Петр Карабин

Язык программирования

Java:

Создание интерактивных приложений для Internet

УДК 681.3
ББК 32.973.26-018.2
К216

Карабин П.Л.

К216 Язык программирования Java: Создание интерактивных приложений для Internet. - М.: Бук-пресс, 2006. - 224 с. - (Хитрости и тонкости).

ISBN 5-8321-0143-9

Учебное пособие по языку программирования Java. Создание языка Java — это один из самых значительных шагов вперед в области разработки сред программирования за последние 20 лет. Язык HTML (Hypertext Markup Language — язык разметки гипертекста) был необходим для статического размещения страниц во «Всемирной паутине» WWW (World Wide Web). Язык Java потребовался для качественного скачка в создании интерактивных продуктов для Internet.

УДК 681.3
ББК 32.973.26-018.2

ISBN 5-8321-0143-9

Содержание

Java

Введение в язык Java

Исходный файл на языке Java — это текстовый файл, содержащий в себе одно или несколько описаний классов. Транслятор Java предполагает, что исходный текст программ хранится в файлах с расширениями Java. Получаемый в процессе трансляции код для каждого класса записывается в отдельном выходном файле, с именем совпадающем с именем класса, и расширением **class**.

Начнем с упражнения: оттранслируем и запустим каноническую программу «Hello World». После этого рассмотрим все существенные лексические элементы, воспринимаемые Java-транслятором: пробелы, комментарии, ключевые слова, идентификаторы, литералы, операторы и разделители.

Итак, вот ваша первая Java-программа:

```
class HelloWorld {
public static void main (String
args []) {
System. out. println ("Hello
World");
```



```
}
}
```

Приведенный выше текст примера надо записать в файл HelloWorld.java. Обязательно проверьте соответствие прописных букв в имени файла тому же в названии содержащегося в нем класса. Для того, чтобы оттранслировать этот пример необходимо запустить транслятор Java — **javac**, указав в качестве параметра имя файла с исходным текстом:

```
C: \> javac HelloWorld.Java
```

Транслятор создаст файл **HelloWorld.class** с независимым от процессора байт-кодом нашего примера. Для того, чтобы исполнить полученный код, необходимо иметь среду времени выполнения языка Java (в нашем случае это программа java), в которую надо загрузить новый класс для исполнения. Подчеркнем, что указывается имя класса, а не имя файла, в котором этот класс содержится.

```
C: > java HelloWorld
Hello World
```

Полезного сделано мало, однако мы убедились, что установленный Java-транслятор и среда времени выполнения работают.

Шаг за шагом

Конечно, HelloWorld — это тривиальный пример. Однако даже такая простая программа новичку в языке Java может показаться пугающе сложной, поскольку она знакомит вас с массой новых понятий и деталей синтаксиса языка. Давайте внимательно пройдемся по каждой строке нашего первого примера, анализируя те элементы, из которых состоит Java-программа.

Строка 1

```
class HelloWorld {
```

В этой строке использовано зарезервированное слово **class**. Оно говорит транслятору, что мы собираемся описать новый класс. Полное описание класса располагается между открывающей фигурной скобкой в первой строке и парной ей закрывающей фигурной скобкой в строке 5. Фигурные скобки в Java используются точно так же, как в языках C и C++.

Строка 2

```
public static void main (String
args []) {
```

Такая, на первый взгляд, чрезмерно сложная строка примера является следствием важного требования, заложенного при разработке языка Java. Дело в том, что в Java отсутствуют глобальные функции. Поскольку

подобные строки будут встречаться в большинстве примеров первой части книги, давайте пристальнее рассмотрим каждый элемент второй строки.

public

Это — модификатор доступа, который позволяет программисту управлять видимостью любого метода и любой переменной. В данном случае модификатор доступа **public** означает, что метод **main** виден и доступен любому классу.

static

Следующее ключевое слово — **static**. С помощью этого слова объявляются методы и переменные класса, используемые для работы с классом в целом. Методы, в объявлении которых использовано ключевое слово **static**, могут непосредственно работать только с локальными и статическими переменными.

void

У вас нередко будет возникать потребность в методах, которые возвращают значение того или иного типа: например, **int** для целых значений, **float** — для вещественных или имя класса для типов данных, определенных программистом. В нашем случае нужно просто вывести на экран строку, а возвращать значение из метода **main** не требуется. Именно поэтому и был использован модификатор **void**.

main

Наконец, мы добрались до имени метода **main**. Здесь нет ничего необычного, просто все существующие реализации Java-интерпретаторов, получив команду интерпретировать класс, начинают свою работу с вызова метода **main**. Java-транслятор может оттранслировать класс, в котором нет метода **main**. А вот Java-интерпретатор запускать классы без метода **main** не умеет.

Все параметры, которые нужно передать методу, указываются внутри пары круглых скобок в виде списка элементов, разделенных символами «**;**» (точка с запятой). Каждый элемент списка параметров состоит из разделенных пробелом типа и идентификатора. Даже если у метода нет параметров, после его имени все равно нужно поставить пару круглых скобок. В примере, который мы сейчас обсуждаем, у метода main только один параметр, правда довольно сложного типа.

Элемент **String args[]** объявляет параметр с именем **args**, который является массивом объектов — представителей класса **String**. Обратите внимание на квадратные скобки, стоящие после идентификатора **args**. Они говорят о том, что мы имеем дело с массивом, а не с одиночным элементом указанного типа.

Строка 3

```
System. out. prlntln("Hello World!");
```


В этой строке выполняется метод **println** объекта **out**. Объект **out** объявлен в классе **OutputStream** и статически инициализируется в классе **System**.

Закрывающей фигурной скобкой в строке 4 заканчивается объявление метода **main**, а такая же скобка в строке 5 завершает объявление класса **HelloWorld**.

Лексические основы

Теперь, когда мы подробно рассмотрели минимальный Java-класс, давайте вернемся назад и рассмотрим общие аспекты синтаксиса этого языка. Программы на Java — это набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей.

Пробелы

Java — язык, который допускает произвольное форматирование текста программ. Для того, чтобы программа работала нормально, нет никакой необходимости выравнивать ее текст специальным образом. Например, класс HelloWorld можно было записать в двух строках или любым другим способом, который придется вам по душе. И он будет работать точно так же при условии, что между отдельными лексемами (между которыми нет операторов или разделителей) имеется по крайней мере по одному пробелу,

символу табуляции или символу перевода строки.

Комментарии

Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования. Комментарии, занимающие одну строку, начинаются с символов // и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:

```
a = 42; // если 42 - ответ, то
каков же был вопрос?
```

Для более подробных пояснений вы можете воспользоваться комментариями, размещенными на нескольких строках, начав текст комментариев символами **/*** и закончив символами ***/**. При этом весь текст между этими парами символов будет расценен как комментарий и транслятор его проигнорирует.

```
/*
* Этот код несколько замысловат...
* Попробую объяснить:
* ….
```



```
*/
```

Третья, особая форма комментариев, предназначена для сервисной программы **javadoc**, которая использует компоненты Java-транслятора для автоматической генерации документации по интерфейсам классов. Соглашение, используемое для комментариев этого вида, таково: для того, чтобы разместить перед объявлением открытого (public) класса, метода или переменной документирующий комментарий, нужно начать его с символов **/**** (косая черта и две звездочки). Заканчивается такой комментарий точно так же, как и обычный комментарий — символами ***/**. Программа **javadoc** умеет различать в документирующих комментариях некоторые специальные переменные, имена которых начинаются с символа **@**. Вот пример такого комментария:

```
/**
* Этот класс умеет делать
замечательные вещи. Советуем
* всякому, кто захочет написать еще
более совершенный класс,
* взять его в качестве базового.
* @see Java. applet. Applet
* ©author Patrick Naughton
* @version 1. 2
*/
```

```
class CoolApplet extends Applet {
/**
* У этого метода два параметра:
* @param key – это имя параметра.
* @param value – это значение
параметра с именем key.
*/ void put (String key, Object
value) {
```

Зарезервированные ключевые слова

Зарезервированные ключевые слова — это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы и средства управления выполнением программы. Эти ключевые слова совместно с синтаксисом операторов и разделителей входят в описание языка Java. Они могут применяться только по назначению, их нельзя использовать в качестве идентификаторов для имен переменных, классов или методов. На сегодняшний день в языке Java имеется 59 зарезервированных слов:

abstract

boolean

break

byte

byvalue

case

cast

catch

char

class

const

continue

default

do

double

else

extends

false

final

finally

float

for

future

generic

goto

if

implements

import

inner

instanceof
int
interface
long
native
new
null
operator
outer
package
private
protected
public
rest
return
short
static
super
switch
syncronized
this
throw
throws
transient
true

try
var
void
volatile
while

Отметим, что слова **byvalue**, **cast**, **const**, **future**, **generic**, **goto**, **inner**, **operator**, **outer**, **rest**, **var** зарезервированы в Java, но пока не используются. Кроме этого, в Java есть зарезервированные имена методов (эти методы наследуются каждым классом, их нельзя использовать, за исключением случаев явного переопределения методов класса **Object**).

Зарезервированные имена методов Java

clone
equals
finalize
getClass
hashCode
notify
notifyAll
toString
wait

Идентификаторы

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _ (подчеркивание) и **$** (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами. Java — язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE — различные идентификаторы.

Литералы

Константы в Java задаются их литеральным представлением. Целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода.

Целые литералы

Целые числа — это тип, используемый в обычных программах наиболее часто. Любое целочисленное значение, например, 1, 2, 3, 42 — это целый литерал. В данном примере приведены десятичные числа, то есть числа с основанием 10 — именно те, которые мы повседневно используем вне мира компьютеров. Кроме десятичных, в качестве целых литералов могут использоваться также числа с основанием 8 и 16 — восьмеричные и

шестнадцатеричные. Java распознает восьмеричные числа по стоящему впереди нулю. Нормальные десятичные числа не могут начинаться с нуля, так что использование в программе внешне допустимого числа 09 приведет к сообщению об ошибке при трансляции, поскольку 9 не входит в диапазон 0...7, допустимый для знаков восьмеричного числа. Шестнадцатеричная константа различается по стоящим впереди символам нуль-х (0х или 0X). Диапазон значений шестнадцатеричной цифры — 0...15, причем в качестве цифр для значений 10...15 используются буквы от A до F (или от a до f). С помощью шестнадцатеричных чисел вы можете в краткой и ясной форме представить значения, ориентированные на использование в компьютере, например, написав Oxffff вместо 65535.

Целые литералы являются значениями типа **int**, которое в Java хранится в 32-битовом слове. Если вам требуется значение, которое по модулю больше, чем приблизительно 2 миллиарда, необходимо воспользоваться константой типа **long**. При этом число будет храниться в 64-битовом слове. К числам с любым из названных выше оснований вы можете приписать справа строчную или прописную букву **L**, указав таким образом, что данное число относится к типу **long**. Например, Ox7fffffffffffffL или 9223372036854775807L — это значение, наибольшее для числа типа **long**.

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения, у которых есть дробная часть. Их можно записывать либо в обычном, либо экспоненциальном форматах. В обычном формате число состоит из некоторого количества десятичных цифр, стоящей после них десятичной точки, и следующих за ней десятичных цифр дробной части. Например, 2.0, 3.14159 и .6667 — это допустимые значения чисел с плавающей точкой, записанных в стандартном формате. В экспоненциальном формате после перечисленных элементов дополнительно указывается десятичный порядок. Порядок определяется положительным или отрицательным десятичным числом, следующим за символом E или e. Примеры чисел в экспоненциальном формате: 6.022e23, 314159E-05, 2e+100. В Java числа с плавающей точкой по умолчанию рассматриваются, как значения типа **double**. Если вам требуется константа типа **float**, справа к литералу надо приписать символ F или f. Если вы любитель избыточных определений — можете добавлять к литералам типа **double** символ D или d. Значения используемого по умолчанию типа **double** хранятся в 64-битовом слове, менее точные значения типа **float** — в 32-битовых.

Логические литералы

У логической переменной может быть лишь два значения — **true** (истина) и **false**

(ложь). Логические значения **true** и **false** не преобразуются ни в какое числовое представление. Ключевое слово **true** в Java не равно 1, а **false** не равно 0. В Java эти значения могут присваиваться только переменным типа **boolean** либо использоваться в выражениях с логическими операторами.

Символьные литералы

Символы в Java — это индексы в таблице символов UNICODE. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов (' '). Все видимые символы таблицы ASCII можно прямо вставлять внутрь пары апострофов: — 'a', 'z', '@'. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей.

Управляющая последовательность	Описание
\ddd	Восьмеричный символ (ddd)
\uxxxx	Шестнадцатеричный символ UNICODE (xxxx)
\'	Апостроф
\"	Кавычка

\\	Обратная косая черта
\r	Возврат каретки (carriage return)
\n	Перевод строки (line feed, new line)
\f	Перевод страницы (form feed)
\t	Горизонтальная табуляция (tab)
\b	Возврат на шаг (backspace)

Строчные литералы

Строчные литералы в Java выглядят точно также, как и во многих других языках — это произвольный текст, заключенный в пару двойных кавычек (""). Хотя строчные литералы в Java реализованы весьма своеобразно (Java создает объект для каждой строки), внешне это никак не проявляется. Примеры строчных литералов:

```
“Hello World!”;

¨две\строки; \ А это в кавычках\¨”.
```

Все управляющие последовательности, восьмеричные и шестнадцатеричные формы записи, которые определены для символьных литералов, работают точно так же и в строках. Строчные литералы в Java должны начинаться и заканчиваться в одной и той же строке

исходного кода. В этом языке, в отличие от многих других, нет управляющей последовательности для продолжения строкового литерала на новой строке.

Операторы

Оператор — это нечто, выполняющее некоторое действие над одним или двумя аргументами и выдающее результат. Синтаксически операторы чаще всего размещаются между идентификаторами и литералами.

Операторы языка Java

```
+
+=
-
-=
*
*=
/
/=
|
|=
^
^=
&
&=
```

%
%=
>
>=
<
<=
!
!=
++
--
>>
>>=
<<
<<=
>>>
>>>=
&&
||
==
=
~
?:
instanceof
[ ]

Разделители

Лишь несколько групп символов, которые могут появляться в синтаксически правильной Java-программе, все еще остались неназванными.
Это — простые разделители, которые влияют на внешний вид и функциональность программного кода.

( ) круглые скобки

Выделяют списки параметров в объявлении и вызове метода, также используются для задания приоритета операций в выражениях, выделения выражений в операторах управления выполнением программы, и в операторах приведения типов.

{ } фигурные скобки

Содержат значения автоматически инициализируемых массивов, также используются для ограничения блока кода в классах, методах и локальных областях видимости.

[ ] квадратные скобки

Используются в объявлениях массивов и при доступе к отдельным элементам массива.

; точка с запятой

Разделяет операторы.

, запятая

Разделяет идентификаторы в объявлениях переменных, также используется

для связи операторов в заголовке цикла **for**.

. точка

Отделяет имена пакетов от имен подпакетов и классов, также используется для отделения имени переменной или метода от имени переменной.

Переменные

Переменная — это основной элемент хранения информации в Java-программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла **for**, либо это может быть переменная экземпляра класса, доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок.

Объявление переменной

Основная форма объявления переменной такова:

```
тип идентификатор [ = значение] [,
идентификатор
[ = значение 7...];
```

Тип — это либо один из встроенных типов, то есть, **byte**, **short**, **int**, **long**, **char**, **float**, **double**, **boolean**, либо имя класса или

интерфейса. Ниже приведено несколько примеров объявления переменных различных типов. Обратите внимание на то, что некоторые примеры включают в себя инициализацию начального значения. Переменные, для которых начальные значения не указаны, автоматически инициализируются нулем.

```
int a, b, c;
```

Объявляет три целых переменных a, b, c.

```
int d = 3, e, f = 5;
```

Объявляет еще три целых переменных, инициализирует d и f.

```
byte z = 22;
```

Инициализирует z.

```
double pi = 3. 14159;
```

Объявляет число пи (не очень точное, но все таки пи).

```
char x = 'x';
```

Переменная x получает значение 'x'.

В приведенном ниже примере создаются три переменные, соответствующие сторонам прямоугольного треугольника, а затем с помощью теоремы Пифагора вычисляется длина гипотенузы, в данном случае числа 5, величины гипотенузы классического прямоугольного треугольника со сторонами 3-4-5.

```
class Variables {
public static void main (String
args []) {
double a = 3;
double b = 4;
double c;
c = Math.sqrt (a* a + b* b);
System.out.println ("c = "+ c); } }
```

Типы

Простые типы

Простые типы в Java не являются объектно-ориентированными, они аналогичны простым типам большинства традиционных языков программирования. В Java имеется восемь простых типов: **byte**, **short**, **int**, **long**, **char**, **float**, **double** и **boolean**. Их можно разделить на четыре группы:

Целые

К ним относятся типы **byte**, **short**, **int** и **long**. Эти типы предназначены для целых чисел со знаком.

Типы с плавающей точкой float и double

Они служат для представления чисел, имеющих дробную часть.

Символьный тип char

Этот тип предназначен для представления элементов из таблицы символов, например, букв или цифр.

Логический тип boolean

Это специальный тип, используемый для представления логических величин.

В Java, в отличие от некоторых других языков, отсутствует автоматическое приведение типов. Несовпадение типов приводит не к предупреждению при трансляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций.

Целые числа

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка — знаковые. Например, если значение переменной типа **byte** равно в шестнадцатеричном виде 0x80, то это — число -1.

Единственная реальная причина использования беззнаковых чисел — это использование иных, по сравнению со знаковыми числами, правил манипуляций с битами при выполнении операций сдвига. Пусть, например, требуется сдвинуть вправо битовый массив **mask**, хранящийся в целой переменной и избежать при этом расширения знакового разряда, заполняющего старшие

биты единицами. Стандартный способ выполнения этой задачи в C — ((unsigned) mask) >> 2. В Java для этой цели введен новый оператор беззнакового сдвига вправо. Приведенная выше операция записывается с его помощью в виде **mask>>>2**.

Отсутствие в Java беззнаковых чисел вдвое сокращает количество целых типов. В языке имеется 4 целых типа, занимающих 1, 2, 4 и 8 байтов в памяти. Для каждого типа — **byte**, **short**, **int** и **long**, есть свои естественные области применения.

byte

Тип **byte** — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла.

```
byte b;
byte c = 0x55;
```

Если речь не идет о манипуляциях с битами, использования типа **byte**, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип **int**.

short

short — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java

тип, поскольку он определен, как тип, в котором старший байт стоит первым.

```
short s;
short t = 0x55aa;
```

Случилось так, что на ЭВМ различных архитектур порядок байтов в слове различается, например, старший байт в двухбайтовом целом **short** может храниться первым, а может и последним. Первый случай имеет место в архитектурах SPARC и Power PC, второй — для микропроцессоров Intel x86. Переносимость программ Java требует, чтобы целые значения одинаково были представлены на ЭВМ разных архитектур.

int

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков. В ближайшие годы этот тип будет прекрасно соответствовать машинным словам не только 32-битовых процессоров, но и 64-битовых с поддержкой быстрой конвейеризации для выполнения 32-битного кода в режиме совместимости. Всякий раз, когда в одном выражении фигурируют переменные типов **byte**, **short**, **int** и целые

литералы, тип всего выражения перед завершением вычислений приводится к **int**.

```
int i;
int j = 0x55aa0000;
```

long

Тип **long** предназначен для представления 64-битовых чисел со знаком. Его диапазон допустимых значений достаточно велик даже для таких задач, как подсчет числа атомов во вселенной.

```
long m;
long n = 0x55aa000055aa0000;
```

Не надо отождествлять разрядность целочисленного типа с занимаемым им количеством памяти. Исполняющий код Java может использовать для ваших переменных то количество памяти, которое сочтет нужным, лишь бы только их поведение соответствовало поведению типов, заданных вами. Фактически, нынешняя реализация Java из соображений эффективности хранит переменные типа **byte** и **short** в виде 32-битовых значений, поскольку этот размер соответствует машинному слову большинства современных компьютеров (CM – 8 бит, 8086 – 16 бит, 80386/486 – 32 бит, Pentium – 64 бит).

Числа с плавающей точкой

Числа с плавающей точкой, часто называемые в других языках вещественными

числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754) набор типов для чисел с плавающей точкой — **float** и **double** и операторов для работы с ними. Характеристики этих типов приведены ниже.

Имя	Разрядность	Диапазон
double	64	1. 7e-308.. 1. 7e+ 308
float	32	3. 4e-038.. 3. 4e+ 038

float

В переменных с обычной, или одинарной точностью, объявляемых с помощью ключевого слова **float**, для хранения вещественного значения используется 32 бита.

```
float f;
float f2 = 3. 14F; // обратите
внимание на F, т.к. по умолчанию
все литералы double
```

double

В случае двойной точности, задаваемой с помощью ключевого слова **double**, для хранения значений используется 64 бита. Все трансцендентные математические функции, такие, как sin, cos, sqrt, возвращают результат типа **double**.

```
double d;
```

```
double pi = 3. 14159265358979323846;
```

Приведение типа

Приведение типов (type casting) — одно из неприятных свойств C++, тем не менее приведение типов сохранено и в языке Java. Иногда возникают ситуации, когда у вас есть величина какого-то определенного типа, а вам нужно ее присвоить переменной другого типа. Для некоторых типов это можно проделать и без приведения типа, в таких случаях говорят об автоматическом преобразовании типов. В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменной-приемника достаточно для хранения исходного значения. Такое преобразование происходит, например, при занесении литеральной константы или значения переменной типа **byte** или **short** в переменную типа **int**. Это называется расширением (**widening**) или повышением (**promotion**), поскольку тип меньшей разрядности расширяется (повышается) до большего совместимого типа. Размера типа **int** всегда достаточно для хранения чисел из диапазона, допустимого для типа **byte**, поэтому в подобных ситуациях оператора явного приведения типа не требуется. Обратное в большинстве случаев неверно, поэтому для занесения значения типа **int** в переменную типа **byte** необходимо использовать оператор

приведения типа. Эту процедуру иногда называют сужением (**narrowing**), поскольку вы явно сообщаете транслятору, что величину необходимо преобразовать, чтобы она уместилась в переменную нужного вам типа. Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. В приведенном ниже фрагменте кода демонстрируется приведение типа источника (переменной типа **int**) к типу приемника (переменной типа **byte**). Если бы при такой операции целое значение выходило за границы допустимого для типа **byte** диапазона, оно было бы уменьшено путем деления по модулю на допустимый для **byte** диапазон (результат деления по модулю на число — это остаток от деления на это число).

```
int a = 100;
byte b = (byte) a;
```

Автоматическое преобразование типов в выражениях

Когда вы вычисляете значение выражения, точность, требуемая для хранения промежуточных результатов, зачастую должна быть выше, чем требуется для представления окончательного результата.

```
byte a = 40;
```

```
byte b = 50;
byte c = 100;
int d = a* b / c;
```

Результат промежуточного выражения (a* b) вполне может выйти за диапазон допустимых для типа **byte** значений. Именно поэтому Java автоматически повышает тип каждой части выражения до типа **int**, так что для промежуточного результата (a* b) хватает места.

Автоматическое преобразование типа иногда может оказаться причиной неожиданных сообщений транслятора об ошибках. Например, показанный ниже код, хотя и выглядит вполне корректным, приводит к сообщению об ошибке на фазе трансляции. В нем мы пытаемся записать значение 50* 2, которое должно прекрасно уместиться в тип byte, в байтовую переменную. Но из-за автоматического преобразования типа результата в int мы получаем сообщение

об ошибке от транслятора — ведь при занесении int в byte может произойти потеря точности.

```
byte b = 50;
b = b* 2;
^ Incompatible type for =. Explicit
cast needed to convert int to byte.
```



```
(Несовместимый тип для =. Необходимо
явное преобразование int в byte)
```

Исправленный текст:

```
byte b = 50;
b = (byte) (b* 2);
```

что приводит к занесению в **b** правильного значения 100.

Если в выражении используются переменные типов **byte**, **short** и **int**, то во избежание переполнения тип всего выражения автоматически повышается до **int**. Если же в выражении тип хотя бы одной переменной — **long**, то и тип всего выражения тоже повышается до **long**. Не забывайте, что все целые литералы, в конце которых не стоит символ L (или l), имеют тип **int**.

Если выражение содержит операнды типа **float**, то и тип всего выражения автоматически повышается до **float**. Если же хотя бы один из операндов имеет тип **double**, то тип всего выражения повышается до **double**. По умолчанию Java рассматривает все литералы с плавающей точкой, как имеющие тип **double**. Приведенная ниже программа показывает, как повышается тип каждой величины в выражении для достижения соответствия со вторым операндом каждого бинарного оператора.

```
class Promote {
public static void main (String
```

```
args []) { byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d =.1234;
double result = (f* b) + (i/ c) -
(d* s);
System. out. println ((f* b)+ "+ "+
(i / c)+ " - " +
(d* s));
System. out. println ("result = "+
result);
}
}
```

Подвыражение f* b — это число типа **float**, умноженное на число типа **byte**. Поэтому его тип автоматически повышается до **float**. Тип следующего подвыражения i/c (int, деленный на char) повышается до **int**. Аналогично этому тип подвыражения d*s (double, умноженный на short) повышается до **double**. На следующем шаге вычислений мы имеем дело с тремя промежуточными результатами типов **float**, **int** и **double**. Сначала при сложении первых двух тип **int** повышается до **float** и получается результат типа **float**. При

вычитании из него значения типа **double** тип результата повышается до **double**. Окончательный результат всего выражения — значение типа **double**.

Символы

Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типа **char** в этом языке — 16 бит. В нем можно хранить десятки тысяч символов интернационального набора символов Unicode. Диапазон типа **char** — 0...65536. Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов.

```
char c;
char c2 = 0xf132;
char c3 = ' a';
char c4 = '\n';
```

Хотя величины типа **char** и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми. Это дает вам возможность сложить два символа вместе, или инкрементировать значение символьной переменной.

В приведенном ниже фрагменте кода мы, располагая базовым символом, прибавляем к

нему целое число, чтобы получить символьное представление нужной нам цифры.

```
int three = 3;
char one = '1';
char four = (char) (three+ one);
```

В результате выполнения этого кода в переменную four заносится символьное представление нужной нам цифры — '4'. Обратите внимание — тип переменной one в приведенном выше выражении повышается до типа **int**, так что перед занесением результата в переменную four приходится использовать оператор явного приведения типа.

Тип boolean

В языке Java имеется простой тип **boolean**, используемый для хранения логических значений. Переменные этого типа могут принимать всего два значения — **true** (истина) и **false** (ложь). Значения типа **boolean** возвращаются в качестве результата всеми операторами сравнения, например (a < b).

```
boolean done = false;
```


Пример создания переменных различных типов

Теперь, когда мы познакомились со всеми простыми типами, включая целые и вещественные числа, символы и логические переменные, давайте попробуем собрать всю информацию вместе. В приведенном ниже примере создаются переменные каждого из простых типов и выводятся значения этих переменных.

```
class SimpleTypes {
public static void main(String args
[]) {
byte b = 0x55;
short s = 0x55ff;
int i = 1000000;
long l = 0xffffffffL;
char c = ' a' ;
float f = .25f;
double d = .00001234;
boolean bool = true;
System.out.println("byte b = " +
b);
System.out.println("short s = " +s);
System.out.println("int i = " + i);
```

```
System.out.println("long l = " +
l);
System.out.println("char c = " +
c);
System.out.println("float f = " +
f);
System.out.println("double d = " +
d);
System.out.println("boolean bool = "
+ bool);
} }
```

Запустив эту программу, вы должны получить результат, показанный ниже:

```
C: \> java SimpleTypes
byte b = 85
short s = 22015
int i = 1000000
long l = 4294967295
char c = a
float f = 0.25
double d = 1.234e-005
boolean bool = true
```


Обратите внимание на то, что целые числа печатаются в десятичном представлении, хотя мы задавали значения некоторых из них в шестнадцатеричном формате.

Массивы

Для объявления типа массива используются квадратные скобки. В приведенной ниже строке объявляется переменная month_days, тип которой — «массив целых чисел типа int».

```
int month_days [];
```

Для того, чтобы зарезервировать память под массив, используется специальный оператор **new**. В приведенной ниже строке кода с помощью оператора **new** массиву month_days выделяется память для хранения двенадцати целых чисел.

```
month_days = new int [12];
```

Итак, теперь month_days — это ссылка на двенадцать целых чисел. Ниже приведен пример, в котором создается массив, элементы которого содержат число дней в месяцах года (невисокосного).

```
class Array {
public static void main (String
args []) {
int month_days[];
```

```
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " +
month_days[3] + " days.");
} }
```

При запуске эта программа печатает количество дней в апреле, как это показано ниже. Нумерация элементов массива в Java начинается с нуля, так что число дней в апреле — это month_days [3].

```
C: \> java Array
April has 30 days.
```

Имеется возможность автоматически инициализировать массивы способом, во

многом напоминающим инициализацию переменных простых типов. Инициализатор массива представляет собой список разделенных запятыми выражений, заключенный в фигурные скобки. Запятые отделяют друг от друга значения элементов массива. При таком способе создания массив будет содержать ровно столько элементов, сколько требуется для хранения значений, указанных в списке инициализации.

```
class AutoArray {
public static void main(String
args[]) {
int month_days[] = { 31, 28, 31,
30, 31, 30, 31, 31, 30, 31, 30, 31
};
System.out.println("April has " +
month_days[3] + " days.");
} }
```

В результате работы этой программы, вы получите точно такой же результат, как и от ее более длинной предшественницы.

Java строго следит за тем, чтобы вы случайно не записали или не попытались получить значения, выйдя за границы массива. Если же вы попытаетесь использовать в качестве индексов значения, выходящие за границы массива — отрицательные числа либо числа, которые больше или равны количеству

элементов в массиве, то получите сообщение об ошибке времени выполнения.

Многомерные массивы

На самом деле, настоящих многомерных массивов в Java не существует. Зато имеются массивы массивов, которые ведут себя подобно многомерным массивам, за исключением нескольких незначительных отличий. Приведенный ниже код создает традиционную матрицу из шестнадцати элементов типа **double**, каждый из которых инициализируется нулем. Внутренняя реализация этой матрицы — массив массивов **double**.

```
double matrix [][] = new double
[4][4];
```

Следующий фрагмент кода инициализирует такое же количество памяти, но память под вторую размерность отводится вручную. Это сделано для того, чтобы наглядно показать, что матрица на самом деле представляет собой вложенные массивы.

```
double matrix [][] = new double
[4][];
matrix [0] = new double[4];
matrix[1] = new double[4];
matrix[2] = new double[4], matrix[3]
= { 0, 1, 2, 3 };
```


В следующем примере создается матрица размером 4 на 4 с элементами типа **double**, причем ее диагональные элементы (те, для которых x==y) заполняются единицами, а все остальные элементы остаются равными нулю.

```
class Matrix {
public static void main(String
args[]) { double m[][];
m = new double[4][4];
m[0][0] = 1;
m[1][1] = 1;
m[2][2] = 1;
m[3][3] = 1;
System.out.println(m[0][0] +" "+
m[0][1] +" "+ m[0][2] +" "+
m[0][3]);
System.out.println(m[1][0] +" "+
m[1][1] +" "+ m[1][2] +" "+
m[1][3]);
System.out.println(m[2][0] +" "+
m[2][1] +" "+ m[2][2] +" "+
m[2][3]);
System.out.println(m[3][0] +" "+
m[3][1] +" "+ m[3][2] +" "+
m[3][3]);
}
```

```
}
```

Запустив эту программу, вы получите следующий результат:

```
C : \> Java Matrix
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Обратите внимание — если вы хотите, чтобы значение элемента было нулевым, вам не нужно его инициализировать, это делается автоматически.

Для задания начальных значений массивов существует специальная форма инициализатора, пригодная и в многомерном случае. В программе, приведенной ниже, создается матрица, каждый элемент которой содержит произведение номера строки на номер столбца. Обратите внимание на тот факт, что внутри инициализатора массива можно использовать не только литералы, но и выражения.

```
class AutoMatrix {
public static void main(String
args[]) { double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 }, { 0*1,
1*1, 2*1, 3*1 }, { 0*2. 1*2, 2*2,
3*2 },
```



```
{ 0*3, 1*3. 2*3, 3*3 } };
System.out.println(m[0][0] +" "+
m[0][1] +" "+ m[0][2] +" "+
m[0][3]);
System.out.println(m[1][0] +"
"+m[1][1] +" "+ m[1][2] +" "+
m[1][3]);
System.out.println(m[2][0] +"
"+m[2][1] +" "+ m[2][2] +" "+
m[2][3]);
System.out.println(m[3][0] +"
"+m[3][1] +" "+ m[3][2] +" "+
m[3][3]);
} }
```

Запустив эту программу, вы получите следующий результат:

```
C: \> Java AutoMatrix
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
```

Операторы

Операторы в языке Java — это специальные символы, которые сообщают транслятору о том, что вы хотите выполнить

операцию с некоторыми операндами. Некоторые операторы требуют одного операнда, их называют унарными. Одни операторы ставятся перед операндами и называются префиксными, другие — после, их называют постфиксными операторами. Большинство же операторов ставят между двумя операндами, такие операторы называются инфиксными бинарными операторами. Существует тернарный оператор, работающий с тремя операндами.

В Java имеется 44 встроенных оператора. Их можно разбить на 4 класса — арифметические, битовые, операторы сравнения и логические.

Арифметические операторы

Арифметические операторы используются для вычислений так же как в алгебре. Допустимые операнды должны иметь числовые типы. Например, использовать эти операторы для работы с логическими типами нельзя, а для работы с типом **char** можно, поскольку в Java тип **char** — это подмножество типа **int**.

+
: Сложение

+ =
: Сложение с присваиванием

-

: Вычитание (также унарный минус)

-=
: Вычитание с присваиванием

*
: Умножение

* =
: Умножение с присваиванием

/
: Деление

/=
: Деление с присваиванием

%
: Деление по модулю

%=
: Деление по модулю с присваиванием

++
: Инкремент

--
: Декремент

Четыре арифметических действия

Ниже, в качестве примера, приведена простая программа, демонстрирующая использование операторов.

Обратите внимание на то, что операторы работают как с целыми литералами, так и с переменными.

```
class BasicMath   { public static
void int a = 1 + 1;
int   b = a *   3;
main(String args[]) {
int c = b / 4;
int d = b - a;
int e = -d;
System.out.println("a =   " +
a);
System.out.println("b =   " +
b);
System.out.println("c =   " +
c);
System.out.println("d =   " +
d);
System.out.println("e =   " +
e);
} }
```

Исполнив эту программу, вы должны получить приведенный ниже результат:

```
C: \> java BasicMath
a = 2
```



```
b = 6
c = 1
d = 4
e = -4
```

Оператор деления по модулю

Оператор деления по модулю, или оператор mod, обозначается символом %. Этот оператор возвращает остаток от деления первого операнда на второй. В отличие от C++, функция mod в Java работает не только с целыми, но и с вещественными типами. Приведенная ниже программа иллюстрирует работу этого оператора.

```
class Modulus {
public static void main (String
args []) {
int x = 42;
double y = 42.3;
System.out.println("x mod 10 = " +
x % 10);
System.out.println("y mod 10 = " +
y % 10);
} }
```

Выполнив эту программу, вы получите следующий результат:

```
C:\> Modulus
```

```
x mod 10 = 2
y mod 10 = 2.3
```

Арифметические операторы присваивания

Для каждого из арифметических операторов есть форма, в которой одновременно с заданной операцией выполняется присваивание. Ниже приведен пример, который иллюстрирует использование подобной разновидности операторов.

```
class OpEquals {
public static void main(String
args[]) {
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
} }
```


А вот и результат, полученный при запуске этой программы:

```
C:> Java OpEquals
a = 6
b = 8
c = 3
```

Инкремент и декремент

В C существует 2 оператора, называемых операторами инкремента и декремента (++ и --) и являющихся сокращенным вариантом записи для сложения или вычитания из операнда единицы. Эти операторы уникальны в том плане, что могут использоваться как в префиксной, так и в постфиксной форме. Следующий пример иллюстрирует использование операторов инкремента и декремента.

```
class IncDec {
public static void main(String
args[]) {
int a = 1;
int b = 2;
int c = ++b;
int d = a++;
c++;
System.out.println("a = " + a);
```

```
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
} }
```

Результат выполнения данной программы будет таким:

```
C:\ java IncDec
a = 2
b = 3
c = 4
d = 1
```

Целочисленные битовые операторы

Для целых числовых типов данных — **long**, **int**, **short**, **char** и **byte**, определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. Операторы битовой арифметики работают с каждым битом как с самостоятельной величиной.

~

Побитовое унарное отрицание (NOT)

&

Побитовое И (AND)

&=

Побитовое И (AND) с присваиванием

|

Побитовое ИЛИ (OR)

|=

Побитовое ИЛИ (OR) с присваиванием

^

Побитовое исключающее ИЛИ (XOR)

^=

Побитовое исключающее ИЛИ (XOR) с присваиванием

>>

Сдвиг вправо

>> =

Сдвиг вправо с присваиванием

>>>

Сдвиг вправо с заполнением нулями

>>>=

Сдвиг вправо с заполнением нулями с присваиванием

<<

Сдвиг влево

<<=

Сдвиг влево с присваиванием

Пример программы, манипулирующей с битами

Ниже показано, как каждый из операторов битовой арифметики воздействует на возможные комбинации битов своих операндов. Приведенный далее пример иллюстрирует использование этих операторов в программе на языке Java.

A	B	OR	AND	XOR	NOT A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
class Bitlogic {
public static void main(String args []) {
String binary[] = { "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111" };
int a = 3;     //        0+2+1   или двоичное 0011
int b = 6;     //        4+2+0   или двоичное 0110
int c = a | b;
int d = a & b;
```



```
int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" ab = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a^~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
} }
```

Ниже приведен результат, полученный при выполнении этой программы:

```
C: \> Java BitLogic
a = 0011
b = 0110
a | b = 0111
a & b = 0010
```

```
a ^ b = 0101
~a & b | a & ~b = 0101
~a = 1100
```

Сдвиги влево и вправо

Оператор << выполняет сдвиг влево всех битов своего левого операнда на число позиций, заданное правым операндом. При этом часть битов в левых разрядах выходит за границы и теряется, а соответствующие правые позиции заполняются нулями. Если хотя бы один из операндов в выражении имеет тип **long**, то и тип всего выражения повышается до **long**.

Оператор >> означает в языке Java сдвиг вправо. Он перемещает все биты своего левого операнда вправо на число позиций, заданное правым операндом. Когда биты левого операнда выдвигаются за самую правую позицию слова, они теряются. При сдвиге вправо освобождающиеся старшие (левые) разряды сдвигаемого числа заполняются предыдущим содержимым знакового разряда. Такое поведение называют расширением знакового разряда.

В следующей программе байтовое значение преобразуется
в строку, содержащую его шестнадцатеричное представление. Обратите внимание — сдвинутое значение приходится маскировать, то есть логически умножать на значение 0x0f, для

того, чтобы очистить заполняемые в результате расширения знака биты и понизить значение до пределов, допустимых при индексировании массива шестнадцатеричных цифр.

```
class   HexByte {
static public void main(String
args[]) {
char hex[] = { '0', '1', '2', '3',
'4', '5', '6', '7', '8', '9', 'a',
'b', 'c', 'd', 'e', 'f' };
byte b = (byte) 0xf1;
System.out.println("b = 0x" + hex[(b
>> 4) & 0x0f] +    hex[b & 0x0f]);
} }
```

Ниже приведен результат работы этой программы:

```
C:\> java HexByte
b = 0xf1
```

Беззнаковый сдвиг вправо

Часто требуется, чтобы при сдвиге вправо расширение знакового разряда не происходило, а освобождающиеся левые разряды просто заполнялись бы нулями.

```
class ByteUShift    {
static public void main(String
args[]) {
```

```
char hex[] = { '0', '1', '2', '3',
'4', '5', '6', '7', '8', '9', 'a',
'b', 'c', 'd', 'e', 'f' };
byte b = (byte) 0xf1;
byte c = (byte) (b >> 4);
byte d = (byte) (b >> 4);
byte e = (byte) ((b & 0xff) >> 4);
System.out.println(" b = 0x" +
hex(b >> 4) & 0x0f] +
hex[b & 0x0f]);
System.out.println(" b >> 4 =  0x"
+ hex[(c >> 4) & 0x0f] + hex[c &
0x0f]);
System.out.println("b >>> 4 = 0x" +
hex[(d >> 4) & 0x0f] + hex[d &
0x0f]);
System.out.println("(b & 0xff) >> 4
= 0x" +
hex[(e >> 4) & 0x0f] + hex[e &
0x0f]);
} }
```

Для этого примера переменную b можно было бы инициализировать произвольным отрицательным числом, мы использовали число с шестнадцатеричным представлением 0xf1. Переменной c присваивается результат знакового сдвига b вправо на 4 разряда. Как и

ожидалось, расширение знакового разряда приводит к тому, что 0xf1 превращается в 0xff. Затем в переменную d заносится результат беззнакового сдвига b вправо на 4 разряда. Можно было бы ожидать, что в результате d содержит 0x0f, однако на деле мы снова получаем 0xff. Это — результат расширения знакового разряда, выполненного при автоматическом повышении типа переменной b до int перед операцией сдвига вправо. Наконец, в выражении для переменной e нам удается добиться желаемого результата — значения 0x0f. Для этого нам пришлось перед сдвигом вправо логически умножить значение переменной b на маску 0xff, очистив таким образом старшие разряды, заполненные при автоматическом повышении типа. Обратите внимание, что при этом уже нет необходимости использовать беззнаковый сдвиг вправо, поскольку мы знаем состояние знакового бита после операции AND.

```
C: \> java ByteUShift
b = 0xf1
b >> 4 = 0xff
b >>> 4 = 0xff
b & 0xff) >> 4 = 0x0f
```

Битовые операторы присваивания

Так же, как и в случае арифметических операторов, у всех бинарных битовых операторов есть родственная форма, позволяющая автоматически присваивать результат операции левому операнду. В следующем примере создаются несколько целых переменных, с которыми с помощью операторов, указанных выше, выполняются различные операции.

```
class OpBitEquals {
public static void main(String
args[]) {
int a = 1;
int b = 2;
int c = 3;
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
} }
```


Результаты исполнения программы таковы:

```
C:\> Java OpBitEquals
a = 3
b = 1
c = 6
```

Операторы отношения

Для того, чтобы можно было сравнивать два значения, в Java имеется набор операторов, описывающих отношение и равенство. Список таких операторов приведен ниже.

Оператор	Результат
==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно

Значения любых типов, включая целые и вещественные числа, символы, логические значения и ссылки, можно сравнивать, используя оператор проверки на равенство == и неравенство !=. Обратите внимание — в языке Java, так же, как в C и C++ проверка на равенство обозначается последовательностью (==). Один знак (=) — это оператор присваивания.

Булевы логические операторы

Булевы логические операторы, сводка которых приведена ниже, оперируют только с операндами типа **boolean**. Все бинарные логические операторы воспринимают в качестве операндов два значения типа **boolean** и возвращают результат того же типа.

Оператор	Результат
&	логическое И (AND)
&=	И (AND) с присваиванием
|	логическое ИЛИ (OR)
=	ИЛИ (OR) с присваиванием
^	логическое исключающее ИЛИ (XOR)
^=	исключающее ИЛИ (XOR) с присваиванием
||	оператор OR быстрой оценки выражений (short circuit OR)
==	равно
&&	оператор AND быстрой оценки выражений short circuit AND)
!=	не равно
!	логическое унарное отрицание (NOT)
?:	тернарный оператор if-then-else

Результаты воздействия логических операторов на различные комбинации значений операндов показаны в таблице.

A	B	OR	AND	XOR	NOT A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true

true	true	true	true	false	false

Программа, приведенная ниже, практически полностью повторяет уже знакомый вам пример BitLogic. Только но на этот раз мы работаем с булевыми логическими значениями.

```
class BoolLogic {
public static void main(String
args[]) {
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " +
f);
System.out.println(" !a = " + g);
} }
```

Результат:

```
C: \> Java BoolLogic
a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
```

Операторы быстрой оценки логических выражений (short circuit logical operators)

Существуют два интересных дополнения к набору логических операторов. Это — альтернативные версии операторов AND и OR, служащие для быстрой оценки логических выражений. Вы знаете, что если первый операнд оператора OR имеет значение true, то независимо от значения второго операнда результатом операции будет величина true. Аналогично в случае оператора AND, если первый операнд — false, то значение второго операнда на результат не влияет — он всегда будет равен false. Если вы в используете операторы **&&** и || вместо обычных форм **&** и |, то Java не производит оценку правого операнда логического выражения, если ответ ясен из значения левого операнда. Общепринятой

практикой является использование операторов **&&** и || практически во всех случаях оценки булевых логических выражений. Версии этих операторов **&** и | применяются только в битовой арифметике.

Тернарный оператор if-then-else

Общая форма оператора **if-then-use** такова:

```
выражение1? выражение2: выражение3
```

В качестве первого операнда — «выражение1» — может быть использовано любое выражение, результатом которого является значение типа **boolean**. Если результат равен true, то выполняется оператор, заданный вторым операндом, то есть, «выражение2». Если же первый операнд равен false, то выполняется третий операнд — «выражение3». Второй и третий операнды, то есть «выражение2» и «выражение3», должны возвращать значения одного типа и не должны иметь тип **void**.

В приведенной ниже программе этот оператор используется для проверки делителя перед выполнением операции деления. В случае нулевого делителя возвращается значение 0.

```
class Ternary {
public static void main(String
```

```
args[]) {
int a = 42;
int b = 2;
int c = 99;
int d = 0;
int e = (b == 0) ? 0 : (a / b);
int f = (d == 0) ? 0 : (c / d);
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("a / b = " + e);
System.out.println("c / d = " + f);
} }
```

При выполнении этой программы исключительной ситуации деления на нуль не возникает и выводятся следующие результаты:

```
C: \>java Ternary
a = 42
b = 2
c = 99
d = 0
a / b = 21
c / d   = 0
```


Приоритеты операторов

В Java действует определенный порядок, или приоритет, операций. В элементарной алгебре нас учили тому, что у умножения и деления более высокий приоритет, чем у сложения и вычитания. В программировании также приходится следить и за приоритетами операций. Ниже указаны в порядке убывания приоритеты всех операций языка Java.

Высший

( )	[ ]	.	
~	!		
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		

Низший

В первой строке приведены три необычных оператора, о которых мы пока не говорили. Круглые скобки () используются для явной установки приоритета. Как вы узнали из предыдущей главы, квадратные скобки [] используются для индексирования переменной-массива. Оператор . (точка) используется для выделения элементов из ссылки на объект.

Явные приоритеты

Поскольку высший приоритет имеют круглые скобки, вы всегда можете добавить в выражение несколько пар скобок, если у вас есть сомнения по поводу порядка вычислений или вам просто хочется сделать свои код более читабельным.

```
a >> b + 3
```

Какому из двух выражений, a >> (b + 3) или (a >> b) + 3, соответствует первая строка? Поскольку у оператора сложения более высокий приоритет, чем у оператора сдвига, правильный ответ — a>> (b + a). Так что если вам требуется выполнить операцию (a>>b)+ 3 без скобок не обойтись.

Управление выполнением программы

Управление в Java почти идентично средствам, используемым в C и C++.

Условные операторы

if-else

В обобщенной форме этот оператор записывается следующим образом:

```
if (логическое выражение) оператор1;
[ else оператор2;]
```

Раздел **else** необязателен. На месте любого из операторов может стоять составной оператор, заключенный в фигурные скобки. Логическое
выражение — это любое выражение, возвращающее значение типа **boolean**.

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
} else
waitForMoreData();
```

А вот полная программа, в которой для определения, к какому времени года относится тот или иной месяц, используются операторы **if-else**.

```
class IfElse {
public static void main(String
args[]) { int month = 4;
String season;
if (month == 12 || month == 1 ||
month == 2) {
season = "Winter";
} else if (month ==3 || month == 4
|| month == 5) {
season = "Spring";
} else if (month == 6 || month ==
7 || month == 8) {
season = "Summer";
} else if (month == 9 || month ==
10 || month == 11) {
season = "Autumn";
} else {
season = "Bogus Month";
}
System.out.println( "April is in the
" + season + ".");
```



```
} }
```

После выполнения программы вы должны получить следующий результат:

```
C: \> java IfElse
April is in the Spring.
```

break

В языке Java отсутствует оператор **goto**. Для того, чтобы в некоторых случаях заменять **goto**, в Java предусмотрен оператор **break**. Этот оператор сообщает исполняющей среде, что следует прекратить выполнение именованного блока и передать управление оператору, следующему за данным блоком. Для именования блоков в языке Java используются метки. Оператор **break** при работе с циклами и в операторах **switch** может использоваться без метки. В таком случае подразумевается выход из текущего блока.

Например, в следующей программе имеется три вложенных блока, и у каждого своя уникальная метка. Оператор **break**, стоящий во внутреннем блоке, вызывает переход на оператор, следующий за блоком b. При этом пропускаются два оператора **println**.

```
class Break {
public static void main(String
args[]) { boolean t = true;
a:          { b:              { c:
{
```

```
System.out.println("Before the
break"); // Перед break
          if (t)
          break b;
System.out.println("This won't
execute"); // Не будет выполнено
}
System.out.println("This won't
execute"); // Не будет выполнено
}
System.out.println("This is after
b"); //После b
} } }
```

В результате исполнения программы вы получите следующий результат:

```
C:\> Java Break
Before the break
This is after b
```

Вы можете использовать оператор **break** только для перехода за один из текущих вложенных блоков. Это отличает **break** от оператора **goto** языка C, для которого возможны переходы на произвольные метки.

switch

Оператор **switch** обеспечивает ясный способ переключения между различными частями программного кода в зависимости от

значения одной переменной или выражения. Общая форма этого оператора такова:

```
switch ( выражение ) { case
значение1:
break;
case значение2:
break;
case значением:
break;
default:
}
```

Результатом вычисления выражения может быть значение любого простого типа, при этом каждое из значений, указанных в операторах **case**, должно быть совместимо по типу с выражением в операторе **switch**. Все эти значения должны быть уникальными литералами. Если же вы укажете в двух операторах **case** одинаковые значения, транслятор выдаст сообщение об ошибке.

Если же значению выражения не соответствует ни один из операторов **case**, управление передается коду, расположенному после ключевого слова **default**. Отметим, что оператор **default** необязателен. В случае, когда ни один из операторов **case** не соответствует значению выражения и в **switch** отсутствует оператор **default** выполнение программы

продолжается с оператора, следующего за оператором **switch**.

Внутри оператора **switch** (а также внутри циклических конструкций) **break** без метки приводит к передаче управления на код, стоящий после оператора **switch**. Если **break** отсутствует, после текущего раздела **case** будет выполняться следующий. Иногда бывает удобно иметь в операторе **switch** несколько смежных разделов **case**, не разделенных оператором **break**.

```
class SwitchSeason { public static
void main(String args[]) {
int month = 4;
String season;
switch (month) {
case 12: // FALLSTHROUGH
case 1: // FALLSTHROUGH
case 2:
season = "Winter";
break;
case 3: // FALLSTHROUGH
case 4: // FALLSTHROUGH
case 5:
season = "Spring";
break;
```



```
case 6: // FALLSTHROUGH
case 7: // FALLSTHROUGH
case 8:
season = "Summer";
break;
case 9: // FALLSTHROUGH
case 10: // FALLSTHROUGH
case 11:
season = "Autumn";
break;
default:
season = "Bogus Month";
}
System.out.println("April is in the
" +      season +        ".");
} }
```

Ниже приведен еще более полезный пример, где оператор **switch** используется для передачи управления в соответствии с различными кодами символов во входной строке. Программа подсчитывает число строк, слов и символов в текстовой строке.

```
class WordCount {
static String text = "Now is the
tifne\n" +
```

```
                                    "for
all good men\n" +
                                    "to
come to the aid\n" +
                                    "of
their country\n"+
                                    "and
pay their due taxes\n";
static int len = text.length();
public static void main(String
args[]) {
boolean inWord = false;
int numChars = 0;
int numWords = 0;
int numLines = 0;
for (int i=0; i < len; i++) {
        char c = text.charAt(i);
        numChars++;
        switch (c) {
                    case '\n':
numLines++; // FALLSTHROUGH
                    case '\t': //
FALLSTHROUGH
                    case ' ' : if
(inWord) {
```



```
        numWords++;

        inWord = false;

        }

        break;
                        default: inWord =
        true;
                                }
                }
        System.out.println("\t" + numLines
        +"\t" + numWords + "\t" +
        numChars);
        } }
```

В этой программе для подсчета слов использовано несколько концепций, относящихся к обработке строк.

return

В Java для реализации процедурного интерфейса к объектам классов используется разновидность подпрограмм, называемых методами. Подпрограмма **main**, которую мы использовали до сих пор — это статический метод соответствующего класса-примера. В любом месте программного кода метода можно поставить оператор **return**, который приведет к

немедленному завершению работы и передаче управления коду, вызвавшему этот метод. Ниже приведен пример, иллюстрирующий использование оператора **return** для немедленного возврата управления, в данном случае — исполняющей среде Java.

```
class ReturnDemo {
public static void main(String
args[]) {
boolean t = true;
System.out.println("Before the
return"); //Перед оператором return
if (t) return;
System.out.println("This won't
execute"); //Это не будет выполнено
} }
```

Зачем в этом примере использован оператор **if (t)**? Дело в том, не будь этого оператора, транслятор Java догадался бы, что последний оператор **println** никогда не будет выполнен. Такие случаи в Java считаются ошибками, поэтому без оператора **if** оттранслировать этот пример нам бы не удалось.

Циклы

Любой цикл можно разделить на 4 части — инициализацию, тело, итерацию и условие завершения. В Java есть три циклические конструкции: **while** (с предусловием), **do-while** (с постусловием) и **for** (с параметром).

while

Этот цикл многократно выполняется до тех пор, пока значение логического выражения равно true. Ниже приведена общая форма оператора **while**:

```
[ инициализация; ]
while ( завершение ) {
тело;
[итерация;] }
```

Инициализация и итерация необязательны. Ниже приведен пример цикла **while** для печати десяти строк «tick».

```
class WhileDemo {
public static void main(String
args[]) {
int n = 10;
while (n > 0) {
          System.out.println("tick "
+ n);
          n--;
```

```
        }
    } }
```

do-while

Иногда возникает потребность выполнить тело цикла по крайней мере один раз — даже в том случае, когда логическое выражение с самого начала

принимает значение false. Для таких случаев в Java используется циклическая конструкция **do-while**. Ее общая форма записи такова:

```
[ инициализация; ] do { тело;
[итерация;] } while ( завершение );
```

В следующем примере тело цикла выполняется до первой проверки условия завершения. Это позволяет совместить код итерации с условием завершения:

```
class DoWhile {
public static void main(String
args[]) {
int n = 10;
do {
      System.out.println("tick " +
n);
    } while (--n > 0);
} }
```

for

В этом операторе предусмотрены места для всех четырех частей цикла. Ниже приведена общая форма оператора записи **for**.

```
for ( инициализация; завершение;
итерация ) тело;
```

Любой цикл, записанный с помощью оператора **for**, можно записать в виде цикла **while**, и наоборот. Если начальные условия таковы, что при входе в цикл условие завершения не выполнено, то операторы тела и итерации не выполняются ни одного раза. В канонической форме цикла **for** происходит увеличение целого значения счетчика с минимального значения до определенного предела.

```
class ForDemo {
public static void main(String
args[]) {
for (int i = 1; i <= 10; i++)
System.out.println("i = " + i);
} }
```

Следующий пример — вариант программы, ведущей обратный отсчет.

```
class ForTick {
public static void main(String
args[]) {
for (int n = 10; n > 0; n--)
System.out.println("tick " + n);
```

```
} }
```

Обратите внимание — переменные можно объявлять внутри раздела инициализации оператора **for**. Переменная, объявленная внутри оператора **for**, действует в пределах этого оператора.

А вот — новая версия примера с временами года, в которой используется оператор **for**.

```
class Months {
static String months[] = {
"January", "February", "March",
"April", "May", "June", "July",
"August", "September", "October",
"November", "December" };
static int monthdays[]    = { 31,
28,   31, 30, 31,   30, 31, 31, 30,
31, 30, 31 };
static String spring = "spring";
static String summer = "summer";
static String autumn = "autumn";
static String winter = "winter";
static String seasons[]   = {
winter, winter, spring, spring,
spring, summer, summer, summer,
autumn, autumn, autumn, winter };
public static void main(String
```



```
args[]) {
for (int month = 0; month < 12;
month++) {
System.out.println(months[month] + "
is a "         +
seasons[month] + " month with " +
monthdays[month] + " days.");
} } }
```

При выполнении эта программа выводит следующие строки:

```
C:\> Java Months
January is a winter month with 31
days.
February is a winter month with 28
days.
March is a spring month with 31
days.
April is a spring month with 30
days.
May is a spring month with 31
days.
June is a summer month with 30
days.
July is a summer month with 31
days.
August is a summer month with 31
```

```
days.
September is a autumn month with 30
days.
October is a autumn month with 31
days.
November is a autumn month with 30
days.
December a winter month with 31
days.
```

Оператор запятая

Иногда возникают ситуации, когда разделы инициализации или итерации цикла **for** требуют нескольких операторов. Поскольку составной оператор в фигурных скобках в заголовок цикла **for** вставлять нельзя, Java предоставляет альтернативный путь. Применение запятой (,) для разделения нескольких операторов допускается только внутри круглых скобок оператора **for**. Ниже приведен тривиальный пример цикла **for**, в котором в разделах инициализации и итерации стоит несколько операторов.

```
class Comma {
public static void main(String
args[]) {
int a, b;
for (a = 1, b = 4; a < b; a++,
b--) {
```



```
System.out.println("a = " +
a);
System.out.println("b = " +
b);
}
} }
```

Вывод этой программы показывает, что цикл выполняется всего два раза.

```
C: \> java Comma
a = 1
b = 4
a = 2
b = 3
```

continue

В некоторых ситуациях возникает потребность досрочно перейти к выполнению следующей итерации, проигнорировав часть операторов тела цикла, еще не выполненных в текущей итерации. Для этой цели в Java предусмотрен оператор **continue**. Ниже приведен пример, в котором оператор **continue** используется для того, чтобы в каждой строке печатались два числа.

```
class ContinueDemo {
public static void main(String
args[]) {
for (int i=0; i < 10; i++) {
```

```
            System.out.print(i + " ");
            if (i % 2 == 0) continue;
            System.out.println("");
          }
    }}
```

Если индекс четный, цикл продолжается без вывода символа новой строки. Результат выполнения этой программы таков:

```
C: \> java ContinueDemo
0 1
2 3
4 5
5 7
8 9
```

Как и в случае оператора **break**, в операторе **continue** можно задавать метку, указывающую, в каком из вложенных циклов вы хотите досрочно прекратить выполнение текущей итерации. Для иллюстрации служит программа, использующая оператор **continue** с меткой для вывода треугольной таблицы умножения для чисел от 0 до 9:

```
class ContinueLabel {
public static void main(String
args[]) {
outer:     for (int i=0; i < 10;
```



```
i++) {
                        for (int j =
0; j < 10; j++) {
                                if (j >
i) {

System.out.println("");

continue outer;
                                }

System.out.print(" " + (i * j));
                        }
                }
}}
```

Оператор **continue** в этой программе приводит к завершению внутреннего цикла со счетчиком j и переходу к очередной итерации внешнего цикла со счетчиком i. В процессе работы эта программа выводит следующие строки:

```
C:\> Java ContinueLabel
0
0 1
0 2 4
0 3 6 9
```

```
0  4  8  12  16
0  5  10  15  20  25
0  6  12  18  24  30  36
0  7  14  21  28  35  42  49
0  8  16  24  32  40  48  56  64
0  9  18  27  36  45  54  63  72  81
```

Классы

Базовым элементом объектно-ориентированного программирования в языке Java является класс. Классы в Java не обязательно должны содержать метод **main**. Единственное назначение этого метода — указать интерпретатору Java, откуда надо начинать выполнение программы. Для того, чтобы создать класс, достаточно иметь исходный файл, в котором будет присутствовать ключевое слово **class**, и вслед за ним — допустимый идентификатор и пара фигурных скобок для его тела.

```
class Point {
}
```

Имя исходного файла Java должно соответствовать имени хранящегося в нем класса. Регистр букв важен и в имени класса, и в имени файла.

Класс определяет структуру объекта и его методы, образующие функциональный интерфейс. В процессе выполнения Java-программы система использует определения классов для создания представителей классов. Представители являются реальными объектами. Термины «представитель», «экземпляр» и «объект» взаимозаменяемы. Ниже приведена общая форма определения класса.

```
class имя_класса extends
имя_суперкласса { type
переменная1_объекта:
type переменная2_объекта:
type переменнаяN_объекта:
type имяметода1(список_параметров) {
тело метода;
}
type имяметода2(список_параметров) {
тело метода;
}
type имя методаM(список_параметров) {
тело метода;
}
}
```

Ключевое слово **extends** указывает на то, что «имя_класса» — это подкласс класса «имя_суперкласса». Во главе классовой

иерархии Java стоит единственный ее встроенный класс — **Object**. Если вы хотите создать подкласс непосредственно этого класса, ключевое слово **extends** и следующее за ним имя суперкласса можно опустить — транслятор включит их в ваше определение автоматически. Примером может служить класс **Point**.

Переменные представителей (instance variables)

Данные инкапсулируются в класс путем объявления переменных между открывающей и закрывающей фигурными скобками, выделяющими в определении класса его тело. Эти переменные объявляются точно так же, как объявлялись локальные переменные в предыдущих примерах. Единственное отличие состоит в том, что их надо объявлять вне методов, в том числе вне метода **main**. Ниже приведен фрагмент кода, в котором объявлен класс **Point** с двумя переменными типа **int**.

```
class Point { int x, y;
}
```

В качестве типа для переменных объектов можно использовать как любой из простых типов, так и классовые типы. Скоро мы добавим к приведенному выше классу метод **main**, чтобы его можно было запустить из командной строки и создать несколько объектов.

Оператор new

Оператор **new** создает экземпляр указанного класса и возвращает ссылку на вновь созданный объект. Ниже приведен пример создания и присваивание переменной p экземпляра класса **Point**.

```
Point p = new Point();
```

Вы можете создать несколько ссылок на один и тот же объект. Приведенная ниже программа создает два различных объекта класса
Point и в каждый из них заносит свои собственные значения. Оператор точка используется для доступа к переменным и методам объекта.

```
class TwoPoints {
public static void main(String
args[]) {
Point p1 = new Point();
Point p2 = new Point();
p1.x = 10;
p1.y = 20;
p2.x = 42;
p2.y = 99;
System.out.println("x = " + p1.x +
" y = " + p1.y);
System.out.println("x =  " + p2.x +
```

```
" y = " + p2.y);
} }
```

В этом примере снова использовался класс **Point**, было создано два объекта этого класса, и их переменным x и y присвоены различные значения. Таким образом мы продемонстрировали, что переменные различных объектов независимы на самом деле. Ниже приведен результат, полученный при выполнении этой программы.

```
C:\> Java TwoPoints
x = 10 y = 20
x = 42 y = 99
```

Поскольку при запуске интерпретатора мы указали в командной строке не класс **Point**, а класс **TwoPoints**, метод **main** класса **Point** был полностью проигнорирован. Добавим в класс **Point** метод **main** и, тем самым, получим законченную программу.

```
class Point { int x, y;
public static void main(String
args[]) {
Point p = new Point();
p.x = 10;
p.y = 20;
System.out.println("x =  " + p.x +
" y = " + p.y);
```



```
} }
```

Объявление методов

Методы — это подпрограммы, присоединенные к конкретным определениям классов. Они описываются внутри определения класса на том же уровне, что и переменные объектов. При объявлении метода задаются тип возвращаемого им результата и список параметров. Общая форма объявления метода такова:

```
тип имя_метода (список формальных
параметров) {
тело метода:
}
```

Тип результата, который должен возвращать метод может быть любым, в том числе и типом **void** — в тех случаях, когда возвращать результат не требуется. Список формальных параметров — это последовательность пар тип-идентификатор, разделенных запятыми. Если у метода параметры отсутствуют, то после имени метода должны стоять пустые круглые скобки.

```
class Point { int x, y;
void init(int a, int b) {
x = a;
y = b;
```

```
} }
```

Вызов метода

В Java отсутствует возможность передачи параметров по ссылке на примитивный тип. В Java все параметры примитивных типов передаются по значению, а это означает, что у метода нет доступа к исходной переменной, использованной в качестве параметра. Заметим, что все объекты передаются по ссылке, можно изменять содержимое того объекта, на который ссылается данная переменная

Скрытие переменных представителей

В языке Java не допускается использование в одной или во вложенных областях видимости двух локальных переменных с одинаковыми именами. Интересно отметить, что при этом не запрещается объявлять формальные параметры методов, чьи имена совпадают с именами переменных представителей. Давайте рассмотрим в качестве примера иную версию метода **init**, в которой формальным параметрам даны имена x и y, а для доступа к одноименным переменным текущего объекта используется ссылка **this**.

```
class Point { int x, y;
void init(int x, int y) {
this.x = x;
```



```
this.y = y } }
class TwoPointsInit {
public static void main(String
args[]) {
Point p1 = new Point();
Point p2 = new Point();
p1.init(10,20);
p2.init(42,99);
System.out.println("x = " + p1.x +
" y = •• + p-l.y);
System.out.printlnC'x = " + p2.x +
" y = •• + p2.y);
} }
```

Конструкторы

Инициализировать все переменные класса всякий раз, когда создается его очередной представитель — довольно утомительное дело даже в том случае, когда в классе имеются функции, подобные методу **init**. Для этого в Java предусмотрены специальные методы, называемые конструкторами. Конструктор — это метод класса, который инициализирует новый объект после его создания. Имя конструктора всегда совпадает с именем класса, в котором он расположен (также, как и в C++). У конструкторов нет типа возвращаемого результата — никакого,

даже **void**. Заменим метод **init** из предыдущего примера конструктором.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
} }
class PointCreate {
public static void main(String
args[]) {
Point p = new Point(10,20);
System.out.println("x = " + p.x + "
y = " + p.y);
} }
```

Программисты на Pascal (Delphi) для обозначения конструктора используют ключевое слово constructor.

Совмещение методов

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется совмещением методов (method overloading). В качестве примера приведена версия класса **Point**, в которой совмещение методов использовано для определения альтернативного конструктора, который

инициализирует координаты x и y значениями по умолчанию (-1).

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
Point() {
x = -1;
y = -1;
} }
class PointCreateAlt {
public static void main(String
args[]) {
Point p = new Point();
System.out.println("x = " + p.x + "
y = " + p.y);
} }
```

В этом примере объект класса **Point** создается не при вызове первого конструктора, как это было раньше, а с помощью второго конструктора без параметров. Вот результат работы этой программы:

```
C:\> java PointCreateAlt
x = -1 y = -1
```

Решение о том, какой конструктор нужно вызвать в том или ином случае, принимается в соответствии с количеством и типом параметров, указанных в операторе **new**. Недопустимо объявлять в классе методы с одинаковыми именами и сигнатурами. В сигнатуре метода не учитываются имена формальных параметров учитываются лишь их типы и количество.

this в конструкторах

Очередной вариант класса **Point** показывает, как, используя **this** и совмещение методов, можно строить одни конструкторы на основе других.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
Point() {
this(-1, -1);
} }
```

В этом примере второй конструктор для завершения инициализации объекта обращается к первому конструктору.

Методы, использующие совмещение имен, не обязательно должны быть

конструкторами. В следующем примере в класс **Point** добавлены два метода **distance**. Функция **distance** возвращает расстояние между двумя точками. Одному из совмещенных методов в качестве параметров передаются координаты точки x и y, другому же эта информация передается в виде параметра-объекта **Point**.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this. y = y;
}
double distance(int x, int y) {
int dx = this.x - x;
int dy = this.y - y;
return Math.sqrt(dx*dx + dy*dy);
}
double distance(Point p) {
return distance(p.x, p.y);
} }
class PointDist {
public static void main(String
args[]) {
Point p1 = new Point(0, 0);
```

```
Point p2 = new Point(30, 40);
System.out.println("p1 = " + pi.x +
", " + p1.y);
System.out.println("p2 = " + p2.x +
", " + p2.y);
System.out.println("p1.distance(p2) =
" + p1.distance(p2));
System.out.println("p1.distance(60,
80) = " + p1.distance(60, 80));
} }
```

Обратите внимание на то как во второй форме метода **distance** для получения результата вызывается его первая форма. Ниже приведен результат работы этой программы:

```
C:\> java PointDist
p1 = 0, 0
p2 = 30, 40
p1.distance(p2) = 50.0
p1.distance(60, 80) = 100.0
```

Наследование

Вторым фундаментальным свойством объектно-ориентированного подхода является наследование (первый – инкапсуляция). Классы-потомки имеют возможность не только создавать свои собственные переменные и методы, но и наследовать переменные и

методы классов-предков.
Классы-потомки принято называть подклассами. Непосредственного предка данного класса называют его суперклассом. В очередном примере показано, как расширить класс **Point** таким образом, чтобы включить в него третью
координату z.

```
class Point3D extends Point { int
z;
Point3D(int x, int y, int z) {
this.x = x;
this.y = y;
this.z = z; }
Point3D() {
this(-1,-1,-1);
} }
```

В этом примере ключевое слово **extends** используется для того, чтобы сообщить транслятору о намерении создать подкласс класса **Point**. Как видите, в этом классе не понадобилось объявлять переменные x и y, поскольку Point3D унаследовал их от своего суперкласса **Point**.

super

В примере с классом Point3D частично повторялся код, уже имевшийся в суперклассе. Вспомните, как во втором конструкторе мы

использовали **this** для вызова первого конструктора того же класса. Аналогичным образом ключевое слово **super** позволяет обратиться непосредственно к конструктору суперкласса (в Delphi / C++ для этого используется ключевое слово inherited).

```
class Point3D extends Point { int
z;
Point3D(int x, int y, int z) {
super(x, y);        // Здесь мы
вызываем конструктор суперкласса
this.z=z;
public static void main(String
args[]) {
Point3D p = new Point3D(10, 20,
30);
System.out.println( " x = " +   p.x
+ " y = " + p.y +
" z = " + p.z);
} }
```

Вот результат работы этой программы:

```
C:\> java Point3D
x = 10 y = 20 z = 30
```

Замещение методов

Новый подкласс Point3D класса **Point** наследует реализацию метода

distance своего суперкласса (пример PointDist.java). Проблема заключается в том, что в классе **Point** уже определена версия метода **distance(mt x, int y)**, которая возвращает обычное расстояние между точками на плоскости. Мы должны заместить (override) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и совмещение (overloading), и замещение (overriding) метода **distance**.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x;
this.y = y;
}
double distance(int x, int y) {
int dx = this.x - x;
int dy = this.y - y:
return Math,sqrt(dx*dx + dy*dy);
}
double distance(Point p) {
return distance(p.x, p.y);
}
}
class   Point3D extends Point { int
```

```
z;
Point3D(int x, int y, int z)    {
super(x, y);
this.z = z;
(
double distance(int x, int y,   int
z) {
int dx = this.x - x;
int dy = this.y - y;
int dz = this.z -   z;
return Math.sqrt(dx*dx + dy*dy +
dz*dz);
}
double distance(Point3D other) {
return distance(other.x, other.y,
other.z);
}
double distance(int x, int y)   {
double dx = (this.x / z) - x;
double dy = (this.y / z) - y;
return Math.sqrt(dx*dx + dy*dy);
}
}
class Point3DDist {
```



```
public static void main(String
args[]) {
Point3D p1 = new Point3D(30, 40,
10);
Point3D p2 = new Point3D(0, 0, 0);
Point p = new Point(4, 6);
System.out.println("p1 = " + p1.x +
", " + p1.y + ", " + p1.z);
System.out.println("p2 = " + p2.x +
", " + p2.y + ", " + p2.z);
System.out.println("p = " + p.x +
", " + p.y);
System.out.println("p1.distance(p2) =
" + p1.distance(p2));
System.out.println("p1.distance(4, 6)
= " + p1.distance(4, 6));
System.out.println("p1.distance(p) =
" + p1.distance(p));
} }
```

Ниже приводится результат работы этой программы:

```
C:\> Java Point3DDist
p1 = 30, 40, 10
p2 = 0, 0, 0
p = 4, 6
```

```
p1.distance(p2) = 50.9902
p1.distance(4, 6) = 2.23607
p1.distance(p) = 2.23607
```

Обратите внимание — мы получили ожидаемое расстояние между трехмерными точками и между парой двумерных точек. В примере используется механизм, который называется динамическим назначением методов (dynamic method dispatch).

Динамическое назначение методов

Давайте в качестве примера рассмотрим два класса, у которых имеют простое родство подкласс/суперкласс, причем единственный метод суперкласса замещен в подклассе.

```
class A { void callme() {
System.out.println("Inside A's
callrne method");
class B extends A { void callme()
{
System.out.println("Inside B's callme
method");
} }
class Dispatch {
public static void main(String
args[]) {
```



```
A a = new B();
a.callme();
} }
```

Обратите внимание — внутри метода **main** мы объявили переменную а класса A, а проинициализировали ее ссылкой на объект класса B. В следующей строке мы вызвали метод **callme**. При этом транслятор проверил наличие метода **callme** у класса A, а исполняющая система, увидев, что на самом деле в переменной хранится представитель класса B, вызвала не метод класса A, а **callme** класса B. Ниже приведен результат работы этой программы:

```
C:\> Java Dispatch
Inside B's calime method
```

final

Все методы и переменные объектов могут быть замещены по умолчанию. Если же вы хотите объявить, что подклассы не имеют права замещать
какие-либо переменные и методы вашего класса, вам нужно объявить их как **final**.

```
final int FILE_NEW = 1;
```

По общепринятому соглашению при выборе имен переменных типа
final — используются только символы верхнего регистра (т.е. используются как аналог препроцерных констант C++). Использование

final-методов порой приводит к выигрышу в скорости выполнения кода — поскольку они не могут быть замещены, транслятору ничто не мешает заменять их вызовы встроенным (in-line) кодом (байт-код копируется непосредственно в код вызывающего метода).

finalize

В Java существует возможность объявлять методы с именем **finalize**. Методы **finalize** аналогичны деструкторам в C++ (ключевой знак ~) и Delphi (ключевое слово destructor). Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

static

Иногда требуется создать метод, который можно было бы использовать вне контекста какого-либо объекта его класса. Так же, как в случае **main**, все, что требуется для создания такого метода — указать при его объявлении модификатор типа **static**. Статические методы могут непосредственно обращаться только к другим статическим методам, в них ни в каком виде не допускается использование ссылок **this** и **super**. Переменные также могут иметь тип static, они подобны глобальным переменным, то есть доступны из любого места кода. Внутри статических методов недопустимы ссылки на переменные представителей. Ниже приведен пример класса, у которого есть статические

переменные, статический метод и статический блок инициализации.

```
class Static {
static int a = 3;
static int b;
static void method(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("static block
initialized");
b = a * 4;
}
public static void main(String
args[]) {
method(42);
} }
```

Ниже приведен результат запуска этой программы.

```
C:\> java Static static block
initialized
X = 42
```

```
A = 3
B = 12
```

В следующем примере мы создали класс со статическим методом и несколькими статическими переменными. Второй класс может вызывать статический метод по имени и ссылаться на статические переменные непосредственно через имя класса.

```
class StaticClass {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
} }
class StaticByName {
public static void main(String
args[]) {
staticClass.callme();
System.out.println("b = " +
StaticClass.b);
} }
```

А вот и результат запуска этой программы:

```
C:\> Java StaticByName
a = 42 b = 99
```


abstract

Бывают ситуации, когда нужно определить класс, в котором задана структура какой-либо абстракции, но полная реализация всех методов отсутствует. В таких случаях вы можете с помощью модификатора типа **abstract** объявить, что некоторые из методов обязательно должны быть замещены в подклассах. Любой класс, содержащий методы **abstract**, также должен быть объявлен, как **abstract**. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора **new**. Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

```
abstract class A {
abstract void callme();
void metoo() {
System.out.println("Inside A's metoo
method");
} }
class B extends A {
void callme() {
System.out.println("Inside B's callme
method");
```

```
} }
class Abstract {
public static void main(String
args[]) {
A a = new B():
a.callme():
a.metoo():
} }
```

В нашем примере для вызова реализованного в подклассе класса A метода **callme** и реализованного в классе A метода **metoo** используется динамическое назначение методов, которое мы обсуждали раньше.

```
C:\> Java Abstract
Inside B's callrne method Inside
A's metoo method
```

Пакеты и интерфейсы

Пакет (package) — это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс **List**, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть если бы кто-нибудь еще создал класс с именем **List**.

Интерфейс — это явно указанная спецификация набора методов, которые должны быть представлены в классе, который реализует эту спецификацию. Реализация же этих методов в интерфейсе отсутствует. Подобно абстрактным классам интерфейсы обладают замечательным дополнительным свойством — их можно многократно наследовать. Конкретный класс может быть наследником лишь одного суперкласса, но зато в нем может быть реализовано неограниченное число интерфейсов.

Пакеты

Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты — это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

- одиночный оператор package (необязателен)
- любое количество операторов import (необязательны)

■ одиночное объявление открытого (public) класса

■ любое количество закрытых (private) классов пакета (необязательны)

Оператор package

Первое, что может появиться в исходном файле Java — это оператор **package**, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор раздельных пространств имен, в которых хранятся имена классов. Если оператор **package** не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если вы объявляете класс, как принадлежащий определенному пакету, например,

```
package java.awt.image;
```

то и исходный код этого класса должен храниться в каталоге java/awt/image.

Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Представьте себе, что вы написали класс с именем PackTest в пакете test. Вы создаете каталог test, помещаете в этот каталог файл PackTest.Java и транслируете. Пока — все в порядке. Однако при попытке запустить его вы получаете от интерпретатора сообщение «can't find class PackTest» («Не могу найти класс PackTest»). Ваш новый класс теперь хранится в пакете с именем test, так что теперь надо указывать всю иерархию пакетов, разделяя их имена точками — test.PackTest. Кроме того, вам надо либо подняться на уровень выше в иерархии каталогов и снова набрать «java test.PackTest», либо внести в переменную CLASSPATH каталог, который является вершиной иерархии разрабатываемых вами классов.

Оператор import

После оператора **package**, но до любого определения классов в исходном Java-файле, может присутствовать список операторов **import**. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора **import** такова:

```
import пакет1
[.пакет2].(имякласса|*);
```

Здесь пакет1 — имя пакета верхнего уровня, пакет2 — это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания

пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора **import**:

```
import java.util.Date
import java.io.*;
```

Все встроенные в Java классы, которые входят в комплект поставки, хранятся в пакете с именем java. Базовые функции языка хранятся во вложенном пакете java.lang. Весь этот пакет автоматически импортируется транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

```
import java.lang.*;
```

Если в двух пакетах, подключаемых с помощью формы оператора import со звездочкой, есть классы с одинаковыми именами, однако вы их не используете, транслятор не отреагирует. А вот при попытке использовать такой класс, вы сразу получите сообщение об ошибке, и вам придется переписать операторы **import**, чтобы явно указать, класс какого пакета вы имеете ввиду.

```
class MyDate extends Java.util.Date
```



```
{ }
```

Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать еще с четырьмя категориями видимости между элементами классов:

- Подклассы в том же пакете.
- Не подклассы в том же пакете.
- Подклассы в различных пакетах.
- Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: **private** (закрытый), **public** (открытый) и **protected** (защищенный), которые употребляются в различных комбинациях.

Элемент, объявленный **public**, доступен из любого места. Все, что объявлено **private**, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того

класса, которому он принадлежит, вам нужно объявить такой элемент **protected**. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет — используйте комбинацию **private protected**.

Ниже приведен довольно длинный пример, в котором представлены все допустимые комбинации модификаторов уровня доступа. В исходном коде первого пакета определяется три класса: **Protection**, **Derived** и **SamePackage**. В первом из этих классов определено пять целых переменных — по одной на каждую из возможных комбинаций уровня доступа. Переменной n приписан уровень доступа по умолчанию, n_pri — уровень **private**, n_pro — **protected**, n_pripro — **private protected** и n_pub — **public**. Во всех остальных классах мы пытаемся использовать переменные первого класса. Те строки кода, которые из-за ограничения доступа привели бы к ошибкам при трансляции, закомментированы с помощью однострочных комментариев (//) — перед каждой указано, откуда доступ при такой комбинации модификаторов был бы возможен. Второй класс — **Derived** — является подклассом класса **Protection** и расположен в том же пакете p1. Поэтому ему доступны все перечисленные переменные за исключением n_pri. Третий класс, **SamePackage**, расположен в том же

пакете, но при этом не является подклассом **Protection**. По этой причине для него недоступна не только переменная n_pri, но и n_pripro, уровень доступа которой — **private protected**.

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
private protected int n_pripro = 4;
public int n_pub = 5;
public Protection() {
System.out.println("base
constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " +
n_pri);
System.out.println("n_pro = "   +
n_pro);
System.out.println("n_pripro = " +
n_pripro);
System.out.println("n_pub = " +
n_pub);
} }
```

```
class Derived extends Protection {
Derived() {

System.out.println("derived
constructor");
                System.out.println("n
= " + n);
                // только в классе
                //
System.out.println("n_pri = " +
n_pri);

System.out.println("n_pro = "      +
n_pro);

System.out.println("n_pripro = " +
n_pripro);

System.out.println("n_pub = "      +
n_pub);
} }
class   SamePackage {
SamePackage() {
                      Protection p =
new Protection();
```



```
System.out.println("same package
                  constructor");

System.out.println("n = " + p.n);
                      // только в
классе
                      //
System.out.println("n_pri = " +
p.n_pri);

System.out.println("n_pro = " +
p.n_pro);
                      // только в
классе и подклассе
                      //
System.out.println("n_pripro = " +
                  p.n_pripro):

System.out.println("n_pub = " +
p.n_pub):
} }
```

Интерфейсы

Интерфейсы Java созданы для поддержки динамического выбора
(resolution) методов во время выполнения программы. Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет

переменных представителей, а в объявлениях методов отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать — это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов. Интерфейсы являются аналогом механизма множественного наследования в C++, но использовать их намного легче.

Оператор interface

Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов. Общая форма интерфейса приведена ниже:

```
interface имя {
тип_результата имя_метода1(список
параметров);
тип имя_final1-переменной = значение;
}
```


Обратите внимание — у объявляемых в интерфейсе методов отсутствуют операторы тела. Объявление методов завершается символом ; (точка с запятой). В интерфейсе можно объявлять и переменные, при этом они неявно объявляются **final**-переменными. Это означает, что класс реализации не может изменять их значения. Кроме того, при объявлении переменных в интерфейсе их обязательно нужно инициализировать константными значениями. Ниже приведен пример определения интерфейса, содержащего единственный метод с именем **callback** и одним параметром типа **int**.

```
interface Callback {
void callback(int param);
}
```

Оператор implements

Оператор **implements** — это дополнение к определению класса, реализующего некоторый интерфейс(ы).

```
class имя_класса [extends суперкласс]
[implements интерфейс0 [,
интерфейс1...]] { тело класса }
```

Если в классе реализуется несколько интерфейсов, то их имена разделяются запятыми. Ниже приведен пример класса, в котором реализуется определенный нами интерфейс:

```
class Client implements Callback {
void callback(int p) {
System.out.println("callback called
with " + p);
} }
```

В очередном примере метод **callback** интерфейса, определенного ранее, вызывается через переменную — ссылку на интерфейс:

```
class TestIface {
public static void main(String
args[]) { Callback c = new
client();
c.callback(42);
} }
```

Ниже приведен результат работы программы:

```
C:\> Java TestIface
callback called with 42
```

Переменные в интерфейсах

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант. В том случае, когда вы реализуете в классе какой-либо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы. Это

аналогично использованию файлов-заголовков для задания в C и C++ констант с помощью директив #define или ключевого слова const в Pascal/Delphi.

Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией этого интерфейса, может вообще ничего не реализовывать. Для импорта констант в пространство имен класса предпочтительнее использовать переменные с модификатором **final**. В приведенном ниже примере проиллюстрировано использование интерфейса для совместно используемых констант.

```
import java.util.Random;
interface SharedConstants { int NO
= 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5; }
class Question implements
SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 *
```

```
rand.nextDouble());
if (prob < 30)
return NO; // 30% else if (prob <
60)
return YES; // 30% else if (prob <
75)
return LATER; // 15% else if (prob
< 98)
return SOON; // 13% else
return NEVER; // 2% } }
class AskMe implements
SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
          System.out.println("Yes");
          break;
case MAYBE:

System.out.println("Maybe");
          break;
```



```
case LATER:

System.out.println("Later");
          break;
case SOON:
          System.out.priniln("Soon");
          break;
case NEVER:

System.out.println("Never");
          break;
} }
public static void main(String
args[]) {
Question q = new Question();
          answer(q.ask());
          answer(q.ask());
          answer(q.askO);
          answer(q.ask());
} }
```

Обратите внимание на то, что результаты при разных запусках программы отличаются, поскольку в ней используется класс генерации случайных чисел **Random** пакета java.util.

```
C:\> Java AskMe
```

```
Later
Scon
No
Yes
```

Теперь вы обладаете полной информацией для создания собственных пакетов классов. Легко понимаемые интерфейсы позволят другим программистам использовать ваш код для самых различных целей. Инструменты, которые вы изучили, должны вам помочь при разработке любых объектно-ориентированных приложений.

Работа со строками

В языках C и C++ отсутствует встроенная поддержка такого объекта, как строка. В них при необходимости передается адрес последовательности байтов, содержимое которых трактуется как символы до тех пор, пока не будет встречен нулевой байт, отмечающий конец строки. В пакет java.lang встроен класс, инкапсулирующий структуру данных, соответствующую строке. Этот класс, называемый **String**, не что иное, как объектное представление неизменяемого символьного массива. В этом классе есть методы, которые позволяют сравнивать строки, осуществлять в них поиск и извлекать определенные символы и подстроки. Класс **StringBuffer** используется тогда, когда строку после создания требуется изменять.

И **String**, и **StringBuffer** объявлены **final**, что означает, что ни от одного из этих классов нельзя производить подклассы. Это было сделано для того, чтобы можно было применить некоторые виды оптимизации позволяющие увеличить производительность при выполнении операций обработки строк.

Конструкторы

Как и в случае любого другого класса, вы можете создавать объекты типа **String** с помощью оператора **new**. Для создания пустой строки используется конструктор без параметров:

```
String s = new String():
```

Приведенный ниже фрагмент кода создает объект **s** типа **String** инициализируя его строкой из трех символов, переданных конструктору в качестве параметра в символьном массиве.

```
char chars[] = { 'a', 'b', 'c' }:
String s = new String(chars);
System.out.println(s):
```

Этот фрагмент кода выводит строку «abc». Итак, у этого конструктора — 3 параметра:

```
String(char chars[], int
начальныйИндекс, int числоСимволов);
```

Используем такой способ инициализации в нашем очередном примере:

```
char chars[] = { 'a', 'b', 'c',
'd', 'e', 'f' };
String s = new String(chars,2,3);
System.out.println(s);
```

Этот фрагмент выведет «cde».

Специальный синтаксис для работы со строками

В Java включено несколько приятных синтаксических дополнений, цель которых — помочь программистам в выполнении операций со строками. В числе таких операций создание объектов типа **String** слияние нескольких строк и преобразование других типов данных в символьное представление.

Создание строк

Java включает в себя стандартное сокращение для этой операции — запись в виде литерала, в которой содержимое строки заключается в пару двойных кавычек. Приводимый ниже фрагмент кода эквивалентен одному из предыдущих, в котором строка инициализировалась массивом типа **char**.

```
String s = "abc";
```



```
System.out.println(s);
```

Один из общих методов, используемых с объектами **String** — метод **length**, возвращающий число символов в строке. Очередной фрагмент выводит число 3, поскольку в используемой в нем строке — 3 символа.

```
String s = "abc";
System.out.println(s.length);
```

В Java интересно то, что для каждой строки-литерала создается свой представитель класса **String**, так что вы можете вызывать методы этого класса непосредственно со строками-литералами, а не только со ссылочными переменными. Очередной пример также выводит число 3.

```
System.out.println("abc".Length());
```

Слияние строк

Строку

```
String s = «He is » + age + "
years old.";
```

в которой с помощью оператора + три строки объединяются в одну, прочесть и понять безусловно легче, чем ее эквивалент, записанный с явными вызовами тех самых методов, которые неявно были использованы в первом примере:

```
String s = new StringBuffer("He is
```

```
").append(age);
s.append(" years old.").toString();
```

По определению каждый объект класса **String** не может изменяться. Нельзя ни вставить новые символы в уже существующую строку, ни поменять в ней одни символы на другие. И добавить одну строку в конец другой тоже нельзя. Поэтому транслятор Java преобразует операции, выглядящие, как модификация объектов **String**, в операции с родственным классом **StringBuffer**.

Все это может показаться вам необоснованно сложным. А почему нельзя обойтись одним классом **String**, позволив ему вести себя примерно так же, как **StringBuffer**? Все дело в производительности. Тот факт, что объекты типа **String** в Java неизменны, позволяет транслятору применять к операциям с ними различные способы оптимизации.

Последовательность выполнения операторов

Давайте еще раз обратимся к нашему последнему примеру:

```
String s = "He is " + age + "
years old.";
```

В том случае, когда age — не **String**, а переменная, скажем, типа **int**, в этой строке кода заключено еще больше магии транслятора. Целое значение переменной **int** передается

совмещенному методу **append** класса **StringBuffer**, который преобразует его в текстовый вид и добавляет в конец содержащейся в объекте строки. Вам нужно быть внимательным при совместном использовании целых выражений и слияния строк, в противном случае результат может получиться совсем не тот, который вы ждали. Взгляните на следующую строку:

```
String s = "four: " + 2 + 2;
```

Быть может, вы надеетесь, что в s будет записана строка «four: 4»? Не угадали — с вами сыграла злую шутку последовательность выполнения операторов. Так что в результате получается "four: 22".

Для того, чтобы первым выполнилось сложение целых чисел, нужно использовать скобки:

```
String s = "four: " + (2 + 2);
```

Преобразование строк

В каждом классе **String** есть метод **toString** — либо своя собственная реализация, либо вариант по умолчанию, наследуемый от класса **Object**. Класс в нашем очередном примере замещает наследуемый метод **toStrring** своим собственным, что позволяет ему выводить значения переменных объекта.

```
class Point {
int x, y;
```

```
Point(int x, int y) {
this.x = x;
this.y = y;
}
public String toString() {
return "Point[" + x + ", " + y +
"]";
} }
class toStringDemo {
public static void main(String
args[]) {
Point p = new Point(10, 20);
System.out.println("p = " + p);
} }
```

Ниже приведен результат, полученный при запуске этого примера.

```
C:\> Java toStringDemo
p = Point[10, 20]
```

Извлечение символов

Для того, чтобы извлечь одиночный символ из строки, вы можете сослаться непосредственно на индекс символа в строке с помощью метода **charAt**. Если вы хотите в один прием извлечь несколько символов, можете воспользоваться методом **getChars**. В

приведенном ниже фрагменте показано, как следует извлекать массив символов из объекта типа **String**.

```
class getCharsDemo {
public static void main(String
args[]) {
String s = "This is a demo of the
getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
} }
```

Обратите внимание — метод **getChars** не включает в выходной
буфер символ с индексом **end**. Это хорошо видно из вывода нашего
примера — выводимая строка состоит из 4 символов.

```
C:\> java getCharsDemo
demo
```

Для удобства работы в **String** есть еще одна функция — **toCharArray**, которая возвращает в выходном массиве типа **char** всю строку. Альтернативная форма того же самого механизма позволяет записать содержимое

строки в массив типа **byte**, при этом значения старших байтов в 16-битных символах отбрасываются. Соответствующий метод называется **getBytes**, и его параметры имеют тот же смысл, что и параметры **getChars**, но с единственной разницей — в качестве третьего параметра надо использовать массив типа **byte**.

Сравнение

Если вы хотите узнать, одинаковы ли две строки, вам следует воспользоваться методом **equals** класса **String**. Альтернативная форма этого метода называется **equalsIgnoreCase**, при ее использовании различие регистров букв в сравнении не учитывается. Ниже приведен пример, иллюстрирующий использование обоих методов:

```
class equalDemo {
public static void main(String
args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals "
+ s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals "
+ s3 + " -> " + s1.equals(s3));
```



```
System.out.println(s1 + " equals "
+ s4 + " -> " + s1.equals(s4));
System.out.println(s1 + "
equalsIgnoreCase " + s4 + " -> " +

s1.equalsIgnoreCase(s4));
} }
```

Результат запуска этого примера:

```
C:\> java equalsDemo
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

В классе **String** реализована группа сервисных методов, являющихся специализированными версиями метода **equals**. Метод **regionMatches** используется для сравнения подстроки в исходной строке с подстрокой в строке-параметре. Метод **startsWith** проверяет, начинается ли данная подстрока фрагментом, переданным методу в качестве параметра. Метод **endsWith** проверяет совпадает ли с параметром конец строки.

Равенство

Метод **equals** и оператор **==** выполняют две совершенно различных проверки. Если метод **equal** сравнивает символы внутри строк,

то оператор == сравнивает две переменные-ссылки на объекты и проверяет, указывают ли они на разные объекты или на один и тот же. В очередном нашем примере это хорошо видно — содержимое двух строк одинаково, но, тем не менее, это — различные объекты, так что **equals** и == дают разные результаты.

```
class EqualsNotEqualTo {
public static void main(String
args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals "
+ s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " == " + s2
+ ", -> " + (s1 == s2));
} }
```

Вот результат запуска этого примера:

```
C:\> java EqualsNotEqualTo
Hello equals Hello -> true
Hello == Hello -> false
```

Упорядочение

Зачастую бывает недостаточно просто знать, являются ли две строки идентичными. Для приложений, в которых требуется сортировка, нужно знать, какая из двух строк меньше другой. Для ответа на этот вопрос

нужно воспользоваться методом **compareTo** класса **String**. Если целое значение, возвращенное методом, отрицательно, то строка, с которой был вызван метод, меньше строки-параметра, если положительно — больше. Если же метод **compareTo** вернул значение 0, строки идентичны. Ниже приведена программа, в которой выполняется пузырьковая сортировка массива строк, а для сравнения строк используется метод **compareTo**. Эта программа выдает отсортированный в алфавитном порядке список строк.

```
class SortString {
static String arr[] = {"Now", "is",
"the", "time", "for", "all",
"good", "men", "to", "come", "to",
"the","aid", "of", "their", "country"
};
public static void main(String
args[]) {
for (int j = 0; i < arr.length;
j++) {
        for (int i = j + 1; i <
arr.length; i++) {
                if
(arr[i].compareTo(arr[j]) < 0) {
                    String t =
arr[j];
```

```
                    arr[j] =
arr[i];
                    arr[i] = t;
                }
        }
        System.out.println(arr[j]);
}
} }
```

indexOf и lastIndexOf

В класс **String** включена поддержка поиска определенного символа или подстроки, для этого в нем имеются два метода — **indexOf** и **lastIndexOf**. Каждый из этих методов возвращает индекс того символа, который вы хотели найти, либо индекс начала искомой подстроки. В любом случае, если поиск оказался неудачным методы возвращают значение -1. В очередном примере показано, как пользоваться различными вариантами этих методов поиска.

```
class indexOfDemo {
public static void main(String
args[]) {
String s = "Now is the time for
all good men " +
                "to come to the
aid of their country " +
```



```
                "and pay their due
taxes.";
System.out.println(s);
System.out.println("indexOf(t) = " +
s.indexOf('f'));
System.out.println("lastIndexOf(t) =
" + s.lastIndexOf('f'));
System.out.println("indexOf(the) = "
+ s.indexOf("the"));
System.out.println("lastIndexOf(the) =
" + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) =
" + s.indexOf('f' , 10));
System.out.println("lastIndexOf(t, 50)
= " + s.lastIndexOf('f' , 50));
System.out.println("indexOf(the, 10)
= " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the,
50) = " + s.lastIndexOf("the",
50));
} }
```

Ниже приведен результат работы этой программы. Обратите внимание на то, что индексы в строках начинаются с нуля.

```
C:> java indexOfDemo
Now is the time for all good men
```

```
to come to the aid of their
country
and pay their due taxes.
indexOf(t) = 7
lastIndexOf(t) = 87
indexOf(the) = 7
lastIndexOf(the) = 77
indexOf(t, 10) = 11
lastIndexOf(t, 50) = 44
indexOf(the, 10) = 44
lastIndexOf(the, 50) = 44
```

Модификация строк при копировании

Поскольку объекты класса **String** нельзя изменять, всякий раз, когда вам захочется модифицировать строку, придется либо копировать ее в объект типа **StringBuffer**, либо использовать один из описываемых ниже методов класса **String**, которые создают новую копию строки, внося в нее ваши изменения.

substring

Вы можете извлечь подстроку из объекта **String**, используя метод **substring**. Этот метод создает новую копию символов из того диапазона индексов оригинальной строки, который вы указали при вызове. Можно

указать только индекс первого символа нужной подстроки — тогда будут скопированы все символы, начиная с указанного и до конца строки. Также можно указать и начальный, и конечный индексы — при этом в новую строку будут скопированы все символы, начиная с первого указанного, и до (но не включая его) символа, заданного конечным индексом.

```
"Hello World".substring(6) -> "World"
"Hello World".substring(3,8) -> "lo
Wo"
```

concat

Слияние, или конкатенация строк выполняется с помощью метода **concat**. Этот метод создает новый объект **String**, копируя в него содержимое исходной строки и добавляя в ее конец строку, указанную в параметре метода.

```
"Hello".concat(" World") -> "Hello
World"
```

replace

Методу **replace** в качестве параметров задаются два символа. Все символы, совпадающие с первым, заменяются в новой копии строки на второй символ.

```
"Hello".replace('l' , 'w') ->
"Hewwo"
```

toLowerCase и toUpperCase

Эта пара методов преобразует все символы исходной строки в нижний и верхний регистр, соответственно.

```
"Hello".toLowerCase()  ->  "hello"
"Hello".toUpperCase()  ->  "HELLO"
```

trim

И, наконец, метод **trim** убирает из исходной строки все ведущие и замыкающие пробелы.

```
“Hello  World      “.trirn()  ->
"Hello  World"
```

valueOf

Если вы имеете дело с каким-либо типом данных и хотите вывести значение этого типа в удобочитаемом виде, сначала придется преобразовать это значение в текстовую строку. Для этого существует метод **valueOf**. Такой статический метод определен для любого существующего в Java типа данных (все эти методы совмещены, то есть используют одно и то же имя). Благодаря этому не составляет труда преобразовать в строку значение любого типа.

StringBuffer

StringBuffer — близнец класса **String**, предоставляющий многое из того, что обычно требуется при работе со строками. Объекты класса **String** представляют собой строки

фиксированной длины, которые нельзя изменять. Объекты типа **StringBuffer** представляют собой последовательности символов, которые могут расширяться и модифицироваться. Java активно использует оба класса, но многие программисты предпочитают работать только с объектами типа **String**, используя оператор **+**. При этом Java выполняет всю необходимую работу со **StringBuffer** за сценой.

Объект **StringBuffer** можно создать без параметров, при этом в нем будет зарезервировано место для размещения 16 символов без возможности изменения длины строки. Вы также можете передать конструктору целое число, для того чтобы явно задать требуемый размер буфера. И, наконец, вы можете передать конструктору строку, при этом она будет скопирована в объект и дополнительно к этому в нем будет зарезервировано место еще для 16 символов. Текущую длину **StringBuffer** можно определить, вызвав метод **length**, а для определения всего места, зарезервированного под строку в объекте **StringBuffer** нужно воспользоваться методом **capacity**. Ниже приведен пример, поясняющий это:

```
class  StringBufferDemo  {
public  static  void  main(String
args[])  {
StringBuffer  sb  =  new
```

```
StringBuffer("Hello");
System.out.println("buffer = " +
sb);
System.out.println("length = " +
sb.length());
System.out. println("capacity = " +
sb.capacity());
} }
```

Вот вывод этой программы, из которого видно, что в объекте **StringBuffer** для манипуляций со строкой зарезервировано дополнительное место.

```
C:\> java StringBufferDemo
buffer = Hello
length = 5
capacity = 21
```

ensureCapacity

Если вы после создания объекта **StringBuffer** захотите зарезервировать в нем место для определенного количества символов, вы можете для установки размера буфера воспользоваться методом **ensureCapacity**. Это бывает полезно, когда вы заранее знаете, что вам придется добавлять к буферу много небольших строк.

setLength

Если вам вдруг понадобится в явном виде установить длину строки в буфере, воспользуйтесь методом **setLength**. Если вы зададите значение, большее чем длина содержащейся в объекте строки, этот метод заполнит конец новой, расширенной строки символами с кодом нуль. В приводимой чуть дальше программе **setCharDemo** метод **setLength** используется для укорачивания буфера.

charAt и setCharAt

Одиночный символ может быть извлечен из объекта **StringBuffer** с помощью метода **charAt**. Другой метод **setCharAt** позволяет записать в заданную позицию строки нужный символ. Использование обоих этих методов проиллюстрировано в примере:

```
class setCharAtDemo {
public static void main(String
args[]) {
StringBuffer sb = new
StringBuffer("Hello");
System.out.println("buffer before = "
+ sb);
System.out.println("charAt(1) before
= " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
```

```
System.out.println("buffer after = "
+ sb);
System.out.println("charAt(1) after =
" + sb.charAt(1));
} }
```

Вот вывод, полученный при запуске этой программы.

```
C:\> java setCharAtDemo
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

append

Метод append класса **StringBuffer** обычно вызывается неявно при использовании оператора **+** в выражениях со строками. Для каждого параметра вызывается метод **String.valueOf** и его результат добавляется к текущему объекту **StringBuffer**. К тому же при каждом вызове метод **append** возвращает ссылку на объект **StringBuffer**, с которым он был вызван. Это позволяет выстраивать в цепочку последовательные вызовы метода, как это показано в очередном примере.

```
class appendDemo {
public static void main(String
args[]) {
```



```
String s;
int a = 42;
StringBuffer sb = new
StringBuffer(40);
s = sb.append("a =
").append(a).append("!").toString();
System.out.println(s);
} }
```

Вот вывод этого примера:

```
C:\> Java appendDemo
a = 42!
```

insert

Метод **insert** идентичен методу **append** в том смысле, что для каждого возможного типа данных существует своя совмещенная версия этого метода. Правда, в отличие от **append**, он не добавляет символы, возвращаемые методом **String.valueOf**, в конец объекта **StringBuffer**, а вставляет их в определенное место в буфере, задаваемое первым его параметром. В очередном нашем примере строка «there» вставляется между «hello» и «world!».

```
class insertDemo {
public static void
main(String args[]) {
StringBuffer sb = new
StringBuffer("hello world !");
```

```
sb.insert(6,"there ");
System.out.println(sb);
} }
```

При запуске эта программа выводит следующую строку:

```
C:\> java insertDemo
hello there world!
```

Почти любой аспект программирования в Java на каком либо этапе подразумевает использование классов **String** и **StringBuffer**. Они понадобятся и при отладке, и при работе с текстом, и при указании имен файлов и адресов URL в качестве параметров методам. Каждый второй байт большинства строк в Java — нулевой (Unicode пока используется редко). То, что строки в Java требуют вдвое больше памяти, чем обычные ASCII, не очень пугает, пока вам для эффективной работы с текстом в редакторах и других подобных приложениях не придется напрямую работать с огромным массивом типа **char**.

Обработка исключений

Исключение в Java — это объект, который описывает исключительное состояние, возникшее в каком-либо участке программного кода. Когда возникает исключительное состояние, создается объект класса **Exception**. Этот объект пересылается в метод,

обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вручную» для того, чтобы сообщить о некоторых нештатных ситуациях.

К механизму обработки исключений в Java имеют отношение 5 ключевых слов: **try**, **catch**, **throw**, **throws** и **finally**. Схема работы этого механизма следующая. Вы пытаетесь (**try**) выполнить блок кода, и если при этом возникает ошибка, система возбуждает (**throw**) исключение, которое в зависимости от его типа вы можете перехватить (**catch**) или передать умалчиваемому (**finally**) обработчику.

Ниже приведена общая форма блока обработки исключений.

```
try {
// блок кода }
catch (ТипИсключения1 e) {
// обработчик исключений типа
ТипИсключения1 }
catch (ТипИсключения2 e) {
// обработчик исключений типа
ТипИсключения2
  throw(e)     // повторное
возбуждение исключения }
finally {
}
```

Типы исключений

В вершине иерархии исключений стоит класс **Throwable**. Каждый из типов исключений является подклассом класса **Throwable**. Два непосредственных наследника класса **Throwable** делят иерархию подклассов исключений на две различные ветви. Один из них — класс **Exception** — используется для описания исключительных ситуации, которые должны перехватываться программным кодом пользователя. Другая ветвь дерева подклассов **Throwable** — класс **Error**, который предназначен для описания исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

Неперехваченные исключения

Объекты-исключения автоматически создаются исполняющей средой Java в результате возникновения определенных исключительных состояний. Например, очередная наша программа содержит выражение, при вычислении которого возникает деление на нуль.

```
class Exc0 {
public static void main(string
args[]) {
int d = 0;
int a = 42 / d;
} }
```


Вот вывод, полученный при запуске нашего примера.

```
C:\> java Exc0
java.lang.ArithmeticException: / by
zero
at Exc0.main(Exc0.java:4)
```

Обратите внимание на тот факт что типом возбужденного исключения был не **Exception** и не **Throwable**. Это подкласс класса **Exception**, а именно: **ArithmeticException**, поясняющий, какая ошибка возникла при выполнении программы. Вот другая версия того же класса, в которой возникает та же исключительная ситуация, но на этот раз не в программном коде метода **main**.

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String
args[]) {
Exc1.subroutine();
} }
```

Вывод этой программы показывает, как обработчик исключений исполняющей системы Java выводит содержимое всего стека вызовов.

```
C:\> java Exc1
java.lang.ArithmeticException: / by
zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

try и catch

Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово **try**. Сразу же после try-блока помещается блок **catch**, задающий тип исключения которое вы хотите обрабатывать.

```
class Exc2 {
public static void main(String
args[]) {
try {
      int d = 0;
       int a = 42 / d;
      }
catch (ArithmeticException e) {
System.out.println("division by
zero");
}
} }
```

Целью большинства хорошо сконструированных catch-разделов должна быть

обработка возникшей исключительной ситуации и приведение переменных программы в некоторое разумное состояние — такое, чтобы программу можно было продолжить так, будто никакой ошибки и не было (в нашем примере выводится предупреждение – division by zero).

В некоторых случаях один и тот же блок программного кода может возбуждать исключения различных типов. Для того, чтобы обрабатывать подобные ситуации, Java позволяет использовать любое количество catch-разделов для try-блока. Наиболее специализированные классы исключений должны идти первыми, поскольку ни один подкласс не будет достигнут, если поставить его после суперкласса. Следующая программа перехватывает два различных типа исключений, причем за этими двумя специализированными обработчиками следует раздел **catch** общего назначения, перехватывающий все подклассы класса **Throwable**.

```
class MultiCatch {
public static void main(String
args[]) {
try {
        int a = args.length;
        System.out.println("a = " +
a);
        int b = 42 / a;
```

```
        int c[] = { 1 };
        c[42] = 99;
      }
catch (ArithmeticException e) {
System.out.println("div by 0: " +
e);
}
catch(ArrayIndexOutOfBoundsException
e) {
System.out.println("array index oob:
" + e);
}
} }
```

Этот пример, запущенный без параметров, вызывает возбуждение исключительной ситуации деления на нуль. Если же мы зададим в командной строке один или несколько параметров, тем самым установив a в значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива **ArrayIndexOutOf Bounds**. Ниже приведены результаты работы этой программы, запущенной и тем и другим способом.

```
C:\> java MultiCatch
a = 0
```



```
div by 0:
java.lang.ArithmeticException: / by
zero
C:\> java MultiCatch 1
a = 1
array index oob:
java.lang.ArrayIndexOutOfBoundsExcepti
on: 42
```

Операторы **try** можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора **try** низкого уровня нет раздела **catch**, соответствующего возбужденному исключению, стек будет развернут на одну ступень выше, и в поисках подходящего обработчика будут проверены разделы **catch** внешнего оператора **try**. Вот пример, в котором два оператора **try** вложены друг в друга посредством вызова метода.

```
class MultiNest {
static void procedure() {
try {
        int c[] = { 1 };
        c[42] = 99;
}
catch(ArrayIndexOutOfBoundsException
e) {
```

```
System.out.println("array index oob:
" + e);
} }
public static void main(String
args[]) {
try {
        int a = args.length();
        System.out.println("a = " +
a);
        int b = 42 / a;
        procedure();
}
catch (ArithmeticException e) {
System.out.println("div by 0: " +
e);
}
} }
```

throw

Оператор **throw** используется для возбуждения исключения «вручную». Для того, чтобы сделать это, нужно иметь объект подкласса класса **Throwable**, который можно либо получить как параметр оператора **catch**, либо создать с помощью оператора **new**. Ниже приведена общая форма оператора **throw**.

```
throw ОбъектТипаThrowable;
```


При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок **try** проверяется на наличие соответствующего возбужденному исключению обработчика **catch**. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов **try**, и так до тех пор пока либо не будет найден подходящий раздел **catch**, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приведен пример, в котором сначала создается объект-исключение, затем оператор **throw** возбуждает исключительную ситуацию, после чего то же исключение возбуждается повторно — на этот раз уже кодом перехватившего его в первый раз раздела **catch**.

```
class ThrowDemo {
static void demoproc() {
try {
throw new
NullPointerException("demo");
}
catch (NullPointerException e) {
System.out.println("caught inside
demoproc");
```

```
throw e;
} }
public static void main(String
args[]) {
try {
demoproc();
}
catch(NulPointerException e) {
System.out.println("recaught: " + e);
}
} }
```

В этом примере обработка исключения проводится в два приема. Метод **main** создает контекст для исключения и вызывает **demoproc**. Метод **demoproc** также устанавливает контекст для обработки исключения, создает новый объект класса **NullPointerException** и с помощью оператора **throw** возбуждает это исключение. Исключение перехватывается в следующей строке внутри метода **demoproc**, причем объект-исключение доступен коду обработчика через параметр **e**. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощью оператора **throw**, в результате чего оно передается обработчику исключений в методе **main**. Ниже приведен результат, полученный при запуске этого примера.


```
C:\> java ThrowDemo
caught inside demoproc
recaught:
java.lang.NullPointerException: demo
```

throws

Если метод способен возбуждать исключения, которые он сам не обрабатывает, он должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений. Для задания списка исключений, которые могут возбуждаться методом, используется ключевое слово **throws**. Если метод в явном виде (т.е. с помощью оператора **throw**) возбуждает исключение соответствующего класса, тип класса исключений должен быть указан в операторе **throws** в объявлении этого метода. С учетом этого наш прежний синтаксис определения метода должен быть расширен следующим образом:

```
тип имя_метода(список аргументов)
throws список_исключений {}
```

Ниже приведен пример программы, в которой метод **procedure** пытается возбудить исключение, не обеспечивая ни программного кода для его перехвата, ни объявления этого исключения в заголовке метода. Такой программный код не будет оттранслирован.

```
class ThrowsDemo1 {
```

```
static void procedure() {
System.out.println("inside
procedure");
throw new
IllegalAccessException("demo");
}
public static void main(String
args[]) {
procedure();
} }
```

Для того, чтобы мы смогли оттранслировать этот пример, нам придется сообщить транслятору, что **procedure** может возбуждать исключения типа **IllegalAccessException** и в методе **main** добавить код для обработки этого типа исключений:

```
class ThrowsDemo {
static void procedure() throws
IllegalAccessException {
System.out.println(" inside
procedure");
throw new
IllegalAccessException("demo");
}
public static void main(String
```



```
args[]) {
try {
procedure();
}
catch (IllegalAccessException e) {
System.out.println("caught " + e);
}
} }
```

Ниже приведен результат выполнения этой программы.

```
C:\> java ThrowsDemo
inside procedure
caught
java.lang.IllegalAccessException: demo
```

finally

Иногда требуется гарантировать, что определенный участок кода будет выполняться независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода используется ключевое слово **finally**. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела **catch**, блок **finally** будет выполнен до того, как управление перейдет к операторам, следующим за разделом **try**. У каждого раздела **try** должен быть по крайней мере или один раздел **catch** или блок **finally**.

Блок **finally** очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода. Ниже приведен пример класса с двумя методами, завершение которых происходит по разным причинам, но в обоих перед выходом выполняется код раздела **finally**.

```
class FinallyDemo {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally {
System.out.println("procA's finally");
} }
static void procB() {
try {
System.out.println("inside procB");
return;
}
finally {
System.out.println("procB's finally");
} }
```



```
public static void main(String
args[]) {
try {
procA();
}
catch (Exception e) {}
procB();
} }
```

В этом примере в методе procA из-за возбуждения исключения происходит преждевременный выход из блока **try**, но по пути «наружу» выполняется раздел **finally**. Другой метод procB завершает работу выполнением стоящего в try-блоке оператора **return**, но и при этом перед выходом из метода выполняется программный код блока **finally**. Ниже приведен результат, полученный при выполнении этой программы.

```
C:\> java FinallyDemo
inside procA
procA's finally
inside procB
procB's finally
```

Подклассы Exception

Только подклассы класса **Throwable** могут быть возбуждены или перехвачены. Простые

типы — **int**, **char** и т.п., а также классы, не являющиеся подклассами **Throwable**, например, **String** и **Object**, использоваться в качестве исключений не могут. Наиболее общий путь для использования
исключений — создание своих собственных подклассов класса **Exception**.
Ниже приведена программа, в которой объявлен новый подкласс класса **Exception**.

```
class MyException extends Exception
{
private int detail;
MyException(int a) {
detail = a:
}
public String toString() {
return "MyException[" + detail +
"]";
}
}
class ExceptionDemo {
static void compute(int a) throws
MyException {
System.out.println("called computer +
a + ").");
if (a > 10)
```



```
throw new MyException(a);
System.out.println("normal exit.");
}
public static void main(String
args[]) {
try {
compute(1);
compute(20);
}
catch (MyException e) {
System.out.println("caught" + e);
}
} }
```

Этот пример довольно сложен. В нем сделано объявление подкласса **MyException** класса **Exception**. У этого подкласса есть специальный конструктор, который записывает в переменную объекта целочисленное значение, и совмещенный метод **toString**, выводящий значение, хранящееся в
объекте-исключении. Класс **ExceptionDemo** определяет метод **compute**, который возбуждает исключение типа **MyExcepton**. Простая логика метода **compute** возбуждает исключение в том случае, когда значение параметра метода больше 10. Метод **main** в защищенном блоке вызывает метод **compute** сначала с допустимым

значением, а затем — с недопустимым (больше 10), что позволяет продемонстрировать работу при обоих путях выполнения кода. Ниже приведен результат выполнения программы.

```
C:\> java ExceptionDemo
called compute(1).
normal exit.
called compute(20).
caught MyException[20]
```

Обработка исключений предоставляет исключительно мощный механизм для управления сложными программами. **Try**, **throw**, **catch** дают вам простой и ясный путь для встраивания обработки ошибок и прочих нештатных ситуаций в программную логику. Если вы научитесь должным образом использовать рассмотренные механизмы, это придаст вашим классам профессиональный вид, и любые будущие пользователи вашего программного кода, несомненно, оценят это.

Легковесные процессы и синхронизация

Параллельное программирование, связанное с использованием легковесных процессов, или подпроцессов (**multithreading**, **light-weight processes**) — концептуальная парадигма, в

которой вы разделяете свою программу на два или несколько процессов, которые могут исполняться одновременно.

Во многих средах параллельное выполнение заданий представлено в том виде, который в операционных системах называется многозадачностью. Это совсем не то же самое, что параллельное выполнение подпроцессов. В многозадачных операционных системах вы имеете дело с полновесными процессами, в системах с параллельным выполнением подпроцессов отдельные задания называются легковесными процессами (**light-weight processes**, **threads**).

Цикл обработки событий в случае единственного подпроцесса

В системах без параллельных подпроцессов используется подход, называемый циклом обработки событий. В этой модели единственный подпроцесс выполняет бесконечный цикл, проверяя и обрабатывая возникающие события. Синхронизация между различными частями программы происходит в единственном цикле обработки событий. Такие среды называют синхронными управляемыми событиями системами. Apple Macintosh, Microsoft Windows, X11/Motif — все эти среды построены на модели с циклом обработки событий.

Если вы можете разделить свою задачу на независимо выполняющиеся подпроцессы и можете автоматически переключаться с одного подпроцесса, который ждет наступления события, на другой, которому есть чем заняться, за тот же промежуток времени вы выполните больше работы. Вероятность того, что больше чем одному из подпроцессов одновременно надолго потребуется процессор, мала.

Модель легковесных процессов в Java

Исполняющая система Java в многом зависит от использования подпроцессов, и все ее классовые библиотеки написаны с учетом особенностей программирования в условиях параллельного выполнения подпроцессов. Java использует подпроцессы для того, чтобы сделать среду программирования асинхронной. После того, как подпроцесс запущен, его выполнение можно временно приостановить (**suspend**). Если подпроцесс остановлен (**stop**), возобновить его выполнение невозможно.

Приоритеты подпроцессов

Приоритеты подпроцессов — это просто целые числа в диапазоне от 1 до 10 и имеет смысл только соотношения приоритетов различных подпроцессов. Приоритеты же используются для того, чтобы решить, когда нужно остановить один подпроцесс и начать

выполнение другого. Это называется переключением контекста. Правила просты. Подпроцесс может добровольно отдать управление — с помощью явного системного вызова или при блокировании на операциях ввода-вывода, либо он может быть приостановлен принудительно. В первом случае проверяются все остальные подпроцессы, и управление передается тому из них, который готов к выполнению и имеет самый высокий приоритет. Во втором случае, низкоприоритетный подпроцесс, независимо от того, чем он занят, приостанавливается принудительно для того, чтобы начал выполняться подпроцесс с более высоким приоритетом.

Синхронизация

Поскольку подпроцессы вносят в ваши программы асинхронное поведение, должен существовать способ их синхронизации. Для этой цели в Java реализовано элегантное развитие старой модели синхронизации процессов с помощью монитора.

Сообщения

Коль скоро вы разделили свою программу на логические части — подпроцессы, вам нужно аккуратно определить, как эти части будут общаться друг с другом. Java предоставляет для этого удобное средство — два подпроцесса могут «общаться»

друг с другом, используя методы **wait** и **notify**. Работать с параллельными подпроцессами в Java несложно. Язык предоставляет явный, тонко настраиваемый механизм управления созданием подпроцессов, переключения контекстов, приоритетов, синхронизации и обмена сообщениями между подпроцессами.

Подпроцесс

Класс **Thread** инкапсулирует все средства, которые могут вам потребоваться при работе с подпроцессами. При запуске Java-программы в ней уже есть один выполняющийся подпроцесс. Вы всегда можете выяснить, какой именно подпроцесс выполняется в данный момент, с помощью вызова статического метода **Thread.currentThread**. После того, как вы получите дескриптор подпроцесса, вы можете выполнять над этим подпроцессом различные операции даже в том случае, когда параллельные подпроцессы отсутствуют. В очередном нашем примере показано, как можно управлять выполняющимся в данный момент подпроцессом.

```
class CurrentThreadDemo {
public static void main(String
args[]) {
Thread t = Thread.currentThread();
t.setName("My Thread");
System.out. println("current thread:
```



```
" + t);
try {
for (int n = 5; n > 0; n--) {
System.out.println(" " + n);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("interrupted");
}
} }
```

В этом примере текущий подпроцесс хранится в локальной переменной t. Затем мы используем эту переменную для вызова метода **setName**, который изменяет внутреннее имя подпроцесса на «My Thread», с тем, чтобы вывод программы был удобочитаемым. На следующем шаге мы входим в цикл, в котором ведется обратный отсчет от 5, причем на каждой итерации с помощью вызова метода **Thread.sleep()** делается пауза длительностью в 1 секунду. Аргументом для этого метода является значение временного интервала в миллисекундах, хотя системные часы на многих платформах не позволяют точно выдерживать интервалы короче 10 миллисекунд. Обратите внимание — цикл заключен в **try/catch** блок. Дело в том, что метод **Thread.sleep()** может возбуждать исключение **InterruptedException**.

Это исключение возбуждается в том случае, если какому-либо другому подпроцессу понадобится прервать данный подпроцесс. В данном примере мы в такой ситуации просто выводим сообщение о перехвате исключения. Ниже приведен вывод этой программы:

```
C:\> java CurrentThreadDemo
current thread: Thread[My
Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что в текстовом представлении объекта **Thread** содержится заданное нами имя легковесного процесса — **My Thread**.
Число 5 — это приоритет подпроцесса, оно соответствует приоритету по умолчанию, **main** — имя группы подпроцессов, к которой принадлежит данный подпроцесс.

Runnable

Не очень интересно работать только с одним подпроцессом, а как можно создать еще один? Для этого нам понадобится другой экземпляр класса **Thread**. При создании нового объекта **Thread** ему нужно указать, какой

программный код он должен выполнять. Вы можете запустить подпроцесс с помощью любого объекта, реализующего интерфейс **Runnable**. Для того, чтобы реализовать этот интерфейс, класс должен предоставить определение метода **run**. Ниже приведен пример, в котором создается новый подпроцесс.

```
class ThreadDemo implements Runnable
{
ThreadDemo() {
Thread ct = Thread.currentThread();
System.out.println("currentThread: "
+ ct);
Thread t = new Thread(this, "Demo
Thread");
System.out.println("Thread created: "
+ t);
t.start();
try {
Thread.sleep(3000);
}
catch (InterruptedException e) {
System.out.println("interrupted");
}
System.out.println("exiting main
```

```
thread");
}
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("" + i);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("child
interrupted");
}
System.out.println("exiting child
thread");
}
public static void main(String
args[]) {
new ThreadDemo();
} }
```

Обратите внимание на то, что цикл внутри метода **run** выглядит точно так же, как и в предыдущем примере, только на этот раз он выполняется в другом подпроцессе. Подпроцесс **main** с помощью оператора **new**

Thread создает новый объект класса **Thread**, причем первый параметр конструктора — **this** — указывает, что нам хочется вызвать метод **run** текущего объекта. Затем мы вызываем метод **start**, который запускает подпроцесс, выполняющий метод **run**. После этого основной подпроцесс (**main**) переводится в состояние ожидания на три секунды, затем выводит сообщение и завершает работу. Второй подпроцесс — «**Demo Thread**» — при этом по-прежнему выполняет итерации в цикле метода **run** до тех пор пока значение счетчика цикла не уменьшится до нуля. Ниже показано, как выглядит результат работы этой программы этой программы после того, как она отработает 5 секунд.

```
C:\> java ThreadDemo
Thread created: Thread[Demo
Thread,5,main]
5
4
3
exiting main thread
2
1
exiting child thread
```

Если вы хотите добиться от Java предсказуемого независимого от платформы

поведения, вам следует проектировать свои подпроцессы таким образом, чтобы они по своей воле освобождали процессор. Ниже приведен пример с двумя подпроцессами с различными приоритетами, которые не ведут себя одинаково на различных платформах. Приоритет одного из подпроцессов с помощью вызова **setPriority** устанавливается на два уровня выше **Thread.NORM_PRIORITY**, то есть, умалчиваемого приоритета. У другого подпроцесса приоритет, наоборот, на два уровня ниже. Оба этих подпроцесса запускаются и работают в течение 10 секунд. Каждый из них выполняет цикл, в котором увеличивается значение переменной-счетчика. Через десять секунд после их запуска основной подпроцесс останавливает их работу, присваивая условию завершения цикла **while** значение **true** и выводит значения счетчиков, показывающих, сколько итераций цикла успел выполнить каждый из подпроцессов.

```
class Clicker implements Runnable {
int click = 0;
private Thread t;
private boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
```



```
}
public void run() {
while (running) {
click++;
} }
public void stop() {
running = false; }
public void start() {
t.start();
} }
class HiLoPri {
public static void main(String
args[]) {
Thread.currentThread().setPriority(Thr
ead.MAX_PRIORITY);
clicker hi = new
clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new
clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try Thread.sleep(-10000) {
}
catch (Exception e) {
```

```
}
lo.stop();
hi.stop();
System.out.println(lo.click + " vs.
" + hi.click);
} }
```

По значениям, фигурирующим в распечатке, можно заключить, что подпроцессу с низким приоритетом достается меньше на 25 процентов времени процессора:

```
C:\>java HiLoPri
304300 vs. 4066666
```

Синхронизация

Когда двум или более подпроцессам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), нужно позаботиться о том, чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из подпроцессов. Java для такой синхронизации предоставляет уникальную, встроенную в язык программирования поддержку. В других системах с параллельными подпроцессами существует понятие монитора. Монитор — это объект, используемый как защелка. Только один из подпроцессов может в данный момент времени владеть монитором. Когда подпроцесс получает эту защелку,

говорят, что он вошел в монитор. Все остальные подпроцессы, пытающиеся войти в тот же монитор, будут заморожены до тех пор пока подпроцесс-владелец не выйдет из монитора.

У каждого Java-объекта есть связанный с ним неявный монитор, а для того, чтобы войти в него, надо вызвать метод этого объекта, отмеченный ключевым словом **synchronized**. Для того, чтобы выйти из монитора и тем самым передать управление объектом другому подпроцессу, владелец монитора должен всего лишь вернуться из синхронизованного метода.

```
class Callme {
void call(String msg) {
System.out.println("[" + msg);
try Thread.sleep(-1000) {}
catch(Exception e) {}
System.out.println("]");
} }
class Caller implements Runnable {
String msg;
Callme target;
public Caller(Callme t, String s) {
target = t;
msg = s;
```

```
new Thread(this).start();
}
public void run() {
target.call(msg);
} }
class Synch {
public static void main(String
args[]) {
Callme target = new Callme();
new Caller(target, "Hello.");
new Caller(target, "Synchronized");
new Caller(target, "World");
}
}
```

Вы можете видеть из приведенного ниже результата работы программы, что **sleep** в методе **call** приводит к переключению контекста между подпроцессами, так что вывод наших 3 строк-сообщений перемешивается:

```
[Hello.
[Synchronized
]
[World
]
]
```


Это происходит потому, что в нашем примере нет ничего, способного помешать разным подпроцессам вызывать одновременно один и тот же метод одного и того же объекта. Для такой ситуации есть даже специальный термин — **race condition** (состояние гонки), означающий, что различные подпроцессы пытаются опередить друг друга, чтобы завершить выполнение одного и того же метода. В этом примере для того, чтобы это состояние было очевидным и повторяемым, использован вызов **sleep**. В реальных же ситуациях это состояние, как правило, трудноуловимо, поскольку непонятно, где именно происходит переключение контекста, и этот эффект менее заметен и не всегда воспроизводятся от запуска к запуску программы. Так что если у вас есть метод (или целая группа методов), который манипулирует внутренним состоянием объекта, используемого в программе с параллельными подпроцессами, во избежание состояния гонки вам следует использовать в его заголовке ключевое слово **synchronized**.

Приоритеты подпроцессов

В Java имеется элегантный механизм общения между подпроцессами, основанный на методах **wait**, **notify** и **notifyAll**. Эти методы реализованы, как
final-методы класса **Object**, так что они имеются в любом Java-классе. Все эти методы должны

вызываться только из синхронизованных методов. Правила использования этих методов очень просты:

wait

Приводит к тому, что текущий подпроцесс отдает управление и переходит в режим ожидания — до тех пор пока другой подпроцесс не вызовет метод **notify** с тем же объектом.

notify

Выводит из состояния ожидания первый из подпроцессов, вызвавших **wait** с данным объектом.

notifyAll

Выводит из состояния ожидания все подпроцессы, вызвавшие **wait** с данным объектом.

Ниже приведен пример программы с наивной реализацией проблемы поставщик-потребитель. Эта программа состоит из четырех простых классов: **класса Q**, представляющего собой нашу реализацию очереди, доступ к которой мы пытаемся синхронизовать; **поставщика** (класс **Producer**), выполняющегося в отдельном подпроцессе и помещающего данные в очередь; **потребителя** (класс **Consumer**), тоже представляющего собой подпроцесс и извлекающего данные из очереди; и, наконец, крохотного **класса PC**, который создает по одному объекту каждого из

перечисленных классов.

```
class Q {
int n;
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out. println("Put: " + n);
} }
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while (true) {
q.put(i++);
} } }
class Consumer implements Runnable {
```

```
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while (true) {
q.get();
}
} }
class PC {
public static void main(String
args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
} }
```

Хотя методы **put** и **get** класса Q синхронизованы, в нашем примере нет ничего, что бы могло помешать поставщику переписывать данные по того, как их получит потребитель, и наоборот, потребителю ничего не мешает многократно считывать одни и те же данные. Так что вывод программы содержит вовсе не ту последовательность сообщений, которую нам бы хотелось иметь:


```
C:\> java PC
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

Как видите, после того, как поставщик помещает в переменную n значение 1, потребитель начинает работать и извлекает это значение 5 раз подряд. Положение можно исправить, если поставщик будет при занесении нового значения устанавливать флаг, например, заносить в логическую переменную значение true, после чего будет в цикле проверять ее значение до тех пор пока поставщик не обработает данные и не сбросит флаг в false.

Правильным путем для получения того же результата в Java является использование вызовов **wait** и **notify** для передачи сигналов в обоих направлениях. Внутри метода **get** мы ждем (вызов **wait**), пока **Producer** не известит нас (**notify**), что для нас готова очередная порция данных. После того, как мы обработаем эти данные в методе **get**, мы извещаем объект класса **Producer** (снова вызов **notify**) о том, что он может передавать следующую порцию данных. Соответственно, внутри метода **put**, мы ждем (**wait**), пока **Consumer** не обработает данные, затем мы передаем новые данные и извещаем (**notify**) об этом объект-потребитель. Ниже приведен переписанный указанным образом класс Q.

```
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if (!valueSet)
try wait();
catch(InterruptedException e):
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
```



```
synchronized void put(int n) {
if (valueSet)
try wait();
catch(InterruptedException e);
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
} }
```

А вот и результат работы этой программы, ясно показывающий, что синхронизация достигнута.

```
C:\> java Pcsynch
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Клинч (deadlock)

Клинч — редкая, но очень трудноуловимая ошибка, при которой между двумя легковесными процессами существует кольцевая зависимость от пары синхронизированных объектов. Например, если один подпроцесс получает управление объектом X, а другой — объектом Y, после чего X пытается вызвать любой синхронизированный метод Y, этот вызов, естественно блокируется. Если при этом и Y попытается вызвать синхронизированный метод X, то программа с такой структурой подпроцессов окажется заблокированной навсегда. В самом деле, ведь для того, чтобы один из подпроцессов захватил нужный ему объект, ему нужно снять свою блокировку, чтобы второй подпроцесс мог завершить работу.

Сводка функций программного интерфейса легковесных процессов

Методы класса

Методы класса — это статические методы, которые можно вызывать непосредственно с именем класса **Thread**.

currentThread

Статический метод **currentThread** возвращает объект **Thread**, выполняющийся в данный момент.

yield

Вызов метода **yield** приводит к тому, что исполняющая система переключает контекст с текущего на следующий доступный подпроцесс. Это один из способов гарантировать, что низкоприоритетные подпроцессы когда-нибудь получат управление.

sleep(int n)

При вызове метода **sleep** исполняющая система блокирует текущий подпроцесс на n миллисекунд. После того, как этот интервал времени закончится, подпроцесс снова будет способен выполняться. В большинстве исполняющих систем Java системные часы не позволяют точно выдерживать паузы короче, чем 10 миллисекунд.

Методы объекта

start

Метод **start** говорит исполняющей системе Java, что необходимо создать системный контекст подпроцесса и запустить этот подпроцесс. После вызова этого метода в новом контексте будет вызван метод run вновь созданного подпроцесса.

Вам нужно помнить о том, что метод **start** с данным объектом можно вызвать только один раз.

run

Метод **run** — это тело выполняющегося подпроцесса.
Это — единственный метод интерфейса **Runnable**. Он вызывается из метода **start** после того, как исполняющая среда выполнит необходимые операции по инициализации нового подпроцесса. Если происходит возврат из метода **run**, текущий подпроцесс останавливается.

stop

Вызов метода **stop** приводит к немедленной остановке подпроцесса.
Это — способ мгновенно прекратить выполнение текущего подпроцесса, особенно если метод выполняется в текущем подпроцессе. В таком случае строка, следующая за вызовом метода **stop**, никогда не выполняется, поскольку контекст подпроцесса «умирает» до того, как метод **stop** возвратит управление. Более аккуратный способ остановить выполнение подпроцесса — установить значение какой-либо переменной-флага, предусмотрев в методе **run** код, который, проверив состояние флага, завершил бы выполнение подпроцесса.

suspend

Метод **suspend** отличается от метода **stop** тем, что метод приостанавливает выполнение подпроцесса, не разрушая при этом его системный контекст. Если выполнение подпроцесса приостановлено вызовом **suspend**, вы можете снова активизировать этот подпроцесс, вызвав метод **resume**.

resume

Метод **resume** используется для активизации подпроцесса, приостановленного вызовом **suspend**. При этом не гарантируется, что после вызова **resume** подпроцесс немедленно начнет выполняться, поскольку в этот момент может выполняться другой более высокоприоритетный процесс. Вызов **resume** лишь делает подпроцесс способным выполняться, а то, когда ему будет передано управление, решит планировщик.

setPriority(int p)

Метод **setPriority** устанавливает приоритет подпроцесса, задаваемый целым значением передаваемого методу параметра. В классе **Thread** есть несколько предопределенных приоритетов-констант: **MIN_PRIORITY**, **NORM_PRIORITY** и **MAX_PRIORITY**, соответствующих соответственно значениям 1, 5 и 10. Большинство пользовательских приложений должно выполняться на уровне **NORM_PRIORITY** плюс-минус 1. Приоритет

фоновых заданий, например, сетевого ввода-вывода или перерисовки экрана, следует устанавливать в **MIN_PRIORITY**. Запуск подпроцессов на уровне **MAX_PRIORITY** требует осторожности. Если в подпроцессах с таким уровнем приоритета отсутствуют вызовы **sleep** или **yield**, может оказаться, что вся исполняющая система Java перестанет реагировать на внешние раздражители.

SetPriority

Этот метод возвращает текущий приоритет подпроцесса — целое значение в диапазоне от 1 до 10.

setName(String name)

Метод **setName** присваивает подпроцессу указанное в параметре имя. Это помогает при отладке программ с параллельными подпроцессами. Присвоенное с помощью **setName** имя будет появляться во всех трассировках стека, которые выводятся при получении интерпретатором неперехваченного исключения.

getName

Метод **getName** возвращает строку с именем подпроцесса, установленным с помощью вызова **setName**.

Простые в использовании встроенные в исполняющую среду и в синтаксис Java легковесные процессы — одна из наиболее

веских причин, по которым стоит изучать этот язык. Освоив однажды параллельное программирование, вы уже никогда не захотите возвращаться назад к программированию с помощью модели, управляемой событиями.

Утилиты

Библиотека классов языка включает в себя набор вспомогательных классов, широко используемых в других встроенных пакетах Java. Эти классы расположены в пакетах java.lang и java.util. Они используются для работы с наборов объектов, взаимодействия с системными функциями низкого уровня, для работы с математическими функциями, генерации случайных чисел и манипуляций с датами и временем.

Простые оболочки для типов

Как вы уже знаете, Java использует встроенные примитивные типы данных, например, **int** и **char** ради обеспечения высокой производительности. Эти типы данных не принадлежат к классовой иерархии Java. Они передаются методам по значению, передать их по ссылке невозможно. По этой причине для каждого примитивного типа в Java реализован специальный класс.

Number

Абстрактный класс **Number** представляет собой интерфейс для работы со всеми стандартными скалярными типами: — **long**, **int**, **float** и **double**.

У этого класса есть методы доступа к содержимому объекта, которые возвращают (возможно округленное) значение объекта в виде значения каждого из примитивных типов:

- **doubleValue()** возвращает содержимое объекта в виде значения типа **double**.
- **floatValue()** возвращает значение типа **float**.
- **intValue()** возвращает значение типа **int**.
- **longValue()** возвращает значение типа **long**.

Double и Float

Double и **Float** — подклассы класса **Number**. В дополнение к четырем методам доступа, объявленным в суперклассе, эти классы содержат несколько сервисных функций, которые облегчают работу со значениями **double** и **float**. У каждого из классов есть конструкторы, позволяющие инициализировать объекты значениями типов **double** и **float**, кроме того, для удобства пользователя, эти объекты можно инициализировать и объектом **String**,

содержащим текстовое представление вещественного числа. Приведенный ниже пример иллюстрирует создание представителей класса **Double** с помощью обоих конструкторов.

```
class DoubleDemo {
public static void main(String
args[]) {
Double d1 = new Double(3.14159);
Double d2 = new Double("314159E-5");
System.out.println(d1 + " = " + d2
+ " -> " + d1.equals(d2));
} }
```

Как вы можете видеть из результата работы этой программы, метод **equals** возвращает значение true, а это означает, что оба использованных в примере конструктора создают идентичные объекты класса **Double**.

```
C:\> java DoubleDemo
3.14159 = 3.14159 -> true
```

Бесконечность и NaN

В спецификации IEEE для чисел с вещественной точкой есть два значения типа **double**, которые трактуются специальным образом: бесконечность и NaN (Not a Number — неопределенность). В классе **Double** есть тесты для проверки обоих этих условий, причем в двух формах — в виде методов (статических), которым значение **double**

передается в качестве параметра, и в виде методов, проверяющих число, хранящееся в объекте класса **Double**.

- **islnfinite(d)** возвращает true, если абсолютное значение указанного числа типа **double** бесконечно велико.
- **islnfinite()** возвращает true, если абсолютное значение числа, хранящегося в данном объекте **Double**, бесконечно велико.
- **isNaN(d)** возвращает true, если значение указанного числа типа **double** не определено.
- **isNaN()** возвращает true, если значение числа, хранящегося в данном объекте **Double**, не определено.

Очередной наш пример создает два объекта **Double**, один с бесконечным, другой с неопределенным значением.

```
class InfNaN {
public static void main(String
args[]) {
Double d1 = new Double(1/0.);
Double d2 = new Double(0/0.);
System.out.println(d1 + ": " +
d1.isInfinite() + ", " +
d1.isNaN());
System.out.println(d2 + ": " +
```



```
d2.isInfinite() + ", " +
d2.isNaN());
} }
```

Ниже приведен результат работы этой программы:

```
C:\> java InfNaN
Infinity: true, false
NaN: false, true
```

Integer и Long

Класс **Integer** — класс-оболочка для чисел типов **int**, **short** и **byte**, а класс **Long** — соответственно для типа **long**. Помимо наследуемых методов своего суперкласса **Number**, классы **Integer** и **Long** содержат методы для разбора текстового представления чисел, и наоборот, для представления чисел в виде текстовых строк. Различные варианты этих методов позволяют указывать основание (систему счисления), используемую при преобразовании. Обычно используются двоичная, восьмеричная, десятичная и шестнадцатеричная системы счисления.

- **parseInt(String)** преобразует текстовое представление целого числа, содержащееся в переменной **String**, в значение типа **int**. Если строка не содержит представления целого числа, записанного в допустимом формате, вы получите исключение **NumberFormatException**.

■ **parseInt(String, radix)** выполняет ту же работу, что и предыдущий метод, но в отличие от него с помощью второго параметра вы можете указывать основание, отличное от 10.

■ **toString(int)** преобразует переданное в качестве параметра целое числов текстовое представление в десятичной системе.

■ **toString(int, radix)** преобразует переданное в качестве первогопараметра целое число в текстовое представление в задаваемой вторым параметром системе счисления.

Character

Character — простой класс-оболочка типа **char**. У него есть несколько полезных статических методов, с помощью которых можно выполнять над символом различные проверки и преобразования.

■ **isLowerCase(char ch)** возвращает true, если символ-параметр принадлежит нижнему регистру (имеется в виду не просто диапазон a-z, но и символы нижнего регистра в кодировках, отличных от ISO-Latin-1).

■ **isUpperCase(char ch)** делает то же самое в случае символов верхнего регистра.

■ **isDigit(char ch)** и **isSpace(char ch)** возвращают true для цифр и пробелов, соответственно.

■ **toLowerCase(char ch)** и **oupperCase(char ch)** выполняют преобразования символов из верхнего в нижний регистр и обратно.

Boolean

Класс **Boolean** — это очень тонкая оболочка вокруг логических значений, она бывает полезна лишь в тех случаях, когда тип **boolean** требуется передавать по ссылке, а не по значению.

Перечисления

В Java для хранения групп однородных данных имеются массивы. Они очень полезны при использовании простых моделей доступа к данным. Перечисления же предлагают более совершенный объектно-ориентированный путь для хранения наборов данных сходных типов. Перечисления используют свой собственный механизм резервирования памяти, и их размер может увеличиваться динамически. У них есть интерфейсные методы для выполнения итераций и для просмотра. Их можно индексировать чем-нибудь более полезным, нежели простыми целыми значениями.

Интерфейс Enumeration

Enumeration — простой интерфейс, позволяющий вам обрабатывать элементы любой коллекции объектов. В нем задается два метода. Первый из них — метод **hasMoreElements**, возвращающий значение типа **boolean**. Он возвращает значение true, если в перечислении еще остались элементы, и false, если у данного элемента нет следующего.

Второй метод — **nextElement** — возвращает обобщенную ссылку на объект класса **Object**, которую, прежде чем использовать, нужно преобразовать к реальному типу содержащихся в коллекции объектов.

Ниже приведен пример, в котором используется класс **Enum**, реализующий перечисление объектов класса **Integer**, и класс **EnumerateDemo**, создающий объект типа **Enum**, выводящий все значения перечисления. Обратите внимание на то, что в объекте **Enum** не содержится реальных данных, он просто возвращает последовательность создаваемых им объектов **Integer**.

```
import java.util.Enumeration;
class Enum implements Enumeration {
private int count = 0;
private boolean more = true;
public boolean hasMoreElements() {
return more;
}
public Object nextElement() {
count++;
if (count > 4) more = false;
return new Integer(count);
} }
```



```
class EnumerateDemo {
public static void main(String
args[]) {
Enumeration enum = new Enum();
while (enum.hasMoreElements()) {
System.out.println(enum.nextElement())
;
}
} }
```

Вот результат работы этой программы:

```
C:\> java EnumerateDemo
1
2
3
4
5
```

Vector

Vector — это способный увеличивать число своих элементов массив ссылок на объекты. Внутри себя **Vector** реализует стратегию динамического расширения, позволяющую минимизировать неиспользуемую память и количество операций по выделению памяти. Объекты можно либо записывать в конец объекта **Vector** с помощью метода **addElement**, либо вставлять в указанную

индексом позицию методом **insertElementAt**. Вы можете также записать в **Vector** массив объектов, для этого нужно воспользоваться методом **copyInto**. После того, как в **Vector** записана коллекция объектов, можно найти в ней индивидуальные элементы с помощью методов **Contains**, **indexOf** и **lastIndexOf**. Кроме того методы **elementAt**, **firstElement** и **lastElement** позволяют извлекать объекты из нужного положения в объекте **Vector**.

Stack

Stack — подкласс класса **Vector**, который реализует простой механизм типа «первым вошел — первым вышел» (FIFO). В дополнение к стандартным методам своего родительского класса, **Stack** предлагает метод **push** для помещения элемента в вершину стека и **pop** для извлечения из него верхнего элемента. С помощью метода **peek** вы можете получить верхний элемент, не удаляя его из стека. Метод **empty** служит для проверки стека на наличие элементов — он возвращает true, если стек пуст. Метод **search** ищет заданный элемент в стеке, возвращая количество операция **pop**, которые требуются для того чтобы перевести искомый элемент в вершину стека. Если заданный элемент в стеке отсутствует, этот метод возвращает -1.

Ниже приведен пример программы, которая создает стек, заносит в него несколько объектов типа **Integer**, а затем извлекает их.


```
import java.util.Stack;
import java.util.EmptyStackException;
class StackDemo {
static void showpush(Stack st, int
a) {
st.push(new Integer(a));
System.out.println("push(" + a +
")");
System.out.println("stack: " + st);
}
static void showpop(Stack st) {
System.out.print("pop -> ");
Integer a = (Integer) st.pop();
System.out.println(a);
System.out.println("stack: " + st);
}
public static void main(String
args[]) {
Stack st = new Stack();
System.out.println("stack: " + st);
showpush(st, 42);
showpush(st, 66);
showpush(st, 99);
showpop(st);
```

```
showpop(st);
showpop(st);
try {
showpop(st);
}
catch (EmptyStackException e) {
System.out.println("empty stack");
} }
}
```

Ниже приведен результат, полученный при запуске этой программы. Обратите внимание на то, что обработчик исключений реагирует на попытку извлечь данные из пустого стека. Благодаря этому мы можем аккуратно обрабатывать ошибки такого рода.

```
C:\> java StackDemo
stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
```



```
pop -> 66
stack: [42]
pop -> 42
stack: []
pop -> empty stack
```

Dictionary

Dictionary (словарь) — абстрактный класс, представляющий собой хранилище информации типа «ключ-значение». Ключ — это имя, по которому осуществляется доступ к значению. Имея ключ и значение, вы можете записать их в словарь методом **put(key, value)**. Для получения значения по заданному ключу служит метод **get(key)**. И ключи, и значения можно получить в форме перечисления (объект **Enumeration**) методами **keys** и **elements**. Метод **size** возвращает количество пар «ключ-значение», записанных в словаре, метод **isEmpty** возвращает true, если словарь пуст. Для удаления ключа и связанного с ним значения предусмотрен метод **remove(key)**.

HashTable

HashTable — это подкласс **Dictionary**, являющийся конкретной реализацией словаря. Представителя класса **HashTable** можно использовать для хранения произвольных объектов, причем для индексации в этой коллекции также годятся любые объекты. Наиболее часто **HashTable** используется для

хранения значений объектов, ключами которых служат строки (то есть объекты типа **String**). В очередном нашем примере в HashTable хранится информация об этой книге.

```
import java.util.Dictionary;
import java.util.Hashtable;
class HTDemo {
public static void main(String
args[]) {
Hashtable ht = new Hashtable();
ht.put("title", "The Java Handbook");
ht.put("author", "Patrick Naugnton");
ht.put("email",
"naughton@starwave.com");
ht.put("age", new Integer(30));
show(ht);
}
static void show(Dictionary d) {
System.out.println("Title: " +
d.get("title"));
System.out.println("Author: " +
d.get("author"));
System.out.println("Email: " +
d.get("email"));
System.out.println("Age: " +
```



```
d.get("age"));
} }
```

Результат работы этого примера иллюстрирует тот факт, что метод **show**, параметром которого является абстрактный тип **Dictionary**, может извлечь все значения, которые мы занесли в **ht** внутри метода **main**.

```
C:\> java HTDemo
Title: The Java Handbook
Author: Patrick Naughton
Email: naughton@starwave.com
Age: 30
```

Properties

Properties — подкласс **HashTable**, в который для удобства использования добавлено несколько методов, позволяющих получать значения, которые,

возможно, не определены в таблице. В методе **getProperty** вместе с именем можно указывать значение по умолчанию:

```
getProperty("имя","значение_по_умолчан
ию");
```

При этом, если в таблице свойство «имя» отсутствует, метод вернет «значение_по_умолчанию». Кроме того, при создании нового объекта этого класса конструктору в качестве параметра можно

передать другой объект **Properties**, при этом его содержимое будет использоваться в качестве значений по умолчанию для свойств нового объекта. Объект **Properties** в любой момент можно записать либо считать из потока — объекта **Stream**. Ниже приведен пример, в котором создаются и впоследствии считываются некоторые свойства:

```
import java.util.Properties;
class PropDemo {
static Properties prop = new
Properties();
public static void main(String
args[]) {
prop.put("Title", "put title here");
prop.put("Author", "put name here");
prop.put("isbn", "isbn not set");
Properties book = new
Properties(prop);
book.put("Title", "The Java
Handbook");
book.put("Author", "Patrick
Naughton");
System.out.println("Title: " +
book.getProperty("Title"));
System.out.println("Author: " +
```



```
book.getProperty("Author"));
System.out.println("isbn: " +
book.getProperty("isbn"));
System.out.println("ean: " +
book.getProperty("ean", "???"));
} }
```

Здесь мы создали объект **prop** класса **Properties**, содержащий три значения по умолчанию для полей **Title**, **Author** и **isbn**. После этого мы создали еще один объект **Properties** с именем **book**, в который мы поместили реальные значения для полей **Title** и **Author**. В следующих трех строках примера мы вывели результат, возвращенный методом **getProperty** для всех трех имеющихся ключей. В четвертом вызове **getProperty** стоял несуществующий ключ «ean». Поскольку этот ключ отсутствовал в объекте **book** и в объекте по умолчанию **prop**, метод **getProperty** выдал нам указанное в его вызове значение по умолчанию, то есть «**???**»:

```
C:\> java PropDemo
Title: The Java Handbook
Author: Patrick Naughton
isbn: isbn not set
ean: ???
```

StrinsTokenizer

Обработка текста часто подразумевает разбиение текста на последовательность лексем — слов (**tokens**). Класс **StringTokenizer** предназначен для такого разбиения, часто называемого лексическим анализом или сканированием. Для работы **StringTokenizer** требует входную строку и строку символов-разделителей. По умолчанию в качестве набора разделителей используются обычные символы-разделители: пробел, табуляция, перевод строки и возврат каретки. После того, как объект **StringTokenizer** создан, для последовательного извлечения лексем из входной строки используется его метод **nextToken**. Другой метод — **hasMoreTokens** — возвращает true в том случае, если в строке еще остались неизвлеченные лексемы. **StringTokenizer** также реализует интерфейс **Enumeration**, а это значит, что вместо методов **hasMoreTokens** и **nextToken** вы можете использовать методы **hasMoreElements** и **nextElement**, соответственно.

Ниже приведен пример, в котором для разбора строки вида «ключ=значение» создается и используется объект **StringTokenizer**. Пары «ключ=значение» разделяются во входной строке двоеточиями.

```
import java.util.StringTokenizer;

class STDemo {

static String in = "title=The Java
Handbook:" + "author=Patrick
```



```
Naughton:" + "isbn=0-07-882199-1:" +
"ean=9 780078 821998:" +
"email=naughton@starwave. corn";

public static void main(String
args[]) {

StringTokenizer st = new
StringTokenizer(in, "=:");

while (st.hasMoreTokens()) {

String key = st.nextToken();

String val = st.nextToken();

System.out.println(key + "\t" +
val);

}

} }
```

Runtime

Класс **Runtime** инкапсулирует интерпретатор Java. Вы не можете создать нового представителя этого класса, но можете, вызвав его статический метод, получить ссылку на работающий в данный момент объект **Runtime**. Обычно апплеты и другие непривелигированные программы не могут вызвать ни один из методов этого класса, не возбудив при этом исключения **SecurityException**. Одна из простых вещей, которую вы можете проделать с объектом **Runtime** — его останов, для этого достаточно вызвать метод **exit(int code)**.

Управление памятью

Хотя Java и представляет собой систему с автоматической сборкой мусора, вы для проверки эффективности своего кода можете захотеть узнать, каков размер «кучи» и как много в ней осталось свободной памяти. Для получения этой информации нужно воспользоваться методами **totalMemory** и **freeMemory**.

При необходимости вы можете «вручную» запустить сборщик мусора, вызвав метод **gc**. Если вы хотите оценить, сколько памяти требуется для работы вашему коду, лучше всего сначала вызвать **gc**, затем **freeMemory**, получив тем самым оценку свободной памяти, доступной в системе. Запустив после этого свою программу и вызвав **freeMemory** внутри нее, вы увидите, сколько памяти использует ваша программа.

Выполнение других программ

В безопасных средах вы можете использовать Java для выполнения других полновесных процессов в своей многозадачной операционной системе. Несколько форм метода **exec** позволяют задавать имя программы и ее параметры.

В очередном примере используется специфичный для Windows вызов **exec**,

запускающий процесс **notepad** — простой текстовый редактор. В качестве параметра редактору передается имя одного из исходных файлов Java. Обратите внимание — exec автоматически преобразует в строке-пути символы «/» в разделители пути в Windows — «\».

```
class ExecDemo {
public static void main(String
args[]) {
Runtime r = Runtime. getRuntime();
Process p = null;
String cmd[] = { "notepad",
"/java/src/java/lang/Runtime.java" };
try {
p = r.exec(cmd);
} catch (Exception e) {
System.out.println("error executing "
+ cmd[0]);
}
} }
```

System

Класс **System** содержит любопытную коллекцию глобальных функций и переменных. В большинстве примеров этой книге для операций вывода мы использовали метод **System.out.println()**.

Метод **currentTimeMillis** возвращает текущее системное время в виде миллисекунд, прошедших с 1 января 1970 года.

Метод **arraycopy** можно использовать для быстрого копирования массива любого типа из одного места в памяти в другое. Ниже приведен пример копирования двух массивов с помощью этого метода.

```
class ACDemo {
static byte a[] = { 65, 66, 67,
68, 69, 70, 71, 72, 73, 74 };
static byte b[] = { 77, 77, 77,
77, 77, 77, 77, 77, 77, 77 };
public static void main(
String args[]) {
System.out.println("a = " + new
String(a, 0));
System.out.println("b = " + new
String(b, 0));
System.arraycopy(a, 0, b, 0,
a.length);
System.out.println("a = " + new
String(a, 0));
System.out.println("b = " + new
String(b, 0));
System.arraycopy(a, 0, a, 1,
a.length - 1);
```



```
System.arraycopy(b, 1, b, 0,
b.length - 1);
System.out.println("a = " + new
String(a, 0));
System.out.println("b = " + new
String(b, 0));
} }
```

Как вы можете заключить из результата работы этой программы, копирование можно выполнять в любом направлении, используя в качестве источника и приемника один и тот же объект.

```
C:\> java ACDemo
a = ABCDEFGHIJ
b = MMMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Апплеты

Апплеты — это маленькие приложения, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте, как часть документа HTML. Когда

апплет прибывает к клиенту, его доступ к ресурсам ограничен.

Ниже приведен исходный код канонической программы HelloWorld, оформленной в виде апплета:

```
import java.awt.*;
import java.applet.*;
public class HelloWorldApplet extends
Applet {
public void paint(Graphics g) {
g.drawString("Hello World!", 20, 20);
} }
```

Этот апплет начинается двумя строками, которые импортируют все пакеты иерархий java.applet и java.awt. Дальше в нашем примере присутствует метод **paint**, замещающий одноименный метод класса **Applet**. При вызове этого метода ему передается аргумент, содержащий ссылку на объект класса **Graphics**. Последний используется для прорисовки нашего апплета. С помощью метода **drawString**, вызываемого с этим объектом типа **Graphics**, в позиции экрана (20,20) выводится строка «Hello World».

Для того, чтобы с помощью браузера запустить этот апплет, нам придется написать несколько строк html-текста.

```
<applet code="HelloWorldApplet"
width=200 height=40>
```



```
</applet>
```

Вы можете поместить эти строки в отдельный html-файл (HelloWorldApplet.html), либо вставить их в текст этой программы в виде комментария и запустить программу appletviewer с его исходным текстом в качестве аргумента.

Для Ваших заметок

Научно-популярное издание

Серия книг «Хитрости и тонкости»

Карабин Петр Леонидович

Язык программирования Java: Создание интерактивных приложений для Internet

Налоговая льгота
«Общероссийский классификатор
ОК 005-93-ТОМ2 953000 — Книги и брошюры»

Лицензия ЛР № 068246 от 20.03.1999 г. Подписано в печать
20.03.2006. Формат 70x100/32. Бумага газетная.
Кол-во п.л. 7. Тираж 3000 экз.