка $O(1)$. Вставка и удаление элементов требует времени порядка $O(N)$.

Для создания экземпляров класса **ArrayList<E>** используются следующие конструкторы:

```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
```

Последний конструктор позволяет задать количество памяти, первоначально выделяемое под массив.

Итераторы обходят списки и динамические массивы в порядке увеличения индексов элементов. Коллекции, реализующие интерфейс **List<E>**, поддерживают специальные *списочные итераторы* (интерфейс **ListIterator<E>**), имеющие более широкую функциональность, чем обычные. В частности, такие итераторы могут двигаться по коллекции в обоих направлениях. Для проверки наличия предыдущего элемента и его получения предназначены методы

```
boolean hasPrevious()
E previous()
```

аналогичные методам **hasNext()** и **next()**. Также списочные итераторы позволяют выполнять замену последнего полученного элемента с помощью метода

```
void set(E o)
```

а также вставку элемента в текущую позицию итератора с помощью метода

```
void add(E o)
```

После вставки итератор размещается после вставленного элемента.

8.4. Упорядочение объектов. Многие задачи обработки контейнеров связаны с размещением объектов в некотором порядке. Для этого необходимо определить для объектов некоторую операцию, которая позволит выяснить, какой из объектов больше или меньше другого. В Java существует два стандартных способа определения такой операции. В первом случае класс, объекты которого подлежат упорядочению, должен реализовывать интерфейс **Comparable<T>**. В качестве параметра **T** должно подставляться имя класса, реализующего данный интерфейс. Например:

```
class MyClass implements Comparable<MyClass>
```

Интерфейс **Comparable<T>** определяет метод

```
int compareTo(T obj)
```

который должен осуществлять сравнение объекта класса, на котором этот объект вызывается (**this**), с переданным данному методу объектом и возвращать значение, меньшее нуля, если объект **this** больше объекта **obj**, равное нулю, если **this.equals(obj) == true** и большее нуля, если объект **obj** больше объекта **this**. Само отношение «больше» может быть произвольным и соответствует критерию, по которому производится упорядочение.

Например, если есть класс «комплексное число» с двумя вещественными полями **re** и **im** и требуется выполнять упорядочение объектов этого класса по возрастанию или убыванию модулей соответствующих комплексных чисел, операцию **compareTo** можно определить следующим образом:

```
class ComplexNumber implements Comparable<ComplexNumber>
{
    public double re, im;
    public ComplexNumber(double re, double im)
    {
        this.re = re; this.im = im;
    }
    public int compareTo(ComplexNumber c)
    {
        double diff = (re * re + im * im) – (c.re * c.re + c.im * c.im);
        return (int)(Math.signum(diff));
    }
}
```

Способ упорядочения объектов, основанный на определённой указанным образом операции сравнения, называется в Java *естественным упорядочением*. Каждый класс может определить не более одного способа естественного упорядочения.

Другой способ упорядочения заключается в определении внешнего класса (*компаратора*), объекты которого будут осуществлять сравнение объектов требуемого класса. Классы-компараторы должны реализовывать интерфейс **Comparator<T>**, где параметр **T** определяет тип объектов, сравниваемых данным компаратором. Интерфейс **Comparator<T>** определяет метод

```
int compare(T o1, T o2)
```

который осуществляет сравнение двух объектов, передаваемых ему в качестве аргументов, и возвращает в результате сравнения положительное, отрицательное или равное нулю целое число аналогично методу **compare()** интерфейса **Comparable<T>**.

Так, в примере, приведённом выше, вместо метода **compareTo()** можно реализовать компаратор следующим образом:

```
class ComplexNumberLenComp implements Comparator<ComplexNumber>
{
    public int compare(ComplexNumber c1, ComplexNumber c2)
    {
        double diff = (c1.re * c1.re + c1.im * c1.im)
            – (c2.re * c2.re + c2.im * c2.im);
        return (int)(Math.signum(diff));
    }
}
```

Легко видеть, что для одного класса можно определить любое количество компараторов, поэтому данный способ обычно применяется в тех случаях, когда программа использует разные критерии упорядочения объектов.

Многие стандартные алгоритмы обработки контейнеров, а также сами контейнеры требуют определения способа упорядочения объектов. Пример использования различных способов упорядочения при хранении и обработке данных приведён в п. 8.8.

8.5. Множества и упорядоченные множества. Множества и упорядоченные множества — это коллекции, обеспечивающие быстрое добавление и удаление элементов, однако не позволяющие устанавливать порядок их следования и хранить дубликаты.

Множества представлены интерфейсом **Set<E>**, а также реализующим его классом **HashSet<E>**. Ни один из них не добавляет новых методов по сравнению с интерфейсом **Collection<E>**. Для хранения элементов класс **HashSet<E>** использует хэширование, поэтому объекты, хранимые в коллекциях-множествах, обязательно должны реализовывать методы **equals()** и **hashCode()** (см. п. 3.12) Благодаря хэшированию время выполнения основных операций (добавление, удаление и проверка вхождения элемента в множество) практически не зависит от размера множества. Порядок обхода элементов множества итератором оказывается достаточно произвольным, так как определяется хэш-

кодами элементов. Для создания экземпляров класса **HashSet<E>** используются следующие конструкторы:

```
HashSet()
HashSet(Collection<? extends E> c)
```

Упорядоченные множества представлены интерфейсом **SortedSet** и реализующем его классом **TreeSet<E>**. Последний использует для хранения элементов сбалансированные бинарные деревья, что позволяет гарантировать порядок обхода множества итератором в возрастающем порядке. Операции добавления, удаления и проверки вхождения элемента в множество требуют времени порядка $N \log N$.

Для создания экземпляров класса **TreeSet<E>** используются следующие конструкторы:

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(SortedSet<E> s)
TreeSet(Comparator<? super E> c)
```

Последний из них позволяет создавать множества, упорядоченные в произвольном порядке, определяемом передаваемым в конструктор компаратором (см. п. 8.4). Во всех остальных случаях множества упорядочиваются в естественном порядке. В этом случае классы элементов множества обязаны реализовывать интерфейс **Comparable<T>**.

8.6. Ассоциативные массивы. Как уже отмечалось, ассоциативные массивы можно рассматривать как массивы, использующие вместо индексов произвольные объекты. Можно считать, что они задают отображение множества *объектов-ключей* во множество *объектов-значений*. Они не являются коллекциями в том смысле, что не реализуют интерфейс **Collection<E>**. Его заменяет интерфейс **Map<K, V>**, имеющий два параметра. Первый определяет тип данных ключей ассоциативного массива, а второй — тип его значений. Основные методы интерфейса **Map<K, V>** приведены в табл. 8.3.

Ассоциативные массивы используют для хранения пар значений контейнеры-множества. Поэтому как иерархия, так и свойства тех и других совпадают: классам **HashSet** и **TreeSet** соответствуют классы **HashMap** и **TreeMap** соответственно, а интерфейсу **SortedSet** — интерфейс **SortedMap**. Оценки времени выполнения основных операций для ассоциативных списков такие же, как и для соответствующих им множеств.

V put(K key, V value)	Добавление пары ключ-значение (возвращает старое значение, ассоциированное с ключом или **null**, если такого ключа не было)
V get(Object key)	Получение значения, соответствующего ключу (возвращает **null**, если ключа нет в ассоциативном массиве)
V remove(Object key)	Удаление пары ключ-значение (возвращает значение удаляемой пары)
boolean containsKey(Object key)	Проверка, содержит ли ассоциативный массив заданный ключ
int size()	Получение размера ассоциативного массива
boolean isEmpty()	Проверка ассоциативного массива на пустоту
void clear()	Очистка ассоциативного массива
boolean equals(Object o)	Сравнение ассоциативных массивов на равенство
Set<K> keySet()	Получение набора ключей в виде множества

Таблица 8.3. Основные методы интерфейса **Map<K, V>**

Ассоциативные списки не имеют собственных итераторов. Для их обхода используются итераторы множеств их ключей, возвращаемые методом **keySet()**. Следующий фрагмент кода демонстрирует использование ассоциативных массивов:

```
TreeMap<String, String> capitals = new TreeMap<String, String>();
capitals.put("Россия", "Москва");
capitals.put("Франция", "Париж");
capitals.put("Италия", "Лондон"); // неправильно
capitals.put("Италия", "Рим"); // исправление ошибки
System.out.println("Столица Италии – " + capitals.get("Италия"));
// Вывод всего массива, отсортированного по названиям стран
for(String s : capitals.keySet())
    System.out.println(s + ": " + capitals.get(s));
```

8.7. Унаследованные (legacy) классы-контейнеры. Кроме классов **ArrayList<E>**, **HashSet<E>** и **HashMap<E>**, существуют близкие им по функциональности классы **Vector<E>** и **Hashtable<E>** и **Dictionary<E>**. Это так называемые унаследованные (legacy) классы из старых версий

E max(Collection<? **extends** E> coll) E max(Collection<? **extends** T> coll, Comparator<? **super** T> comp) E min(Collection<? **extends** E> coll) E min(Collection<? **extends** T> coll, Comparator<? **super** T> comp)	Нахождение наибольшего/наименьшего элемента коллекции в смысле естественного порядка элементов/порядка элементов, определяемого компаратором
void sort(List<T> list) **void** sort(List<T> list, Comparator<? **super** T> c)	Сортировка коллекции с произвольным порядком следования элементов в естественном порядке/в порядке, определяемом компаратором
int binarySearch(List<? **extends** T> list, T key) **int** binarySearch(List<? **extends** T> list, T key, Comparator<? **super** T> c)	Бинарный поиск индекса элемента в упорядоченной коллекции. Если элемент не найден, возвращается величина, равная *(минус позиция, в которой должен был быть элемент, минус единица)*
Collection<T> unmodifiableCollection(Collection<? **extends** T> c) List<T> unmodifiableList(List<? **extends** T> c) и т. д.	Получение неизменяемой обертки коллекции (существуют аналогичные методы для всех стандартных типов контейнеров)

Таблица 8.4. Основные методы класса **Collections**. Все методы являются статическими

библиотеки. Основное их назначение — обеспечивать обратную совместимость. Хотя они не считаются устаревшими (deprecated), вероятно, лучше избегать использования их в новых программах.

8.8. Стандартные алгоритмы обработки контейнеров. Стандартная библиотека Java содержит реализацию некоторых стандартных алгоритмов обработки контейнеров, а также некоторых сервисных операций. Все они представлены в виде статических методов классов **Collections** (для коллекций и ассоциативных списков) и **Arrays** (для массивов). Соответствующие методы приведены в табл. 8.4, 8.5.

Особо отметим семейство методов, возвращающих неизменяемые обёртки контейнеров (**unmodifiableCollection()**, **unmodifiableList()** и т. д.). Эти методы возвращают объект, реализующий интерфейс контейнера соответствующего типа. При использовании методов перемещения и получения значений возвращённого объекта последний ведёт себя точно так же, как и исходный объект. При попытке вызова любо-

void sort(**int**[] a)	Сортировка массива
void sort(T[] a, Comparator<? **super** T> c)	Сортировка массива объектов в порядке, определяемом компаратором
int binarySearch(**int**[] a, **int** key)	Бинарный поиск индекса элемента в упорядоченном массиве. Если элемент не найден, возвращается величина, равная *(минус позиция, в которой должен был быть элемент, минус единица)*
int binarySearch(T[] a, T key, Comparator<? **super** T> c)	
boolean equals(**int**[] a, **int**[] a2)	Сравнение массивов на равенство
void fill(**int**[] a, **int** val)	Заполнение всех элементов массива заданным значением
List<T> asList(T[] a)	Преобразование массива/набора элементов в список
List<T> asList(T. . . a)	

Таблица 8.5. Основные методы класса **Arrays**. Приведены методы для работы с массивами типа **int[]**. Существуют аналогичные методы для работы с массивами всех остальных примитивных типов и типа **Object**. Все методы являются статическими

го из методов изменения контейнера выбрасывается исключение типа **UnsupportedOperationException**.

Следующий пример демонстрирует возможности средств, описанных в данной главе.

```
import java.util.*;
/** Класс "анкетные данные человека" */
class Person implements Comparable<Person>
{
    /** Данные и методы для их получения */
    private String name;
    public String getName() { return name; }
    private int age;
    public int getAge() { return age; }
    /** Конструктор */
    public Person(String name, int age) { this.name = name; this.age = age; }
    /** Преобразование объекта к строке */
    public String toString() { return name + " (" + age + ")"; }
    /** Сравнение объектов на равенство */
    public boolean equals(Object obj)
    {
        if(!(obj instanceof Person))
            return false;
```

```
        return ((Person)obj).name.equals(name) && ((Person)obj).age == age;
    }
    /** Получение хэш–кода объекта */
    public int hashCode() { return name.hashCode() + age; }
    /** Сравнение объектов по фамилиям */
    public int compareTo(Person p) { return name.compareTo(p.name); }
}
/** Класс "компаратор для сравнения по возрасту" */
class PersonAgeComparator implements Comparator<Person>
{
    public int compare(Person p1, Person p2)
    {
        return p1.getAge() – p2.getAge();
    }
}
public class PersonsSortExample
{
    public static void main(String args[])
    {
        TreeSet<Person> data = new TreeSet<Person>();
        data.add(new Person("Петров", 31));
        data.add(new Person("Иванов", 45));
        data.add(new Person("Смирнов", 20));
        data.add(new Person("Смирнов", 20)); // такой элемент уже есть
        System.out.println(data); // вывод отсортирован по фамилиям
        // Копирование содержимого множества в список
        ArrayList<Person> data2 = new ArrayList<Person>(data);
        Collections.sort(data2, new PersonAgeComparator());
        System.out.println(data2); // вывод отсортирован по возрасту
    }
}
```

Вывод этой программы:

```
[Иванов (45), Петров (31), Смирнов (20)]
[Смирнов (20), Петров (31), Иванов (45)]
```

9. Многопоточное программирование

9.1. Создание потоков и управление ими. *Многопоточность* — это способ организации выполнения программы, при котором отдельные её части выполняются одновременно или псевдоодновременно.

Класс **Thread** инкапсулирует поток выполнения. Основные методы этого класса перечислены в табл. 9.1. Каждому потоку выполнения в программе соответствует экземпляр этого класса или его субклассов. Как правило, в программах создаются субклассы класса **Thread**, переопределяющие метод

```
void run()
```

который является точкой входа в поток. При создании экземпляров потокового класса соответствующие им потоки выполнения не получают управления. Чтобы запустить поток, следует вызвать метод

```
void start()
```

соответствующего ему объекта. При этом поток, на котором был вызван метод **start()**, продолжает выполнение, и одновременно с ним на другом потоке начинает выполняться метод **run()** соответствующего объекта («одновременно» в данном случае обычно означает, что потоки последовательно получают небольшие кванты процессорного времени, что создаёт иллюзию их одновременного выполнения). Как только выполнение метода **run()** завершится, соответствующий ему поток выполнения прекратит своё существование. Метод **start()** можно вызывать лишь один раз для каждого объекта-потока.

Для того чтобы проверить, существует ли ещё некоторый поток выполнения, используется метод **isAlive()** соответствующего объекта. Существует также возможность приостановления текущего потока до момента завершения другого. Для этого используется метод **join()**, вызываемый на объекте, соответствующем потоку, завершение которого ожидается.

Выполняющийся поток сам может перейти в режим ожидания путём вызова метода **sleep()**. В этом случае процессорное время приостановленного потока будет использоваться другими потоками. Кроме того,

void start()	Запуск потока
void run()	Точка входа в поток
static void yield()	Добровольная кратковременная передача управления другим потокам
static void sleep(**long** millis) **throws** InterruptedException	Приостановление потока на указанный промежуток времени
void join() **throws** InterruptedException	Приостановление текущего потока до момента завершения другого потока
boolean isAlive()	Проверка, не завершился ли поток
void interrupt()	Установка флага **interrupted**
boolean isInterrupted()	Получение состояния флага **interrupted**
void setPriority(**int** newPriority)	Изменение приоритета потока
int getPriority()	Получение приоритета потока
void setDaemon(**boolean** on)	Объявление потока демоном

Таблица 9.1. Основные методы класса **Thread**

поток может отказаться от своего кванта времени путём вызова метода **yield()**. В этом случае его квант также перераспределяется между другими потоками. Обычно это используется в тех случаях, когда поток ожидает наступления некоторого события, инициируемого другим потоком. Например:

```
if(!somethingHappened)
    yield();
```

Использование **yield()** позволяет повысить быстродействие программы, поскольку избегаются многократные бесполезные проверки значения переменной, пока другие потоки не получат возможность изменения этого значения.

Соотношение между временем, которое получают различные потоки, определяется их приоритетами. Приоритет — это величина в диапазоне от 1 (минимальный приоритет) до 10 (максимальный приоритет). Для установки и получения значения приоритета используются методы **setPriority()** и **getPriority()** соответственно. По умолчанию потоки создаются с приоритетом, равным 5.

В любой программе всегда существует по крайней мере один поток выполнения, на котором выполняется метод **main()** (точка входа в программу). Выполнение программы завершается лишь тогда, когда все потоки, запущенные программой, завершатся. Исключением из этого правила являются лишь потоки-демоны. Для того чтобы объявить

некоторый поток демоном, следует вызвать метод **setDaemon()** до запуска этого потока. Потоки-демоны прерываются виртуальной машиной автоматически, когда завершаются все остальные потоки. Основное назначение таких потоков — предоставлять остальной части программы некоторые сервисы в фоновом режиме.

Иногда возникает необходимость досрочно прервать работу потока и, возможно, возобновить его выполнение в дальнейшем. Для этого в класс **Thread** были введены методы **stop()**, **suspend()** и **resume()**, однако позднее они были признаны небезопасными, поскольку не позволяют перед остановкой потока осуществить такие действия, как, например, освобождение заданных ресурсов, и потому могут привести к зависанию или некорректному функционированию программы. Эти методы не должны использоваться в новых программах. Альтернативным способом управления потоком является использование специального флага **interrupted** класса **Thread**. Для того чтобы прервать выполнение потока, используется метод **setInterrupted()**. Если поток в этот момент является приостановленным (вызовом метода **sleep()**, **join()** и др.), то он возобновляет свою работу, а соответствующий метод выбрасывает исключение. В противном случае устанавливается специальный флаг **interrupted**. Поток должен самостоятельно проверять состояние этого флага путём вызова метода **isInterrupted()** и осуществлять завершение работы в случае, если он установлен, например:

```
void run()
{
    boolean done = false;
    while(!done) // главный цикл потока
    {
        if(isInterrupted())
        {
            // провести завершающие действия
            return;
        }
    }
}
```

Каждый вызов метода **isInterrupted()** сбрасывает значение флага **interrupted**. Отметим, что, если поток не проверяет значение флага **interrupted**, может возникнуть ситуация, когда его вообще нельзя будет остановить.

Следующий пример демонстрирует использование различных методов класса **Thread**. В этом примере главный поток создаёт экземпляр класса **HelloThread**, выводящий на экран фразу «Hello, world!» на отдельном потоке выполнения и делающий задержку перед своим завершением. Главный поток, в свою очередь, ждёт завершения дочернего потока, после чего выводит сообщение, что дочерний поток завершился.

```
class HelloThread extends Thread
{
    /** Точка входа в поток */
    public void run()
    {
        System.out.println("Hello, world!");
        try
        {
            sleep(1000);
        }
        catch(InterruptedException e)
        {
            System.err.println("Поток MyHelloThread прерван");
        }
    }
}
public class SimpleThreadsExample
{
    public static void main(String args[])
    {
        Thread t = new HelloThread();
        t.start(); // запуск потока
        try
        {
            t.join(); // ожидание завершения потока
        }
        catch(InterruptedException e)
        {
            System.err.println("Главный поток прерван");
        }
        System.out.println("Поток MyHelloThread завершился");
    }
}
```

9.2. Интерфейс Runnable. Иногда бывает необходимо добавить функциональность класса **Thread** к уже существующему классу. Однако выполнять наследование от класса **Thread** нельзя, поскольку класс уже является субклассом некоторого другого класса. В этом случае может использоваться интерфейс **Runnable**. Этот интерфейс определяет единственный метод

```
void run()
```

аналогичный одноимённому методу класса **Thread**.

Класс, реализующий интерфейс **Runnable**, используется следующим образом: создаётся экземпляр этого класса, который передаётся в конструктор класса **Thread**:

```
MyRunnableClass r = new MyRunnableClass();
Thread t = new Thread(r);
t.start();
```

Иногда объект класса **Thread** размещается непосредственно внутри класса, реализующего интерфейс **Runnable**, и инициализируется в конструкторе этого класса:

```
class MyRunnableClass implements Runnable
{
    private Thread thread;
    /** Конструктор: создаёт объект класса Thread и запускает его */
    public MyThread()
    {
        thread = new Thread(this);
        thread.start();
    }
    /** Точка входа в поток */
    public void run()
    {
        // . . .
    }
}
```

9.3. Синхронизация. В многопоточных приложениях часто бывает необходимо, чтобы один и тот же объект или метод одновременно использовался лишь одним потоком. Для того чтобы обеспечить эту возможность, используется механизм *синхронизации*.

Рассмотрим простейший пример, где может потребоваться синхронизация.

```
class MyThread extends Thread
{
    /** Конструктор (сохраняет имя потока и запускает поток) */
    public MyThread(String name) { super(name); start(); }
    /** Точка входа в поток */
    public void run()
    {
        try
        {
            System.out.print("[");
            Thread.sleep(1);
            System.out.print(getName());
            Thread.sleep(1);
            System.out.print("]");}
        }
        catch(InterruptedException e)
        {
            System.err.println("Поток " + getName() + " прерван");
        }
    }
}
public class SyncExample
{
    public static void main(String[] args)
    {
        for(int i = 0; i < 10; ++i)
            new MyThread(String.valueOf(i));
    }
}
```

В этом примере создаётся 10 потоков, каждый из которых выводит на консоль свой номер в квадратных скобках и завершается. Пример вывода этой программы:

```
[[[[[[[[[1[4320586]79]]]]]]]]]
```

Поскольку все потоки одновременно используют один тот же объект (**System.out**), то результаты вывода различных потоков перепутываются.

Нетрудно представить себе ситуацию, когда подобное совместное использование объектов может приводить к ошибкам. Следует также иметь в виду, что поскольку распределение времени между потоками носит недетерминированный характер, то ошибки могут проявляться

не при каждом запуске программы, что усложняет их нахождение. Например, если в приведённом выше примере убрать задержки (**sleep**), то указанный эффект *почти никогда* не возникает, поскольку одного кванта времени, выделяемого потоку, почти всегда бывает достаточно для того, чтобы осуществить требуемый вывод.

Основным понятием синхронизации является монитор. *Монитор* — это (неявный) объект, который используется для взаимоисключающей блокировки. В Java каждый объект изначально имеет свой собственный монитор. Когда поток входит в монитор, соответствующий объект блокируется. Если в то же время другой поток пытается войти в тот же монитор, то он приостанавливается до тех пор, пока первый поток не выйдет из монитора и тем самым не снимет блокировку с объекта.

В Java существуют два вида синхронизации: посредством синхронизированных методов и синхронизированных блоков. Вход (выход) в синхронизированный метод или блок автоматически приводит к входу текущего потока в монитор (выходу из монитора) соответствующего объекта.

Чтобы сделать метод синхронизированным, достаточно добавить в его заголовок ключевое слово **synchronized**. Синхронизированные блоки (также иногда называемые критическими секциями) имеют следующий синтаксис.

```
synchronized(имя_объекта)
{
    // . . .
}
```

В частности, в приведённом выше примере достаточно поместить содержимое блока **try** в синхронизированный блок:

```
synchronized(System.in)
{
    System.out.print("[");
    Thread.sleep(1);
    System.out.print(getName());
    Thread.sleep(1);
    System.out.print("]");
}
```

В этом случае поток, вошедший первым в монитор объекта **System.in**, приостанавливает все остальные потоки, пытающиеся войти в тот же

монитор, и освобождает монитор только после того, как осуществляет свой вывод полностью. Затем в монитор входит следующий поток и т. д.

Вывод программы после добавления в неё синхронизированного блока:

```
[0][9][8][7][6][5][4][3][2][1]
```

В заключение отметим, что Java предоставляет значительно более широкие возможности управления синхронизацией потоков. В частности, поток, вошедший в монитор, может на время уступить монитор другому потоку и вновь получить управление после того, как последний освободит монитор. Эти возможности, а также характерные ошибки межпоточного программирования рассмотрены в книге [2].

10. Технология доступа к базам данных JDBC

10.1. Архитектура JDBC. JDBC (Java DataBase Connectivity) — это платформо-независимая технология доступа к базам данных из Java-приложений. Она предоставляет набор интерфейсов доступа к данным, посредством которых Java-приложения взаимодействуют с внешними драйверами баз данных, а также средства, обеспечивающие интеграцию этих внешних драйверов в среду Java.

Такой подход является достаточно гибким, поскольку Java-приложения оказываются отделены от деталей взаимодействия с БД, а драйверам БД достаточно обеспечить реализацию определённых JDBC-интерфейсов для того, чтобы быть совместимыми с Java-приложениями.

Классы и интерфейсы JDBC размещаются в пакете **java.sql**. Дополнительные сведения о технологии JDBC можно найти в книгах [4, 5].

10.2. Драйверы баз данных. Для соединения с базой данных необходимо сначала загрузить драйвер, соответствующий используемой БД. В стандартную поставку Java входит лишь один такой драйвер — JDBC–ODBC Bridge. Этот драйвер организует мост между интерфейсом JDBC и стандартным интерфейсом подключения к базам данных Microsoft ODBC (Open DataBase Connectivity). Такой способ является достаточно универсальным, поскольку для большинства баз данных существуют соответствующие драйверы ODBC. Однако предпочтительным способом подключения всё же считается использование JDBC-драйверов конкретных баз данных, поскольку такие драйверы, как правило, являются более функциональными и быстродействующими по сравнению с JDBC–ODBC Bridge. Кроме того, они могут обеспечить кросс-платформенность приложений (использование ODBC, как правило, возможно только в ОС MS Windows). JDBC-драйверы могут быть найдены в составе поставки большинства существующих СУБД, либо в Интернете по адресу *http://developers.sun.com/product/jdbc/drivers*.

JDBC-драйверы обычно поставляются в виде jar-архивов, которые необходимо разместить в любом из каталогов, перечисленных в **CLASSPATH** (см. п. 3.9), чтобы JVM смогла найти требуемый драйвер.

Загрузка драйвера осуществляется посредством вызова метода статического метода класса **Class**:

```
static Class forName(String className) throws ClassNotFoundException
```

В качестве аргумента методу передаётся имя класса-драйвера, например:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // драйвер ODBC–JDBC Bridge
Class.forName("com.mysql.jdbc.Driver"); // драйвер MySQL Connector/J
```

В случае невозможности загрузки драйвера будет выброшено исключение типа **ClassNotFoundException**.

10.3. Подключение к базе данных. Подключению к базе данных соответствуют объекты классов, реализующих интерфейс **Connection**. Для создания этих объектов используется статический метод

```
static Connection getConnection(String url, String user, String password)
    throws SQLException
```

класса **DriverManager**. Первый аргумент этого метода определяет строку подключения к базе данных, два других — имя пользователя и его пароль.

Строка подключения имеет следующий формат:

```
jdbc:имя_подпротокола:имя_источника_данных
```

Здесь «имя подпротокола» определяет драйвер или протокол подключения к БД, а «имя источника данных» — как правило, имя базы данных. Строка подключения может содержать также дополнительные параметры. Их формат, равно как и формат имени источника данных, зависит от конкретного драйвера. Например, следующая строка может быть использована для подключения посредством драйвера JDBC–ODBC Bridge к БД с именем «Clients», расположенной на локальной машине:

```
jdbc:odbc:Clients
```

Соответствующая БД должна быть предварительно зарегистрирована в ODBC Administrator. Ещё один пример:

```
jdbc:mysql://http://mydbserv.ru:3105/Students?useUnicode=true
```

Такая строка подключения используется для соединения с удалённой БД «Students», находящейся на сервере MySQL по адресу *http://mydbserv.ru*. Подключение будет осуществляться через порт 3105, для кодирования передаваемых строк будет использоваться Unicode.

10.4. Создание и выполнение запросов к базе данных. Запрос к базе данных инкапсулируется классами, реализующими интерфейс **Statement**. Каждому объекту такого класса соответствует не более одного объекта, представляющего результирующий набор данных. Если требуется иметь более одного результирующего набора данных, каждый из них должен быть сгенерирован отдельным **Statement**-объектом. Выполнение нового SQL-запроса закрывает существующий набор данных, созданный тем же **Statement**-объектом.

Для создания запроса используется метод

```
Statement createStatement() throws SQLException
Statement createStatement(int resultSetType, int resultSetConcurrency,
    int resultSetHoldability) throws SQLException
```

вызываемый на объекте-соединении. Вторая версия этого метода позволяет установить параметры результирующего набора данных. Возможные значения этих параметров приведены в табл. 10.1 (см. также п. 10.5).

Для выполнения запроса к БД используются следующие методы интерфейса **Statement**:

```
boolean execute(String sql) throws SQLException
ResultSet executeQuery(String sql) throws SQLException
int executeUpdate(String sql) throws SQLException
```

Первый метод обычно применяется для запросов, не возвращающих результирующих значений (например, **DROP**), второй метод — для запросов, возвращающих набор данных (например, **SELECT**). Третий метод применяется для выполнения SQL-операторов **INSERT**, **UPDATE** и **DELETE**. Он возвращает количество добавленных, удалённых или изменённых в результате запроса записей. При возникновении ошибок все методы выбрасывают исключение типа **SQLException**.

После обработки результатов запроса последний должен быть закрыт посредством вызова метода

```
void close() throws SQLException
```

Если в результате запроса был создан набор данных, он закрывается автоматически при закрытии запроса.

Следующий пример демонстрирует создание, заполнение и модификацию таблицы «people» посредством SQL-запросов.

```
// Загрузка драйвера и соединение с БД
```

```
String dbUrl = "jdbc:mysql://localhost/test?useUnicode=true";
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManager.getConnection(dbUrl, "pgm", "");
Statement s = conn.createStatement();
// Создание таблицы и заполнение её
s.execute("CREATE TABLE people "
        + "(Id INTEGER AUTO_INCREMENT PRIMARY KEY,"
        + "Name VARCHAR(40), "
        + "Telephone VARCHAR(10),"
        + "Age INTEGER)");
s.execute("INSERT INTO people(Name, Telephone, Age)"
        + "VALUES('Иванов', '12–02–00', 37)");
s.execute("INSERT INTO people(Name, Telephone, Age)"
        + "VALUES('Петров', '43–55–47', 26)");
s.execute("INSERT INTO people(Name, Telephone, Age)"
        + "VALUES('Сидоров', '43–88–90', 59)");
// Изменение и удаление записей
System.out.println("Обновлено записей: "
        + s.executeUpdate("UPDATE people SET Telephone='63–16–00'"
        + "WHERE Name='Петров'"));
System.out.println("Удалено записей: "
        + s.executeUpdate("DELETE FROM people WHERE Name='Иванов'"));
s.close();
```

10.5. Навигация по наборам данных. Набор данных — это объект класса, реализующего интерфейс **ResultSet**, инкапсулирующий результат выполнения запроса к БД. Возможности набора данных определяются его параметрами. Эти параметры задаются при создании объекта-запроса методами **createStatement()** или **prepareStatement()** объекта-соединения (см. пп. 10.4, 10.7). Параметр **rsType** определяет тип набора данных, **rsConcurrency** — режим обновления набора, а **rsHoldability** — режим сохранения курсора после фиксации транзакции. Возможные значения этих параметров приведены в табл. 10.1.

Набор данных содержит *курсор*, позволяющий перемещаться *между строками* этого набора. Техника работы с курсорами во многом аналогична технике работы с итераторами коллекций (см. п. 8.2). Первоначально курсор размещается перед первой строкой набора данных. Вызовы методов

```
boolean next() throws SQLException
boolean previous() throws SQLException
```

Значение	Описание
*Параметр **rsType***	
TYPE_FORWARD_ONLY (по умолч.)	набор данных является однонаправленным; курсор может двигаться по набору данных только вперёд
TYPE_SCROLL_INSENSITIVE	допускается движение курсора в произвольном направлении, а также произвольный доступ к содержимому набора данных
TYPE_SCROLL_SENSITIVE	допускается движение курсора в произвольном направлении, а также произвольный доступ к содержимому набора данных; содержимое набора данных автоматически актуализируется, когда содержимое БД изменяется в результате выполнения других запросов
*Параметр **rsConcurrency***	
CONCUR_READ_ONLY (по умолч.)	режим «только для чтения»; изменения набора данных запрещены
CONCUR_UPDATABLE	изменения набора данных разрешены; эти изменения автоматически вносятся в БД
*Параметр **rsHoldability***	
HOLD_CURSORS_OVER_COMMIT	после фиксации транзакции набор данных остаётся открытым, и положение курсора в нём сохраняется
CLOSE_CURSORS_AT_COMMIT (по умолч.)	после фиксации транзакции набор данных закрывается

Таблица 10.1. Значения параметров набора данных. Все значения являются статическими константами класса **ResultSet**

позволяют выполнять перемещение на одну строку набора вперёд и назад соответственно (курсор остаётся между строками!). Методы

```
boolean relative(int rows) throws SQLException
boolean absolute(int row) throws SQLException
```

позволяют осуществлять относительное и абсолютное позиционирование курсора в наборе данных. Указанные методы возвращают значение

Тип данных SQL	Тип данных Java	Тип данных SQL	Тип данных Java
BOOLEAN	boolean	DOUBLE	double
TINYINT	byte	VARCHAR	String
SMALLINT	short	NUMERIC	BigDecimal
INTEGER	int	DATE	Date
BIGINT	long	TIME	Time
FLOAT	float	BLOB	Blob

Таблица 10.2. Соответствие между типами данных SQL и Java

true, если перемещение закончилось успешно, и **false**, если требуемый элемент отсутствует. Отметим, что *нумерация записей набора осуществляется с единицы!* Для позиционирования курсора перед первым и после последнего элемента набора используются методы

```
void beforeFirst() throws SQLException
void afterLast() throws SQLException
```

Все перечисленные выше методы, кроме метода **next()**, работают только в том случае, если набор данных не является однонаправленным (значение параметра **rsType** не равно **TYPE_FORWARD_ONLY**).

Для получения значений полей последней строки набора данных, через которую переместился любой из методов позиционирования (**next()**, **previous()** и т. д.), используются методы

```
*** get***(int columnIndex) throws SQLException
*** get***(String columnName) throws SQLException
```

Здесь вместо *** подставляется имя типа данных Java, соответствующее типу данных SQL параметра (см. табл. 10.2). Требуемое поле определяется по либо по его имени в БД, либо по номеру в запросе (*нумерация полей начинается с единицы!*). Вместо методов **get***** можно использовать универсальный метод

```
Object getObject(int columnIndex) throws SQLException
Object getObject(String columnName) throws SQLException
```

Он возвращает значение запрошенного поля в виде объекта соответствующего типа (табл. 10.2). Если этот тип является примитивным, возвращается его обёртка (**Integer** вместо **int** и т. д.; см. также п. 3.13).

Следующий метод выводит на экран содержимое произвольной таблицы, имя которой передаётся методу в качестве аргумента. В примере используется объект класса **ResultSetMetadata**, позволяющий получить

количество и названия полей набора данных. Более подробную информацию об этом классе можно получить в документации по JDBC.

```
/** Печать содержимого таблицы БД */
public static void printDB(Connection conn, String tableName)
throws SQLException
{
    Statement s = conn.createStatement();
    ResultSet r = s.executeQuery("SELECT * FROM " + tableName);
    // Получение и вывод названий столбцов
    ResultSetMetaData md = r.getMetaData();
    for(int i = 1; i <= md.getColumnCount(); ++i)
        System.out.print(md.getColumnName(i) + "\t");
    System.out.println();
    // Вывод данных таблицы
    while(r.next())
    {
        for(int i = 1; i <= md.getColumnCount(); ++i)
            System.out.print(r.getObject(i) + "\t");
        System.out.println();
    }
    s.close();
}
```

10.6. Модифицируемые наборы данных. Если набор данных содержит поля только одной таблицы БД, среди полей набора содержится первичный ключ этой таблицы, а параметр **rsConcurrency** установлен равным **CONCUR_UPDATABLE**, то такой набор данных можно использовать для модификации соответствующей таблицы БД, минуя явные вызовы SQL-операторов. Для этого используются методы

```
void update***(int columnIndex, *** x) throws SQLException
void update***(String columnName, *** x) throws SQLException
```

Изменения записываются в БД путём вызова метода

```
void updateRow() throws SQLException
```

Для удаления текущей записи набора данных из БД используется метод

```
void deleteRow() throws SQLException
```

Несколько более сложным образом осуществляется добавление записи. Сначала необходимо переместить курсор на специальную строку

набора данных, предназначенную для вставки. Для этого используется метод

```
void moveToInsertRow() throws SQLException
```

Затем с помощью методов **update***()** осуществляется заполнение полей добавляемой записи. Добавление записи в таблицу производит метод

```
void insertRow() throws SQLException
```

Для того чтобы вернуть курсор в позицию, в которой он находился до вставки, используется метод

```
void moveToCurrentRow() throws SQLException
```

Следующий пример модифицирует БД из примера п. 10.4 посредством изменения набора данных.

```
Statement s = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
ResultSet r = s.executeQuery("SELECT Id, Name, Telephone, Age "
        + "FROM people");
// Добавление единицы к возрасту всех людей
while(r.next())
{
    r.updateInt(4, r.getInt(4) + 1);
    r.updateRow();
}
// Добавление новой записи
r.moveToInsertRow();
r.updateString("Name", "Смирнов");
r.updateObject("Telephone", "26–08–58");
r.updateInt("Age", 43);
r.insertRow();
r.moveToCurrentRow();
s.close();
```

10.7. Использование прекомпилированных запросов. В тех случаях, когда осуществляется множество однотипных запросов к БД, отличающихся лишь параметрами, для повышения быстродействия можно использовать прекомпилированные запросы. Прекомпилированным запросам соответствуют объекты классов, реализующих интерфейс **PreparedStatement**. Создаются такие объекты методом

```
PreparedStatement prepareStatement(String sql) throws SQLException
PreparedStatement prepareStatement(String sql, int resultSetType,
    int resultSetConcurrency, int resultSetHoldability) throws SQLException
```

на объекте-соединении. Эти методы принимают на входе строку-шаблон запроса, содержащую знаки вопроса на месте параметров, например,

```
INSERT INTO people(Name, Telephone, Age) VALUES(?, ?, ?)
```

Значения параметров задаются посредством вызова методов

```
void set***(int parameterIndex, *** x) throws SQLException
void setObject(int parameterIndex, Object x) throws SQLException
```

Вместо *** подставляется имя типа данных Java, соответствующее типу данных SQL параметра (см. табл. 10.2). Аргумент **parameterIndex** определяет номер задаваемого параметра прекомпилированного запроса. *Нумерация параметров начинается с единицы!*

После того как значения параметров заданы, запрос выполняется посредством вызова одного из методов:

```
boolean execute() throws SQLException
ResultSet executeQuery() throws SQLException
int executeUpdate() throws SQLException
```

аналогичных одноимённым методам интерфейса **Statement** (см. п. 10.4).

Следующий фрагмент заполняет таблицу из примера п. 10.4 данными, хранящимися в массиве объектов, используя при этом прекомпилированный запрос.

```
PreparedStatement s = conn.prepareStatement(
        "INSERT INTO people(Name, Telephone, Age) VALUES(?, ?, ?)");
Object[][] data = {
        { "Иванов", "12–02–00", 37 },
        { "Петров", "43–55–47", 26 },
        { "Сидоров", "43–88–90", 59 } };
for(int i = 0; i < data.length; ++i)
{
    for(int j = 0; j < data[i].length; ++j)
        s.setObject(j + 1, data[i][j]);
    s.execute();
}
s.close();
```

10.8. Управление транзакциями. По умолчанию действует режим автоматической фиксации транзакций. Это означает, что каждый запрос к БД рассматривается как отдельная автоматически фиксируемая транзакция. Однако этот режим можно отключить, вызвав метод

```
void setAutoCommit(boolean autoCommit) throws SQLException
```

В этом случае все транзакции должны фиксироваться явно посредством вызова метода

```
void commit() throws SQLException
```

либо отклоняться посредством вызова метода

```
void rollback() throws SQLException
```

Если в процессе выполнения транзакции происходит ошибка, транзакция автоматически отклоняется.

В JDBC не требуется явно определять начало транзакции. Первая транзакция начинается в момент вызова метода **setAutoCommitMode()**, все последующие — непосредственно после вызова методов **commit()** или **rollback()** для фиксации или отклонения предыдущей транзакции.

11. Сетевое программирование

11.1. Обзор средств сетевого программирования. Java поддерживает широкий спектр средств и технологий разработки сетевых приложений. Сюда входят: средства непосредственной передачи данных с использованием протоколов TCP и UDP, средства поддержки протоколов уровня приложений http, https, ftp и др., технологии, расширяющие функциональность web-страниц на стороне клиента (апплеты), а также функциональность web-сервера (сервлеты и JSP), технология удалённого вызова процедур RMI (Remote Method Invocation), а также технология построения распределённых приложений Enterprise Java Beans.

Настоящая глава представляет собой лишь краткое введение в сетевое программирование в Java. В ней рассматриваются средства взаимодействия приложений посредством сокетов TCP/IP, а также средства получения web-ресурсов с использованием протоколов уровня приложений. Все они размещаются в пакете **java.net**.

Более детальное описание технологий разработки сетевых приложений можно найти в книгах [1, 3, 4, 5].

11.2. Класс InetAddress. Класс **InetAddress** инкапсулирует IP-адрес и доменное имя хоста. Он не имеет видимых (**public**) конструкторов, и для создания экземпляров этого класса используются следующие производственные методы:

```
static InetAddress getByName(String host) throws UnknownHostException
static InetAddress getByAddress(byte[] addr) throws UnknownHostException
static InetAddress getLocalHost() throws UnknownHostException
```

Первый из них создаёт экземпляр по доменному имени хоста (например, *www.uniyar.ac.ru*), второй — по его IP-адресу, задаваемому массивом из четырёх чисел. Аналогичное поведение можно получить передав первому методу вместо доменного имени строку из четырёх чисел, разделённых точками (например, "193.233.51.122"). Третий метод создаёт экземпляр, инкапсулирующий адрес локального хоста (как правило, 127.0.0.1). Если для создания экземпляра используется доменное имя хоста, то метод **getByName()** осуществляет запрос к DNS-серверу, для того чтобы осуществить преобразование этого имени к IP-адресу. Если

требуемый хост не найден или сетевое соединение неактивно, выбрасывается исключение типа **UnknownHostException**. Это же исключение выбрасывается при возникновении любых других ошибок при выполнении описанных методов.

11.3. TCP-сокеты. *Сокет* — это абстракция, используемая для представления «гнезда», в которое включается «кабель», соединяющий хосты в сети TCP/IP. Поскольку один и тот же хост может предоставлять различные сервисы, для идентификации определённой службы используется номер *порта*, к которому осуществляется подключение. Номера от 1 до 1024 зарезервированы для системных служб.

Класс **Socket** представляет собой класс TCP-сокетов, используемых для подключения к удалённому или локальному хосту. Для создания экземпляров этого класса используются следующие конструкторы:

```
Socket(String host, int port) throws UnknownHostException, IOException
Socket(InetAddress address, int port) throws IOException
```

Аргументами конструктора являются строка, содержащая доменное имя хоста, или объект типа **InetAddress**, инкапсулирующий его IP-адрес, а также номер порта хоста, к которому производится подключение. При создании объекта-сокета сразу же предпринимается попытка установления соединения с заданным хостом. Если она заканчивается неудачей, выбрасывается исключение.

Когда соединение установлено, методы

```
InputStream getInputStream() throws IOException
OutputStream getOutputStream() throws IOException
```

возвращают байтовые потоки ввода и вывода (см. п. 7.1), которые могут быть использованы для получения и отправления данных хосту.

После использования сокет должен быть обязательно закрыт посредством вызова метода

```
void close() throws IOException
```

Отметим, что экземпляры класса **Socket** используются для установления соединения как на стороне клиента, так и на стороне сервера. При этом взаимодействие клиента с сервером на стороне клиента, как правило, сводится просто к созданию экземпляра класса **Socket**, получению потоков ввода/вывода и обмену информацией с сервером через эти потоки.

11.4. Установление соединения на стороне сервера. Клиент-серверное взаимодействие на стороне сервера обычно состоит из этапов прослушивания некоторого порта (через этот порт к серверу могут обращаться клиенты), установления соединения и обмена данными с клиентом. Вторую из этих задач выполняет рассмотренный выше класс **Socket** (как уже отмечалось выше, клиентские и серверные сокеты между собой не различаются), третью — классы потоков ввода/ вывода. Для решения первой задачи используется специальный класс **ServerSocket**. Название этого класса является неудачным, поскольку никакого сокета он не инкапсулирует. Более точно его было бы назвать «слушателем порта». Для создания экземпляров класса используется конструктор

```
ServerSocket(int port)
```

принимающий на входе номер порта, подлежащего прослушиванию. Основным методом класса **ServerSocket** является метод

```
Socket accept() throws IOException
```

который блокирует поток выполнения программы до тех пор, пока какой-либо клиент не запросит соединения с сервером. При этом автоматически устанавливается соединение и возвращается объект-сокет, через который осуществляется соединение. Обычно сервер вызывает метод **accept()** в цикле, передавая объекты-сокеты другим потокам выполнения, которые осуществляют обслуживание клиентов.

По окончании работы необходимо всегда закрывать объект-слушатель с помощью метода

```
void close() throws IOException
```

11.5. Пример. Данный пример демонстрирует простейшее клиент-серверное приложение. «Математический сервер» принимает от клиента два числа, вычисляет их сумму и возвращает её клиенту.

```
import java.io.*;
import java.net.*;
/** Класс "Математический сервер" */
public class MathServer
{
    /** Номер порта, который слушает сервер */
    public static final int PORT = 21370;
    /** Точка входа в программу */
    public static void main(String[] args) throws IOException
```

```
{
    ServerSocket serv = new ServerSocket(PORT);
    try
    {
        // Главный цикл сервера
        while(true)
        {
            // Соединение и получение потоков ввода/вывода
            Socket s = serv.accept();
            try
            {
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(s.getInputStream()));
                PrintWriter out = new PrintWriter(
                        new OutputStreamWriter(s.getOutputStream()),
                        true);
                // Получение данных, обработка и возврат результата
                try
                {
                    if(!in.ready())
                        throw new NumberFormatException();
                    int a = Integer.parseInt(in.readLine());
                    if(!in.ready())
                        throw new NumberFormatException();
                    int b = Integer.parseInt(in.readLine());
                    out.println(a + b);
                    in.close();
                    out.close();
                }
                catch(NumberFormatException e)
                {
                    out.println("Неверный формат входных данных");
                }
            }
            catch(Exception e)
            {
                System.err.println("Ошибка ввода/вывода");
            }
            finally
            {
                s.close(); // гарантированное закрытие клиентского сокета
            }
```

```
                }
            }
            finally
            {
                serv.close(); // гарантированное закрытие слушателя порта
            }
        }
    }
```

Сервер открывает порт 21370, после чего входит в бесконечный цикл обслуживания клиентов (в реальных серверах следует предусматривать штатный способ завершения работы). Как только поступает запрос от клиента, метод **accept()** открывает соответствующий сокет. Сервер читает числа, переданные ему клиентом, вычисляет их сумму и возвращает её клиенту. Если клиент переслал серверу данные в неверном формате (например, одно число вместо двух, либо переданы не числа), сервер посылает клиенту сообщение об ошибке. Если в процессе ввода/вывода возникает ошибка, сервер выводит сообщение в стандартный поток ошибки. Независимо от того, произошла ошибка или нет, клиентский сокет закрывается и сервер переходит к ожиданию следующего запроса.

Если порт не открылся или метод **accept()** завершился неуспешно, соответствующее исключение будет перехвачено обработчиком по умолчанию и работа сервера завершится.

Клиент «математического сервера» представляет собой консольное приложение. Входные данные передаются клиенту через аргументы командной строки. Предполагается, что клиент выполняется на той же машине, что и сервер, однако такое поведение легко изменить путём модификации адреса, передаваемого методу **getByName()**.

```
import java.io.*;
import java.net.*;
/** Класс "Клиент математического сервера" */
public class MathClient
{
    public static void main(String[] args) throws IOException
    {
        if(args.length != 2)
        {
            System.out.println("Неверный вызов программы");
            return;
        }
```

```
            // Соединение и получение потоков ввода/вывода
            InetAddress addr = InetAddress.getByName("127.0.0.1");
            Socket s = new Socket(addr, MathServer.PORT);
            try
            {
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(s.getInputStream()));
                PrintWriter out = new PrintWriter(
                        new OutputStreamWriter(s.getOutputStream()), true);
                // Отправка данных серверу
                out.println(args[0] + "\n" + args[1]);
                // Получение ответа и печать его на экран
                System.out.println(in.readLine());
            }
            finally
            {
                s.close(); // гарантированное закрытие клиентского сокета
            }
        }
    }
```

11.6. Поддержка протоколов прикладного уровня. Одним из наиболее важных классов, поддерживающих сетевые протоколы прикладного уровня, является класс **URL**. Он инкапсулирует URL ресурса World Wide Web, а также позволяет получать содержимое этого ресурса по одному из следующих протоколов: http, https, ftp, file, jar.

Для создания экземпляров класса, как правило, используется конструктор

```
URL(String url) throws MalformedURLException
```

В качестве аргумента конструктору передаётся URL ресурса, обязательно включающий имя протокола (например, *http://www.uniyar.ac.ru*).

Для установления соединения и открытия входного потока, связанного с требуемым ресурсом, используется метод

```
InputStream openStream() throws IOException
```

Он возвращает поток, который можно использовать для получения содержимого запрошенного ресурса.

Следующий пример демонстрирует использование класса **URL**. Он читает содержимое ресурса, определённого URL, и записывает его в

файл. URL ресурса и имя выходного файла задаются параметрами входной строки.

```
import java.io.*;
import java.net.*;

class GetFile
{
    public static void main(String args[])
    {
        if(args.length != 2)
        {
            System.out.println("Неверный вызов программы");
            return;
        }
        try
        {
            URL url = new URL(args[0]);
            String filename = args[1];
            InputStream in = url.openStream();
            FileOutputStream out = new FileOutputStream(filename);
            int b;
            while((b = in.read()) != –1)
                out.write(b);
            in.close();
            out.close();
        }
        catch(IOException e)
        {
            System.err.println("Ошибка ввода/вывода");
        }
    }
}
```

Литература

Основная литература

[1] Эккель, Б. Философия Java / Б. Эккель. — СПб.: Питер, 2003. — 976 с.

[2] Ноутон, П. Java 2: Наиболее полное руководство / П. Ноутон, Г. Шилдт. — СПб.: БХВ-Петербург, 2005. — 1072 с.

[3] Тейт, Б. Горький вкус Java / Б. Тейт. — СПб.: Питер, 2003. — 333 с.

[4] Риккарди, Г. Системы баз данных. Теория и практика использования Internet в среде Java / Г. Риккарди. — М.: Вильямс, 2001. — 480 с.

[5] Дейтел, Х. М. Технологии программирования на Java 2 / Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. — М.: Бином-Пресс, 2003. — 672 с.

Дополнительная литература

[6] Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка / Дж. Рамбо, М. Блаха. — СПб.: Питер, 2006. — 544 с.

[7] Гранд, М. Шаблоны проектирования в Java / М. Гранд. — М.: Новое знание, 2004. — 559 с.

[8] Лафоре, Р. Объектно-ориентированное программирование в C++ / Р. Лафоре. — СПб.: Питер, 2004. — 923 с.

[9] Портянкин, И. А. Swing: Эффектные пользовательские интерфейсы / И. А. Портянкин. — СПб.: Питер, 2005. — 524 с.

[10] Фридл, Дж. Регулярные выражения / Дж. Фридл. — СПб.: Питер, 2003. — 464 с.

[11] Bracha, G. Generics in the Java Programming Language. *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*

Учебное издание

Парамонов Илья Вячеславович

Язык программирования Java и Java-технологии

Учебное пособие

Редактор, корректор В. Н. Чулкова
Компьютерный набор, вёрстка И. В. Парамонова

Подписано в печать 25.10.2006 г. Формат 60×84/16.
Бумага тип. Усл. печ. л. 5,35. Уч.-изд. л. 5,0.
Тираж 100 экз. Заказ

Оригинал-макет подготовлен в редакционно-издательском отделе
Ярославского государственного университета.

Отпечатано на ризографе ЯрГУ.

Ярославский государственный университет
150000, Ярославль, ул. Советская, 14.